TRESDLIB.DLL API

Contents:

****

1. Disclaimer

****

****

2. List of files

****

The ESD directory in Program Files includes the following files:

* TRESDLIB.DLL (The actual WIN32 DLL module)
* TRESDLIB.H (A 'C' header file for DLL)
* TRESDCMD.EXE (WIN32 executable command interpreter)
* DEV_SERIAL.TXT (Sample device descriptor file for COM1)
* DEV_PROXY.TXT (Device descriptor using proxy server)
* TRESDLIB_API.TXT (This document)
* TEST.BAT (Batch file starting TRESDCMD.EXE)

****

3. Purpose of this document

This text document serves as a guide for the functionality provided by the TRESDLIB dynamic library.

Throughout this document it is assumed that the reader is familiar with embedding DLL files into its developement suite. Also it is assumed that the reader has a fair knowledge of the legal requirements of fiscal invoice signing (Ref# POL 1257)

4.1. Functionality -- Overview

The DLL provides a set of functions that implement all the fiscal signature device related scopes of functionality, ie communication and legal requirements as follows:

* Implements all communication layers, based on Micrelec signature device protocol specification (Ref# mfsps_serial_v0224.TXT), including error handling and user notifications on errorous conditions.
* Implements the communication specification for redirection of the sign functions to a server on a local network or the internet (sign proxy).
* Implements the digital signature filekeeping requirements that the law specifies (Ref# POL 1257)
* Provides a way to retrieve visual information of a requested signature device.
* provides a minimal user interface of signature device control and status.

From developer point of view, the application can (by means of this library):

a) Sign a document and get the generated signature information.
b) Issue a closing Z (Daily fiscal closure) report
c) Popup a signature device control dialog for status view and header setting
d) Retrieve the time/date setting of the device
e) Retrieve information from the device
f) Enable/Disable or redirect the DLL's user interface

Here is the DLL function list (later paragraphs describe each in detail):

* FSL_Command
* FSL_SelectDevice
* FSL_SignDocument
* FSL_IssueZreport
* FSL_PopupControl
* FSL_ReleaseDevice
* FSL_ErrorToString
* FSL_SetProgress
* FSL_SetUI
* FSL_GetTime
* FSL_GetStat
* FSL_SetUnicode
* FSL_SetBackup
* FSL_Reports

****
4.1.1. Functionality -- Direct & proxy devices
****

The DLL supports two categories of connections, the direct connection, ie devices that are accessed directly by the DLL through their protocol interface, and the proxy connections, which are devices that are connected to a server process and the DLL forwards the application's requests to this server process.

Direct connection functionality differs from proxy connections because in proxy connections all maintainance, backup and file generation functionality is performed by the sign proxy server itself. Some functions that provided by this DLL are behaving differently due to this fact.

- Direct connection operating aspects:

* The connection interface details must be known to the DLL.
* DLL creates the required workspace.
* DLL creates the produced fiscal files.
* DLL handles device recovery operations.
* DLL perfrorms file checking for fiscal file integrity.
* Real-time data retrieved (status and time/date) are accurate.
* When using '**********' (the 'any' serial no) at a specific connection, the same device is accessed.

- Proxy connection operating aspects:

* The connection interface details are known only to the server.
* Server creates the required workspace, DLL does not.
* Server creates the produced fiscal files, DLL does not.
* Server handles device recovery operations.
* Server performs file checking for fiscal file integrity.
* Real-time data retrieved are not guaranteed to be up to date, especially when the device is being used by many clients on the sign proxy server.
* When using '**********' (the 'any' serial no) the least busy device is accessed (ie can be different).

****
4.2. Functionality -- Operation modes
****

The library provides two operating modes that give the developer the freedom of choice between implementation simplicity over speed and efficiency.

These operation modes are:

a) The 'single command mode', where the application may call only the 'FSL_Command' function, which executes a complete fiscal signature operation, ie like sign a document or issue Z closure. For example, the application calling the DLL in single command mode will have the following execution profile (using simplified parameters):

```
FSL_Command("SIGN", "document_1")
FSL_Command("SIGN", "document_2")
FSL_Command("SIGN", "document_3")
```

..
FSL_Command("SIGN", "document_N")
FSL_Command("Z")

In this mode, *each* 'FSL_Command()' call performs the following steps:

1) Loads a device descriptor file (see 'Device files' for details)
2) Selects the device (Opens comms, prepares workspace)
3) Executes the specified command (Z Closures, document signing, etc)
4) Releases the device (Closes comms, releasing resouces)

As we can see, the advantage of this mode is that if we want a single command to be executed, this can be accomplished with a single function call; the disadvantage is that the steps 1,2 and 4 have to be repeated when more than one commands are desired to be executed at once. Another feature of this mode is that all parameters and data expected and returned by the command dispatcher are through files.

b) The 'block mode' is a mode that allows various operations to be executed over a selected device (via 'FSL_SelectDevice'), one after the other. In this case, one or more documents may be signed before releasing the device. This is provided because the 'FSL_SelectDevice' process performs various tasks (ie, like file integrity checking) which may proove a reason for delay when the file list grows longer. For example, the application calling the DLL using this mode will have the following execution profile:

```
+-- FSL_SelectDevice()                  // Select desired device
|    FSL_SignDocument()                  //   Sign document 1
|    FSL_SignDocument()                  //   Sign document 2
|    FSL_SignDocument()                  //   Sign document 3
|    ..                              ..
|    FSL_SignDocument()                  //   Sign document N
|    FSL_IssueZreport()                //   Close the day
|    ..
|    ..
+-- FSL_ReleaseDevice()                  // Release the device
```

The advantage of this mode is that it executes faster than a sequence of many 'FSL_Command()' calls that open, close and check the device every time. The disadvantage is that the minimum operation requires at least 3 steps to be completed:

1) FSL_SelectDevice()
2) FSL_SignDocument() (or others)
3) FSL_ReleaseDevice()

Another advantage of this mode is that the application may choose to select the device once at program start, and release the device before program termination.
The last difference with the 'single command mode' is that using this mode, the parameters are passed through structures and pointers rather than files.

