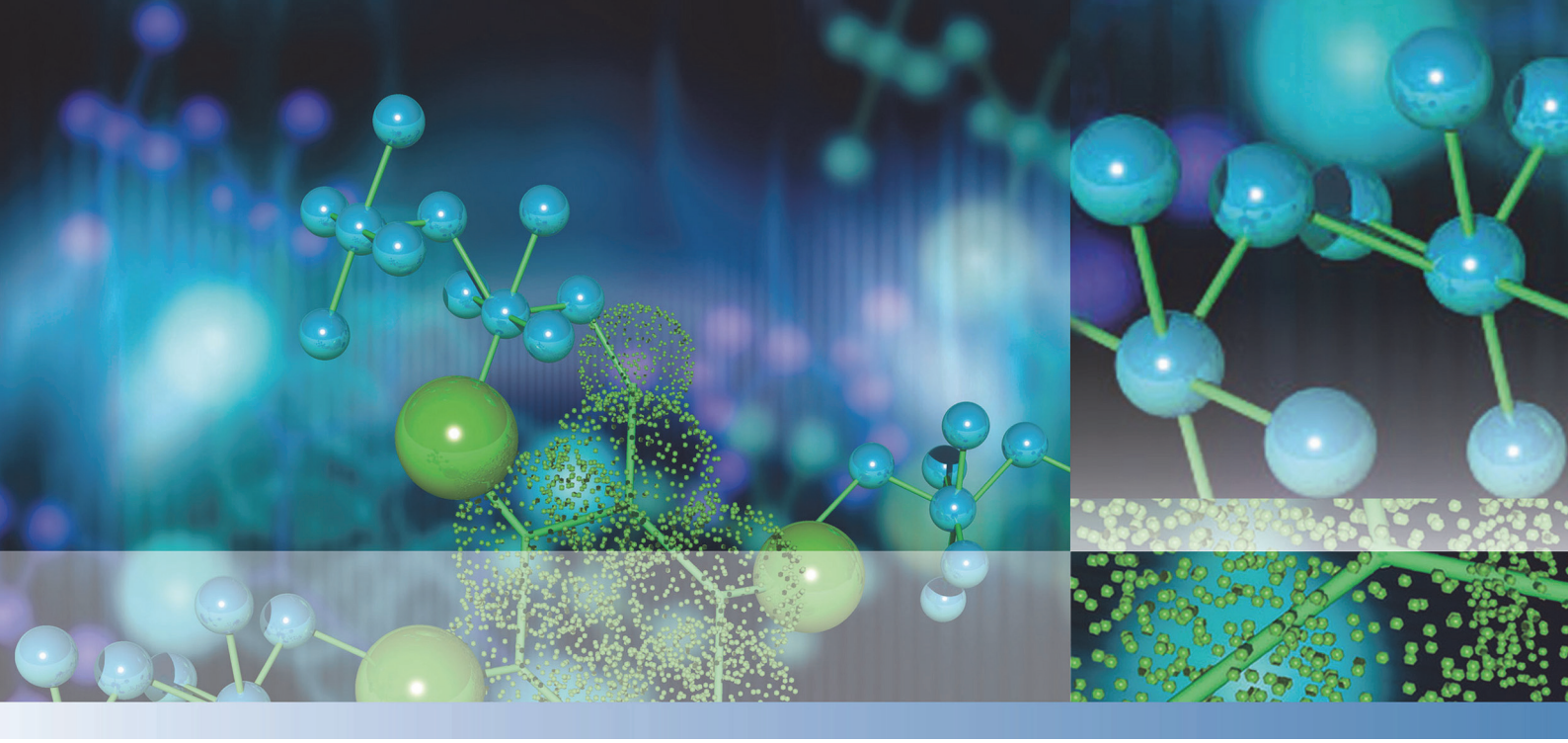

Thermo Scientific

KingFisher Presto

Integration Guide

Cat. No. N17647 Rev 1.0 2016

Thermo SCIENTIFIC

Release history:

For Research Use Only. Not for use in diagnostic procedures.

Introduction

This guide includes all documentation related to integrating the KingFisher Presto instrument to an automation environment.

The following documents are included in this guide:

- *KingFisher Presto Developer's Guide*
- *KingFisher Presto Interface Specification*
- *KFModule.dll Interface Specification*
- *ThermoUSB.dll Interface Specification*
- *ThermoLAN.dll Interface Specification*
- *ThermoCOM.dll Interface Specification*

Related Documentation

In addition to this guide, Thermo Fisher Scientific provides the following documents for KingFisher Presto:

- *Thermo Scientific™ KingFisher™ Presto User Manual* (Cat.no. N17413)
- *Thermo Scientific™ BindIt™ Software User Manual* (Cat. no. N07974)

Safety and Special Notices

Make sure you follow the precautionary statements presented in this guide. The safety and other special notices appear in boxes.

Safety and special notices include the following:

CAUTION Highlights hazards to humans, property, or the environment. Each CAUTION notice is accompanied by an appropriate CAUTION symbol.

IMPORTANT Highlights information necessary to prevent damage to software, loss of data, or invalid test results; or might contain information that is critical for optimal performance of the system.

Note Highlights information of general interest.

Tip Highlights helpful information that can make a task easier.

Contacting Us

For the latest information on products and services, visit our website at:

`www.thermofisher.com/kingfisher`

Thermo Scientific

KingFisher Presto

Developer's Guide

Thermo SCIENTIFIC

Contents

1 KingFisher Presto Developer's Guide 1
- 1.1 Contents . . . 1
- 1.2 Confidential . . . 1
- 1.3 Introduction . . . 2

2 System Description 3

3 Interfacing to KFModule.dll 7
- 3.1 Building the Integration Sample . . . 7
- 3.2 Running Tests with KingFisher Presto Simulator . . . 8

4 Creating and Uploading Protocols 9
- 4.1 BindIt Protocol Editor . . . 10
- 4.2 Creating "My Test Protocol" . . . 10
- 4.3 Uploading "My Test Protocol" . . . 11
- 4.4 Uploading and Downloading Protocols . . . 11

5 Protocol Execution Methods 13
- 5.1 Step-by-Step Execution . . . 13
- 5.2 Event-Based Execution . . . 15
- 5.3 Executing Multiple Overlapping Protocols . . . 17
- 5.4 Precautions for Heater Control . . . 22

6 Error Handling 25
- 6.1 Communication Interfaces . . . 25
- 6.2 Communication Protocol . . . 25
- 6.3 KingFisher Presto Protocol Integrity . . . 26

Chapter 1

KingFisher Presto Developer's Guide

1.1 Contents

- Introduction
- System Description
- Interfacing to KFModule.dll
- Creating and Uploading Protocols
- Protocol Execution Methods
- Error Handling

1.2 Confidential

1.3 Introduction

This document is intended for software designers writing computer programs for controlling the instrument. It contains information necessary to know in order to be able to write such a program. It is assumed that the reader of this document is familiar with the function of the instrument.

Chapter 2

System Description

Figure 3-1 shows the basic architecture of a system using the KingFisher instrument. The system is constructed into three layers: **Application layer**, **Communication layer** and **Hardware layer**. Components using the layers are **Automation System**, **BindIt Software** and **KingFisher Presto Instrument**.

System Components

Automation System refers to a process management system (PMS) software module provided by an automation integrator and running on a computer with a MS Windows operating system.

BindIt Software is used to create KingFisher Presto protocols. It can also upload and download protocols to and from the KingFisher Presto instrument using KFModule.dll, but it is not part of the automation system and it is not used to run the protocols in the automation system.

KingFisher Presto instrument provides communication interfaces for transferring and executing KingFisher Presto protocols created in BindIt software.

Application Services Layer

KFModule.dll dynamic link library provides easy access to the KingFisher Presto instrument interface through communication libraries. The physical connection between a computer and the instrument can be USB, RS232 or LAN.

Communication Services Layer

This is a middle layer between the KingFisher Presto instrument and the KFModule.dll and can be ignored by the Automation System. Depending on a physical connection, one of the communication libraries will be used: **ThermoUSB.dll**, **ThermoCOM.dll** or **ThermoLAN.dll**.

Hardware Interfaces

The KingFisher Presto instrument has three alternate communication ports for connecting to a computer: RS232, USB and LAN. All connections can be accessed either through higher level service layers or directly through physical ports.

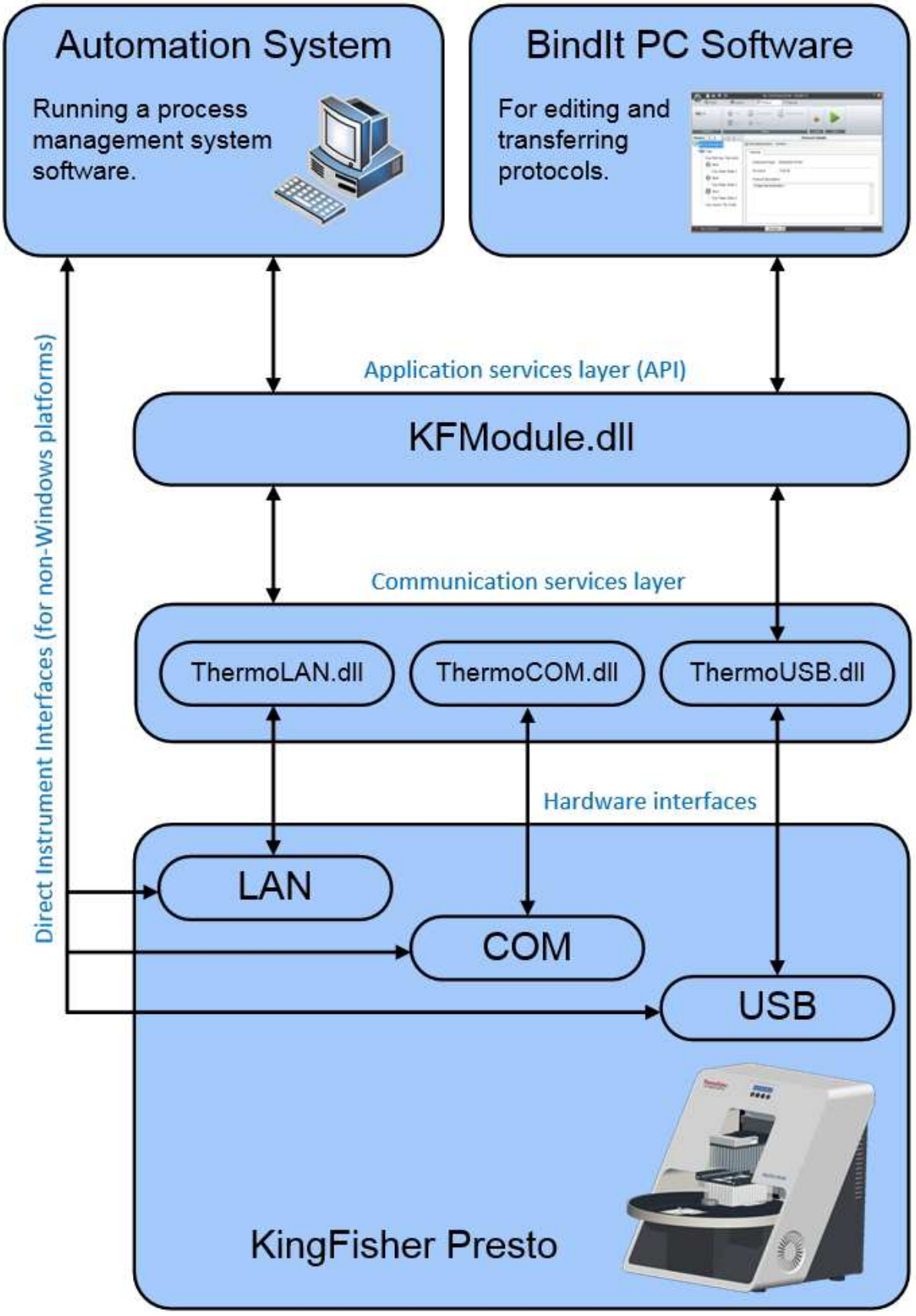

Figure 2.1: System description

Related Documents

BindIt User Manual: Thermo Scientific BindIt Software for KingFisher Instruments User Manual (Cat. No. N07974)

KingFisher Presto Developer's Guide

KingFisher Presto Interface Specification

KFModule.dll Interface Specification

ThermoUSB.dll Interface Specification

ThermoLAN.dll Interface Specification

ThermoCOM.dll Interface Specification

Chapter 3

Interfacing to KFModule.dll

This chapter introduces an integration sample (C#) which demonstrates how to use the functions of the KFModule.dll.

Note. The sample solution is created with Microsoft Visual Studio 2012 and provided as is, it has not been fully tested for stability and may malfunction when used in unexpected way.

For detailed information of the KFModule.dll exported functions and instrument commands, see: KFModule.dll Interface Specification and KingFisher Presto Interface Specification.

3.1 Building the Integration Sample

Look for a folder named **IntegrationSample** provided together with this document. From that folder, open the **Sample.sln** with Microsoft Visual Studio 2012 or later.

