

TMS International BV

TMSi Data Acquisition SDK

Application interface to real-time data

TMSi develops, produces and sells, signal acquisition devices for medical use. These devices have one or more options of communication with a PC: Bluetooth, WLAN or USB 2.0. The options depend on the type of device.

This Software Development Kit is intended for people who want to write their own application software for viewing or analyzing the signals measured with the TMSi front ends. This SDK is installed when installing driver software for the your frontend, and contains an include file and a DLL.

This SDK supports only front ends which uses WLAN, Bluetooth and USB to connect to the PC.

For supported OS versions, see the installation manual.

This document belongs to SDK revision 7.2.144.0.

[TMS International BV](#)

Zuthpenstraat 57, 7575 EJ, Oldenzaal

The Netherlands

tel: +31 (0) 541 534603

fax: +31 (0) 541 534628

e-mail: info@tmsi.com

For comments or questions about this SDK
please contact: support@tmsi.com

xt : [Getting Started](#)

TMS International BV

etting started

The driver package provides support for a large variety of TMSi data acquisition devices. To simplify application development all communication with the devices is performed through a single Dynamic Link Library (DLL) that exports a limited number of functions.

In the directory ExampleCode on the Driver CD, you will find example code written in C, and the TMSiSDK.h include file with the prototypes of the functions.

The frontends can be connected to the PC using an WLAN, Bluetooth or USB connection.

All communication with the frontends is performed through a single DLL. This DLL is named '*TMSiSDK.DLL*' and is automatically installed in the Windows system directory during driver installation. This library exports functions for controlling the front ends. The functions use the Microsoft `__stdcall` calling convention, and have a "C" declaration, so they can be called from a C program.

Because the location of the Windows System directory may vary from system to system it is best to use the WIN32 API **GetSystemDirectory** to locate this library. A handle to the TMSiSDK.DLL can now be opened and the pointers to these functions loaded using the **GetProcAddress** function. The following example shows how a pointer to an exported function can be obtained.

Excerpt from the example code:

```
typedef BOOL ( __stdcall * PCLOSE )(HANDLE hHandle); // Prototype fr  
  
TCHAR Path[ MAX_PATH ];  
GetSystemDirectory(Path, sizeof(Path) / sizeof(TCHAR) );  
lstrcat(Path, "\\TMSiSDK.dll");  
HINSTANCE LibHandle = LoadLibrary( Path );  
PCLOSE Close = (PCLOSE) GetProcAddress( LibHandle , "Close" );
```

Refer to your Microsoft Windows Platform SDK for more information

about using dynamic link libraries and runtime linking. For a complete list of all available functions see the [functions](#) part of the SDK documentation.

In case loading any function exported by the DLL fails, use the WIN32 function **GetLastError** for more information.

The TMSiSDK.dll is not thread-safe. When using threads in your application, you must take care that functions are not called concurrently.

The example code demonstrates how several common tasks are done, like getting information about the serial number and name of the connected frontend, the channel layout and properties, setting the RTC clock etc.

TMS International BV

Changes between WDM driver v6.* and TMSi SDK 7.*

This TMSi SDK version 7 is the successor of WDM driver 6.*.

TMSi SDK version 7 is newly build from the ground up to provide an easier way to connect to TMSi front ends, and to unify access methods over different communication interfaces such as WLAN, Bluetooth and USB.

As a result of this, there are several significant differences between this driver and the WDM driver v6.*, which makes it impossible to simply recompile your current software to use this TMSi SDK version 7.

Compared to WDM driver 6.*, all functions which were marked as deprecated in the SDK of WDM driver v6.* are removed. These are functions `GetNrDevices`, `GetDevicePath`, `GetSADHandle`, `GetManufacturer`, `GetDescription`, `GetId`, `GetSignalInfo`, `GetDeviceKey`, and structures `DEVICE_FEATURE_GAIN`, `DEVICE_FEATURE_HIGHPASS`, `DEVICE_FEATURE_LOWPASS`, `DEVICE_FEATURE_OFFSET`.

It is no longer necessary to interrogate the registry to find out if a TMSi USB device is connected (like `Fusbi`, `Synfi` or `Mobita`).

Functions `GetInstanceId`, `OpenRegKey`, structures `SP_DEVICE_PATH` are not present in this SDK.

It is no longer possible to use master/slave devices directly using Fiber cables.

Functions `AddSlave`, `GetSlaveHandle` are not present in this SDK.

Function `GetSamples` is changed and now always delivers a sample value of 4 bytes per channel.

New functions are added for acquiring samples that are stored in the

internal memory of Mobita: CloseCardFile, OpenCardFile, StartCardFile, StopCardFile, GetCardFileSamples, GetCardFileSignalFormat, GetCardFileList.

For USB connected devices (like Fusbi, Synfi or Mobita) Windows only displays the device itself in the Windows device manager, and not the frontend(s) that is (are) connected to this device. So if you connect a Refa to a Fusbi, you will only see the Fusbi in the Windows device manager, not the Refa.

The average reference calculation in the driver is now deactivated by default for all devices. This means that all EXG channels now have the unmodified signals, i.e. without the common average removed from the signals. If you want the driver to perform the average reference calculation use the SetRefCalculation() function to turn it on.

WARNING: when using Synfi to acquire EXG signals from two TMSi frontends simultaneously, the average reference calculation in the driver should always be switched off. Please refer to the Synfi User Manual for more details.

Compilation

First install the TMSi SDK on your PC using the CD delivered with your frontend. Please follow the instructions in the installation manual carefully.

After installation, TMSiSDK.dll is located in the C:/Windows/system32 directory. The TMSi SDK also contains a TMSiSDK.h include file, which contains definitions of all structures and prototypes of all functions. This file is not installed, but must be copied from installation CD to the directory in which you keep your development files.

The TMSi SDK files are created using Visual Studio 2010 on a Windows 7 64-bit platform, and are compiled for Windows 7 and 8.1.

In order to test if you can compile, link, and run the SDK, first use the example code on the CD to see if you can communicate with your frontend. The example code has been tested thoroughly, which means that any error occurs during compilation, linking and running are most likely due to a misconfiguration in your development environment.

WARNING: If you run the ExampleCode.exe without Visual Studio 2010 installed, loading the TmsiSDK.dll will fail. Please install the vccredist_vs2010_x86.exe from the ExampleCode directory.

If the test with the example code was successful, you can start with writing code for this version of the SDK.

WARNING: The TMSiSDK.h file and SADIO.h file from the WDM 6.* driver have the same C structures. This is done to minimize the amount of work needed for development with this driver. As a result, including both TMSiSDK.h file and SADIO.h in one C file result in compile errors and warnings.

The concept of the SDK is that you start the library with the desired communication interface(WLAN, Bluetooth, USB), perform the desired

actions (acquiring samples, reading samples from memory card) and close the library. For each desired communication interface you need to start the library, but it is possible to start several libraries parallel to each other, with each library using a different communication interface. In order to select a device, call `GetDeviceList` to get a list of all connected TMSi devices.

The error handling of the library is designed to be as uniform as possible. For each function which returns a Boolean `FALSE`, you can call the `GetErrorCode()` function to retrieve the specific error code for that function.

The order in which you need to call functions from the SDK, and the naming of those functions is mostly unchanged with respect to the WDM driver 6.*. However, it is possible that parameters or return types have a different type.

If you do not have any code for the previous WDM driver 6.*, the fastest way to get started is to copy the example code, and modify that.

If you already have code used with the previous WDM driver 6.*, the fastest way to get started is to copy your code, rename it, remove the `SADIO.h` from the code, include `TMSiSDK.h`, and recompile. This will not work at once, because parameters or return types of functions can have a different type. First fix all code which gives a compiler error, using the example code and this reference documentation. Then fix all code which gives a compiler warning, using the example code and this reference documentation.

The following functions have changed:

`GetMeasuringMode` is still present, but does not work, and always returns `FALSE`.

`SetMeasuringMode` has changed parameters, and does work.

`SetSignalBuffer` is still present, does set the size of the buffer. Getting and setting the sample rate does work.

`GetSignalFormat` has a parameter changed.

`GetSamples` returns now a signed long.

`GetDeviceState` is removed.

Installation

During installation on a 32 bit platform, the TMSiSDK.dll is installed in the %windir%\System32 directory.