As an implementation directive concerning the operation modes of the DLL, it is advised not combine the 'single' and the 'block' modes (ie: don't call the

'FSL_Command()' after having a device selected).

****
4.3. Functionality -- File keeping topology and backup conciderations
****

One aspect of the library is to automatically generate all files required by specification of the fiscal document signing. To do so, a path is required that point to an existing, specified by application, directory that will be used as 'signature data root directory', ie a workspace under where all files for all connected devices will be located.

For each device (when more than one device is connected) the library creates a subdirectory inside the signature data root named after the device's complete serial number (for example: ABC02000001). To form an example, lets assume that the 'signature data root directory' is given 'C:\My Signature Files' and we have two devices connected with serial numbers ABC02000001 and ABC02000002. In this case, after using the DLL for some time, some files and directories are found under the specified root that look like this:

```
**********************************************************
* <C:\>
*  |
*  +--- <C:\My Signature Files>
*       |
*       +--- <ABC02000001>
*       |    |
*       |    +-- ABC020000103031500010001_A.TXT
*       |    +-- ABC020000103031500010001_B.TXT
*       |    +-- ABC020000103031500010002_A.TXT
*       |    +-- ABC020000103031500010002_B.TXT
*       |    +-- ABC020000103031516580001_C.TXT
*       |    ..
*       |
*       +--- <ABC02000002>
*            |
*            +-- ABC020000203031600010001_A.TXT
*            +-- ABC020000203031600010001_B.TXT
*            +-- ABC020000203031500010002_A.TXT
*            +-- ABC020000203031500010002_B.TXT
*            +-- ABC020000203031610230001_C.TXT
*            ..
**********************************************************
```

These text files are the end user's fiscal documents and for this reason they *must* be protected by means of a safekeeping backup system. Integrity problems (or complete absence) of these files may cause the end user to have problems with fiscal authorities; by specification, losing one of these files is the same as losing a fiscal invoice document. The library makes all possible efforts to ensure the integrity of these files by the following means:

a) Before Z closure, the DLL checks if all daily _A.TXT files are having a companion _B.TXT file of the correct size. If not, the DLL regenerates the _B.TXT file from the device's maintained daily list of signature

entries. This ensures that during a day, if one - or even all - _B files are accidentaly deleted, they will be regenerated by the DLL.

b) The library checks if all closure files (_C.TXT) exist in the device's work directory, if any of those are missing they are regenerated by the DLL by means of reading the device's fiscal records that hold all required information for the regeneration.

Doing so, the DLL accomplishes to ensure that the files are operational and present after a complete work cycle (ie all signatures and files generated between two Z closures, for simplicity 'daily files'). It is *highly* recommended, that the application provides a backup mechanism, automatic or manual, that copies these files in another media (or location) after the end of this cycle. This is recommended for the following reasons:

a) The DLL has no means whatsoever to regenerate or restore files that belong to a previous Z closure (ie other than the current) because the device itself *clears* signature records after the Z closure in order to keep the new list. So, applying a backup procedure right after the daily Z closure, reduces dramatically the possibility of sensitive file loss.

b) After the backup, older _A.TXT and _B.TXT files can be deleted from the device subdirectory, thus preventing the number of generated files to reach the tenths (or even hundreds) of thousands. This helps much in eliminating delays introduced by the file system itself when too many files are kept in a directory that is constantly accessed (not to mention system errors). To have an estimation of the amount of files that are expected over time, lets make an imaginary example: assume a usual daily workload of a shop issuing fiscal invoices to 200, 25 working days per month. This gives a yearly estimation of invoices to:

200 invoices/day * 25 days/month * 12 months/year = 60.000 invoices/year

From file point of view this exceeds the 120.000 files per year (2 files generated for each invoice) all located under the device's subdirectory (if the files are not backed up elsewhere). Considering also the operating system's cluster allocating mechanism and the usual size of the _A and _B files, a *lot* of wasted space is accumulated there over time.

To sum things up, the best way for the application is to backup *_A.TXT and *_B.TXT files right after successful completion of a Z closure and delete them from the device's workspace directory. It is *highly* recommended that the applied backup system must be able to reproduce the _A, _B and _C files of a specific day.

****
4.4. Functionality -- User interface
****

The DLL commands open a small dialog box for informational reasons during the execution of a command. This dialog has no user controls; shows only the current progress of the function being executed. In single command mode, the dialog opens at start and closed just before returning control to the calling process. In block mode, the dialog opens at FSL_DeviceSelect() commands, stays open until the FSL_DeviceRelease() command. This window will be refered as

the 'progress' window.

The DLL provides two functions to control the behavior of the DLL's user interface, The FSL_SetProgress() and the FSL_SetUI() functions. These functions are used to:

a) enable or disable the 'progress' window
b) to specify different (application provided) functions for the simple user interface tasks (such as error message handler, retry/abort selector etc).

With these functions the calling entity can fully control the user interface behavior of the DLL.