Select Debug configuration and build the solution. You should get an output listing similar to one below.

```
1>------ Build started: Project: KFModuleWrapperLibrary, Configuration: Debug Any CPU ------
1>  KFModuleWrapperLibrary
    -> C:\temp\IntegrationSample\KFModuleWrapperLibrary\bin\Debug\KFModuleWrapperLibrary.dll
2>------ Build started: Project: Sample, Configuration: Debug Any CPU ------
2>  Sample -> C:\temp\IntegrationSample\Sample\bin\Debug\Sample.dll
2>          1 file(s) copied.
2>          1 file(s) copied.
2>          1 file(s) copied.
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Note the project named **KFModuleWrapperLibrary**. It provides an interface to the KFModule.dll and can be used as a starting point when designing your own interface library.

The project **Sample** contains test methods demonstrating how to communicate with a KingFisher Presto Simulator. The difference between connecting to the simulator or to a real instrument is very small. You just choose to call a function KFModule_OpenSimulator() instead of KFModule_Open(). The simulator is included in the **IntegrationSample**.

3.2 Running Tests with KingFisher Presto Simulator

Open Test Explorer from the Visual Studio and you should see these four unit tests:

1. TEST1_SimulatorStarted
2. TEST2_UploadProtocol
3. TEST3_RunProtocol
4. TEST4_RemoveProtocol

The KingFisher Presto simulator **KFPresto.exe** needs to be started before running the tests. See also **StartSimulator.bat** in the IntegrationSample package. It can be used to start the simulator with logging enabled. Then all communication messages between IntegrationSample and simulator are written to a file named **kfm.log**.

After simulator is started, run the tests in the given order and you should succeed.

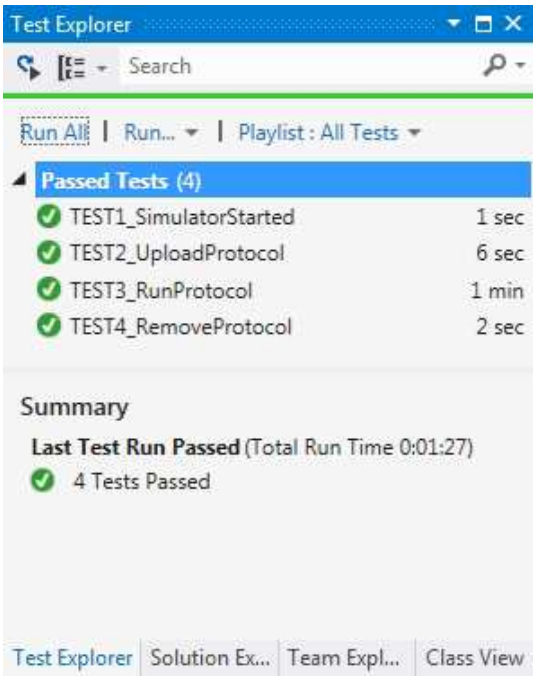

Figure 3.1: Passed tests

Chapter 4

Creating and Uploading Protocols

This chapter contains instructions how to create KingFisher Presto protocols and upload them into the instrument using BindIt PC software or KFModule.dll API. An example protocol named "My Test Protocol" is introduced. The example protocol is addressed later on in this document, when it is explained how to execute uploaded protocols in the KingFisher Presto instrument.

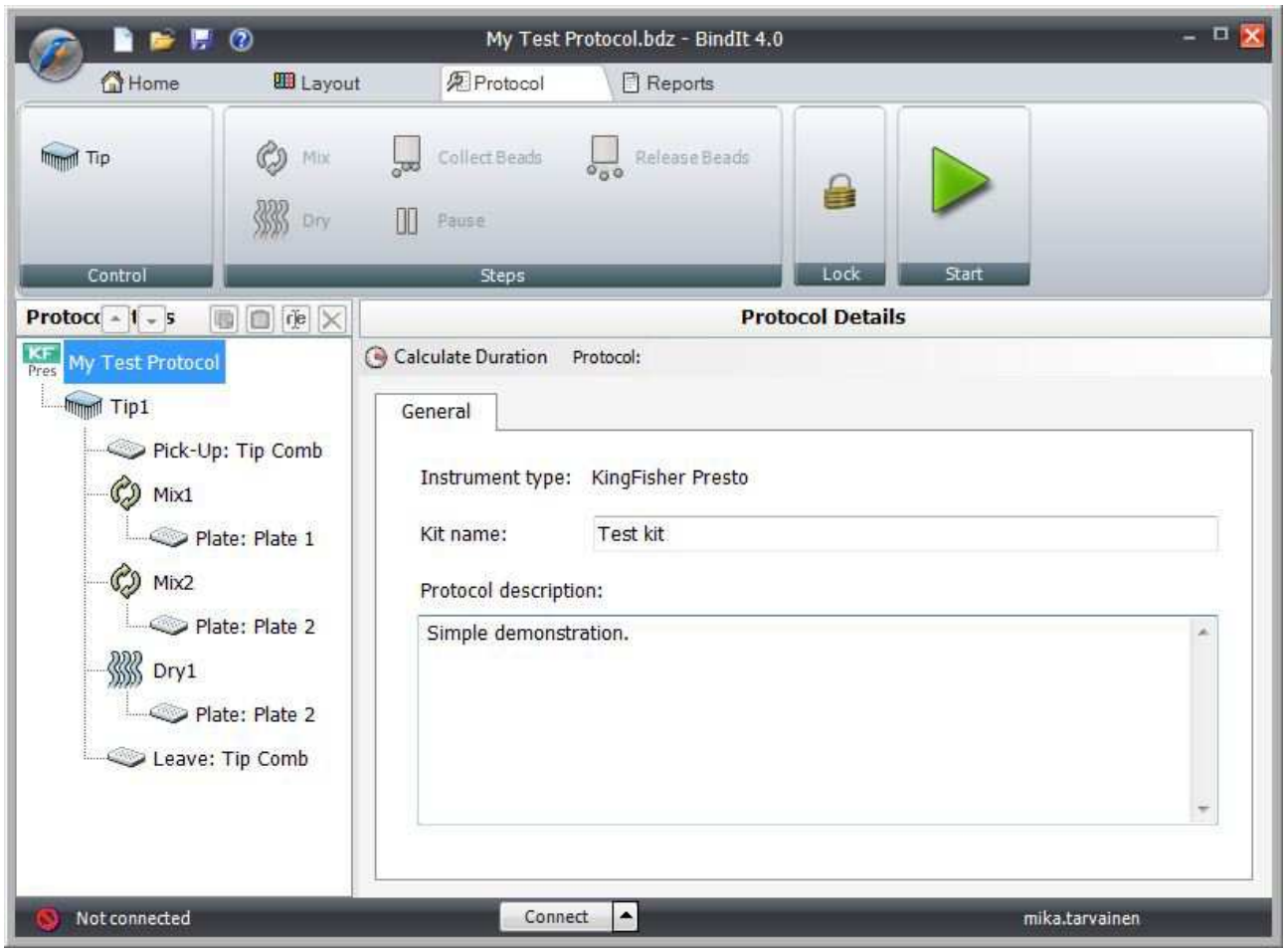

Figure 4.1: BindIt protocol editor

4.1 BindIt Protocol Editor

KingFisher Presto protocols area created with **BindIt** PC software. It can also upload and dowload the protocols to and from the KingFisher Presto instrument but it is not part of the Automation System. It can be used to run protocols "manually", meaning that a human person needs to handle all plate load/remove events.

4.2 Creating ”My Test Protocol”

Here are listed the steps needed to create the "My Test Protocol" example protocol. The outcome is a protocol structure with few plates and steps as shown in figure 5.2. Detailed instructions of how to create protocols can be found in the Thermo Scientific Bindit Software for KingFisher Instruments User Manual.

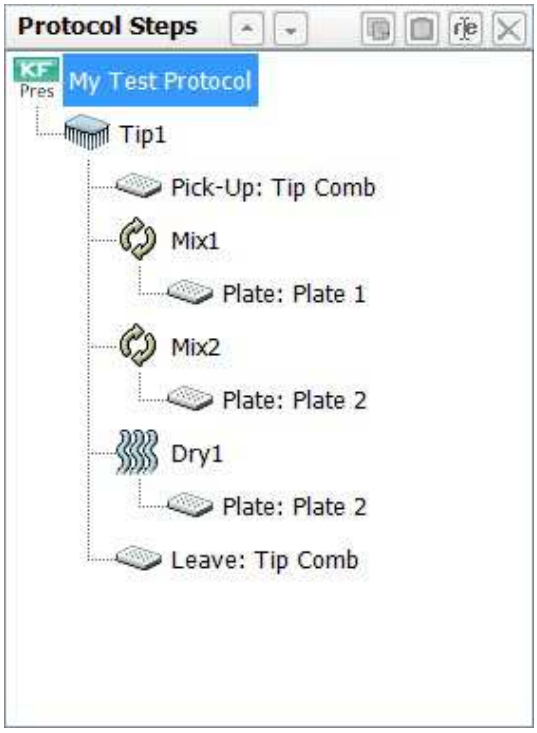

Figure 4.2: My Test Protocol

1. Open BindIt and select "Home" tab.
2. Press button "New" from the "Protocol" button group.
3. Select KingFisher Presto from the opened list.
4. Press button "New" from the "Plate" button group in "Layout" tab.
5. Select "96 DW plate"
6. Type "Tip Comb" to the "Plate name:" text box.
7. Add a dummy reagent by typing "None" to the reagent name row with guide text "Type a name to add new...". Default 50 µl volume is ok.
8. Create two more plates in a similar way, but name them "Plate 1" and "Plate 2".
9. Select "Protocol" tab.

10. Select "Tip1" from the "Protocol Steps" list and then "96 DW tip comb" from the "Tip:" dropdown list on the right on "General" tab.
11. Select "Pick-Up" from the "Protocol Steps" list and then "Tip Comb" from the "Plate:" dropdown list on the right on "Details" tab.
12. Select "Leave" from the "Protocol Steps" list and then "Tip Comb" from the "Plate:" dropdown list on the right on "Details" tab.
13. Select "Tip1" from the "Protocol Steps" list.
14. Press button "Mix" from the "Steps" button group in "Protocol" tab to create a mix step. Default parameters are ok.
15. Select "Plate" under the "Mix1" in the "Protocol Steps" list.
16. Select "Plate 1" from the from the "Plate:" dropdown list on the right on "Details" tab.
17. Create another Mix step in a similar way, but Select "Plate 2" for it.
18. Create a Dry step in a similar way and select "Plate 2" for it.
19. Save the protocol as "My Test Protocol" by pressing the save symbol on top and left of the window.
20. Verify that you created identical step list as in figure 5.2

4.3 Uploading ”My Test Protocol”

The easiest way to upload and test a protocol with a KingFisher Presto instrument is by connecting the instrument to BindIt and pressing a green Start symbol. However, the protocol is not saved permanently to the memory of the instrument and is available only during the execution of the protocol in BindIt software.

The green Start symbol can be found from the "Start" group, which is visible in all tabs and enabled after a protocol is created and saved. See quick guide steps below.

1. Open BindIt.
2. Power on a KingFisher Presto instrument and attach it to the computer running the BindIt with USB cable.
3. Connect BindIt to the instrument, see Thermo Scientific BindIt Software for KingFisher Instruments User Manual.
4. Open "My TestProtocol".
5. Press Start symbol from the "Start" group.

4.4 Uploading and Downloading Protocols

Connect to a KingFisher Presto instrument and open protocol transfer view in BindIt Software by pressing the "Transfer" symbol in the "Instrument" group of the "Home" tab. In this view it is possible to upload, download and remove protocols to/from the instrument. Note that the "Instrument" group is enabled only after the KingFisher Presto instrument is connected to the BindIt Software. Notice the button "Connect" or "Disconnect" in the status bar in the bottom of the window. See Thermo Scientific BindIt Software for KingFisher Instuments User Manual for more details about connecting to the instrument.

Protocols can be also saved to the file system folders of the computer running the BindIt Software. The file format of these files is "BindIt export data file" and the file extension is ".bdz". BindIt protocols can be uploaded and downloaded to/from the KingFisher Presto instrument through KFModule.dll API functions KFModule_UploadProtocol() and KFModule_Download-Protocol. See KFModule.dll Interface Specification for more details.

Chapter 5

Protocol Execution Methods

The automation interface offers two distinct ways to execute protocols on the KingFisher Presto instrument. The integrator can decide which option suits his/her needs the best. The first **Step-by-step** approach leaves the total control of the process to the automation system. The instrument control can be seen as a master-slave relationship between automation system and instrument. Alternatively, KingFisher Presto protocols can be executed using an **event-based** approach where the automation system has to react to events sent by the instrument. KingFisher Presto supports also overlapped multi-protocol executions as an application of Step-by-step method.

 **Caution**. The validities of the KingFisher Presto protocols provided by ThermoFisher Scientific are verified using Event-Based Execution method. It is up to the integrator to ensure that the protocols are executed correctly when using Step-by-Step Execution method or when Executing Multiple Overlapping Protocols.

5.1 Step-by-Step Execution

The automation can be build to run sequential step execution scripts where the turntable rotation commands are placed between the steps. The system does not need to react to the events coming from the instrument, it can poll the status of the execution instead.

KingFisher Presto protocols are created with BindIt protocol editor and uploaded to the instrument through the communication interface of the instrument. Protocol steps are executed by addressing them by name. As said before, turntable rotation is handled by the automation or the process management system. Understanding of the content of the protocol is not required for an integrator. Only the order of the steps needs to be understood. The execution of the "My Test Protocol" example protocol is descibed below. It assumes that the protocol is already uploaded, see Creating and Uploading Protocols.

 **Note.** The example demonstrates how the KingFisher Presto instrument replies to the commands and how the automation system needs to poll the instrument status before sending new commands. Not all of these repeatedly needed commands and responses are shown in order to keep the example readable. Furthermore, all event messages are stripped off. See KingFisher Presto Interface Specification for more details.

1. Rotate turntable to a startup position, meaning that nest 1 will be in the processing position. If the command can be executed, then the KingFisher Presto instrument replies immediately with "ok" response and starts rotating the turntable.
   <Cmd name="Rotate" nest="1" position="1" />
   <Res name="Rotate" ok="true" />

2. Read the status of the instrument until the reply from the instrument is "Idle" instead of "Busy“.
   <Cmd name="GetStatus" />
   <Res name="GetStatus" ok="true"><Status>**Busy**</Status></Res>
   <Cmd name="GetStatus" />
   <Res name="GetStatus" ok="true"><Status>**Busy**</Status></Res>
   ...
   ...
   ...
   <Cmd name="GetStatus" />
   <Res name="GetStatus"ok="true"><Status>**Idle**</Status></Res>

3. Load Tip Comb plate to the nest 2, which is in the load position, and rotate it to the processing position. Poll again for the instrument to return to the "Idle" state.
   <Cmd name="Rotate" nest="2" position="1" />
   <Res name="Rotate" ok="true" />
   <Cmd name="GetStatus" />
   <Res name="GetStatus" ok="true"><Status>**Idle**</Status></Res>

4. Start the "Pick-Up" step. Again, wait for the "Idle" status after the instrument has first replied with "ok" to the start command.
   <Cmd name="StartProtocol" protocol="My Test Protocol" tip="Tip1" step="Pick-Up" />
   <Res name="StartProtocol" ok="true" />
   <Cmd name="GetStatus" />
   <Res name="GetStatus"ok="true"><Status>**Busy**</Status></Res>
   <Cmd name="GetStatus" />
   <Res name="GetStatus" ok="true"><Status>**Busy**</Status></Res>
   ...
   ...
   ...
   <Cmd name="GetStatus" />
   <Res name="GetStatus" ok="true"><Status>**Idle**</Status></Res>

5. Place "Plate 1" to the nest 1, which is now in the load position and rotate it to the processing position.
   Note. The plate could have been loaded also during the time when the instrument was running the "Pick-Up" step.
   <Cmd name="Rotate" nest="1" position="1" />

6. Unload "Tip Comb" plate from the nest 2 in the load/unload position of the instrument and start the first mix step "Mix1" for the "Plate 1", which is now in the processing position.
   <Cmd name="StartProtocol" protocol="My Test Protocol" tip="Tip1" step="Mix1" />

7. Place "Plate 2" to the nest 2, which is now in the load position and rotate it to the processing position.
   <Cmd name="Rotate" nest="2" position="1" />

8. Unload "Plate 1" from the nest 1 and start step "Mix2" for the "Plate 2" in the processing position.
   <Cmd name="StartProtocol" protocol="My Test Protocol" tip="Tip1" step="Mix2" />

9. Start step "Dry1" for the "Plate 2" in the processing position.
   <Cmd name="StartProtocol" protocol="My Test Protocol" tip="Tip1" step="Dry1" />

10. Load Tip Comb plate to the nest 1, which is in the load position, and rotate it to the processing position. "Plate 2" will be rotated to the load/unload position.