During installation on a 64 bit platform, the 64 bit TMSiSDK.dll is installed in the %windir%\System32 directory and the 32 bit TMSiSDK32bit.dll is installed in the %windir%\SysWOW64 directory.

See <http://msdn.microsoft.com/en-us/library/aa384249%28v=VS.85%29.aspx> for more information on how Windows handles 32-bit applications and dlls on 64-bit platforms.

Because the Microsoft installer enforces unique names for all files, it is not possible to use the TMSiSDK.dll name in both directories.

TMS International BV

Porting your program from PortiSerial to TMSi SDK v7.*

This TMSi SDK version 7 is the successor of the WDM driver 6.* AND Portiserial.dll. The latter is used to provide an application interface for TMSi frontends that communicate using a Bluetooth interface.

TMSi SDK version 7 is build from scratch to provide an easier way to connect to all TMSi frontends, and to unify access methods over different communication interfaces (WLAN, Bluetooth and USB).

As a result, there are huge differences between TMSi SDK version 7 and the existing PortiSerial.dll, which makes it impossible to just recompile your current software to use this TMSi SDK version 7.

Bluetooth driver

The TMSiSDK.dll works with the Microsoft Bluetooth stack ONLY. As Windows only allows one Bluetooth stack to be installed on the system, this means that a currently installed non-Microsoft Bluetooth stack needs to be deinstalled first. This can be done by removing a currently inserted Bluetooth stick, and deinstalling the non-microsoft Bluetooth driver (such as the Toshiba or Widdcom stack). Then re-insert your Bluetooth stick and install the Microsoft Bluetooth stack.

For internal Bluetooth devices, remove the device using the Windows Device Manager and deinstall the Bluetooth stack. However, we have not tested it, and can not guarantee that it will work properly. If your internal Bluetooth devices refuse to work with the Microsoft Bluetooth stack, use the Device Manager to disable the device, and use the TMSi supplied Bluetooth device.

Compilation

First install the TMSi SDK on your PC using the CD delivered with you frontend. Please follow the instructions in the installation manual carefully.

After installation, the TMSiSDK.dll is located in the C:/Windows/system32 directory. The TMSi SDK also includes the TMSiSDK.h include file, which contains definitions of all structures and prototypes of all functions. This file is not installed, but must be copied from cd to the directory in which you have your development files.

The TMSi SDK files are created using Visual Studio 2010 on a Windows 7 64-bit platform, and are compiled for Windows 7 and 8.1.

In order to test if you can compile, link, and run the SDK, first use the example code on the CD to see if you can communicate with the frontend. Because the example code has been tested thoroughly, it is likely that any error that occurs during compilation, linking and running is due to a misconfiguration in your development environment.

WARNING: If you run the ExampleCode.exe without Visual Studio 2010 installed, loading the TMSiSDK.dll will fail. In that case, install vcredist_vs2010_x86.exe from the ExampleCode directory.

If the test with the example code has been successful, you can start with writing code for this version of the SDK.

The concept of the SDK is that you start the library with the desired communication method (WLAN, Bluetooth, USB), perform the desired actions (acquiring samples, reading samples from memory card) and close the library. For each desired communication interface you need to start the library, but it is possible to start several libraries parallel to each other, with each library using a different communication interface. In order to select a device, call GetDeviceList() to get a list of available devices.

The error handling of the library is designed to be as uniform as possible. For each function that returns a Boolean FALSE, you can call the GetErrorCode() function to retrieve the specific error code for that function.

The order in which you need to call functions from the SDK, and the naming of those functions is different from the function names of PortiSerial.dll.

Porting

The PortiSerial.dll and PortiSerialSDK.dll are written in C++ with a COM interface. The TMSiSDK.dll is written in C code, and does not have a COM interface. Also, it does not need to be registered using regsvr32. The PortiSerial.dll used the Bluetooth stack from the driver CD of the Bluetooth dongle supplied by TMSi. The TMSiSDK.dll uses the Microsoft Bluetooth stack ONLY. If you do not use the Microsoft Bluetooth stack, it is not possible to detect TMSi Bluetooth frontends, and the function will fail.

The functions put_ComPort() and get_Comport() are not needed anymore. The functions GetDeviceList() and Open() are used to find a Bluetooth frontend, Close() is used to close a Bluetooth frontend. InitializeSerialPortAndFrontend() can be replaced by Open() and SetSignalBuffer().

get_Caption() can be replaced by GetSignalFormat().

put_Caption() has no equivalent function.

StartAcq() can be replaced by Start(). The sample rate must be set in SetSignalBuffer() before calling Start().

StopAcq() can be replaced by Stop().

GetSampleRecordAsVariant() can be replaced by GetSamples().

GetStationairSampleRecords() has no equivalent function.

KeepSerialPortInUse() has no equivalent function, as the TMSiSDK.dll performs this function itself.

ReleaseSerialPort() is replaced by Close().

TestBlock() has no equivalent function.

The functions GetFrontendNrOfChannels(),

get_FrontendNrOfChannels(), get_FrontendSerialNumber(),

get_FrontendId(), get_FrontendHardwareVersion(),

get_FrontendSoftwareVersion(), get_FrontendCommandBufferSize(),

get_FrontendSendBufferSize() are replaced by GetFrontEndInfo().

get_SampleRate() has no equivalent function.

put_SampleRate() has no equivalent function.

GetRawData() has no equivalent function.

ResetRawData() has no equivalent function.

Refer the functions part of the SDK documentation for a full description of the functions.

Installation

During installation on a 32 bit platform, the TMSiSDK.dll is installed in the %windir%\System32 directory.

During installation on a 64 bit platform, the 64 bit TMSiSDK.dll is installed in the %windir%\System32 directory and the 32 bit TMSiSDK32bit.dll is installed in the %windir%\SysWOW64 directory.

See <http://msdn.microsoft.com/en-us/library/aa384249%28v=VS.85%29.aspx> for more information on how Windows handles 32 bit applications and DLLs on 64 bit platforms.

Because the Microsoft installer enforces unique names for all files, it is not possible to use the TMSiSDK.dll name in both directories.

TMS International BV

Programming for the Synfi

The Synfi is a device which synchronizes sample data from two identical Signal Acquisition Devices, which are connected to the SynFi using fiber optical cables.

The concept of the Synfi is that it presents itself to the TMSi driver as any other Signal Acquisition Device, but connect two frontends simultaneously. Although we tried to implement that concept to the fullest, some small differences in behavior could not be avoided.

Number of channels, channel names and channel ordering

The Synfi combines (and synchronizes) the outputs of two Signal Acquisition Devices. During that process, there are some rules regarding the order in which the Signal Acquisition Devices and the channel names are displayed.

The SynFi has 2 Fiber ports, labeled “1” and “2”. The channel names of the Signal Acquisition Device connected to port 1 are prefixed by “F1_”. So if the Signal Acquisition Device has a channel name “Exg1”, the SynFi shall list the name of that channel as “F1_Exg1”. The channel names of the Signal Acquisition Device connected to port 2 are prefixed by “F2_”. So if the Signal Acquisition Device has a channel name “Exg7”, the SynFi shall list the name of that channel as “F2_Exg7”. The channel order of the Synfi is that first the channels of the Signal Acquisition Device connected to port 1 are shown, and then the channels of the Signal Acquisition Device connected to port 2 are shown.

The last channel of the Synfi carries the sawtooth, with channel name “/|/|”. The individual saw tooth channels of the connected Signal Acquisition Devices are not displayed. The sawtooth of the SynFi is created based upon the sawtooth of both connected Signal Acquisition Devices. If one or both of the Signal Acquisition Devices misses a sample, the sawtooth of the Synfi will proportionally show this. Example: The slave Signal Acquisition Device misses one sample. Then the sawtooth of the SynFi will also miss a sample.

Some Signal Acquisition Devices have one channel in which the digi input is combined with the sawtooth. The Synfi then removes the sawtooth from that channel, and keep the digi input. As a result of that action, and due to technical reasons, the `GetFrontEndInfo()` will return a lower number of channels than the `GetSignalFormat()`. You should always use the number of channels from the `GetSignalFormat()`.

The Digi channels of each Signal Acquisition Device are the last shown channels for that Signal Acquisition Device.

It is possible that applications of OEM's or customers re-order the channels in a way they find useful. This is the responsibility of the application itself, and should be described in the manual of the application.