****
4.5. Device files
****

The FSL_Command() requires a filename specifying a 'device descriptor file'. This file contains the information required for opening the physical connection with the device, and the device's identification information. The format of a device file is one line of plain ascii, with all information serarated by spaces. For example (quotes NOT included in the actual file):

"ABC02000001 S 1 c:\device_1_result.txt" <CR/LF>

Where:

* The first element is a string with the expected signature device serial number (ex: ABC02000001). If only the port and type is known to the calling entity, the '**********' (11 asterisks) should be used to specify that the serial number should be aquired by the DLL itself.
* The second element is a character specifying the device type, can be either 'S' (serial), or 'P' (Proxy) connections.
* The third element is the port or IP address. For serial type of connections this should be from 1 (for COM1) to N (N being the max COM number available to the specific system). Serial port baud rate can be optionally added (1:9600). For proxy connections, this must be the valid IP address of the proxy server.
* The forth element is the required result file's name and path and is optional. When this filename/path is missing, the default 'smout.txt' is used. If the filename contains absolute paths (eg 'c:\results\'), the file will be saved at the absolute location pointed to by the path part of the filename. If the file contains relative paths (eg 'results\d1.txt') these paths *must* reside in the specified data root directory. If the filename does not contain a path at all, the specified file will be created directly under the data root directory.

In this example, a device with serial number "ABC02000001" is expected to be found connected on port COM1 and the result of the operation will be stored at the root directory of c: with the name 'device_1_result.txt'. The FSL_Command() internaly loads this file into a SIGNDEVICE structure.
The device files are only used by FSL_Command() function. All block mode functions operate with structures directly, so there is no need to generate device files if you plan to use only the block mode.

****

5.1. Library reference -- FSL_Command() function
****

Declared as:
==========

```
SHORT WINAPI FSL_Command( BYTE *strBaseDir,
                          BYTE *strCommand,
                          BYTE *strDevFile,
                          BYTE *strParam1,
                          BYTE *strParam2
                        );
```

This command is used to issue a single complete command to a connected fiscal signature device. It is provided as a simplified alternative to the rest of the functions provided by the DLL. Its usage requires no other function call to complete.

After the function termination, the result file has been created (see description of device files for details). This file contains a string with the completion code of the command in it first two digits and a description of the success or the failure. If the code is other than '00', then an error occured and the command has failed at some point. The result code (as a numeric value) is also returned by the FSL_Command() function to the caller (for details of result codes see 'Function result codes').

All types of commands supported by this function, are performing the following steps:

a) Load and parse the device descriptor file
b) Select the device
c) Execute the specified command
d) Release the device

Parameters:
==========

'strBaseDir' Must be a string pointer to an absolute directory path, in which the library will create the files, depending on command (for more details over the directory usage, refer to #4.3).

'strCommand' Must be a string (upper or lower case) identifying the command. Valid command strings are:

* "SIGN" for generating the signature for a file. The sign command requires 'strParam1' to specify a filename for the input file and 'strParam2' to specify the output filename that will receive the signature stream that is generated. Input files *MUST* be in ELOT-928 encoding to satisfy specification. On successful completion of the command, the output file contains the string to be included at the last line of the printed fiscal document.

* "Z" for issuing the daily Z closure report. In this command, the 'strParam1' and 'strParam2' may be set to NULL (are ignored by the function). See also 5.4, FSL_IssueZreport() function.

* "CTL" for opening a control dialog box with the device.

The 'strParam1' and 'strParam2' may be set to NULL (are ignored by the function). This command is only available in non-proxy devices (ie directly connected to the host). In 'proxy' devices, the command will result success but will do nothing.

* "STAT" for reading the device's current status. The 'strParam1' is ignored and may be set to NULL, the 'strParam2' must contain the output file to store the status information retrieved by the device. The format of the status result is the following (without the quotes):

```
"ABC02000001 162 25093 38 1954 2362"
 |          |  |    | |   |
 |          |  |    | |   +--- The remaining signatures
 |          |  |    | |        until device needs Z
 |          |  |    | +--- The last signature's data size
 |          |  |    +--- # of signatures since last Z
 |          |  +--- # of signatures issued ever
 |          +--- Last Z closure number (1-2000)
 +--- The actual device's serial number
```

Notes for devices in sign proxy servers:
==========
When a device is of type 'proxy' ('P' type in device descriptor), the information returned by the DLL represents the device's status at the time the request was executed in the server. This means that two calls to FSL_Command() with "STAT" may *not* return the same information, if another client of the same network issues a call that affects this information (eg issue a sign command).

* "TIME" for reading the device's current date/time. The 'strParam1' is ignored and may be set to NULL, the 'strParam2' must contain the output file to store the time/date information retrieved by the device. Returned format is:

"DD/MM/YY HH:MM"

Notes for devices in sign proxy servers:
==========
When a device is of type 'proxy' ('P' type in device descriptor), the information returned by the DLL represents the device's time/date at the time the request was executed in the server.

* All other command identifiers are invalid producing a 'ERR_BADARGUMENT' error

'strDevFile' Must be a valid device descriptor file. (See 'device files' for details)

'strParam1' The 'input file' for signing when command is "SIGN", or NULL otherwise. Setting this parameter to NULL when required by

a command, produces a 'ERR_BADARGUMENT' error.

'strParam2' The 'output file' that will be generated after successful completion of the "SIGN" command, or NULL otherwise. Setting this parameter to NULL when required by a command, produces a 'ERR_BADARGUMENT' error.