```
<Cmd name="Rotate" nest="1" position="1" />
```

11. Unload Plate 2 from the load/unload position of the instrument and start step "Leave" to drop off the tips to the "Tip Comb" plate.
```
<Cmd name="StartProtocol" protocol="My Test Protocol" tip="Tip1" step="Leave" />
```

12. Rotate the turntable one more time to get the "Tip Comb" plate to the load/unload position. **Note.** This specific command example addresses the position 2, which is the load/unload position. Command "rotate nest 2 to the position 1" would have the same effect.
```
<Cmd name="Rotate" nest="1" position="2" />
```

13. Unload Tip Comb plate from the load/unload position after the instrument has finished the final rotation.
```
<Cmd name="GetStatus" />
<Res name="GetStatus" ok="true"><Status>Idle</Status></Res>
```

 **Caution**. Minimize delay times between steps.

Tips/magnets of the KingFisher Presto are always lifted up from a microwell plate and thus from the liquids within the plate when a step execution is finished. It is very important to minimize the delay before the next step so that the tips do not dry between the steps. Otherwise some applications may suffer. In practice, all the plates must be prepared beforehand. It is also recommended that the plate changes are designed in the way that a next plate is loaded to the load position during the time when a previous plate is being processed.

5.2 Event-Based Execution

KingFisher Presto protocols can be executed in a mode where the instrument handles turntable rotation by itself and sends events to the automation system whenever it requires attention. See example below demonstrating messaging between the instrument and an automation system running the "My Test Protocol" execution in event mode. The example is explained in detail after the listing. Note that not all events are listed in order to keep the listing readable. These events include **StepStarted ProtocolTimeLeft** and **Temperature**. See KingFisher Presto Interface Specification for more details about events, commands and responses.

```
<Cmd name="StartProtocol" protocol="My Test Protocol" />                    (1)
<Res name="StartProtocol" ok="true" />

<Evt name="LoadPlate" plate="Tip Comb" optional="false" />                  (2)
<Cmd name="Acknowledge" />
<Res name="Acknowledge" ok="true" />

<Evt name="LoadPlate" plate="Plate 1" optional="true" />                    (3)
<Evt name="LoadPlate" plate="Plate 1" optional="false" />
<Cmd name="Acknowledge" />
<Res name="Acknowledge" ok="true" />

<Evt name="ChangePlate" optional="true" >                                   (4)
    <Evt name="RemovePlate" plate="Tip Comb" optional="true" />
    <Evt name="LoadPlate" plate="Plate 2" optional="true" />
</Evt>
<Cmd name="Acknowledge" />
<Res name="Acknowledge" ok="true" />
```

```
<Evt name="ChangePlate" optional="true" >                                    (5)
    <Evt name="RemovePlate" plate="Plate 1" optional="true" />
    <Evt name="LoadPlate" plate="Tip Comb" optional="true" />
</Evt>
<Cmd name="Acknowledge" />
<Res name="Acknowledge" ok="true" />

<Evt name="RemovePlate" plate="Plate 2" optional="true" />                   (6)
<Cmd name="Acknowledge" />
<Res name="Acknowledge" ok="true" />

<Evt name="RemovePlate" plate="Tip Comb" optional="false" />                 (7)
<Cmd name="Acknowledge" />
<Res name="Acknowledge" ok="true" />