Common reference calculation

To get the unmodified signal to the measuring application, the common reference calculation should not be performed by the driver when connecting the Synfi. The reason is that the driver can not make a difference between the channel sets of each frontend connected to the Synfi. So when using the Synfi, turn off the reference calculation by using `SetRefCalculation`.

This means that the Common Reference Derivation must be done in the application. To do this, we assume that the patient is connected as described in the Synfi Manual, i.e. the patient ground is connected to the Signal Acquisition Devices, and the Common Reference electrode is connected to channel 1 on both Signal Acquisition Devices. Channel 1 must be an EXG channel.

Because the signals coming from the Signal Acquisition Device are unprocessed, it is possible that signals show jumps in value when any other channels are switched off when signals on those channels are out of range. After the Common Reference Derivation is done, these jumps are removed.

For each received sample, for the Signal Acquisition Device connected

to port Fiber1, we subtract the value of channel 1 (channel F1_EXG1) from all other EXG channels. Then for the Signal Acquisition Device connected to port Fiber2, we subtract the value of channel 1 (channel F2_EXG1) from all other EXG channels.

After the Common Reference Derivation is done, the resulting values for all channels can be used for any kind of mathematical operation.

To find out if a Synfi is connected, check the serial number in the FrontEndInfo structure. Each TMSi frontend has a 10 digit serial number, starting with a zero. The first 4 digits of the serial number of a Synfi are 0393. Because the serial number in the FrontEndInfo structure is an integer, the leading zero is not shown.

Impedance measurement

The impedance measurement is done in two steps, assuming both Signal Acquisition Devices support impedance measurement. Check the manuals of your Signal Acquisition Devices to see if they do.

During impedance measurement only one Signal Acquisition Device is in impedance mode, and the other Signal Acquisition Device does not do anything. This is for reasons of patient safety.

Keep in mind that during impedance mode, the number of channels produced by the Synfi is still the combined number of channels of both frontends. The sample values of the channels of the frontend that is not in impedance mode are set to zero.

The first time an impedance measurement is done using [SetMeasuringMode](#) with MEASURE_MODE_IMPEDANCE , the Signal Acquisition Device connected to port Fiber_1 is set in impedance mode, as can be seen by the switching on of the amber-colored LED's on the front. The impedance measurement is stopped by using [SetMeasuringMode](#) with MEASURE_MODE_NORMAL, then the impedance mode of the Signal Acquisition Device connected to port Fiber_1 is turned off.

The second time a impedance measurement is done (using [SetMeasuringMode](#) with MEASURE_MODE_IMPEDANCE), the impedance mode of the the Signal Acquisition Device connected to port Fiber_2 is set in impedance mode, as can be seen by the switching on of the amber-colored LED's on the front plate. The impedance measurement is ended by using [SetMeasuringMode](#) with MEASURE_MODE_NORMAL, then the impedance mode of the Signal Acquisition Device connected to port Fiber_2 is turned off.

The third time a impedance measurement is done, the Signal Acquisition Device connected to port Fiber_1 is set in impedance mode again, and so on.

Set/Get the RTC clock

It is not possible to [read out the RTC clock or set the RTC clock](#) of a single or both Signal Acquisition Devices using the Synfi.

xt: [Legal Information](#)

TMS International BV

Start measuring

The data acquisition device supplies its data in a constant stream of data. For each sample, the driver generates a conversion result for each channel. Because the user mode application runs in a multitasking environment, processing time for each sample can not be guaranteed. This is why samples need to be buffered. Before a measurement is started, the size of this buffer needs to be set. We also need to set the required sample rate of the measurement. Both of these settings are done with the function **SetSignalBuffer**. It is not possible to set or select individual channels of the frontend.

After the **SetSignalBuffer** function completes successfully, **Start** can be called. This will start the generation of samples. All samples for all channels of a device are stored in the allocated signal buffer. It is now is the task of the application to read these samples from the buffer as they become available. If the application cannot keep up with generated samples the buffer will overrun and data is lost. The application can call **GetBufferInfo** to get information about the status of the signal buffer.

Reading the samples from the signal buffer is done by calling **GetSamples**. The samples read from the buffer are automatically removed from it. The format of the sample data depends on the frontend that is used. Application needs the information returned by **GetSignalFormat** to interpret this sample data.

so see

[Start](#), [Stop](#), [SetSignalBuffer](#), [GetBufferInfo](#), [GetSamples](#), [GetSignalFormat](#)

TMS International BV

Legal Information

Information in this document is subject to change without notice. The example companies, organizations, products, people and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of TMS International BV.

TMS International BV may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from TMSi, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2001-2012 TMS International BV. All rights reserved

TMS International BV

Instructions

The following functions are exported by **TMSiSDK.DLL**. To get channel information and sample data from the frontend, it is necessary to call several functions in a specific order:

1. LibraryInit() to initialize the SDK library for the chosen communication interface
2. GetDeviceList() to get a list of connected frontends
3. Open() to open the communication path with the chosen frontend
4. FreeDeviceList() to free the device list given by GetDeviceList
5. SetRtcTime() (optional): if the frontend has an internal clock and you want to set this clock
6. GetSignalFormat() to get channel information
7. SetSignalBuffer() to get the maximum sample rate and the recommended buffer size for the maximum sample rate
8. SetSignalBuffer() to set the desired sample rate and the desired buffer size for that sample rate
9. Start() to start sampling
10. GetSamples() to get the samples. Call GetSamples repeatedly until Stop is called.
11. Stop() to stop sampling. Optional you can start again at 9) or 8)
12. Close() to stop the communication path with the chosen frontend. Optional you can start again at 3) or 2)
13. LibraryExit() to exit the library.

Impedance is not available on all frontends, so check the manual of the frontend.

TMS International BV

Close

```
BOOL Close(  
    IN HANDLE Handle  
);
```

Parameters

Handle

The handle of the library

Return Value

If successful this function returns TRUE. If the function returns FALSE, use [GetLastError](#) to get the error code.

Remarks

This function closes the connection to the frontend. To reconnect, call [Open](#).

Also see

[Open](#)

TMS International BV

CloseCardFile

```
BOOL CloseCardFile(  
    IN HANDLE Handle  
);
```

Parameters

Handle

The handle of the library

Return Value

If successful this function returns TRUE. If the function return FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function closes the connection to the file on the card. To reconnect, call OpenCardFile.

This function is for Mobita only.

Also see

[OpenCardFile](#)

TMS International BV

ConvertSignalFormat

```
PSIGNAL_FORMAT ConvertSignalFormat (  
    IN HANDLE Handle,  
    IN PSIGNAL_FORMAT pSignalFormat,  
    IN unsigned int ChannelNumber,  
    IN OUT int *Size,  
    IN OUT int *Format,  
    IN OUT int *Type,  
    IN OUT int *SubType,  
    IN OUT float *UnitGain,  
    IN OUT float *UnitOffset,  
    IN OUT int *UnitId,  
    IN OUT int *UnitExponent,  
    IN OUT char Name[SIGNAL_NAME] );
```

Parameters

Handle

Handle of the library.

pSignalFormat

pointer to an existing array of SIGNAL_FORMAT structures, retrieved by a previous call to GetSignalFormat.

ChannelNumber

Number of the channel you want the information from.

Other parameters

See [SIGNAL_FORMAT](#) for a description of the parameters

Return Value

If successful this function returns TRUE. If unsuccessful (i.e wrong Handle, null pSignalFormat pointer) this function returns FALSE.

Comments

This function was created to help LabView understand the SignalFormat structure. See [SIGNAL_FORMAT](#) for a description of the returned structures.

Also see

SIGNAL FORMAT

TMS International BV

'ee

```
BOOL Free(  
    IN VOID *Memory  
);
```

Parameters

Memory

Pointer to the memory block that has to be released.

Return Value

If successful this function returns TRUE. If the release of the memory block failed for any reason it will return FALSE.

Comments

Use this function to release any memory structures allocated by the TMSiSDK DLL only.

After this function is called, the pointer to the memory should NOT be used anymore.

Also See

[GetSignalFormat](#)

TMS International BV

FreeDeviceList

```
void APIENTRY FreeDeviceList(  
HANDLE Handle,  
int NrOfFrontEnds,  
char** DeviceList )
```

Parameters

Handle

Handle of the library

NrOfFrontEnds

Number of found frontends

DeviceList

Array of found frontends

Return Value

None.

Comments

The given char array should not be used after FreeDeviceList is called.