Return values:
==========
A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
=========

```
//*************************************************************************
// Example of issuing a "SIGN" command.
// > Signs the file 'ELOT_FILE.TXT' using the device 'DEVICE_1.DAT' and
// > gets the signature stream into file 'SIGNATURE.DAT'.
//*************************************************************************
#include "tresdlib.h"
void myfunc(void)
{
 SHORT iRet;
 iRet = FSL_Command( "C:\\My Signature Files\\",         // Sign root dir
                     "SIGN",                            // Command
                     "F:\\My Devices\\Device_1.Dat",     // Use device
                     "F:\\Data Files\\ELOT_FILE.TXT",    // File to sign
                     "C:\\OUT\\Signature.Dat"            // Result file
                   );
 if(iRet == 0) {
   // Success
 } else {
   // Report it
 }
}
//*************************************************************************
// Example of issuing a "Z" closure command.
// > The device prints the closure report and closes the day
//*************************************************************************
#include "tresdlib.h"
void myfunc(void)
{
 SHORT iRet;
 iRet = FSL_Command( "C:\\My Signature Files\\",         // Sign root dir
                     "Z",                               // Command
                     "F:\\My Devices\\Device_1.Dat",     // Use device
                     NULL,                              // Unused
                     NULL                               // Unused
                   );
 if(iRet == 0) {
   // Success
 } else {
   // Report it
 }
```

```
}
//*************************************************************************
// Example of issuing a "CTL" dialog command.
// > Pops up the device specified in 'DEVICE_1' control dialog
//*************************************************************************
#include "tresdlib.h"
void myfunc(void)
{
 SHORT iRet;
 iRet = FSL_Command( "C:\\My Signature Files\\",         // Sign root dir
                     "CTL",                             // Command
                     "F:\\My Devices\\Device_1.Dat",     // Use device
                     NULL,                              // Unused
                     NULL                               // Unused
                   );
 if(iRet == 0) {
   // Success
 } else {
   // Report it
 }
}
//*************************************************************************
// Example of issuing a "STAT" command.
// > Gets the device's current status using device 'DEVICE_1.DAT' and
// > stores status info in file 'C:\STAT.TXT'
//*************************************************************************
#include "tresdlib.h"
void myfunc(void)
{
 SHORT iRet;
 iRet = FSL_Command( "C:\\My Signature Files\\",         // Sign root dir
                     "STAT",                            // Command
                     "F:\\My Devices\\Device_1.Dat",     // Use device
                     NULL,                              // Unused
                     "C:\STAT.TXT"                      // The output file
                   );
 if(iRet == 0) {
   // Success
 } else {
   // Report it
 }
}
//*************************************************************************
// Example of issuing a "TIME" command.
// > Gets the device's current time/date using device 'DEVICE_1.DAT' and
// > stores status info in file 'C:\TIME.TXT'
//*************************************************************************
#include "tresdlib.h"
void myfunc(void)
{
 SHORT iRet;
 iRet = FSL_Command( "C:\\My Signature Files\\",         // Sign root dir
                     "TIME",                            // Command
                     "F:\\My Devices\\Device_1.Dat",     // Use device
                     NULL,                              // Unused
```

```
                "C:\TIME.TXT"                   // The output file
              );
  if(iRet == 0) {
    // Success
  } else {
    // Report it
  }
 }
```

****
5.2. Library reference -- FSL_SelectDevice() function
****

Declared as:
==========

```
SHORT WINAPI FSL_SelectDevice(SIGNDEVICE *Dev, BYTE *strBaseDir);
```

This function is essential for preparing a device for any operation using the 'block mode' (refer to #4.2 for more info on operation modes). Without calling the FSL_SelectDevice(), all subsequent calls to block mode functions will fail with 'ERR_NOTSELECTED' error. Once this function is succesfully completed, the calling entity may issue any number of calls to the specific device until the FSL_ReleaseDevice() is finaly called.

Upon successfull return of the 'FSL_DeviceSelect()' function, members of 'Dev' structure are filled with data. Further calls to DLL's 'block mode' functions must have this structure as the device descriptor.

- Behavior in direct connected devices (SDTYPE_SERIAL):

The 'FSL_SelectFunction()' performs a series of procedures over the specified device. To be more specific, the function does the following:

* Opens the device's port. Essential for further device operation.
* Reads and verifies the device's expected serial number (if not 'any' device requested).
* Cancels any accidental signature block in progress. This is possible when the calling entity has crashed in the middle of a signature operation.
* Creates the device's data workspace (directory) if not present. The device workspace is the directory under the specified 'signature data root directory' named after the signature device's complete serial number (for example, ABC02000001).
* If the device has a CMOS RESET condition (as a result from a service operation), the DLL performs a full recovery of the daily files (ie the process of re-signing all daily files) as the legal specification requires.
* Changes the current directory to the device's workspace.

- Behavior in sign proxy type of devices (SDTYPE_PROXY):

The 'FSL_SelectFunction()' does nothing in proxy devices, but must be called whatsoever.

Parameters:
==========

'Dev' The device descriptor (refer to data types for more info).
'strBaseDir' Must be a string pointer to an absolute directory path, in which the library will find the device workspace (for more details over the directory usage, refer to #4.3).