<Evt name="Ready" />                                                         (8)
```

1. Protocol is started by sending command "StartProtocol" to which the instrument immediately replies.
2. The first "LoadPlate" event is sent by the instrument when it is requesting "Tip Comb" plate to be inserted to the load position of the turntable. Value "false" of the attribute "optional" means that the instrument does not continue its operation until the automation system acknowledges the event. After a command "Acknowledge" is received and replied, the instrument rotates the turntable and starts processing the plate, which in this example means picking up the tip comb.
3. The second "LoadPlate" event is sent immediately after the processing of the tip comb is started. Attribute "optional" is now "true" indicating that the "Plate 1" can be placed to the load position but it is not obligatory. This event is ignored by the automation system in this example and so the instrument continues processing. After finished with the "Tip Comb" plate, the instrument sends another event requiring the "Plate 1" to be loaded. Attribute "optional" is now "false" and the plate needs to be placed to the load position. The instrument continues after it receives command "Acknowledge" and sends a reply to it. Again, the turntable is rotated and processing of the "Plate 1" starts.
4. There are plates in both nests of the turntable now. Processing of the "Plate 1" is under progress and the next plate needed is "Plate 2". Instrument sends a "ChangePlate" event to the automation system, telling that now it is a good time to remove "Tip Comb" from the load position and replace it with "Plate 2". Automation system does that and acknowledges. Instrument replies and sends no further events of this particular need, but rotates the turntable automatically after processing of the "Plate 1" is finished and starts processing "Plate 2" without any delay.
5. See previous. Event "ChangePlate" is sent in order to remove "Plate 1" and load "Tip Comb".
6. At this phase of the protocol execution the instrument has rotated the turntable again and is starting to drop off the tip comp to the "Tip Comb" plate. Before that, it sends a "RemovePlate" event suggesting that the "Plate 2" could be removed. Note that attribute "optional" is "true" meaning that it is just a suggestion, automation system could also remove the plate later. In this example the plate is removed by this time and command "Acknowledge" is sent and replied.
7. The instrument is finished with the "Tip Comb" plate and is now requesting it to be removed. Event "RemovePlate" is sent with attribute "optional" value "false". Automation system removes the last plate and sends a final "Acknowledge" command. Instrument replies.
8. Event "Ready" is sent by the instrument when the protocol execution is completed. This event needs not to be acknowledged.

5.3 Executing Multiple Overlapping Protocols

KingFisher Presto supports overlapped multi-protocol executions.

Caution. If a protocol contains **Mix** steps with pre-heating option enabled, then there is some differences in heater control between straightforward protocol execution and overlapped multi-protocol executions. See Precautions for Heater Control for more details.

See figure 6.1 below for an example protocol created in the BindIt protocol editor. The name of the protocol is "Protocol A" and it uses two microplates: one for a tip comb and another for mixing. Actual applications would never use just one plate for processing, but this way the following discussion is not too exhaustive.

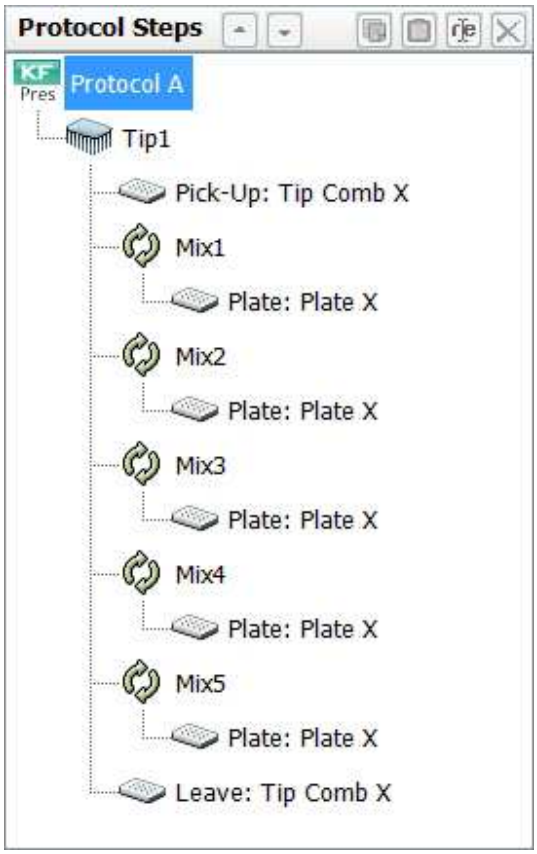

Figure 5.1: Protocol A

Probably the most likely scenario for multiprotocol execution is to run the protocols sequentially. See Figure 6.2 for example. However, some automation system integrators may find it useful that Step-by-Step Execution enables multi-protocol execution in an overlapped way, meaning that an automating system can execute several instances of a same protocol or different protocols by simply switching between the step lists. See Figure 6.3 for example.

It should be noted that typically every separate execution uses different tips and plates than the other executions, meaning that the context switch between executions requires "**Pick-Up**" and "**Leave**" steps around the actual step to be executed. Conseptually, this means that the context switch is actually a plastics switch where plastics are microwell plates and tip combs for the magnets. Need for a plastics switch is escpecially clear in Figure 6.3 in comparision with straighforward repeated execution as in the Figure 6.2 below.

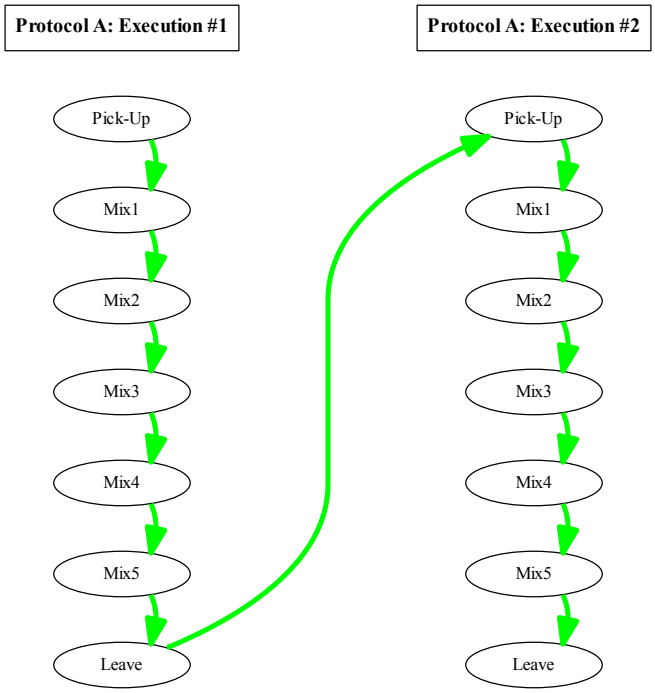

Figure 6.2: Consecutive executions of Protocol A

Figure 6.2 above and the command listing on the next page illustrate how a **Protocol A** is executed twice using Step-by-step execution method. The example protocol executions contain tips in a "Tip Comb" plates 1 and 2 and the actual plates to be processed are named as "Plate 1" and "Plate 2".

Note that the execution status polling and command reply messages are not listed in order to keep the example more readable. See Step-by-Step Execution for more details on those.

1. <Cmd name="Rotate" nest="1" position="2" />
2. Place "Tip Comb 1" plate to the turntable.
3. <Cmd name="Rotate" nest="2" position="2" />
4. Place "Plate 1" to the turntable.
5. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Pick-Up" />
6. <Cmd name="Rotate" nest="2" position="1" />
7. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix1" />
8. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix2" />
9. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix3" />

10. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix4" />
11. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix5" />
12. <Cmd name="Rotate" nest="2" position="2" />
13. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Leave" />
14. Remove "Plate 1" from the turntable.
15. <Cmd name="Rotate" nest="1" position="2" />
16. Remove "Tip Comb 1" plate from the turntable.
17. Place "Tip Comb 2" plate to the turntable.
18. <Cmd name="Rotate" nest="2" position="2" />
19. Place "Plate 2" to the turntable.
20. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Pick-Up" />
21. <Cmd name="Rotate" nest="2" position="1" />
22. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix1" />
23. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix2" />
24. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix3" />
25. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix4" />
26. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix5" />
27. <Cmd name="Rotate" nest="2" position="2" />
28. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Leave" />
29. Remove "Plate 2" from the turntable.
30. <Cmd name="Rotate" nest="1" position="2" />
31. Remove "Tip Comb 2" plate from the turntable.

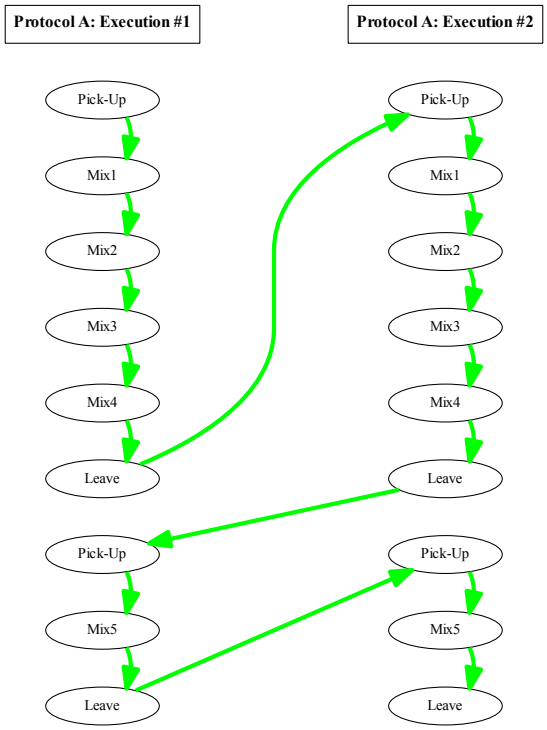

Figure 6.3: Context switch between two executions of Protocol A

Figure 6.3 above and the command listing below demonstrates switching of plastics when two instances or executions of a **Protocol A** are executed in an overlapped fashion.

The example protocol executions contain tips in a "Tip Comb" plates 1 and 2 and the actual plates to be processed are named as "Plate 1" and "Plate 2". Note that the execution status polling and command reply messages are not listed in order to keep the example more readable. See Step-by-Step Execution for more details on those.

1. <Cmd name="Rotate" nest="1" position="2" />
2. Place "Tip Comb 1" plate to the turntable.
3. <Cmd name="Rotate" nest="2" position="2" />
4. Place "Plate 1" to the turntable.
5. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Pick-Up" />
6. <Cmd name="Rotate" nest="2" position="1" />
7. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix1" />
8. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix2" />
9. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix3" />
10. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix4" />
11. <Cmd name="Rotate" nest="1" position="1" />
12. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Leave" />

13. Remove "Plate 1" from the turntable.
14. Place "Tip Comb 2" plate to the turntable.
15. <Cmd name="Rotate" nest="2" position="1" />
16. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Pick-Up" />
17. Remove "Tip Comb 1" from to the turntable.
18. Place "Plate 2" to the turntable.
19. <Cmd name="Rotate" nest="1" position="1" />
20. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix1" />
21. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix2" />
22. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix3" />
23. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix4" />
24. <Cmd name="Rotate" nest="2" position="1" />
25. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Leave" />
26. Remove "Plate 2" from the turntable.
27. Place "Tip Comb 1" plate to the turntable.
28. <Cmd name="Rotate" nest="1" position="1" />
29. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Pick-Up" />
30. Remove "Tip Comb 2" from to the turntable.
31. Place "Plate 1" to the turntable.
32. <Cmd name="Rotate" nest="2" position="1" />
33. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix5" />
34. <Cmd name="Rotate" nest="1" position="1" />
35. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Leave" />
36. Remove "Plate 1" from the turntable.
37. Place "Tip Comb 2" plate to the turntable.
38. <Cmd name="Rotate" nest="2" position="1" />
39. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Pick-Up" />
40. Remove "Tip Comb 1" from to the turntable.
41. Place "Plate 2" to the turntable.
42. <Cmd name="Rotate" nest="1" position="1" />
43. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Mix5" />
44. <Cmd name="Rotate" nest="2" position="1" />
45. <Cmd name="StartProtocol" protocol="Protocol A" tip="Tip1" step="Leave" />
46. Remove "Plate 2" from the turntable.
47. <Cmd name="Rotate" nest="2" position="2" />
48. Remove "Tip Comb 2" from to the turntable.

5.4 Precautions for Heater Control

Mix steps in KingFisher Presto protocols may have heating option enabled, meaning that a heating block is heated and lifted up against a plate during mixing. Furthermore, a preheat option can be enabled to ensure that the heating block will reach the target temperature just before the actual step where the heating is required. The heating block is positioned some distance below the plate during preheating.

There is some differences in heater control between straightforward protocol execution and overlapped multi-protocol executions when the preheat option is enabled. See Figure 6.4 for an example where pre-heat is enabled for the step **Mix5**. When the protocol is executed using Event-Based Execution method, then the KingFisher Presto instrument calculates the moment when the preheating needs to be started and then turns the heater on automatically. The moment arises in our example during step **Mix4**.

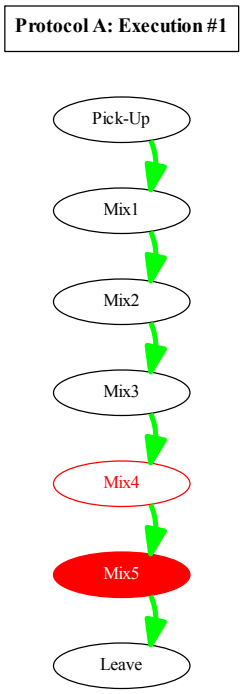

Figure 6.4: Protocol A with preheat enabled in Mix5

If Step-by-Step Execution method is used or when Executing Multiple Overlapping Protocols, then the KingFisherPresto follows the same automatic preheat control logic as when using Event-Based Execution method but with some restrictions:

1. If preheat is on and the step to be executed is not from the same protocol as previous executed step, then the preheat is turned off unless:
   - Step to be executed is "Pick-Up" or "Leave".
2. If preheat is on and the step to be executed is from the same protocol as previous executed step then the preheat is turned off unless:
   - Step to be executed is "Pick-Up" or "Leave".
   - Step to be executed is same as previous executed step.
   - Step to be executed is next in the protocol after the previous executed step.

3. Preheat is turned off automatically if a new step execution is not started within 10 minutes from the end of the previous step execution.
4. Preheat is never turned on during "Pick-Up" or "Leave" steps when Step-by-Step Execution method is used or when Executing Multiple Overlapping Protocols. This may cause a small delay to the preheating when compared to Event-Based Execution method.
5. If the step to be executed is the first step in a protocol, then the heater will be turned off at the beginning of the step execution.
6. If the step to be executed is the last step in a protocol and the step to be executed is from the same protocol as previous executed step, then the heater will be turned off at the end of the step execution.

 **Note**. If preheat is turned off, because of the above mentioned restrictions, then a warning event is sent to the automation system. Same warning is also included to the reply message of instrument status query command as long as the step is being executed.

 **Caution**. If a **Mix** step with heating and preheating contains long **End of step** actions e.q. **Postmix** or/and **Collect beads** and is switched between multiple execution instances, then the preheating is compromized. That is because the heater is always turned off after **Mixing** (e.q. before **Postmix**) and the heating block will be cooled down during the **Postmix**.

Chapter 6

Error Handling

There are several different error handling mechanisms in the communication interfaces of the KingFisher Presto instrument. Some of them are implemented in DLL level while some others are features of the instrument itself. See related interface specifications for detailed information:

- KFModule.dll Interface Specification

- KingFisher Presto Interface Specification

6.1 Communication Interfaces

Low level error handling is based on the mechanisms of the underlaying communication protocol, for example on guaranteed data integrity of the USB protocol. Some connection errors can be captured in DLL level, whereas some others require polling of the instrument status.

KFModule.dll is able to detect when USB connection is lost. Function **KFModule_AttachEvent()** can be used to subscribe **KFMODULE_ERROR** events. If the connection is lost, then event **KFM_ERROR_DISCONNECTED** will be sent.

There is no detection for failed connection when using RS232 interface. Then the user application e.q. an automation system needs to poll the instrument status with **GetStatus** command in order to be sure that the instrument is still online.

It is recommended to use the above mentioned **GetStatus** method also when using LAN interface, because the KFModule.dll can only detect a failed connection after some timeout when attemping to send commands to the instrument. Meaning that any spontaneous events or some replies are lost if the physical connection is disconnected.

6.2 Communication Protocol

KFModule.dll function calls all return error codes if they do not succeed.

Responses to instrument commands use "ok" attribute for replying if a command is accepted or rejected. Responses may also contain error and/or warning codes.

Reply to **GetStatus** instrument status query command may indicate "In error" status and contain error and/or warning codes. The controlling automation system needs to send an error acknowledge command to the instrument in order to get the instrument out of the error state.

Failed protocol and step executions and other spontaneous errors are reported with error events which may require that the controlling system sends an error acknowledge command before the instrument continues its operation.

Command set includes **Stop** and **Abort** commands for stopping and disconnecting the instrument. Abort command can be generated also from the physical maintenance interface by pressing the red stop symbol.

6.3 KingFisher Presto Protocol Integrity

Protocol upload command of the KingFisher Presto includes a 32-bit cyclic redundancy check (CRC) error-detecting code from the protocol data to be transferred. The instrument verifies that the CRC matches to the CRC of the received data. The CRC value is saved to the internal memory of the instrument together with the protocol data. The KFModule.dll API function KFModule_UploadProtocol() calculates CRC codes automatically.

Furthermore, the instrument calculates CRC from a saved protocol every time a protocol or a step from a protocol is about to be started. An error is generated if the calculated CRC does not match to the CRC value that was saved when the protocol in question was uploaded to the instrument.

Thermo Scientific

KingFisher Presto

Interface Specification

Thermo SCIENTIFIC

Contents

1 KingFisher Presto Interface Specification 1
- 1.1 Contents . . . 1
- 1.2 Confidential . . . 1
- 1.3 Introduction . . . 2

2 Hardware Requirements 3
- 2.1 RS232 . . . 3
  - 2.1.1 Abort . . . 3
- 2.2 USB . . . 3
  - 2.2.1 Sending commands to the instrument . . . 4
  - 2.2.2 Receiving responses from the instrument . . . 4
  - 2.2.3 Abort . . . 4
  - 2.2.4 Flow Control . . . 4
- 2.3 LAN . . . 4
  - 2.3.1 Abort . . . 5

3 Communication Protocol 7
- 3.1 General . . . 7
- 3.2 Character Encoding . . . 7
- 3.3 Maximum Line Width . . . 7
- 3.4 XML Format . . . 7
- 3.5 Attributes and Tag Data . . . 7
- 3.6 Root Tag Types . . . 8
  - 3.6.1 <Cmd> - Commands . . . 8
  - 3.6.2 <Res> - Responses . . . 8
  - 3.6.3 <Evt> - Events . . . 8
- 3.7 Error Handling . . . 9
- 3.8 Error Codes . . . 10
- 3.9 Warning Codes . . . 10

4 Commands and Responses 11
- 4.1 Abort . . . 12
- 4.2 Acknowledge . . . 14
- 4.3 Connect . . . 15
- 4.4 Disconnect . . . 17
- 4.5 DownloadProtocol . . . 18
- 4.6 ErrorAcknowledge . . . 19
- 4.7 GetProtocolDuration . . . 20
- 4.8 GetProtocolTimeLeft . . . 22
- 4.9 GetStatus . . . 23
- 4.10 ListProtocols . . . 24
- 4.11 RemoveProtocol . . . 25
- 4.12 Rotate . . . 26
- 4.13 SetTemperatureReporting . . . 27
- 4.14 StartProtocol . . . 28
- 4.15 Stop . . . 29

4.16 UploadProtocol 30

5 Events 31
5.1 Aborted 32
5.2 ChangeMagnets 33
5.3 ChangePlate 34
5.4 Error 35
5.5 LoadPlate 36
5.6 Pause 37
5.7 ProtocolTimeLeft 38
5.8 Ready 39
5.9 RemovePlate 40
5.10 StepStarted 41
5.11 Temperature 42

6 Appendix - XML path syntax 43

Chapter 1

KingFisher Presto Interface Specification

1.1 Contents

- Introduction
- Hardware Requirements
- Communication Protocol
- Commands and Responses
- Events

1.2 Confidential

1.3 Introduction

This document is intended for software designers writing computer programs for controlling the instrument. It contains information necessary to know in order to be able to write such a program. This document can also be used for black box testing of the instrument. It is assumed that the reader of this document is familiar with the function of the instrument.

Syntaxes of commands, responses and events are documented using a style similar to the `Xpath` syntax. For examples and more detailed description of the usage, see Appendix - XML path syntax.

Chapter 2

Hardware Requirements

The KingFisher Presto instrument has three alternate communication ports for connecting to a computer: RS232, USB and LAN.

2.1 RS232

Serial port parameters are fixed to the following values: baud rate: 115200, 1 start bit, 8 data bits, 1 stop bit and no parity.

The serial connector on the instrument is a 9 pin male D connector. Reception is through pin 3, transmission through pin 2 and signal ground is at pin 7.

XON/XOFF flow control is used. No hardware handshaking is used. The XON character is 0x11 and the XOFF character 0x13. No frames or checksums are used in the RS232 protocol. The instrument ignores any NUL bytes (0x00) it receives.

Maximum length of a command line is limited by the receive buffer size, which is 512 bytes. Note however that XOFF will be sent when the buffer is half full (256 characters).

2.1.1 Abort

To send the Abort command to the instrument, the computer must first flush it's transmit buffer and force transmit flow control to enabled state. Only then is the Abort command sent.

Abort command is a single character, the escape control code, which is decimal 27 or hex 0x1B.

2.2 USB

The instrument has a standard USB series "B" receptacle for connecting to a PC. The USB interface complies with the USB 2.0 full speed device specification. The device class of the instrument is HID (Human Interface Device), and it has only one configuration which is numbered 1. In this configuration there is one interface with an interrupt IN endpoint (0x81) and an interrupt OUT endpoint (0x01).

The HID Report descriptor of the instrument defines three reports: A 64 byte Input report, a 64 byte Output report and a 2-byte Feature report. As there is only one type of each report, no report ID:s are used. The usage of all the reports is vendor specific. Read the USB Device Class Definition for Human Interface Devices for more information of the HID device interface.

The following identification information is returned during enumeration:

Vendor id: 0x0AB6

Product id: 0x02C9

Manufacturer: Thermo Fisher Scientific Oy

Serial number: The serial number string of the instrument.

2.2.1 Sending commands to the instrument

Commands are sent using the 64 byte HID class Output report. The first byte of the report must be the number of actual data (command) bytes in the report. The command data starts from the second byte of the report. If the command data does not fill up the whole report, the instrument discards the remaining bytes of the report.

2.2.2 Receiving responses from the instrument

The PC software receives the instrument responses in a 64 byte HID class Input report. The first byte of the report is the number of actual response bytes in the report. The response data starts from the second byte of the report. If the response data does not fill up the whole report, the PC software must ignore the remaining bytes of the report.

2.2.3 Abort

Because there is no way the PC application can force an Abort to the interrupt OUT endpoint past previous commands, aborting must be performed in two phases.

First the PC must send the two-byte Feature report through the control endpoint. The first byte of the report must be nonzero and the second zero. This causes the instrument to discard all commands it may already have received and also discard all commands it receives until the Abort command is received.

Then the PC must send the Abort character through the interrupt OUT endpoint just like any other command.

2.2.4 Flow Control

When the receive buffer of the instrument becomes full, it simply stops reading the Output reports. This means that any command writes the PC software makes do not complete until the instrument resumes reading the reports.

If the PC application cannot process the Input reports at the same rate as the instrument sends them, it should send a Feature report to the instrument to request it stop sending the Input reports. The first byte of the Feature report must be zero and the second nonzero. When the PC software can again receive more Input reports, it should send a Feature report with both bytes zero.

2.3 LAN

The instrument can optionally be connected through a Local Area Network. The interface is IEEE 802.3 compliant and has a RJ-45 connector for connection to a 10BASE-T network.

Before the LAN interface can be used, the Ethernet interface MAC address and the TCP port number must be programmed using the PAR 38 and PAR 39 commands, respectively.

The instrument gets an IP address dynamically from a DHCP server. Unless the server is configured to give a fixed IP address to the instrument, a PC software wishing to connect to the instrument through LAN must use the WS-Discovery protocol to find the transport address of the instrument.

There are three strings the instrument uses for matching the Types of a WS-Discovery Probe message: ThermoDevice, KingFisherPresto and SN_$*$, where $*$ is the serial number string of the instrument. An example Probe looking for KingFisherPresto instrument serial number 12345:

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope
 xmlns:s="http://www.w3.org/2003/05/soap-envelope"
```

```
 xmlns:a="http://www.w3.org/2005/08/addressing">
 xmlns:d="http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01">
    <s:Header>
        <a:Action>http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Probe</a:Action>
        <a:MessageID>urn:uuid:dc280767-49b0-4c5e-97f3-95f961df0053</a:MessageID>
        <a:ReplyTo>
            <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
        </a:ReplyTo>
        <a:To>urn:docs-oasis-open-org:ws-dd:ns:discovery:2009:01</a:To>
    </s:Header>
    <s:Body>
        <Probe xmlns="http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01">
            <d:Types>ThermoDevice KingFisherPresto SN_12345</Types>
        </Probe>
    </s:Body>
</s:Envelope>
```

If the instrument is connected to LAN, it responds with a ProbeMatches message. The <d:Xaddrs> field of the response is the transport address to use to communicate with the instrument. The transport address consists of the instrument IP address and the TCP port number the instrument is listening.