Also see

[Start](#), [SetSignalBuffer](#)

TMS International BV

etBufferInfo

```
BOOLEAN GetBufferInfo(  
    IN HANDLE Handle,  
    OUT PULONG Overflow,  
    OUT PULONG PercentFull  
);
```

Parameters

Handle

Handle of the library

Overflow

This value will increment each time the input buffer overruns. This happens if the rate at which the driver fills the buffer exceeds the rate at which it is read by the application. If a buffer overflow occurs all data in that buffer is lost. This value is reset automatically at Start() and Stop().

PercentFull

Can be anything between 0 and 100%. Used to indicate the amount of samples available to the application. If 100% is reached the *Overflow* will increment and the *PercentFull* parameter will be reset to 0. In that case one buffer of samples is lost. If this percentage increases during a measurement, the application cannot keep up with the data rate of the drivers. To prevent data loss the application can increase its thread priority, or use a larger buffer when calling GetSamples.

Return Value

If successful this function returns TRUE. If the function return FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function only works after Start() is called and before Stop() is called.

Also see

[SetSignalBuffer](#), [GetSamples](#)

TMS International BV

etCardFileList

```
TMSiFileInfoType GetCardFileList (
    IN HANDLE Handle,
    IN OUT int* NrOfFiles )
```

Parameters

Handle

Handle of the library.

NrOfFiles

Pointer to the number of files found on the card

Return Value

If successful the pointer to the first element in the array is returned. If unsuccessful this function returns NULL, in that case use [GetErrorCode](#) to get the error code.

Comments

The internal card on the device can have multiple files. This function is used to get information about each of these files. This function returns a structure for each file on the card. This array of structures is allocated by the SDK and must be freed using Free().

This function is for Mobita only.

Also see

[OpenCardFile](#) , [CloseCardFile](#)

TMS International BV

etCardFileSamples

```
LONG GetCardFileSamples(  
    IN HANDLE Handle,  
    OUT PULONG SampleBuffer,  
    IN ULONG Size  
);
```

Parameters

Handle

Handle of the library

SampleBuffer

Pointer to a buffer where this function will store the samples

Size

Size of this buffer in bytes

Return Value

Returns the total number of bytes read from the internal card. Returns 0 if no new data is available, and a negative value (which is the Error code) if an error occurred.

Comments

This function is used to get one or more samples from the internal card. A pointer to a buffer allocated by the application and the size of this buffer in bytes should be used as input parameters.

The driver will fill this buffer with the available samples. The number of samples returned depends on the size of the given buffer and the amount of samples available. One sample means one conversion result from each channel of the device.

Example: a measurement on the device has 6 input signals, while running at a sample rate of 100 Hz. As stated before, each signal needs

4 bytes for storing the conversion result. The buffer must be at least $6 \times 4 = 24$ bytes for one sample. When getting 100 samples, you need a buffer of $6 \times 4 \times 100 = 2400$ bytes.

When a channel is not connected or out-of-range, the returned value is 0x80000000.

The only allowed function after this call is the StopCardFile function.

This function is for Mobita only.

so see

[StopCardFile](#), [GetCardFileSignalFormat](#)

TMS International BV

etConnectionProperties

```
BOOL GetConnectionPropertie(  
    HANDLE Handle,  
    int *SignalStrength,  
    unsigned int *NrOfCRCErrors,  
    unsigned int *NrOfSampleBlocks )
```

Parameters

Handle

Handle of the library

SignalStrength

For WLAN connections : the signal strength of the connection to the frontend, between 0 and 100.

For USB connections : the signal strength is not available and the variable is set to -1.

For Bluetooth connections : the signal strength is not available and the variable is set to -1.

NrOfCrcErrors

The number of CRC errors in the received data blocks from the Frontend since the last call of Start().

NrOfSampleBlocks

The number received data blocks from the Frontend since the last call of Start().

Return Value

If successful this function returns TRUE. If the function return FALSE, use [GetErrorCode](#) to get the error code.

Comments

The Open function must have been called before.

It is recommended to call this function only when the frontend is not sampling, i.e. do NOT call this function when Start() is called, and Stop() is not called yet.

The Start() function resets the *NrOfSampleBlocks* and *NrOfCrcErrors* counters.

Data blocks are send with a Cyclic Redundancy Check (CRC) to the PC. In this way, it is verified if the data has been corrupted during the data transport from the frontend to the PC. When a CRC error is found, the data block is discarded, and the *NrOfCRCErrors* counter is incremented.

so see

[Open](#)

TMS International BV

etDeviceList

```
const char** APIENTRY GetDeviceList(  
HANDLE Handle,  
int *NrOfFrontEnds)
```

Parameters

Handle

Handle of the library

NrOfFrontEnds

Number of found frontends

Return Value

If successful this function returns an array of *NrOfFrontEnds* strings that contain connection identifiers. If the function return NULL, no frontends are found, and use [GetErrorCode](#) to get the error code.

Comments

The array must be freed using `FreeDeviceList`.

For Bluetooth devices to show up in the list, they must be paired before this function is called.

For WLAN devices to show up in the list, they must be a point-to-point connection configured before this function is called.

For USB devices to show up in the list, they must be connected before this function is called. For any connected USB device, the sampling is stopped.

Also see

[Start](#), [SetSignalBuffer](#)

TMS International BV

GetErrorCode

```
int WINAPI GetErrorCode(  
IN HANDLE Handle  
);
```

Parameters

Handle

Handle of the library

Return Value

This function returns the error code of the previously called function.

Comments

This function should only be called if the previously called SDK function returns FALSE.

Use the [GetErrorCodeMessage](#) function to get a text string (in English) describing the error code.

Error codes can be negative or positive values. A negative code means something went wrong in the SDK. A zero error code means there is no error.

Error Codes

The following is a list of TMSi internal error codes which are returned by GetErrorCode.

Positive error codes > 255 are from the OS.

Code	Message	Possible cause
-1	Undefined error	Unclear
-2	A NULL parameter is given to the API where not allowed	Programming error of the calling application
-3	IO port on which is read/write is not initialized	There is no longer a connection to the frontend
-4	A received data-block contains a CRC error	Due to transmission errors the data block was not received correctly.
-5	Not enough data for block-type	Due to transmission errors the data block was not received correctly.
-6	No data received from frontend	The frontend does not respond
-7	Buffer given in function call is too small	Programming error of the calling application
-8	Connection has been broken for unknown reasons	For wireless connections: The device has moved out-of-range.
-9	Unexpected block-type received	Device is busy processing older requests
-10	Invalid handle given to function	Function is called with an invalid handle/pointer
-11	Allocation of memory failed	Programming error of the calling application: Too much memory is used
	Initialization of	Missing libraries on the OS

-12	(underlying) library failed	
-13	No USB device found	Dongle has been removed by user, or frontend has been switched off
-14	No WLAN device found	Dongle has been removed by user, or frontend has been switched off
-15	Parameter in data-block Out-Of-Range	Development firmware in the device
-16	Function Parameter1 Out-Of-Range	The value of the first parameter after the handle is invalid
-17	Function Parameter2 Out-Of-Range	The value of the second parameter after the handle is invalid
-18	Function Parameter3 Out-Of-Range	The value of the third parameter after the handle is invalid
-19	Bluetooth Name Not Found	Name of the Bluetooth frontend is not found
-20	Not Implemented	This function is not implemented
-21	No WLAN Dongle	No WLAN Dongle on PC found
-22	No Bluetooth Dongle	No Bluetooth Dongle on PC found
-23	Wrong Order Of Calls	The order in which the API functions are called is wrong
0	No error, positive acknowledge	All is OK
1	Unknown or not implemented block-type	Function is not supported by the frontend. Examples could be Get/Set Clock, Set Impedance.
2	CRC error in received block	

3	Error in command data (can't do that for this communication method)	Application calls a function which is not supported on the selected communication method, like calling a function using a Bluetooth connection, while the function is only supported on a USB connection.
4	Invalid block size (too large)	
17	No external power supplied	You try to do something for which the frontend needs external power, and the power supply is not connected or not working
18	Not possible because the frontend is recording	Application calls a function to change some configuration settings, while the frontend is busy recording sample data on (internal) storage.
19	Storage medium is busy	Application calls a function to do something with the (internal) storage, while that storage is busy doing something
20	Flash memory not present	Application calls a function to change data on flash memory, while the device has no flash memory at all.
21	Number of words to read from flash memory out of range	Application calls a function to read data on flash memory, but ask more data than available
22	Flash memory is write protected	Application calls a function to write data on flash memory, but the flash memory is write protected.
23	Incorrect value for initial inflation pressure	
24	Wrong size or values in BP cycle list	
	Selected sample	You try to use a sample rate which is too low