Examples:
=========

```
//*************************************************************************
// Example of selecting a device with FSL_SelectDevice()
// The 'ABC02000001' device at COM1 is selected
//*************************************************************************
#include "tresdlib.h"
SIGNDEVICE Device;

void myfunc(void)
{
 SHORT     iRet;

 //=========================================================
 // Create the device connection info
 //=========================================================
 Device.Conn.iType = SDTYPE_SERIAL;                  // A serial device
 Device.Conn.wPort = 1;                          // At to COM1
 strcpy(Device.strSerialNo, "ABC02000001");          // Has this s/n

 //=========================================================
 // Select the device
 //=========================================================
 iRet = FSL_SelectDevice(&Device, "C:\\My Files");   // This does it

 //=========================================================
 // If succeeded, proceed with other device commands.
 // At this point we can use the 'Device' structure
 // for subsequent command
 //=========================================================
 if(iRet == 0) {
  // Other 'block mode' functions may follow
 } else {
  // inform for error
 }
}
```

****
5.3. Library reference -- FSL_SignDocument() function
****

Declared as:
============

```
SHORT WINAPI FSL_SignDocument(  SIGNDEVICE   *Dev,
                                BYTE         *strInfile,
                                BYTE         *strResult
                              );
```

The FSL_SignDocumet() function is used to a selected device for signing a document. The provided input file MUST be in ELOT928 encoding, containing only printable characters. The result string returned is the information that has to be printed at the last line of the fiscal document.

- Behavior in direct connected devices (SDTYPE_SERIAL):

After successful completion of this command, the DLL has created two files as the specification requires, the _A.TXT and _B.TXT files, which are located at the device's workspace. Also, the signature has been registered to the signarure device's daily list and a receipt has been issued by the device's printer mechanism.

- Behavior in sign proxy type of devices (SDTYPE_PROXY):

No fiscal files are generated localy; the fiscal files are generated at the sign proxy server's storage space.

Parameters:
==========

'Dev' The device descriptor (refer to data types for more info).

'strInfile' The file to generate signature. Can have a complete path, and *must* be absolute (ie: not having '..' parts).
For example:
"c:\test1\test2\file.txt" is valid
"..\file.txt" is invalid
"test\file.txt" is invalid

'strResult' Is a caller provided pointer to a string that will receive the generated signature string that the specification requires to be printed at the fiscal document's last line. The string has to have sufficient space to hold 78 characters (77 for the string data and one for the '\0' string terminator).

Return values:
==========

A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
=========

```
//*****************************************************************************
// Example of signing a document with FSL_SignDocument().
// > Assumes that the 'Device' is previously selected
//*****************************************************************************
#include "tresdlib.h"
```

```
SIGNDEVICE  Device;

void myfunc(void)
{
 SHORT           iRet;
 static BYTE      strSignature[78];

 iRet = FSL_SignDocument(
  &Device,
  "c:\\My Files\\Document1.TXT",
  strSignature
 );
 if(iRet == 0) {
  // Success
 } else {
  // Error: report it
 }
}
```

****
5.4. Library reference -- FSL_IssueZreport() function
****

Declared as:
==========

```
SHORT WINAPI FSL_IssueZreport( SIGNDEVICE *Dev );
```

The FSL_IssueZreport() command is used to issue a Z closure report on the selected device. After the successful completion of the command, the device will have printed the closure report by its printing mechanism. The device's day list of signatures is cleared so a new 'day' or 'work cycle' can begin.

- Behavior in directly connected devices (SDTYPE_SERIAL):

The DLL have generated a _C.TXT file with the required closure data. Also, some steps are taken automatically by the DLL to ensure that the files in the device's folder are present:

* If a day is open (ie at least a signature has been issued after a closure), performs an integrity check on daily signature files and regenerates them when damaged or missing. Specificaly, for every daily _A file existing, a check for existance and correct size is performed in the corresponding _B file. If the _B file is not okay or missing, the file is regenerated.
* Performs an integrity checking on closure _C files and regenerates them when missing reading the Z closure data from the device's fiscal memory.

- Behavior in sign proxy type of devices (SDTYPE_PROXY):

No files are generated localy, nor steps for for file regeneration; these actions are performed by the sign proxy server. It is possible for the sign proxy server to ignore or delay the closure request for sharing purposes.

Parameters:
==========
'Dev' The device descriptor (refer to data types for more info).

Return values:
==========
A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
==========

```
//*************************************************************************
// Example of issuing a Z report with FSL_IssueZreport().
// > Assumes that the 'Device' is previously selected
//*************************************************************************
#include "tresdlib.h"
SIGNDEVICE  Device;

void myfunc(void)
{
  SHORT iRet;

  iRet = FSL_IssueZreport( &Device );
  if(iRet == 0) {
    // Success
  } else {
    // Error: report it
  }
}
```

****
5.5. Library reference -- FSL_PopupControl() function
****

Declared as:
==========

```
SHORT WINAPI FSL_PopupControl( SIGNDEVICE *Dev );
```

The FSL_PopupControl() command is used to open a dialog box of the selected device which is usefull for finding out various signature device's information, status, and settings. Additionaly, at this dialog, the user can setup the printed header. This function does nothing in devices that are of 'proxy' type.

Parameters:
==========
'Dev' The device descriptor (refer to data types for more info).

Return values:
==========
A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
==========

```
//*************************************************************************
// Example of opening the control dialog with FSL_IssueZreport().
// > Assumes that the 'Device' is previously selected
//*************************************************************************
#include "tresdlib.h"
SIGNDEVICE  Device;

void myfunc(void)
{
 SHORT iRet;

 iRet = FSL_PopupControl( &Device );
 if(iRet == 0) {
  // Success
 } else {
  // Error: report it
 }
}
```

****
5.6. Library reference -- FSL_ReleaseDevice() function
****

Declared as:
==========

```
SHORT WINAPI FSL_ReleaseDevice( SIGNDEVICE *Dev );
```

The FSL_ReleaseDevice() command is used to close the physical connection with the signature device and release the resources allocated for it. If the specified device was not selected, the function will return success but no action has taken place.