Although the instrument always responds to WS-Discovery messages, it only lets one client at a time to connect to the TCP port. The client should close the port when done communicating with the instrument to allow another client to connect.

2.3.1 Abort

Sending the Abort character to the instrument is a two phase process.

First the PC must send an UDP message containing text "Abort" (without quotes or newline) to the same UDP port number as is used for the TCP and wait a moment for an identical UDP response from the instrument. The UDP response is sent to the same UDP port as the PC used for sending the UDP message. If no UDP response, retry at least two times using short timeout.

Then, when an UDP response is received or no response after retries, send Abort character (0x1B) to the TCP port.

Chapter 3

Communication Protocol

3.1 General

This document describes the commands and the responses as they are seen by the controlling software and the instrument without the framing added by the data transport layer.

3.2 Character Encoding

All commands and responses are ASCII encoded except values, which are encoded using UTF-8 character encoding.

3.3 Maximum Line Width

Communication is based on CR or/and LF terminated lines.


Note. The maximum lenght of a input line including carriage return and line feed characters is **200**.

3.4 XML Format

All commands, responses and events are XML elements. Elements may or may not contain line feed and carriage return characters. Both start-tag-end-tag and empty-element-tag formats are supported. Tag and parameter names are not case sensitive.

Syntaxes of commands, responses and events are documented using a style similar to the `Xpath` syntax. For examples and more detailed description of the usage, see Appendix - XML path syntax.

3.5 Attributes and Tag Data

Attribute values can be booleans, integers, floating point numbers or strings. Each command and response contains different amount of attributes and tags and the order of those is not fixed.

3.6 Root Tag Types

See table below for the three different kind of root tags used in the communication protocol between the KingFisher Presto instrument and a controlling system.

Tag	Description
Cmd	Command from the controlling system to the KingFisher Presto instrument
Res	Response to a command
Evt	Event message from the KingFisher Presto instrument to the controlling system

3.6.1 <Cmd> - Commands

Commands are sent from a controlling system to the KingFisher Presto instrument. They are used for example to start a protocol execution.

Examples:

```
<Cmd name="MyCommand" parameter1=”Something”>
        <Something>More complex value</Something>
</Cmd>

<Cmd name="MyOtherCommand" parameter1=”Nonething” />
```

See Commands and Responses for the complete list of available commands.

3.6.2 <Res> - Responses

Responses to the commands contain attribute named "ok", which can be used to quickly check if a command was succesfull.

Examples:

```
<Res name="MyCommand" ok=”false”>
        <Thing>Lots of things</Thing>
</Res>

<Res name="MyOtherCommand" ok=”true” />
```

See Commands and Responses for detailed information.

3.6.3 <Evt> - Events

Events are used to indicate data periodically or in the case of events occurring. Some events require acknowledgement from the controlling system, see command Acknowledge for more information.

Examples:

```
<Evt name="Temperature" value=”23”/>

<Evt name="LoadPlate" plate="Plate 1"/>

<Evt name="LoadPlate" plate="Plate 2"/>
```

```
<Evt name="ChangePlate">
    <Evt name="RemovePlate" plate="Plate 2"/>
    <Evt name="LoadPlate" plate="Plate 3"/>
</Evt>

<Evt name="RemovePlate" plate="Plate 3"/>
```

See Events for the complete list of events sent by the KingFisher Presto instrument.

3.7 Error Handling

Low level error handling is based on the mechanisms of the underlaying communication protocol, for example on guaranteed data integrity of the USB protocol.

Commands and Responses use "ok" attribute for replying if a command is accepted or rejected, see <Res> - Responses. Responses to commands may also contain Error Codes and/or Warning Codes.

Failed protocol and step executions and other spontaneous errors are reported with Error events which may require that the controlling system sends an ErrorAcknowledge command before the instrument continues its operation.

Examples

```
<Cmd name="StartProtocol" protocol=”KF Blood 12 DW”/>

<Res name="StartProtocol" ok=”true” />

<Cmd name="StartProtocol" protocol=”KF Blood 12 DW”/>

<Res name="StartProtocol" ok=”false”>
    <Error code="124">Protocol already running.</Error>
</Res>

<Evt name="Error" ack="true">
    <Error code="321">Execution failed</Error>
</Evt>
```

3.8 Error Codes

Code	Description
2	Received an unknown command.
3	Already connected to another port.
4	Head position error.
5	Magnets position error.
6	Turntable position error.
7	Heater unit position error.
8	Lock position error.
11	Invalid command argument.
13	Protocol memory error.
14	Protocol memory is full.
15	No protocols found from the protocols memory.
16	Protocol was not found from the protocols memory.
17	Given tip name was not found from the protocol.
18	Given step name was not found from the given tip of the protocol.
19	A name of a step to start was not given.
20	A name of a tip where to start the step was not given.
23	Protocol name is invalid. Maximum lenght of the name is 100 bytes e.q. 100 ASCII characters.
24	Invalid protocol file.
25	Protocol is not executable.
27	Protocol is too large and can't be loaded.
28	Instrument is executing, please wait.
32	No protocol is currently running.
33	Data transmit to USB port failed (timed out).
34	Cannot run magnets down without tips.
35	Magnetic head is missing.
38	Plate not detected in processing position.
39	Plate detected in processing position.
40	Plate not detected in load position.
41	Plate detected in load position.
43	Protocol is not for this instrument.

3.9 Warning Codes

Code	Description
101	Instrument is already connected.
102	Previous command was incomplete.
103	Date/time string is invalid, instrument time was not set.
104	Protocol execution was aborted by the user.
105	Step execution was aborted by the user.
106	Existing protocol was overwritten.
107	Heater preheat was turned off between single step executions.
108	Heater was turned off after 10 minutes since last single step execution.

Chapter 4

Commands and Responses

Note. Remember to terminate commands with a new line character (ASCII 10).

- Abort
- Acknowledge
- Connect
- Disconnect
- DownloadProtocol
- ErrorAcknowledge
- GetProtocolDuration
- GetProtocolTimeLeft
- GetStatus
- ListProtocols
- RemoveProtocol
- Rotate
- SetTemperatureReporting
- StartProtocol
- Stop
- UploadProtocol

4.1 Abort

Aborts any ongoing operation in the instrument.

Syntax

The Abort command is an exception to the normal command syntax. For example for RS232 interface, it is a single character, the escape control code, which is decimal 27 or hex 0x1B. When the Abort character is received, the execution of a possible ongoing protocol, step or some other process is stopped. Heating is turned off and all the motors are stopped. Heating block is driven down if possible. Response to the Abort command is replied immediately and event Ready is sent after motors are stopped/driven. In case of errors during driving, event Error is sent.

Abort command differs from the command Stop in the way, that the message buffers of the communication port are flushed and an existing communication connection is disconnect. Event Aborted is sent before disconnection to a connected communication port if the Abort command was received from some other communication port.

Immediate response to the Abort command follows the same syntax as the other Commands and Responses use.

Syntax

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "Abort"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if the command was accepted

Details

For the instrument to react to the Abort command immediately it must reach the instrument promptly after it is sent. This means that the Abort command must override all previous commands queued to the instrument. This is not trivial and is done differently for different interfaces.

Serial port Before sending the Abort character, the flow control must be forced to XON state so that the Abort character will actually be sent.

USB port Because there is no way the PC application can force an Abort to the interrupt OUT endpoint past previous commands, aborting must be performed in two phases.

First the PC must send the two-byte Feature report through the control endpoint. The first byte of the report must be nonzero and the second zero. This causes the instrument to discard all commands it may already have received and also discard all commands it receives until the Abort command is received.

LAN port First the PC must send an UDP message containing text "Abort" (without quotes or newline) to the same UDP port number as is used for the TCP and wait a moment for an identical UDP response from the instrument. The UDP response is sent to the same UDP port as the PC used for sending the UDP message.

If no UDP response, retry at least two times using short timeout.

When an UDP response is received or no response after retries, send Abort character (0x1B) to the TCP port.

Using KFModule.dll By far the easiest way to send Abort to the instrument is to use the KFModule_Abort() function of the KFModule.dll library. It will automatically select the right Abort procedure depending on which interface is used.

Example

```
<Res name="Abort" ok="true"/>
```

4.2 Acknowledge

General notification message for acknowledging the KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "Acknowledge"	Name of the command

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "Acknowledge"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

This command is used to acknowledge the instrument in various different kind of situations, for example after LoadPlate event.

Example

```
<Cmd name="Acknowledge"/>

<Res name="Acknowledge" ok="true"/>
```

4.3 Connect

Establish a connection between a controlling system and a KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String:"Connect"	Name of the command
Cmd@setTime	Date/time string in format: "YYYY-MM-DD hh:mm:ss"	Optional attribute for setting the instrument date and time

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "Connect"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull
Res/Instrument/text()	String: Max 100 chars	Name/type of the instrument
Res/Version/text()	String: Max 100 chars	Firmware version string
Res/Serial/text()	String: Max 100 chars	Instrument serial number

Details

The instrument does not accept any other commands before this command is succesfully received. Instrument serial number, type and firware version number is returned in the reply message.

Optional attribute "setTime" can be used to set the date and time of the instrument. There is no battery backup for the calendar of the instrument, so the date and time needs to be re-set every time the instrument powers on. Note that for example KingFisherDll Windows driver does this automatically.

Response message may contain warning and error codes depending on the state of the instrument. See command Get-Status. See also examples below for most typical use cases.

Example 1 - Succesfully connected

```
<Cmd name="Connect"/>

<Res name="Connect" ok="true">
    <Instrument>KingFisher Presto</Instrument>
    <Version>0.0.0</Version>
    <Serial>123-456</Serial>
</Res>
```

Example 2 - Connect with date/time setting

```
<Cmd name="Connect" setTime="2015-09-04 09:29:59"/>

<Res name="Connect" ok="true">
    <Instrument>KingFisher Presto</Instrument>
    <Version>0.0.0</Version>
    <Serial>123-456</Serial>
</Res>
```

Example 3 - Allready connected

```
<Cmd name="Connect"/>

<Res name="Connect" ok="true">
    <Warning code="101">Instrument is already connected.</Warning>
    <Instrument>KingFisher Presto</Instrument>
    <Version>0.0.0</Version>
    <Serial>123-456</Serial>
</Res>
```

Example 4 - Connection is reserved for another communication port

```
<Cmd name="Connect"/>

<Res name="Connect" ok="false">
    <Error code="3">Allready connected to another port.</Error>
</Res>
```

Example 5 - Connection is established but the instrument is in error state

```
<Cmd name="Connect"/>

<Res name="Connect" ok="true">
    <Error code="4">Head position error.</Error>
    <Instrument>KingFisherPresto</Instrument>
    <Version>0.0.1</Version>
    <Serial>123-456</Serial>
</Res>
```

4.4 Disconnect

Disconnect an established connection between a controlling system and a KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "Disconnect"	Name of the command

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "Disconnect"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

Note that the instrument does not reply to this command if the connection is already closed. See command Connect for more details.

Example

```
<Cmd name="Disconnect"/>

<Res name="Disconnect" ok="true"/>
```

4.5 DownloadProtocol

Download a protocol from the internal memory of the KingFisher Presto instrument.

Note! It's much easier to use **KFModule_DownloadProtocol()** API function from **KFModule.dll** than this command directly. KFModule_DownloadProtocol handles base64 decoding and outputs BindIt .bdz files.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "DownloadProtocol"	Name of the command
Cmd@protocol	String: Max 100 bytes	Name of the protocol to be trasferred, case sensitive.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "DownloadProtocol"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull
Res/CDATA	Base64 string/strings	XML character data section containing base64 encoded protocol file

Details

Protocol is received as base64 encoded binary data in a CDATA section of the response message.

Example

```
<Cmd name="DownloadProtocol" protocol="KingFisher Presto Blood 24 DW"/>

<Res name="DownloadProtocol" ok="true">
    <![CDATA[
        PHN0ZXA+DQpoZXJlIGlzIGFu
        IGV4YW1wbGUNCmZvciB0aGUg
        YmFzZTY0IGVuY29kZXINCjwv
        c3RlcD4=
    ]]>
</Res>
```

4.6 ErrorAcknowledge

This command must be used to clear instrument errors.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "ErrorAcknowledge"	Name of the command

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "ErrorAcknowledge"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

When an error is detected during execution, it is reported using Error event. This command must be send in order to clear the error state of the instrument. See also command GetStatus.

Example

```
<Cmd name="ErrorAcknowledge"/>

<Res name="ErrorAcknowledge" ok="true"/>
```

4.7 GetProtocolDuration

Get KingFisher Presto protocol duration.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "GetProtocolDuration"	Name of the command
Cmd@protocol	String: Max 100 chars	Name of a protocol, case sensitive.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "GetProtocolDuration"	Name of the command.
Res@ok	Boolean string: "true" or "false"	Error status quick peek.
Res/Init@protocol	String	Name of the protocol.
Res/Init/TimeStamp[1,2]@at	Time string: hh:mm:ss	Start or stop time of the protocol initialization.
Res/Init/TimeStamp[1,2]@type	Type string of the timestamp: "Start" or "Stop"	Init step uses only "Start" and "Stop" types.
Res/Init/TimeStamp[1,2]@step	String "Init"	Fixed step name.
Res/Init/TimeStamp[2]@duration	XML Duration data type	Duration of the protocol initialization phase.
Res/Tip/TimeStamp[n]@at	Time string: hh:mm:ss	Timestamp referenced to the start time of the protocol initialization.
Res/Tip/TimeStamp[n]@type	Type string of the timestamp: "Start", "Event" or "Stop"	Time of step start/stop or an pause event. See command GetProtocolTimeLeft.
Res/Tip/TimeStamp[n]@step	String	Name of a step, fixed "pseudo" step or user defined, see event StepStarted.
Res/Tip/TimeStamp[n]@duration	XML Duration data type	Duration of a step.
Res/Tip/TimeStamp[n]@msg	String	Message string for "Event" type.
Res/Tip/TimeStamp[n]@plate	String	Name of a plate for "Event" type.
Res/Tip/TimeStamp[n]@remove	String	Name of a plate to remove for "ChangePlate" event type.
Res/Tip/TimeStamp[n]@load	String	Name of a plate to load for "ChangePlate" event type.
Res/Init@protocol	String	Name of the protocol.
Res/Finish/TimeStamp[1,2]@at	Time string: hh:mm:ss	Start or stop time of the protocol finalization.
Res/Finish/TimeStamp[1,2]@type	Type string of the timestamp: "Start" or "Stop"	Finish step uses only "Start" and "Stop" types.
Res/Finish/TimeStamp[1,2]@step	String "Init"	Fixed step name.
Res/Finish/TimeStamp[2]@duration	XML Duration data type	Duration of the protocol initialization phase.
Res/Total@duration	XML Duration data type	Total duration of the protocol.