25	frequency out of range for this communication method	for the connected frontend
26	Wrong nr of user channels (≤ 0 , $> \text{maxUSRchan}$)	Invalid number of channels (< 0 or more than the frontend has available)
27	Address flash memory out of range	Application calls a function to read/write data on flash memory, but asks data on a address larger than the amount of flash memory available
28	Erasing not possible because battery low	Application calls a function to erase data on (internal) storage, but there is no sufficient amount of power available to perform the erase. Load the battery or use an external power supply.
29	Frontend locked	The frontend is locked and can only be unlocked by the OEM application
30	No valid data available	There is no valid data available as answer on the send command
31	Requested sample frequency is different from configured ambulant recording frequency	The requested sample frequency for the selected communication method is different from configured ambulant recording frequency on the frontend

so see

[GetErrorMessage](#)

TMS International BV

GetErrorMessage

```
const char* WINAPI GetErrorMessage(  
IN void* Handle,  
IN int ErrorCode )  
);
```

Parameters

Handle

Handle of the library

ErrorCode

Error code from which the message is wanted

Return Value

This function returns the error message (in English) for the given error code.

Comments

This function should only be called if you want the error message string for the error code given by the `GetErrorCode` function. The strings given by `GetErrorMessage` should not be displayed in any application. The returned char pointer should not be freed.

Also see

[GetErrorCode](#)

TMS International BV

etExtFrontEndInfo

```
BOOL GetExtFrontEndInfo(  
    HANDLE Handle,  
    IN OUT TMSiExtFrontendInfoType *ExtFrontEndInfo,  
    TMSiBatReportType *BatteryReport,  
    TMSiStorageReportType *StorageReport,  
    TMSiDeviceReportType *DeviceReport  
)
```

Parameters

Handle

Handle of the library

ExtFrontEndInfo

Extended information of the frontend

BatteryReport

Information on the battery capacity and current charge

StorageReport

Information of the memory card capacity and current usage

DeviceReport

Information of the amount of usage of the FrontEnd

Return Value

If successful this function returns TRUE. If the function return FALSE, use [GetErrorCode](#) to get the error code.

Comments

The Open function must have been called before. For the application, the fields NrOfChannels, Serial, NrExg, NrAux from the FRONTENDINFO are most useful.

This function is for Mobita only.

Also see

[Open](#)

TMS International BV

etFrontEndInfo

```
BOOL GetFrontEndInfo(  
    HANDLE Handle,  
    FRONTENDINFO *FrontEndInfo)
```

Parameters

Handle

Handle of the library

FrontEndInfo

Information of the FrontEnd

Return Value

If successful this function returns TRUE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

The Open function must have been called before. For the application, the fields NrOfChannels, Serial, NrExg, NrAux from the FRONTENDINFO are most useful.

Also see

[Open](#)

TMS International BV

etRecordingConfiguration

```
BOOLEAN GetRecordingConfiguration(  
    IN HANDLE Handle,  
    IN OUT TMSiRecordingConfigType *RecordingConfig,  
    IN OUT unsigned int *ChannelConfig,  
    IN OUT unsigned int *NrOfChannels )  
);
```

Parameters

Handle

Handle of the library

RecordingConfig

Pointer to a recording configuration for the device

ChannelConfig

Reserved, use NULL pointer.

NrOfChannels

Reserved, use 0.

Return Value

If successful this function returns TRUE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is used to get the current measurement recording schedule and settings of the frontend.

The Open function must have been successfully called before using this function.

This function is for Mobita only.

so see

[Open, SetRecordingConfiguration](#)

TMS International BV

etSamples

```
LONG GetSamples(  
    IN HANDLE Handle,  
    OUT PULONG SampleBuffer,  
    IN ULONG Size  
);
```

Parameters

Handle

Handle of the library

SampleBuffer

Pointer to a user-allocated buffer where this function will store the samples

Size

Size of this buffer in bytes

Return Value

Returns the total number of bytes put into the user-allocated buffer. Return 0 if no new data is available. Returns a negative value (which is the error code) if an error occurred.

Comments

This function is used to write one or more samples to the user-allocated buffer. A pointer to a buffer allocated by the application and the size of this buffer in bytes are used as input parameters. The size of the input buffer in samples should be larger or the same as the size of the internal buffer as previously set using the SetSignalBuffer function.

The driver will fill the user-allocated buffer with the available samples. The number of samples returned depends on the size of this user-allocated buffer and the amount of samples available. One sample means one conversion result from each channel of the device. All

samples read by the application are no longer available from the driver. Use the function `GetSignalFormat` to interpret the sample data returned by the `Getsample` function. The values are always stored in the buffer as 32-bit numbers, i.e. they need 4 bytes each, regardless of the channel size as given by the `GetSignalformat`.

Example: a device has 6 input signals, while running at a sample rate of 100 Hz. As stated before, each signal needs 4 bytes for storing the conversion result. The buffer must be at least $6*4 = 24$ bytes for one sample. When acquiring 100 samples, you need a buffer of $6*4*100 = 2400$ bytes.

When a channel of type EXG, BIP, AUX is not connected or out-of-range, the value returned is `0x80000000`.

The only allowed function after this call is the `Stop()` function.

Warning

When using Bluetooth or WLAN, do NOT use the Windows OS functions to scan for other Bluetooth or WLAN frontends. The scan will access the Bluetooth or WLAN dongle, and disturb the received signal from the sampling frontend, with loss of samples as a result.

The `GetSamples` function should be called at least 16 times per second. If you prefer to call it less often, you have to make sure that your defined buffer is large enough to store all incoming samples.

When `GetSamples()` returns a negative value, the application should try to stop the frontend by calling `Stop()` and after that calling `Close()`. Then the application should try to reconnect using `Open()`.

so see

[GetBufferInfo](#), [GetSignalFormat](#)

TMS International BV

etSignalFormat

```
PSIGNAL_FORMAT GetSignalFormat (  
    IN HANDLE Handle,  
    IN OUT char* FrontEndName )
```

Parameters

Handle

Handle of the library

FrontEndName

Pointer to char array of length MAX_FRONTENDNAME_LENGTH, as declared in TMSiSDK.h, or a NULL pointer. When the FrontEndName is given, it is filled by the function with the FrontEndName of the connected FrontEnd.

Return Value

If successful returns the pointer to the first element in the array. If unsuccessful this function returns NULL, in that case use [GetErrorCode](#) to get the error code.

Comments

One device can have multiple channels. This function is used to get information about each of these channels. This function returns a structure for each channel in the system. See [SIGNAL_FORMAT](#) for a description of the returned structures.

Also see

[SIGNAL_FORMAT](#)

TMS International BV

LibraryExit

```
int WINAPI LibraryExit(  
    HANDLE Handle )
```

Parameters

Handle

Handle of the library

Return Value

The return value is an error code. If successful this function returns zero, else a non-zero value. If the function returns a non-zero value, you can NOT use [GetErrorMessage](#) to get the string explaining the error code, because the library is already shut down.

Comments

The handle (and the library) can no longer be used after using this function. The function [Close](#) must be called before calling this function. To use the library again, call [LibraryInit](#) to get a new handle.

Also see

[Close](#), [LibraryInit](#)

TMS International BV

LibraryInit

```
HANDLE WINAPI LibraryInit(  
TMSiConnectionType GivenConnectionType,  
int *ErrorCode )
```

Parameters

GivenConnectionType

Chosen connection type (WLAN, USB, Bluetooth)

ErrorCode

When an error occurs, this code contains the cause of the error.

Return Value

If successful this function returns the handle to the library. If unsuccessful returns **INVALID_HANDLE_VALUE**, and the ErrorCode parameter contains the error code.

Comments

This is the first function to call after the TMSiSDK.dll is loaded and all function pointers are loaded. If the returned handle is no longer needed it can be released using the **LibraryExit** function.

Also see

[Close](#), [LibraryExit](#)

TMS International BV

pen

```
BOOLEAN Open(  
    IN HANDLE Handle  
    IN const char* DeviceLocator  
);
```

Parameters

Handle

Handle of the library

DeviceLocator

Name of the device you want to open, as given by GetDeviceList

Return Value

If successful this function returns TRUE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

Open tries to interrogate the device given by the DeviceLocator. The DeviceLocator string must be a pointer to a string previously given by the GetDeviceList function.