Parameters:
==========

'Dev' The device descriptor (refer to 'data types' for more info).

Return values:
==========

A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
==========

```
//*************************************************************************
// Example of releasing the device with FSL_DeviceRelease().
//*************************************************************************
#include "tresdlib.h"
SIGNDEVICE  Device;
```

```
void myfunc(void)
{
 SHORT iRet;

 iRet = FSL_ReleaseDevice( &Device );
 if(iRet == 0) {
  // Success
 } else {
  // Error: report it
 }
}
```

****
5.7. Library reference -- FSL_ErrorToString() function
****

Declared as:
==========

```
void WINAPI FSL_ErrorToString( SHORT iRet, BYTE *strDescr, int iMaxLen );
```

The FSL_ErrorToString() command is used to convert a numeric, integer value returned by a DLL function to a readable string description. For details on error return values see 'function result codes'.

Parameters:
==========

'iRet' Should be a value returned from a DLL function call
'strDescr' Is a pointer to a string to receive the error description
'iMaxLen' Is the string space limit for the text copy in TCHARs (a supplied space of 100 characters should be enough).

Return values:
==========

None

Examples:
==========

```
//**********************************************************************
// Example of getting error description with FSL_ErrorToString().
//**********************************************************************
#include "tresdlib.h"
SIGNDEVICE  Device;

void myfunc(void)
{
 SHORT iRet;

 iRet = FSL_PopupControl( &Device );
 if(iRet == 0) {
  UserInform("Popup success");
 } else {
  UserInform("Failed opening device dialog: %s", FSL_ErrorToString(iRet));
```

```
    }
  }
```

****
5.8. Library reference -- FSL_SetProgress() function
****

Declared as:
==========

```
  void WINAPI FSL_SetProgress( int fEnable, void *fnProgress );
```

The FSL_SetProgress() function is used for a) enabling/disabling the progress window and/or specifying a different (caller provided) function for displaying progress information.

Parameters:
==========

'fEnable' a flag specifying if progress window is to be displayed (default setting is 1). If set to 0, no progress (user specified or the default) display will be shown.

'fnProgress' a pointer to an alternative user function that will be used to display the progress information. Passing a NULL to this pointer will force DLL to use the default progress window.

Progress function semantics:
==========

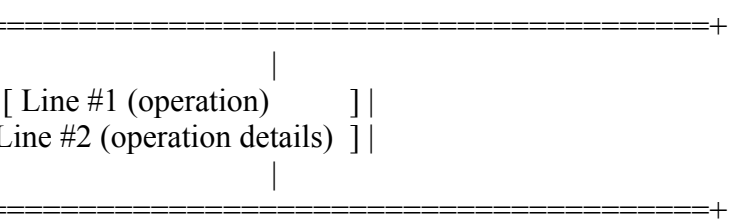
The progress function must be declated as:

```
void WINAPI ProgressInfo(INT32 iOperation, BYTE *strText);
```

where: 'iOperation' may be one of the following:

0 - Open progress indicator ('strText' must be ignored)
1 - Close progress indicator ('strText' must be ignored)
2 - Display text line #1 ('strText' is displayed in line 1)
3 - Display text line #2 ('strText' is displayed in line 2)

The default behavior/appearance of progress window is:



Return values:
==========

None

Examples:
==========

```
  //*********************************************************************
```

```
// Example of enabling progress status with a user specified function
// that redirects all progress messages to STDOUT
//*************************************************************************
#include "tresdlib.h"

void WINAPI MyProgressFn(INT32 iOperation, BYTE *strText)
{
 switch(iOperation) {
   case 0: case 1: /* No open/close required for std stream i/o  */
   return;
   case 2: fprintf(stdout, "* %s\r\n", strText); break;
   case 3: fprintf(stdout, "  %s\r\n", strText); break;
   default: break;
 }
}

void myfunc(void)
{
 FSL_SetProgress(1, MyProgressFn);
 FSL_Command(...);
 ..
 FSL_SetProgress(1, NULL);   // Restore to default (DLL's) function
 ..
 FSL_SetProgress(0, NULL);   // Turn off progress interfacing
}
```

****
5.9. Library reference -- FSL_SetUI() function
****

Declared as:
==========

```
void WINAPI FSL_SetUI( void *fnAbortRetry, void *fnError, void *fnWarning );
```

Provides a method to use alternative (specified by the calling process) routines for the DLL popup selections/warnings and/or error messages. A value of NULL to any of the function pointers causes the DLL to use the default user interface for the specified function.

Parameters:
==========

'fnAbortRetry' Must be a pointer to a function that should display a message to user and give him the choice of either RETRY or ABORT an operation. The function should be declared like:

```
SHORT WINAPI User_GetAbortRetry(  BYTE *strTitle,
                                  BYTE *strMessage );
```

The return value must be: 1 for user selection RETRY
0 for user selection ABORT

'fnError' Must be a pointer to a function that should inform the user for an error condition. No selection from user is required.