Details

This command returns the full time structure of a KingFisher Presto protocol as a step by step list of timestamps in addition to the total duration.

Examples

```
<Cmd name="GetProtocolDuration" protocol="My Test Protocol" />

<Res name="GetProtocolDuration" ok="true">
    <Init protocol="My Test Protocol">
        <TimeStamp at="00:00:00" type="Start" step="Init"/>
        <TimeStamp at="00:00:00" type="Stop"  step="Init" duration="PT0S"/>
    </Init>
    <Tip name="Tip1">
        <TimeStamp at="00:00:00" type="Start" step="Pick-Up"/>
        <TimeStamp at="00:00:00" type="Event" msg="Load plate" plate="Tip Comb"/>
        <TimeStamp at="00:00:08" type="Stop"  step="Pick-Up" duration="PT8S"/>
        <TimeStamp at="00:00:08" type="Start" step="Mix1"/>
        <TimeStamp at="00:00:08" type="Event" msg="Load plate" plate="Plate 1"/>
        <TimeStamp at="00:01:41" type="Stop"  step="Mix1" duration="PT1M33S"/>
        <TimeStamp at="00:01:41" type="Start" step="Mix2"/>
        <TimeStamp at="00:01:41" type="Event" msg="Change plate" remove="Tip Comb" load="Plate 2"/>
        <TimeStamp at="00:03:15" type="Stop"  step="Mix2" duration="PT1M34S"/>
        <TimeStamp at="00:03:15" type="Start" step="Dry1"/>
        <TimeStamp at="00:04:17" type="Stop"  step="Dry1" duration="PT1M2S"/>
        <TimeStamp at="00:04:17" type="Start" step="Leave"/>
        <TimeStamp at="00:04:17" type="Event" msg="Change plate" remove="Plate 1" load="Tip Comb"/>
        <TimeStamp at="00:04:24" type="Stop"  step="Leave" duration="PT7S"/>
        <TimeStamp at="00:04:24" type="Start" step="Unload"/>
        <TimeStamp at="00:04:24" type="Event" msg="Remove plate" plate="Plate 2"/>
        <TimeStamp at="00:04:24" type="Event" msg="Remove plate" plate="Tip Comb"/>
        <TimeStamp at="00:04:26" type="Stop"  step="Unload" duration="PT2S"/>
    </Tip>
    <Finish protocol="My Test Protocol">
        <TimeStamp at="00:04:26" type="Start" step="Finish"/>
        <TimeStamp at="00:04:26" type="Stop"  step="Finish" duration="PT0S"/>
    </Finish>
    <Total duration="PT4M26S"/>
</Res>
```

4.8 GetProtocolTimeLeft

Get time left of a KingFisher Presto protocol or single step execution.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "GetProtocolTimeLeft"	Name of the command
Cmd@protocol	String: Max 100 chars	Name of a protocol, case sensitive.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "GetProtocolTimeLeft"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek.
Res/TimeToPause@value	XML Duration data type	Time to next plate load/change/remove or pause event
Res/TimeLeft@value	XML Duration data type	Total time left of the protocol/step execution

Details

This command returns time left to a next pause event and total time left of a protocol execution.

Note! Element **TimeToPause** is not included when running single step execution. Note also that then the value of **TimeLeft** refers to the total execution time of a step being executed.

A pause event is an event which needs to be acknowledged with Acknowledge command. Non-optional LoadPlate, ChangePlate, RemovePlate and ChangeMagnets events are also pause events in addition to obvious Pause event.

Note that Error is not a pause event and it needs to be acknowledged with a special ErrorAcknowledge command.

See also event ProtocolTimeLeft.

Examples

```
<Cmd name="StartProtocol" protocol="My Test Protocol"/>
    <Res name="StartProtocol" ok="true"/>
    <Cmd name="GetProtocolTimeLeft"/>

    <Res name="GetProtocolTimeLeft" ok="true">
    <TimeToPause value="PT0S"/>
    <TimeLeft value="PT30M53S"/>
</Res>


    <Cmd name="StartProtocol" protocol="My Test Protocol" tip="Tip1" step="Mix1"/>
    <Res name="StartProtocol" ok="true"/>
    <Cmd name="GetProtocolTimeLeft"/>

<Res name="GetProtocolTimeLeft" ok="true">
    <TimeLeft value="PT2M42S"/>
</Res>
```

4.9 GetStatus

Get instrument status: **Idle**, **Busy** or **In error**.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "GetStatus"	Name of the command

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "GetStatus"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull and the instrument is not in an error state.
Res/Status/text()	String: "Idle", "Busy" or "In error"	Status of the instrument. Idle: waiting for a new commmand, Busy: instrument is executing a command, In error: instrument is in an error state.
Res/Error@code	uint16_t: see Error Codes	Error code number
Res/Error/text()	String: Max 100 characters	Description of an error

Details

This command is used to query the status or state of the instrument. When the instrument is executing StartProtocol or Rotate command, it will reply with **Busy** status. When an execution is finished or after succesfull power on initialization, the instrument will reply with **Idle** status. If an error occurs during an execution or initialization, then the instrument will transition to an error state and reply with **In error** status to the GetStatus command. An ErrorAcknowledge command must be send in order to clear the error. No other commands are accepted until the error is cleared. Expect commands Disconnect and Connect. Note that Error event is also sent if and an error occurs during execution.

Examples

```
<Cmd name="GetStatus"/>

<Res name="GetStatus" ok="true">
    <Status>Idle</Status>
</Res>

<Res name="GetStatus" ok="true">
    <Status>Busy</Status>
</Res>

<Res name="GetStatus" ok="false">
    <Error code="5">Turntable position error.</Error>
    <Status>In error</Status>
</Res>
```

4.10 ListProtocols

List protocols in the internal memory of the KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "ListProtocols"	Name of the command

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "ListProtocols"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull
Res/Protocols/Protocol[n]	String: Max 100 chars	Unique name of a protocol "n"
Res/MemoryUsed@value	uint16_t: 0 - 100	Protocol memory usage percentage

Details

This command returns a list of all available protocols in the instrument and a protocol memory usage percentage.

Example

```
<Cmd name="ListProtocols"/>

<Res name="ListProtocols" ok="true">
    <Protocols>
        <Protocol>KingFisher Presto Blood 24 DW</Protocol>
        <Protocol>KingFisher Presto Cells 96</Protocol>
        <Protocol>KingFisher Presto Demo 24 DW</Protocol>
    </Protocols>
    <MemoryUsed value="35"/>
</Res>
```

4.11 RemoveProtocol

Remove a protocol from the internal memory of the KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "RemoveProtocol"	Name of the command
Cmd@protocol	String: Max 100 bytes	Name of the protocol to be removed, case sensitive.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "RemoveProtocol"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

Protocol is removed permanently from the internal memory of the instrument.

Example

```
<Cmd name="RemoveProtocol" protocol="KingFisher Presto Blood 24 DW"/>

<Res name="RemoveProtocol" ok="true"/>
```

4.12 Rotate

Rotate the turntable.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "Rotate"	Name of the command
Cmd@nest	uint16_t: 1, 2	Nest of the turntable to be positioned
Cmd@position	uint16_t: 1, 2	The position where the nest is to be rotated. Position 1 is the procesessing position and position 2 is the load/unload position.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "Rotate"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

Response to the command is replied immediately and event Ready is sent by the instrument after the rotation is completed. If rotation fails, then an Error event will be sent instead. See also command GetStatus.

Example

```
<Cmd name="Rotate" nest="1" position="2" />

<Res name="Rotate" ok="true"/>
```

4.13 SetTemperatureReporting

Enable/disable temperature events with desired interval.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "SetTemperatureReporting"	Name of the command
Cmd@interval	XML Duration data type	Interval of the temperature event. Min 1 second.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "SetTemperatureReporting"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull and the instrument is not in an error state.

Details

This command can be used to trigger Temperature events with a minimum interval of one second. If the interval is set to "PT0S" or zero, then the events are disabled.

Note The interval is fixed to five seconds in KingFisher Presto Windows simulator and it can vary depending on the the operating system load and version.

Examples

```
<Cmd name="SetTemperatureReporting" interval="PT3S"/>

<Res name="SetTemperatureReporting" ok="true" />

<Evt name="Temperature">
   <Ambient value="22.1"/>
   <Heater value="37.7"/>
</Evt>

<Evt name="Temperature">
   <Ambient value="22.2"/>
   <Heater value="37.6"/>
</Evt>
```

4.14 StartProtocol

Start executing a protocol or a step from a protocol.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "StartProtocol"	Name of the command
Cmd@protocol	String: Max 100 chars	Name of the protocol to be started, case sensitive.
Cmd@tip	String: Max 100 chars	Optional, name of a tip containing the step to be started.
Cmd@step	String: Max 100 chars	Optional, name of the step to be started.

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "StartProtocol"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

This command is accepted if the instrument is not already running a step or a protocol. Event Ready is sent by the instrument after the execution. If attributes "tip" and "step" are given, then a single step execution is started instead of a full protocol execution. If execution fails, then an Error event will be sent instead. See also command GetStatus.

Note. The protocol needs to be uploaded to the internal memory of the instrument beforehand. See commands ListProtocols and UploadProtocol.

Note. If this command is used to start single step execution, then the controlling system needs to handle turtable rotations by sending separate Rotate commands.

Event StepStarted is sent for every step in the protocol being executed.

Examples

```
<Cmd name="StartProtocol" protocol="KingFisher Presto Blood 96 DW"/>

<Res name="StartProtocol" ok="true"/>

<Cmd name="StartProtocol" protocol="KingFisher Presto Blood 96 DW" tip="Tip1" step="Mix1"/>

<Res name="StartProtocol" ok="true"/>
```

4.15 Stop

Stop execution.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "Stop"	Name of the command

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "Stop"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

The Stop command is used to stop an ongoing execution of protocol, step or some other process. Response to the command is replied immediately and event Ready is sent after all the motors of the instrument are stopped In case of errors during stopping, event Error is sent.

Note that in addition to the Stop command there is also the isUsbAbort procedure for stopping the instrument. The isUsb-Abort procedure differs from the normal command-reply message exchange, see isPageUsbInterface for more detailed information.

Example

```
<Cmd name="Stop"/>

<Res name="Stop" ok="true"/>
```

4.16 UploadProtocol

Transfer a protocol to the internal memory of the KingFisher Presto instrument.

Note! It's much easier to use **KFModule_UploadProtocol()** API function from **KFModule.dll** than this command directly. KFModule_UploadProtocol takes BindIt .bdz files as input parameter and handles both CRC calculation and base64 encoding.

Syntax

Tag / Attribute	Data type and range/limits	Description
Cmd@name	String: "UploadProtocol"	Name of the command
Cmd@protocol	String: Max 100 bytes	Name of the protocol to be trasferred, case sensitive.
Cmd@crc	uint32_t	CRC value of the BindIt protocol file data.
Cmd[CDATA]	Base64 string/strings	XML character data section containing base64 encoded protocol file

Reply

Tag / Attribute	Data type and range/limits	Description
Res@name	String: "UploadProtocol"	Name of the command
Res@ok	Boolean string: "true" or "false"	Error status quick peek, true if command was succesfull

Details

Existing protocol is overwritten. Possible error codes are listed in the response message.

Note! Lines in the CDATA section must be n x 4 characters of base64 encoded data. Except the last line.

Example

```
<Cmd name="UploadProtocol" protocol="KingFisher Presto Blood 24 DW" crc="123456" >
    <![CDATA[
        PHN0ZXA+DQpoZXJlIGlzIGFu
        IGV4YW1wbGUNCmZvciB0aGUg
        YmFzZTY0IGVuY29kZXINCjwv
        c3RlcD4=
    ]]>
</Cmd>

<Res name="UploadProtocol" ok="true"/>
```

Chapter 5

Events

- Aborted
- ChangeMagnets
- ChangePlate
- Error
- LoadPlate
- Pause
- ProtocolTimeLeft
- Ready
- RemovePlate
- StepStarted
- Temperature

5.1 Aborted

Instrument aborted event.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "Aborted"	Name of the event

Details

This event is sent to a connected communication port if the KingFisher Presto is aborted for some other communication port. See command Abort. More specifically, see Abort for USB, Abort for LAN and Abort for RS232.

Examples

```
<Evt name="Aborted"/>
```

5.2 ChangeMagnets

KingFisher Presto is requesting a changing of the magnet heads.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String : "ChangeMagnets"	Name of the event
Evt@tips	String: Max 100 chars	Name of the tips for the magnet heads

Details

This event is sent by the KingFisher Presto instrument when a protocol with a multiple types of magnet heads is requiring a change of the magnet heads.

Note that the changing of the magnet heads is a manual operation.

Instrument continues the execution of the protocol after the command Acknowledge is sent by the controlling system.

Example

```
<Evt name="ChangeMagnets" tips="Tip 24 DW"/>
```

5.3 ChangePlate

KingFisher Presto is requesting a plate change.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "ChangePlate"	Name of the event
Evt@optional	Boolean string: "true" or "false"	If true, then the plate change is possible but not required. If false, then the instrument will not continue operation untill the plate is changed.
Evt/Evt[1]@name	String: "RemovePlate"	Name of the wrapped event
Evt/Evt[1]@plate	String: Max 100 chars	Plate name
Evt/Evt[1]@optional	Boolean string: "true" or "false"	See Evt/@optional
Evt/Evt[2]@name	String: "LoadPlate"	Name of the wrapped event
Evt/Evt[2]@plate	String: Max 100 chars	Plate name
Evt/Evt[2]@optional	Boolean string: "true" or "false"	See Evt/@optional

Details

The instrument sends this event to a controlling system, when a plate can be changed in the load position. Event parameter "optional" may be set to "true" to indicate that the plate change is not yet obligatory. The controlling system can perform the change operation at this point and notify the instrument using Acknowledge command. The protocol execution time can be minimized by changing the plate immediately after this first event, but the controlling system can also choose to wait when the plate change is actually needed to be done.