Also see

[Close](#)

TMS International BV

penCardFile

```
BOOLEAN OpenCardFile (  
    IN HANDLE Handle,  
    IN short FileId,  
    OUT TMSiFileHeaderType *FileHeader )
```

Parameters

Handle

Handle of the library.

FileId

The file identifier as found in the TMSiFileInfoType structure given by GetCardFileList

FileHeader

The header of the selected file.

Return Value

If successful this function returns TRUE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

The header of the file contains information about the measurement, such as start and stop time, serial number of the device, sample rate, and patient identifier.

This function is for Mobita only.

Also see

[GetCardFileList](#), [TMSiFileHeaderType](#)

TMS International BV

SetMeasuringMode

```
BOOLEAN APIENTRY SetMeasuringMode(  
IN HANDLE Handle,  
IN ULONG *Mode,  
IN int Value )
```

Parameters

Handle

Handle of the library

Mode

The mode in which the frontend should be set

Value

The value which is used by the chosen mode

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

If the returned error code is 1, the frontend can not be set in the requested mode.

Comments

Most Refa systems support impedance measurement. During impedance measurement the device measures the electrode impedance of each input channel. This feature is added to switch between these modes of operation.

Mode	Description
MEASURE_MODE_NORMAL	Normal mode
MEASURE_MODE_IMPEDANCE_EX	Impedance mode

Impedance measurement

To put the device in impedance mode, set the **Mode** parameter to MEASURE_MODE_IMPEDANCE_EX. In this case the **Value** parameter is used to select at which level to LED on the device should indicate that the electrode impedance is too high. The **Value** parameter can be any of the following values:

Value	Description
IC_OHM_002	Impedance limit at 2 kOhm
IC_OHM_005	Impedance limit at 5 kOhm
IC_OHM_010	Impedance limit at 10 kOhm
IC_OHM_020	Impedance limit at 20 kOhm
IC_OHM_050	Impedance limit at 50 kOhm
IC_OHM_100	Impedance limit at 100 kOhm
IC_OHM_200	Impedance limit at 200 kOhm

Before calling this function, stop the frontend by calling [Stop\(\)](#).

Check the user manual of your frontend to see if your frontend supports impedance modes.

To set the frontend back to normal mode (=sampling), use the MEASURE_MODE_NORMAL value for the Mode parameter and 0 for the Value parameter.

plies to

All frontends that support impedance measurements. Refer to the manual of your frontend.

so see

[Open](#)

TMS International BV

SetOEMData

```
BOOLEAN WINAPI SetOEMData(IN HANDLE Handle,  
                          unsigned char *OEMData,  
                          unsigned int OEMDataLengthInBytes )  
);
```

Parameters

Handle

Handle of the library

OEMData

Array in which the data from the OEM storage is stored

LengthInBytes

Length of the array in bytes

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is not supported by all frontends. The [Open\(\)](#) function and [GetOEMSize\(\)](#) function must have been called before. This function will send the data to the OEM storage on the frontend. The length of the buffer must be equal or smaller than the value returned by [GetOEMSize\(\)](#).

Also see

[GetOEMData](#)

TMS International BV

SetRtcTime

```
BOOLEAN APIENTRY SetRtcTime(  
IN HANDLE Handle,  
IN SYSTEMTIME *InTime )
```

Parameters

Handle

Handle of the library

InTime

Win32 defined data structure for holding time and date information

Return Value

If successful this function returns TRUE, else FALSE. If the functions return FALSE, use [GetErrorCode](#) to get the error code.

Comments

RTC stands for Real Time Clock. Most portable devices made by TMSi have an internal RTC. In some cases, the value of the RTC can be displayed on the device's display. It is used to keep track of the start time of locally stored measurements. In some cases it can be used to automatically start a measurement at a programmable date and time (ambulatory recording). This function is used to write clock information to the frontend.

The example below demonstrates how this is done:

```
#include <windows.h>
```

```
SYSTEMTIME Time = {0};
```

```
GetSystemTime( &Time ); // For time in UTC
```

```
GetLocalTime( &Time ); // For time in local timezone
```

```
Status = SetRtcTime( Handle, &Time );
```

Keep in mind that for frontends that do not have an internal clock, the

function returns FALSE, and the error code returned by GetErrorCode will be non-zero.

plies to

All frontends that have an RTC. Refer to the manual of your frontend.

so see

[Open](#)

TMS International BV

SetRefCalculation

```
BOOLEAN SetRefCalculation(  
    IN HANDLE Handle,  
    IN int OnOrOff  
);
```

Parameters

Handle

Handle of the library

OnOrOff

If the average reference calculation is turned on (value != 0), or off (value == 0)

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is used to turn the average reference calculation in the driver ON or OFF.

When using Synfi, the average reference calculation in the driver should always be OFF. Refer to the User Manual of the Synfi for more information on this.

Warning

By default, it is OFF for all devices. This means that all EXG channels carry raw signals, i.e. without the common average removed from the signals, and this will result in showing 50/60 Hz interference in the signals.

Also see

Start

TMS International BV

SetSignalBuffer

```
BOOLEAN SetSignalBuffer(  
    IN HANDLE Handle,  
    IN OUT PULONG SampleRate,  
    IN OUT PULONG BufferSize  
);
```

Parameters

Handle

Handle of the library

SampleRate

The SampleRate argument is a pointer to an unsigned long. Before calling this function this location must be filled with required sample rate. If the hardware can not match the required sample rate, it will be set to the first available sample rate below the requested sample rate. The location used for the input sample rate is then overwritten by this function with the sample rate that is actually set. The sample rate used in this function is defined in milliHertz. So to set a sample rate of 1Hz use 1000 as input value.

BufferSize

This parameter is used to set the buffer size which the application is going to use in the GetSamples function. This size is not defined in bytes but as the amount of samples that can be stored. A sample in this case is one conversion result for all input channels. If this value is set to 100 and the sample rate of the device is 100Hz, 1 second of conversion results can be stored.

Return Value

If successful this function returns TRUE, else FALSE. If the function return FALSE, use [GetErrorCode](#) to get the error code.

Comments

To find out what the maximal sample rate and maximal buffer size is, call

SetSignalBuffer with value 0xFFFFFFFF for both parameters. After return, the parameters will contain the maximal sample rate of the connected device.

Before starting a measurement (by calling Start) call this function to set the sample rate and buffer size.

The minimum required buffer size is at least 16 samples, and can be higher based on the given sample rate. The [GetSamples\(\)](#) function should be called at least 16 times per second.

so see

[Start](#)

TMS International BV

GetRecordingConfiguration

```
BOOLEAN GetRecordingConfiguration(  
    IN HANDLE Handle,  
    IN TMSiRecordingConfigType *RecordingConfig,  
    IN unsigned int *ChannelConfig,  
    IN unsigned int NrOfChannels )  
);
```

Parameters

Handle

Handle of the library

RecordingConfig

Pointer to a recording configuration for the device

ChannelConfig

Reserved, use NULL pointer

NrOfChannels

Reserved, use 0

Return Value

If successful this function returns TRUE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is for the Mobita only.

This function is used to set the measurement recording schedule and settings of the frontend. A previous recording schedule (if present) will be overwritten.

The [Open\(\)](#) function must have been successfully called before using this function.

so see

[Open, GetRecordingConfiguration](#)

TMS International BV

Start

```
BOOLEAN Start(  
    IN HANDLE Handle  
);
```

Parameters

Handle

Handle of the library

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function will start data acquisition. Call this function after the sample rate of the frontend has been configured. Always call [SetSignalBuffer\(\)](#) before (re)starting a new measurement. After starting the device use [GetSamples](#) to get the actual conversion results from the device, or stop the frontend by calling [Stop\(\)](#).

Also see

[Stop](#), [GetSamples](#), [SetSignalBuffer](#)

TMS International BV

StartCardFile

```
BOOLEAN StartCardFile(  
    IN HANDLE Handle  
);
```

Parameters

Handle

Handle of the library

Return Value

If successful this function returns TRUE, else FALSE. If the functions return FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function will start reading data from the file on the card. After starting the device use [GetCardFileSamples\(\)](#) to get the sample data from the card.

This function is for Mobita only.