The function should be declared like:

```
void WINAPI User_MessageError( BYTE *strTitle,
                               BYTE *strMessage );
```

'fnWarning' Must be a pointer to a function that should inform the user for a warning condition. No selection from user is required. The function should be declared like:

```
void WINAPI User_MessageWarning( BYTE *strTitle,
                                 BYTE *strMessage );
```

Return values:
==============
None

Examples:
=========

```
//*************************************************************************
// Example of setting custom popup display functions
//*************************************************************************
#include "tresdlib.h"

SHORT WINAPI MyAbortRetry(BYTE *strTitle, BYTE *strMessage)
{
 int iRet = MessageBox(NULL, strMessage, strTitle, MB_RETRYCANCEL | MB_ICONERROR);
 if(iRet == IDRETRY) return(1);
 return(0);
}

void WINAPI MyMessageError(BYTE *strTitle, BYTE *strMessage)
{
 MessageBox(NULL, strMessage, strTitle, MB_OK | MB_ICONERROR);
}

void WINAPI MyMessageWarning(BYTE *strTitle, BYTE *strMessage)
{
 MessageBox(NULL, strMessage, strTitle, MB_OK | MB_ICONWARNING);
}

void myfunc(void)
{
 FSL_SetUI(MyAbortRetry, MyMessageError, MyMessageWarning);
 FSL_Command(..);
 FSL_Command(..);
 ..
 FSL_SetUI(NULL, NULL, NULL);   // Restore default interface
}
```

****
5.10. Library reference -- FSL_GetTime() function
****

Declared as:
==========

```
SHORT WINAPI FSL_GetTime( SIGNDEVICE *Dev, BYTE *pbResult );
```

The FSL_GetTime() function retrieves the current time/date from the specified device. The time/date information is returned in the string pointer specified by 'pbResult' in the following format: "DD/MM/YY HH:MM".

Parameters:
==========

'Dev' The device descriptor (refer to data types for more info).
'pbResult' A pointer to string of sufficient size to receive the time/date information.

Return values:
==========

A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
==========

```
//*************************************************************************
// Example of getting the time/date information from a (selected) device
//*************************************************************************
#include "tresdlib.h"
SIGNDEVICE Dev;

void myfunc(void)
{
 SHORT    iRet;
 BYTE     strTimeInfo[20];

 iRet = FSL_GetTime(&Dev, strTimeInfo);
 if(iRet == 0) printf("Device time: %s", strTimeInfo);
       else fprintf(stderr, "Failed retrieving device time (%s)", FSL_ErrorToString(iRet));
}
```

****
5.11. Library reference -- FSL_GetStat() function
****

Declared as:
==========

```
SHORT WINAPI FSL_GetStat( SIGNDEVICE *Dev, BYTE *pbResult );
```

The FSL_GetStat() function retrieves the current status of the specified device. The status information is returned in the string pointer specified by 'pbResult' in the following format:

```
"ABC02000001 162 25093 38 1954 2362"
|          | |   | | |
|          | |   | | +--- The remaining signatures
|          | |   | |      until device needs Z
```

```
|         |  |    | +--- The last signature's data size
|         |  |    +--- # of signatures since last Z
|         |  +--- # of signatures issued ever
|         +--- Last Z closure number (1-2000)
+--- The actual device's serial number
```

Parameters:
==========

'Dev' The device descriptor (refer to data types for more info).
'pbResult' A pointer to string of sufficient size to receive the status information.

Return values:
==========

A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
==========

```
//*************************************************************************
// Example of getting the status information from a (selected) device
//*************************************************************************
#include "tresdlib.h"
SIGNDEVICE Dev;

void myfunc(void)
{
 SHORT    iRet;
 BYTE     strStatusInfo[80];

 iRet = FSL_GetStat(&Dev, strStatusInfo);
 if(iRet == 0) printf("Device status: %s", strStatusInfo);
       else fprintf(stderr, "Failed retrieving device status (%s)", FSL_ErrorToString(iRet));
}
```

****
5.12. Library reference -- FSL_SetUnicode() function
****

Declared as:
==========

```
SHORT WINAPI FSL_SetUnicode( int fUnicode, BYTE * reserved );
```

The FSL_SetUnicode() function enables Unicode support for the library. By default for compatibility it is working in ASCII mode. When the function is called with fUnicode parameter = 1, all string parameters of FSL_* commands declared as BYTE * are assumed as Unicode. Exceptions are the strSerialNo member of SIGNDEVICE structure and the contents of input/output text files.

****
5.13. Library reference -- FSL_SetBackup() function
****

Declared as:

==========

VOID WINAPI FSL_SetBackup(BYTE * szBackupDir);

The FSL_SetBackup() function specifies path to fiscal files backup directory. Using backup directory is optional. It is recommended to use different volume than the default fiscal directory to avoid device failure. The function must be called before FSL_SelectDevice()

****
5.14. Library reference -- FSL_Reports() function
****

Declared as:
==========

SHORT WINAPI FSL_Reports(SIGNDEVICE * Dev, int iType, char * szParam);

The FSL_Reports() command is used to issue different report types supported by the device. It is extended version of FSL_IssueZreport() which supports only Z closure report.

Parameters:
==========
'Dev' The device descriptor (refer to data types for more info).
'szParam' Optional parameter.
'iType' The report type code. Possible values are:

0 - Recovery of _b and _c files without any report. 'szParam' is ignored;
1 - Z closure report. This is command 'x' from the serial protocol. 'szParam' contains two numeric parameters separated by '/'. To issue daily closure report set szParam to "0/0". The same report without date warning will be issued when szParam is "2/0". See the serial protocol description for more details;
2 - Fiscal report (date to date). This is command 'f' from the serial protocol. 'szParam' contains the start and end date (DDMMYY) - e.g. "010109/311209";
3 - Fiscal report (Z to Z) - command 'z' from the serial protocol. 'szParam' is Start Z to End Z numbers - e.g. "1/200";
4 - Ranged list of daily signatures - command 'X' from the serial protocol. Signature entries are specified in 'szParam' (e.g. "5/20").