If the event with the "optional" parameter is neglected and not acknowledged by the controlling system, then the instrument will send another event with the parameter "optional" set to "false". Now the controlling system must change the plate and notify the instrument using Acknowledge command so that the protocol under execution can continue.

The change plate event is structured in the way, that it contains RemovePlate and LoadPlate events. Note that only one Acknowledge command is required though.

Example

```
<Evt name="ChangePlate" optional="true">
    <Evt name="RemovePlate" plate="Plate 1" optional="true"/>
    <Evt name="LoadPlate" plate="Plate 2" optional="true"/>
</Evt>

<Evt name="ChangePlate" optional="false">
    <Evt name="RemovePlate" plate="Plate 1" optional="false"/>
    <Evt name="LoadPlate" plate="Plate 2" optional="false"/>
</Evt>
```

5.4 Error

Spontaneous error message from the KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "Error"	Name of the event
Evt/Error@code	uint16_t: see Error Codes	Error code number
Evt/Error/text()	String: Max 100 characters	Description of the error

Details

This event is sent for example after failed turntable rotation, see command Rotate. The instrument will transition to an error state and an ErrorAcknowledge command must be send in order to clear the error.

Examples

```
<Evt name="Error">
    <Error code="5">Turntable position error.</Error>
</Evt>
```

5.5 LoadPlate

KingFisher Presto is requesting a plate to be inserted to the load position.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String : "LoadPlate"	Name of the event
Evt@plate	String: Max 100 chars	Plate name
Evt@optional	Boolean string: "true" or "false"	If true, then the loading of the plate is possible but not required. If false, then the instrument will not continue operation untill the plate is loaded.

Details

When the instrument is ready for a new plate to be loaded during protocol execution, then it sends this event to the controlling system. Event parameter "optional" may be set to "true" to indicate that the plate can be loaded but it is not yet needed. The controlling system can perform the load process at this point and notify the instrument using Acknowledge command. The controlling system can minimize the protocol execution time by loading the plate immediately after this first event, but it can also choose to wait when the plate is actually required.

If the event with the "optional" parameter is neglected and not acknowledged by the controlling system, then the instrument will send another event with the parameter "optional" set to "false". Now the controlling system must load the plate and notify the instrument using Acknowledge command so that the protocol under execution can continue.

Example

```
<Evt name="LoadPlate" plate="Plate 1" optional="true"/>

<Evt name="LoadPlate" plate="Plate 1" optional="false"/>
```

5.6 Pause

Event from a Pause step. User is requested to perform actions to a plate.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "Pause"	Name of the event
Evt@plate	String: Max 100 chars	Plate name
Evt/Message[text()]	String: Max ??? chars	Message for the user

Details

Instrument must be notified using Acknowledge command so that the pause step can continue.

Example

```
<Evt name="Pause" plate="Plate 1" >
    <Message>Dispense 10 µl of ethanol to each well.</Message>
</Evt
```

5.7 ProtocolTimeLeft

Spontaneous time left event from a KingFisher Presto protocol or single step execution.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "ProtocolTimeLeft"	Name of the event
Evt/TimeToPause@value	XML Duration data type	Time to next plate load/change/remove or pause event
Evt/TimeLeft@value	XML Duration data type	Total time left of the protocol/step execution

Details

This event is sent automatically after every StepStarted event. See command GetProtocolTimeLeft for more details.

Note! Element **TimeToPause** is not included when running single step execution. Note also that then the value of **TimeLeft** refers to the total execution time of a step being executed.

Examples

```
<Evt name="ProtocolTimeLeft">
    <TimeToPause value="PT0S"/>
    <TimeLeft value="PT30M53S"/>
</Evt>
```

```
<Evt name="ProtocolTimeLeft">
    <TimeLeft value="PT2M42S"/>
</Evt>
```

5.8 Ready

Step or protocol execution completed.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "Ready"	Name of the event

Details

This event is sent by the instrument after execution of a step or a protocol started with command StartProtocol. See also Rotate.

Examples

```
<Evt name="Ready"/>
```

5.9 RemovePlate

KingFisher Presto is requesting a plate to be removed from the load position.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "RemovePlate"	Name of the event
Evt@plate	String: Max 100 chars	Plate name
Evt@optional	Boolean string: "true" or "false"	If true, then the removing of the plate is possible but not required. If false, then the instrument will not continue operation untill the plate is removed.

Details

The instrument sends this event to a controlling system, when a plate can be removed from the load position. Event parameter "optional" may be set to "true" to indicate that the plate can be removed but it is not obligatory. The controlling system can perform the removal operation at this point and notify the instrument using Acknowledge command. The protocol execution time can be minimized by removing the plate immediately after this first event, but the controlling system can also choose to wait when the plate is actually needed to be removed.

If the event with the "optional" parameter is neglected and not acknowledged by the controlling system, then the instrument will send another event with the parameter "optional" set to "false". Now the controlling system must remove the plate and notify the instrument using Acknowledge command so that the protocol under execution can continue.

Example

```
<Evt name="RemovePlate" plate="Plate 1" optional="true"/>

<Evt name="RemovePlate" plate="Plate 1" optional="false"/>
```

5.10 StepStarted

Step start notification from the KingFisher Presto instrument.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "StepStarted"	Name of the event
Evt/Step@name	String	Name of the step
Evt/Step@pseudo	Boolean string: "true" or "false"	If this attribute is "false", then the step is a normal BindIt Protocol step e.q. Mix, Dry, Pause, Collect or Release. If attribute is "true" then the step is a fixed pseudo step like Init, Pick-Up, Leave and Unload.
Evt/Step@tip	String	Name of the tip goup under the step is in the BindIt protocol
Evt/Step@plate	String	Name of a plate used in the step

Details

This event in sent when a protocol step is started. See also command StartProtocol.

Examples

```
<Evt name="StepStarted">
    <Step name="Init" pseudo="true"/>
</Evt>

<Evt name="StepStarted">
    <Step name="Pick-Up" pseudo="true" tip="Tip1" plate="Tip Comb" />
</Evt>

<Evt name="StepStarted">
    <Step name="Mix1" pseudo="false" tip="Tip1" plate="Plate 1"/>
</Evt>

<Evt name="StepStarted">
    <Step name="Unload" pseudo="true" tip="Tip1" />
</Evt>
```

5.11 Temperature

Temperature measurements event.

Syntax

Tag / Attribute	Data type and range/limits	Description
Evt@name	String: "Temperature"	Name of the event
Evt/Ambient@value	float	Ambient temperature in Celciuss degrees
Evt/Heater@value	float	Heater block temperature in Celciuss degrees

Details

This event reports ambient and heater block temperatures. See command SetTemperatureReporting.

Examples

```
<Evt name="Temperature">
   <Ambient value="22.1"/>
   <Heater value="37.7"/>
</Evt>
```

Chapter 6

Appendix - XML path syntax

Syntaxes of Events, Commands and Responses are documented using a style similar to the `Xpath` syntax.

Here is a few examples of the usage. Note that also the actual syntaxes are explained using detailed examples.

Example XML block:

```
<T1>
    <T2>
        <T3 a1="123">
            This is a sentence.
        </T3>
        <T4>
            <T5 a2="321"/>
            <T5 a2="432">
            <T5 a2="543">
            <T5 a2="654">
        </T4>
    </T2>
    <![CDATA[ABCDEFG]]>
</T1>
```

Path to the T3:

```
T1/T2/T3
```

Path to the text of the tag T3:

```
T1/T2/T3/text()
```

Path to the attribute ”a1” of the tag T3:

```
T1/T2/T3@@a1
```

Path to the attribute ”a2” of the third T5 tag with a value of ”543”

```
T1/T2/T4/T5[3]@@a2
```

Path to the CDATA of the tag T1

```
Note that XPath does not support addressing a CDATA element, we’ll use this syntax:

T1[CDATA]
```

Thermo Scientific

KFModule.dll

Interface Specification

Thermo SCIENTIFIC

Contents

1 KFModuleDll Interface Specification 1

2 Using KFModuleDll 3
2.1 About . . . 3
2.2 Exported functions . . . 4
2.2.1 KFModule_OpenUsb . . . 5
2.2.2 KFModule_OpenSerial . . . 6
2.2.3 KFModule_OpenLan . . . 7
2.2.4 KFModule_ListLanDevices . . . 8
2.2.5 KFModule_OpenSimulator . . . 9
2.2.6 KFModule_Close . . . 10
2.2.7 KFModule_Connect . . . 11
2.2.8 KFModule_Disconnect . . . 12
2.2.9 KFModule_AttachEvent . . . 13
2.2.10 KFModule_AttachMsg . . . 14
2.2.11 KFModule_AttachCallback . . . 15
2.2.12 KFModule_Send . . . 16
2.2.13 KFModule_Abort . . . 17
2.2.14 KFModule_ReadReceived . . . 18
2.2.15 KFModule_ReadResponse . . . 19
2.2.16 KFModule_ReadEvent . . . 20
2.2.17 KFModule_UploadProtocol . . . 21
2.2.18 KFModule_DownloadProtocol . . . 22
2.2.19 KFModule_GetError . . . 23

3 File Index 25
3.1 File List . . . 25

4 File Documentation 27
4.1 api.h File Reference . . . 27
4.2 KFModuleDll.c File Reference . . . 27
4.2.1 Detailed Description . . . 28
4.2.2 Function Documentation . . . 28
4.2.2.1 KFModule_Abort . . . 28
4.2.2.2 KFModule_AttachCallback . . . 28
4.2.2.3 KFModule_AttachEvent . . . 29
4.2.2.4 KFModule_AttachMsg . . . 29
4.2.2.5 KFModule_Close . . . 29
4.2.2.6 KFModule_Connect . . . 30
4.2.2.7 KFModule_Disconnect . . . 30
4.2.2.8 KFModule_DownloadProtocol . . . 30
4.2.2.9 KFModule_GetError . . . 31
4.2.2.10 KFModule_ListLanDevices . . . 31
4.2.2.11 KFModule_OpenLan . . . 32
4.2.2.12 KFModule_OpenSerial . . . 32
4.2.2.13 KFModule_OpenSimulator . . . 33

4.2.2.14 KFModule_OpenUsb . . . 33
4.2.2.15 KFModule_ReadEvent . . . 34
4.2.2.16 KFModule_ReadReceived . . . 34
4.2.2.17 KFModule_ReadResponse . . . 35
4.2.2.18 KFModule_Send . . . 35
4.2.2.19 KFModule_UploadProtocol . . . 36
4.3 KFModuleDll.h File Reference . . . 36
4.3.1 Detailed Description . . . 38
4.3.2 Macro Definition Documentation . . . 38
4.3.2.1 KF_PRESTO_PID . . . 38
4.3.2.2 KFMODULE_ERROR . . . 38
4.3.2.3 KFMODULE_EVENT . . . 38
4.3.2.4 KFMODULE_RECEIVE . . . 38
4.3.2.5 KFMODULE_RESPONSE . . . 38
4.3.2.6 KFMODULE_TRANSMIT . . . 39
4.3.3 Enumeration Type Documentation . . . 39
4.3.3.1 KFM_ERROR . . . 39
4.3.3.2 KFM_SIMULATOR_PORT . . . 39
4.3.4 Function Documentation . . . 39
4.3.4.1 KFModule_Abort . . . 39
4.3.4.2 KFModule_AttachCallback . . . 40
4.3.4.3 KFModule_AttachEvent . . . 40
4.3.4.4 KFModule_AttachMsg . . . 40
4.3.4.5 KFModule_Close . . . 41
4.3.4.6 KFModule_Connect . . . 41
4.3.4.7 KFModule_Disconnect . . . 41
4.3.4.8 KFModule_DownloadProtocol . . . 42
4.3.4.9 KFModule_GetError . . . 42
4.3.4.10 KFModule_ListLanDevices . . . 42
4.3.4.11 KFModule_OpenLan . . . 43
4.3.4.12 KFModule_OpenSerial . . . 43
4.3.4.13 KFModule_OpenSimulator . . . 44
4.3.4.14 KFModule_OpenUsb . . . 44
4.3.4.15 KFModule_ReadEvent . . . 45
4.3.4.16 KFModule_ReadReceived . . . 45
4.3.4.17 KFModule_ReadResponse . . . 46
4.3.4.18 KFModule_Send . . . 47
4.3.4.19 KFModule_UploadProtocol . . . 47
4.4 mainpage.h File Reference . . . 48

Index 49

Chapter 1

KFModuleDll Interface Specification

About

The purpose of the KFModule.dll dynamic link library is to make interface to a KFModule instrument easy. For communication with the instrument the KFModule.dll uses another DLLs: ThermoUSB.dll, ThermoLAN.dll and ThermoCOM.dll. The user of the KFModule.dll does not have to know anything about the actual hardware port. Just use the functions provided by KFModule.dll.

Confidential

Chapter 2

Using KFModuleDll

2.1 About

Because it is not trivial to write PC software from scratch to communicate with a KFModule instrument, two Dynamic Link Libraries are provided which hide much of the complexity of the communication.

The first one is a DLL for the actual HW interface: ThermoUSB.dll, ThermoLAN.dll or ThermoCOM.dll. These DLLs are generic libraries for communicating with several different Thermo Scientific microplate instruments.

The second one is KFModule.dll, which uses DLLs mentioned above to communicate with the KFModule instrument.

To communicate with a KFModule instrument you use the exported functions of KFModule.dll. No knowledge of the other DLLs is required, they only need to be in a folder where Windows can find them (probably the same place where the KFModule.dll is loaded from).

Depending on your project setup, you may find useful a couple of other files which are also provided. The header file KFModuleDll.h contains the prototypes of the exported functions and definitions of constant values used by the dll. File KFModule.lib contains information about the dll the linker uses to add references to the library in the executable. This way the dll is automatically loaded and the exported functions of the library can be called as easy as the functions in the code using the dll.

2.2 Exported functions

- KFModule_OpenUsb
- KFModule_OpenSerial
- KFModule_OpenLan
- KFModule_ListLanDevices
- KFModule_OpenSimulator
- KFModule_Close
- KFModule_Connect
- KFModule_Disconnect
- KFModule_AttachEvent
- KFModule_AttachMsg
- KFModule_AttachCallback
- KFModule_Send
- KFModule_Abort
- KFModule_ReadReceived
- KFModule_ReadResponse
- KFModule_ReadEvent
- KFModule_UploadProtocol
- KFModule_DownloadProtocol
- KFModule_GetError

2.2.1 KFModule_OpenUsb

Open an USB communication channel to a KFModule instrument. Full declaration: KFModule_OpenUsb().