Also see

[StopCardFile](#), [GetCardFileSamples](#)

TMS International BV

Stop

```
BOOLEAN Stop(  
    IN HANDLE Handle  
);
```

Parameters

Handle

Handle of the library

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

Stops the data acquisition. Call [SetSignalBuffer\(\)](#) before starting data acquisition again.

Also see

[Start](#), [SetSignalBuffer](#)

TMS International BV

StopCardFile

```
BOOLEAN StopCardFile(  
    IN HANDLE Handle  
);
```

Parameters

Handle

Handle of the library

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function stops reading data from the file on the card. After stopping the device do not use **GetCardFileSamples** any more.

This function is for Mobita only.

Also see

[StartCardFile](#), [GetCardFileSamples](#)

TMS International BV

SYSTEMTIME

```
typedef struct _SYSTEMTIME {  
    WORD wYear;  
    WORD wMonth;  
    WORD wDayOfWeek;  
    WORD wDay;  
    WORD wHour;  
    WORD wMinute;  
    WORD wSecond;  
    WORD wMilliseconds;  
} SYSTEMTIME;
```

Members

wYear

The current year

wMonth

The current month; January is 1

wDayOfWeek

The current day of the week; Sunday is 0, Monday is 1, and so on

wDay

The current day of the month

wHour

The current hour

wMinute

The current minute

wSecond

The current second

wMilliseconds

The current millisecond

Include

WINDOWS.H

so see

[TMSiRecordingConfigType](#)

TMS International BV

GNAL_FORMAT

```
typedef struct _SignalFormat
{
    ULONG Size;
    ULONG Channels;
    ULONG Type;
    ULONG SubType;
    ULONG Format;
    ULONG Bytes;
    FLOAT UnitGain;
    FLOAT UnitOffset;
    ULONG UnitId;
    LONG UnitExponent;
    WCHAR Name[ 40 ];
    ULONG Port;
    WCHAR PortName[ 40 ];
    ULONG SerialNumber;
} SIGNAL_FORMAT, *PSIGNAL_FORMAT;
```

Members

Size

The size in bytes of this structure.

Channels

Total number of channels in this configuration.

Type

These values identify the channels type. Below a list of all types defined.

Type	Name	Description
0	Unknown	Set if the driver cannot determine the channel type
1	EXG	Electro physiological input on a common average reference amplifier.
2	BIP	Electro physiological input measured on a bipolar amplifier

3	AUX	Signal measured on an auxiliary input
4	DIG	Digital input.
5	TIME	Signal used for synchronization
6	LEAK	Measuring leakage for urology purposes.
7	PRESSURE	Signal for measuring pressure
8	ENVELOPE	EMG Envelope signal derived from an electro physiological input
9	MARKER	Marker input
10	RAMP	Internally generated test signal
11	SAO2	Signals measure with an oxygen saturation measuring device

SubType

By means of this parameter you can get more specific information about a channel. These values are device specific and are subject to change. Refer to your device manual for more information.

Format

Specifies the data format of this channel. Channel data is in integer format. The following table shows all supported formats:

Format	Description
0	Unsigned integer
1	Signed integer

Bytes

This parameter can be ignored. All data in the buffer (see [GetSamples](#)) is 32 bits.

UnitGain & UnitOffset

These are two 32Bit floating point values. Data measured on a channels represents some form of units. For example electro physiological measure there data in micro volts. To use these units the conversion results from a channel need to translated. Translating to units is done by multiplying the result with *UnitGain* and then adding *UnitOffset*. The driver does NOT perform this calculation, but this must be done by the application.

UnitId

The following list shows all defined units

UnitId	Name	Description
0	UNIT_UNKNOWN	If the driver cannot determine the units of a channel
1	UNIT_VOLT	Channel measures voltage
2	UNIT_PERCENT	Channel measures a percentage
3	UNIT_BPM	Beats per minute
4	UNIT_BAR	Pressure in bar
5	UNIT_PSI	Pressure in psi
6	UNIT_MH2O	Pressure calibrated to meters water
7	UNIT_MHG	Pressure calibrated to meters mercury
8	UNIT_BIT	Used for digital inputs

UnitExponent

Used for defining the exponent of a measured unit. e.g if a channel measures micro volts the *UnitExponent* will be -6. For kilo volt *UnitExponent* will be 3, etc. If a driver cannot determine the unit type this value will be zero.

Port

Some devices support multiple input boxes or ports. If so this value

identifies the port number of the device on which this channel is connected.

PortName

Name of the device, or if a device has multiple input ports the name of the input port, on which this channel is connected

SerialNumber

The serial number or id of the source device of this channel.

include

TMSiSDK.h

also see

[GetSignalFormat](#)

TMS International BV

MSiFileHeaderType

```
typedef struct TMSiTDFHeader
{
    unsigned int          NumberOfSamp;
    SYSTEMTIME           StartRecTime;
    SYSTEMTIME           EndRecTime;
    unsigned int         FrontEndSN;
    unsigned int         FrontEndAdpSN;
    unsigned short       FrontEndHWVer;
    unsigned short       FrontEndSWVer;
    unsigned short       FrontEndAdpHWVer;
    unsigned short       FrontEndAdpSWVer;
    unsigned short       ADCSampleRate;
    char                 PatientID[MAX_PATIENTID_LENGTH];
    char                 UserString1[MAX_USERSTRING_LENGTH];
} TMSiFileHeaderType;
```

Members

NumberOfSamples

The number of samples stored in this file

StartRecTime

The Start time of the recording of this file

StopRecTime

The Stop time of the recording of this file

FrontendSN

The serial number of the device for which the recording configuration is to be used. After recording this field contains the complete serial number of the device which was used to do the recording

FrontendAdpSN

The serial number of the adapter for which the recording configuration is to be used. After recording this field contains the complete serial number

of the adapter which was used to do the recording

FrontendHWVers

The hardware version number of the device on which the file is recorded

FrontendSWVers

The firmware version number of the device on which the file is recorded

FrontendAdpHWVers

The hardware version number of the adapter that was connected to the device on which the file is recorded

FrontendAdpSWVers

The firmware version number of the adapter that was connected to the device on which the file is recorded

ADCSampleRate

This integer value contains the sample rate used during recording

PatientId

This is an area that can be used to store patient information. The string is 64 bytes long

UserString1

This is an area that can be used by the initialization/export application to store information about the application name or version etc. The string is 64 bytes long

:lude

TMSiSDK.H

so see

[SetRecordingConfiguration](#), [GetRecordingConfiguration](#)

TMS International BV

MSiFileInfoType

```
typedef struct TMSiFileInfo
{
    unsigned int          FileId;
    SYSTEMTIME           StartRecTime;
    SYSTEMTIME           EndRecTime;
} TMSiFileInfoType;
```

Members

FileId

File identifier that is unique for the files in the internal storage of the Mobita.

StartRecTime

The Start time of the recording of this file.

StopRecTime

The Stop time of the recording of this file.

Include

TMSiSDK.H

Also see

[SetRecordingConfiguration](#), [GetRecordingConfiguration](#)

TMS International BV

MSiRecordingConfig

```
typedef struct TMSiRecordingConfig
{
    unsigned short      StorageType;
    unsigned short      ADCSampRate;
    unsigned short      NumberOfChan;
    unsigned int         StartControl;
    unsigned int         EndControl;
    unsigned int         CardStatus;
    unsigned int         InitIdentifier;
    char                 MeasureFileName[MAX_MEASUREFILENAME_
SYSTEMTIME            AlarmTimeStart;
SYSTEMTIME            AlarmTimeStop;
SYSTEMTIME            AlarmTimeInterval;
    unsigned int         AlarmTimeCount;
    unsigned int         FrontEndSN;
    unsigned int         FrontEndAdpSN; /*!< Serial number of
    unsigned int         RecordCondition;
SYSTEMTIME            RFInterfStartTime;
SYSTEMTIME            RFInterfStopTime;
SYSTEMTIME            RFInterfInterval;
    unsigned int         RFInterfCount;
    char                 PatientID[MAX_PATIENTID LENG
    char                 UserString1[MAX_USERSTRING_L
} TMSiRecordingConfigType;
```

Members

FileType

Reserved

StorageType

These values identify the storage type. Below is a list of all types defined.

Bit	Name	Description
0	Raw Mode	If bit0 is set, then the data as it would be sent over the wireless/USB/Fiber connection will be stored in the measurement file.