Return values:
==========
A returned value of zero means success, otherwise indicates the reason of failure (see 'Function result codes' for details).

Examples:
==========

```
//*****************************************************************
// Example of issuing a X report (List of the registered signatures).
// > Assumes that the 'Device' is previously selected
//*****************************************************************
#include "tresdlib.h"
SIGNDEVICE  Device;
```

```
void myfunc(void)
{
  SHORT iRet;

  iRet = FSL_Reports( &Device, 1, "1/0" );
  if(iRet == 0) {
    // Success
  } else {
    // Error: report it
  }
}
```

****
6. Library reference -- Data types
****

Besides the simple data types used by this library, some structures are used also:

* SIGNDEVICE structure
  ==========
  Declared at TRESDLIB.H as:

```
typedef struct {                // ============= Device data ============
  SDCONN    Conn;               // Connection description
  BYTE      strSerialNo[12];    // Expected device serial number
  SDSTAT    Status;             // Last access device status and codes
  SDSUMMARY Summ;               // Summary info; signatures, closures etc
  BYTE      strWorkDir[2048];   // The directory to keep files
  DWORD     dwSelectMark;       // A value to keep track selected devices
} SIGNDEVICE;                   // ======================================
```

  Used for describing a device and for keeping run-time working information.
  The calling entity does not need to deal with all members of this type,
  just the ones that are required for describing the connection and expected
  serial number. All other structures and members are of no interest of the
  calling entity and should not be accessed/modified by it in any way.

* SDCONN structure
  ==========
  Declared at TRESDLIB.H as:

```
typedef struct {                // ===== Connection data ===========
  SHORT        iType;           // Hardware connection type
  WORD         wPort;           // Port of connection
  DWORD        dwIp;            // IP address of connection
  SOCKET       s;               // Socket of ip connection
  BYTE        *bWorkspace;      // Connection buffer
  BYTE         bConnData[53];   // Additional connection data
  WORD         wConnDataLen;    // Length of additional data
} SDCONN;                       // ================================
```

  Type used to describe the physical connection of the signature device.
  'iType' should be one of the defined (in TRESDLIB.H) types:

```
#define SDTYPE_SERIAL           1    // Serial connection
#define SDTYPE_ETHERNET         2    // Ethernet connection
#define SDTYPE_PROXY            3    // Device in proxy server
```

When type is SDTYPE_SERIAL the 'wPort' must contain the serial port number (eg 1 for COM1, etc) and the 'dwIp' is 0 or baud rate. When type is SDTYPE_PROXY, the 'wPort' is ignored; instead 'dwIp' is used to specify the IP address of the connection in network order.

Examples of setting up device connections:

```
// a serial device with serial number AAA03000009 connected in COM2
strcpy(Dev.strSerialNo, "AAA03000009");
Dev.conn.iType = SDTYPE_SERIAL;
Dev.conn.wPort = 2;
iRet = FSL_SelectDevice(&Dev, "c:\\files");

// Any device in a sign proxy server located at address 10.10.1.1
strcpy(Dev.strSerialNo, "**********");
Dev.conn.iType = SDTYPE_PROXY;
Dev.conn.dwIp  = inet_addr("192.168.0.2");
iRet = FSL_SelectDevice(&Dev, "c:\\files");
```

****

7. Library reference -- Function result codes

****

Most of the functions of this DLL produce a numeric value which indicates the result of the function called. A zero value indicates success and states the fact that the function has completed all work related to it with no errors. A non zero value indicates the failure of the requested operation, and its value is the failure information. The following table describes these reasons of failure.

Value	Macro defined in TRESDLIB.H	Reason
0	ERR_SUCCESS	No errors, success
1	ERR_BADARGUMENT	Bad parameter specified
2	ERR_FILEIO_ERROR	Error in file/filesystem operation
3	ERR_COMMUNICATION_ERROR	Communication with device failed
4	ERR_UNRECOVERABLE_ERROR	Hardware error: device needs service
5	ERR_UNEXPECTED_ERROR	Unexpected error, operation aborted
6	ERR_INVALIDDEVICEID	Serial number is not the one expected
7	ERR_INVALIDBASEDIR	Base directory specified is invalid
8	ERR_DEVFILELOAD_ERR	Failed loading device descriptor file
9	ERR_NOTSELECTED	Device specified is not selected
10	ERR_INVALIDINPUTFILE	File contains invalid characters
11	ERR_DEVICE_NOT_FOUND	The device specified is not connected
12	ERR_NETSYS_FAILED	Network I/O error
13	ERR_SERVER_UNAVAILABLE	Failure contacting proxy server
14	ERR_SERVER_COM_ERROR	Error communicating with proxy server

==========

8. Table of valid characters in input files

This table specifies which characters in range 0x00 to 0xFF are valid
when encountered in a given input file.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[A]	[B]	[C]	[D]	[E]	[F]
[0]	X	X	X	X	X	X	X	X	X	-	-	X	-	-	X	X
[1]	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X	X
[2]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[3]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[4]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[5]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[6]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[7]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X
[8]	-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
[9]	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
[A]	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	
[B]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[C]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[D]	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-
[E]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
[F]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X

'X' denotes an invalid value,
'-' denotes an acceptable value.

9. DLL specification

* The functions have a PASCAL calling convertion (same as win APIs).
* The functions provided by this DLL are *NOT* reentrant.
* Many of these functions are blocking.
* The complex data types (structures) expected must have a member alignment of 1 byte (ie no padding).
* This specification applies to the executables (DLL and LIB) included in this package; any changes at the source code of this DLL may modify these specification.