1	Reserved	
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	
8	Reserved	
9	Reserved	
10	Reserved	
11	Reserved	

ADCSampleRate

This integer value contains the sample rate of ADCs in Hertz. The ADC sample rate is defined by the maximum sample rate of the frontend.

NrOfChannels

This integer value contains the sum of storable channels, ExG, Aux, BIP etc. (analog) and Digi, SaO2, 3D etc. (digital) that are available on the frontend.

StartControl

This integer consists of a number of bits that control the start-up behavior of the system.

WARNING: If both the RTC_SET and ALARM_RECORD_AUTO_START bit are set, both bits will be ignored, an error indication appears, and both bits will be cleared, so that at a next attempt a recording will start in the normal way.

If ALARM_RECORD_AUTO_START, BUTTON_ENABLE and POWER-ON RECORD AUTO START are all 0, then there is no way to start a recording!

Bit	Name	Description
0	RTC_SET	If this bit (least significant bit) is set then the 'Alarm time' settings are used to set the Real Time Clock (RTC). This happens at the moment that the user switches the TMSi frontend ON for the first time, after this bit has been set. At the same time the bit will be cleared, so that the same RTC time will be not be set again.

1 ALARM_RECORD_AUTO_START

If this bit is set the alarm settings of the RTC are used to automatically start the recording. The auto start time is programmed at the moment that the user switches the TMSi frontend on for the first time, after the bit has been set. At the same time the bit will be cleared, so that the same auto start time will be not be set again. After the auto start time, this time is also

		cleared from TMSi frontend internal memory.
2	MAN_RECORD_ENABLE	If this bit is set then the recording can be started and stopped manually. Otherwise, when bit 2 is 0, the user ON/OFF is ignored during a recording.
3	POWER-ON_RECORD_AUTO_START	If this bit is set, recording is started automatically at power on, without the need for user intervention. If this bit is not set, data will not be stored on the SD card.
		If this bit is

4	ALARM_RECURREN	set, the system will reinitialize the RTC ALARM after the previous ALARM, based on the Alarm Time values specified.
5	RF_AUTO_START	If this bit is set, the system will enable the RF module directly after power-up.
6	RF_TIMED_START	If this bit is set, the system will enable the RF module according to the wireless interface parameters.
		If this bit is set, the system will reinitialize

7	RF_RECURRING	the RF ALARM after the previous ALARM, based on the Wireless Time values specified.
8	MAN_SHUTDOWN_ENABLE	If this bit is set, the user can manually shutdown the system. Otherwise, the system can only shut down automatically.
9	Reserved	
10	Reserved	
11	Reserved	

EndControl

If this value is 0 then the recording is continued until the end of the flash memory is reached, until the recording is stopped manually, by RTCor when the system battery is empty.

Otherwise this value determines the recording length. In that case this value represents the number of sample periods of a recording (counted at the sample rate of the ADC Rate).

CardStatus

The following list shows all defined units

UnitId	Name	Description
0	Unknown	Unknown state for internal SD card
1	Formatted	Internal SD card is formatted
2	Filled	Internal SD card has files, but still has space available
3	Full	Internal SD card is full
0x7FFFFFFF	Error	Error on internal SD card
0xFFFFFFFF	Default	

InitIdentifier

Reserved

MeasureFilename

The TMSi frontend uses these fields to name the measurement files that are made. All characters are printable ASCII characters. The default name is YYYYMMDD_HHMMSS

AlarmtimeSTART / ALARMTIMEstop

When bit 0 or bit 1 is set in the field START CONTROL the defined time in this field will be used to program the RTC time or auto start time of real time clock in the ambulatory system. The real time clock is programmed when the system is turned on.

Alarminterval

The alarm repetition interval is relative to the ALARM TIMESTART and must be larger than the time difference : ALARMTIMESTOP –

ALARMTIMESTART.

If ALARMINTERVAL is not all zeroes then after passing ALARMTIMESTOP the value of ALARMINTERVAL is added to ALARMTIMESTART and ALARMTIMESTOP to set the new recording start and stop times. Only Day, Hours, Minutes and Seconds are used, the other fields are ignored.

AlarmCount

The alarm repetition count is decreased every Alarm repetition interval, until it is zero.

FrontendSN

The serial number of the device for which this recording configuration will be used. When initialized this field contains only the first part of the serial number = device identification, e.g. 0710 for a Mobita. After recording this field contains the complete serial number of the device which was used to do the recording.

FrontendadpSN

The serial number of the adapter for which this recording configuration is to be used. When initialized this field contains only the first part of the serial number = device identification, e.g. 0710 for a Mobita. After recording this field contains the complete serial number of the adapter which was used to do the recording.

RFInterfStartTime / StopTime

Not used.

RFInterfCount

Not used.

PatientId

This is an area that is used by the initialization/export program to store

information about the patient or the type of measurement that is done with this card. The frontend itself does not use this information. If the TMSi application is not used to build a measurement and process the measurement data, this field may freely be used for other administration purposes.

USERSTRING1

This is an area that is used by the initialization/export program to store information about the application etc.

:lude

TMSiSDK.H

so see

[SetRecordingConfiguration](#), [GetRecordingConfiguration](#)

TMS International BV

GetCardFileSignalFormat

```
PSIGNAL_FORMAT GetCardFileSignalFormat (  
    IN HANDLE )
```

Parameters

Handle

Handle of the library.

Return Value

If successful returns the pointer to the first element in the array. If unsuccessful this function returns NULL, in that case use [GetErrorCode](#) to get the error code.

Comments

For this function to work, OpenCardFile must be called first.

One file can have multiple channels. This function is used to get information about each of these channels. This function returns a structure for each channel in the system. This array of structures is allocated by the SDK and must be freed using Free(). See [SIGNAL_FORMAT](#) for a description of the returned structures.

This function is for Mobita only.

Also see

[SIGNAL_FORMAT](#)

TMS International BV

etOEMSize

```
BOOLEAN APIENTRY GetOEMSize(  
    void *Handle,  
    unsigned int *LengthInBytes  
);
```

Parameters

Handle

Handle of the library

LengthInBytes

Length of the available OEM storage on the frontend in bytes.

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is not supported by all frontends. The [Open\(\)](#) function must have been called before. This function will retrieve the length of the available OEM storage on the frontend.

Also see

[GetRandomKey](#)

TMS International BV

GetOEMData

```
BOOLEAN WINAPI GetOEMData(IN HANDLE Handle,  
                          unsigned char *OEMData,  
                          unsigned int *OEMDataLengthInBytes )  
);
```

Parameters

Handle

Handle of the library

OEMData

Array in which the data from the OEM storage is stored.

LengthInBytes

Length of the array in bytes.

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is not supported by all frontends. The [Open\(\)](#) function and [GetOEMSize\(\)](#) function must have been called before. This function will retrieve the data from the OEM storage on the frontend. The length of the buffer must be equal or larger than the value returned by [GetOEMSize\(\)](#). After return, the `OEMDataLengthInBytes` is set to the returned number of bytes.

Also see

[SetOEMData](#)

TMS International BV

etRandomKey

```
BOOLEAN WINAPI GetRandomKey(  
    void *Handle,  
    char *Key,  
    unsigned int *LengthKeyInBytes  
);
```

Parameters

Handle

Handle of the library

Key

Array in which the retrieved key is stored.

LengthKeyInBytes

Length of the given array in bytes.

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is not supported by all front ends. The `Open()` function must have been called before. This function will retrieve a random key from the frontend. The given array should have a length of at least 16 bytes (=128 bits). The retrieved key must be transformed in an OEM-specific way. Then the transformed key is given to [UnlockFrontend](#) to unlock the frontend for further use.

Also see

[UnlockFrontend](#)

TMS International BV

UnlockFrontEnd

```
BOOLEAN WINAPI UnlockFrontEnd(  
    void *Handle,  
    char *Key,  
    unsigned int *LengthKeyInBytes  
);
```

Parameters

Handle

Handle of the library

Key

Array in which the transformed key is stored

LengthKeyInBytes

Length of the given array in bytes

Return Value

If successful this function returns TRUE, else FALSE. If the function returns FALSE, use [GetErrorCode](#) to get the error code.

Comments

This function is not supported by all frontends. The [Open\(\)](#) function and [GetRandomKey\(\)](#) function must have been called before. This function will send the given key to the frontend. If the function returns TRUE, the frontend is unlocked will respond to other commands.

Also see

[GetRandomKey](#)