Parameters

SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first device with matching VendorID and ProductID.
productId	Manufacturer product id number of the USB device.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per device is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

2.2.2 KFModule_OpenSerial

Open serial communication port to a KFModule instrument. Full declaration: KFModule_OpenSerial().

Parameters

port	Serial port number.
baud	Serial port baudrate.
DeviceName	Pointer to the device name string of the device. The device must report an identical name to the VER command for the connection to succeed. This parameter may be NULL, in which case the device name is ignored.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number to the VER command for the connection to succeed. This parameter may be NULL, in which case the serial number is ignored.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per serial port is allowed. If both DeviceName and SerialNumber are NULL, no VER command is sent to the instrument.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

2.2.3 KFModule_OpenLan

Open a LAN communication channel to a KFModule instrument. Full declaration: KFModule_OpenLan().

Parameters

instrumentName	Name of the instrument. This must match the device name the instrument uses for matching a WS--Discovery Probe and Resolve.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first found KFModule instrument.
timeout	Timeout which is used to search the instrument. If 0 is given then WS-Discovery will use default 4 sec timeout.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per device is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

2.2.4 KFModule_ListLanDevices

List instruments found on the LAN. Full declaration: KFModule_ListLanDevices().

Parameters

instrumentName	Name of the instrument. Only instruments with a matching name are listed. May be NULL, in which case the found devices are not filtered by the name.
SerialNumber	The serial number of the instrument. Only instruments with a matching serial number are listed. May be NULL, in which case the found devices are not filtered by the serial number.
buf	The buffer to list the found devices to.
bufsize	Size of the buffer in bytes.

Returns

The size in bytes of the complete list of found devices. If the returned value is equal or smaller than the given buffer size, the buffer contains the complete list of devices found on the LAN. If the returned value is higher than the given buffer size, no data is returned and the caller must call the function again with big enough buffer. Return value 0 means that no instruments were found.

On success, the caller's buffer contains zero terminated strings with a combined length of the return value. The string terminating zeros are included in the length.

For each found instrument, the first string is the instrument IPv4 address and the TCP port number it is listening, enclosed in square brackets, e.g. [10.32.196.210:49536].

The IP address string is followed by the WS-Discovery match strings, usually 3 of them. The first one is always 'Thermo-Device', the second one is the instrument name and the third one the instrument serial number string.

If the match strings are followed by a string in angle brackets, e.g. <10.32.196.154:57403>, it means that the instrument is currently connected that IP address and TCP port, and trying to connect to that instrument with function KFModule_-OpenLan() will fail. If there is no string in square brackets, the instrument will accept a connection.

2.2.5 KFModule_OpenSimulator

Open communication channel to KFModule simulator. Full declaration: KFModule_OpenSimulator().

Parameters

port	The simulator port to connect to, one of KFM_SIMULATOR_USB, KFM_SIMULATOR_COM, KFM_SIMULATOR_DBG or KFM_SIMULATOR_LAN.
productId	(USB) product id of the instrument.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per simulator communication channel is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

2.2.6 KFModule_Close

Close a communication channel. Full declaration: KFModule_Close().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function should be called when the communication channel is no longer needed. Failing to close the channel prevents new connections to the channel as long as the dll stays loaded.

Note. When using the XML interface, function KFModule_Disconnect() must be called before this function in order to disconnect the instrument from the open communication channel. Otherwise the instrument may refuse new connections.

2.2.7 KFModule_Connect

Connect to the instrument. Full declaration: KFModule_Connect().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function sends the "Connect" command with a current date/time setting to the instrument. It is recommended to use this function to connect to the instrument so that the calendar of the instrument is always set to the local time automatically.

Note. This funtion returns before the actual connection is done, meaning that the return value indicates merely if the starting of the command is succesfull of not. The instrument will send a separate response after the connection is finished. See interface specification of the instrument in question for more details.

2.2.8 KFModule_Disconnect

Disconnect the instrument. Full declaration: KFModule_Disconnect().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function sends the "disconnect" command to the instrument. This is only needed because of the symmetry in the list of API functions. Disconnect() is the pair to Connect() function.

Note. This funtion returns before the actual connection is closed, meaning that the return value indicates merely if the starting of the command is succesfull of not. The instrument will send a separate response just before the disconnection. See interface specification of the instrument in question for more details.

2.2.9 KFModule_AttachEvent

Attach an event object to a KFModule connection. Full declaration: KFModule_AttachEvent().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
ev	Event(s) to signal.
object	Handle of an event object.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Parameter 'ev' may be any combination of KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT and KFMODULE_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event. If the same event object is used for more than one event, the application must check for all possible events after the event object is signalled.

2.2.10 KFModule_AttachMsg

Attach an event message to the KFModule connection. Full declaration: KFModule_AttachMsg().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
evCode	Event(s) for which a message is sent.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT and KFMODULE_ERROR. The wParam of the sent message will be one of these event codes.

2.2.11 KFModule_AttachCallback

Attach an event callback function to the KFModule connection. Full declaration: KFModule_AttachCallback().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
callback	Address of the callback function.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The callback function will be called whenever any of events KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT or KFMODULE_ERROR occurs. The connection handle and the event code are passed as parameters to the function.

2.2.12 KFModule_Send

Send a NUL terminated ASCII string to the instrument. Full declaration: KFModule_Send().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	The NUL terminated string to send.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Use this function to send commands and acknowledges to the instrument. For uploading and downloading a protocol there are dedicated functions KFModule_UploadProtocol() and KFModule_DownloadProtocol(). Also connecting/disconnecting is recommended to be done with separate KFModule_Connect() and KFModule_Disconnect() functions.

The data to be sent to the instrument is buffered in the dll. A KFMODULE_TRANSMIT event is sent to the application when all data is sent. You may call this function repeatedly without waiting for the event, but a KFM_ERROR_OUT_OF_HEAP error may be returned if you send a lot of data without waiting for the event.

Note. Remember to terminate commands with a new line character (ASCII 10). See interface specification of the instrument in question for more details.

2.2.13 KFModule_Abort

Send Abort command to the instrument. Full declaration: KFModule_Abort().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Aborts any command(s) being executed in the instrument.

2.2.14 KFModule_ReadReceived

Read a received data line from the received data chain. Full declaration: KFModule_ReadReceived().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the line.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_RECEIVE event. The received data is neither a XML Response nor a XML Event. An empty string will be copied to caller's buffer when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent strings from the instrument may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_RECEIVE( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadReceived( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

2.2.15 KFModule_ReadResponse

Read a received XML response from the response chain. Full declaration: KFModule_ReadResponse().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the response.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_RESPONSE event. The received data is a XML Response. The event is not sent until a whole XML Response is received. If the caller's buffer is not large enough to hold the whole response, a partial response is copied. An empty string is copied when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent responses may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_RESPONSE( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadResponse( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

2.2.16 KFModule_ReadEvent

Read a received XML event from the event chain. Full declaration: KFModule_ReadEvent().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the event.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_EVENT event. The received data is a XML Event. The event is not sent until a whole XML Event is received. If the caller's buffer is not large enough to hold the whole Event, a partial Event is copied. An empty string is copied when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent events from the instrument may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_EVENT( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadEvent( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

2.2.17 KFModule_UploadProtocol

Export the requested protocol to the instrument. Full declaration: KFModule_UploadProtocol().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
protocolName	Name of the protocol to export.
path	Complete path of the .bdz file containing the protocol to export.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The given protocol name must match exactly with a protocol name stored in the .bdz file. The .bdz file may contain several protocols, but only the requested protocol is sent to the instrument.

Note. This funtion returns before the actual transfer is completed, meaning that the return value indicates merely if the starting of the transfer is succesfull of not. The instrument will send a separate response after the transfer. See interface specification of the instrument in question for more details.

2.2.18 KFModule_DownloadProtocol

Import the requested protocol to a .bdz file. Full declaration: KFModule_DownloadProtocol().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
protocolName	Name of the protocol to export.
path	Complete path of the .bdz file to receive the protocol.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The given protocol name must match exactly with a protocol name stored in the instrument. The dll must have write access to the given .bdz file. The file will be overwritten by the dll.

Note. This funtion returns before the actual transfer is completed, meaning that the return value indicates merely if the starting of the transfer is succesfull of not. The instrument will send a separate response after the transfer. See interface specification of the instrument in question for more details.

2.2.19 KFModule_GetError

Get the asynchronous error code. Full declaration: KFModule_GetError().

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Most dll functions return an error code, but an error may occur asynchronously in the receive or transmit thread of the dll. This kind of errors are reported with the KFMODULE_ERROR event. The application should then call this function to read the error code and take action. The error usually is KFM_ERROR_DISCONNECTED, in which case the application should close the current connection and try to open a new one.

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

api.h
: Part of KFModuleDll documentation 27

KFModuleDll.c
: Interface to a Thermo Fisher Scientific KingFisher instrument that contains KingFisher module and uses ThermoUSB.dll, ThermoLAN.dll or ThermoCOM.dll 27

KFModuleDll.h
: Functions exported from KFModule.dll 36

mainpage.h
: KFModuleDll Interface Specification 48

Chapter 4

File Documentation

4.1 api.h File Reference

Part of KFModuleDll documentation.

4.2 KFModuleDll.c File Reference

Interface to a Thermo Fisher Scientific KingFisher instrument that contains KingFisher module and uses ThermoUSB.dll, ThermoLAN.dll or ThermoCOM.dll.

Functions

- HANDLE WINAPI KFModule_OpenUsb (LPCSTR SerialNumber, WORD productId)

  Open an USB communication channel to a KFModule instrument.
- HANDLE WINAPI KFModule_OpenSerial (WORD port, DWORD baud, LPCSTR DeviceName, LPCSTR SerialNumber)

  Open serial communication port to a KFModule instrument.
- HANDLE WINAPI KFModule_OpenLan (LPCSTR DeviceName, LPCSTR SerialNumber, UINT timeout)

  Open a LAN communication channel to a KFModule instrument.
- DWORD WINAPI KFModule_ListLanDevices (LPCSTR DeviceName, LPCSTR SerialNumber, LPSTR buf, DWORD bufsize)

  List instruments found on the LAN.
- HANDLE WINAPI KFModule_OpenSimulator (KFM_SIMULATOR_PORT port, WORD productId)

  Open communication channel to KFModule simulator.
- KFM_ERROR WINAPI KFModule_Close (HANDLE hConn)

  Close a communication channel.
- DllExport KFM_ERROR WINAPI KFModule_Connect (HANDLE hConn)

  Connect to the instrument.
- DllExport KFM_ERROR WINAPI KFModule_Disconnect (HANDLE hConn)

  Disconnect the instrument.
- KFM_ERROR WINAPI KFModule_ReadReceived (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received data line from the received data chain.
- KFM_ERROR WINAPI KFModule_ReadResponse (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received XML response from the response chain.
- KFM_ERROR WINAPI KFModule_ReadEvent (HANDLE hConn, LPSTR buf, DWORD bufsize)

Read a received XML event from the event chain.

- KFM_ERROR WINAPI KFModule_AttachEvent (HANDLE hConn, UINT ev, HANDLE object)

  Attach an event object to a KFModule connection.
- KFM_ERROR WINAPI KFModule_AttachMsg (HANDLE hConn, HANDLE hWnd, UINT msg, UINT evCode)

  Attach an event message to the KFModule connection.
- KFM_ERROR WINAPI KFModule_AttachCallback (HANDLE hConn, void(∗callback)(HANDLE conn, UINT ev))

  Attach an event callback function to the KFModule connection.
- KFM_ERROR WINAPI KFModule_Send (HANDLE hConn, LPCSTR buf)

  Send a NUL terminated ASCII string to the instrument.
- KFM_ERROR WINAPI KFModule_Abort (HANDLE hConn)

  Send Abort command to the instrument.
- KFM_ERROR WINAPI KFModule_UploadProtocol (HANDLE hConn, LPCSTR protocolName, LPCSTR path)

  Export the requested protocol to the instrument.
- KFM_ERROR WINAPI KFModule_DownloadProtocol (HANDLE hConn, LPCSTR protocolName, LPCSTR path)

  Import the requested protocol to a .bdz file.
- KFM_ERROR WINAPI KFModule_GetError (HANDLE hConn)

  Get the asynchronous error code.

4.2.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

Definition in file KFModuleDll.c.

4.2.2 Function Documentation

4.2.2.1 KFM_ERROR WINAPI KFModule_Abort (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Aborts any command(s) being executed in the instrument.

Definition at line 2553 of file KFModuleDll.c.

4.2.2.2 KFM_ERROR WINAPI KFModule_AttachCallback (HANDLE *hConn,* void(∗)(HANDLE conn, UINT ev) *callback*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
callback	Address of the callback function.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The callback function will be called whenever any of events KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT or KFMODULE_ERROR occurs. The connection handle and the event code are

passed as parameters to the function.

Definition at line 2451 of file KFModuleDll.c.

4.2.2.3 KFM_ERROR WINAPI KFModule_AttachEvent (HANDLE *hConn,* UINT *ev,* HANDLE *object*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
ev	Event(s) to signal.
object	Handle of an event object.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Parameter 'ev' may be any combination of KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT and KFMODULE_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event. If the same event object is used for more than one event, the application must check for all possible events after the event object is signalled.

Definition at line 2362 of file KFModuleDll.c.

4.2.2.4 KFM_ERROR WINAPI KFModule_AttachMsg (HANDLE *hConn,* HANDLE *hWnd,* UINT *msg,* UINT *evCode*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
evCode	Event(s) for which a message is sent.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT and KFMODULE_ERROR. The wParam of the sent message will be one of these event codes.

Definition at line 2417 of file KFModuleDll.c.

4.2.2.5 KFM_ERROR WINAPI KFModule_Close (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function should be called when the communication channel is no longer needed. Failing to close the channel prevents new connections to the channel as long as the dll stays loaded.

Note. When using the XML interface, function KFModule_Disconnect() must be called before this function in order to disconnect the instrument from the open communication channel. Otherwise the instrument may refuse new connections.

Definition at line 2099 of file KFModuleDll.c.

4.2.2.6 DllExport KFM_ERROR WINAPI KFModule_Connect (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function sends the "Connect" command with a current date/time setting to the instrument. It is recommended to use this function to connect to the instrument so that the calendar of the instrument is always set to the local time automatically.

Note. This funtion returns before the actual connection is done, meaning that the return value indicates merely if the starting of the command is succesfull of not. The instrument will send a separate response after the connection is finished. See interface specification of the instrument in question for more details.

Definition at line 2133 of file KFModuleDll.c.

4.2.2.7 DllExport KFM_ERROR WINAPI KFModule_Disconnect (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function sends the "disconnect" command to the instrument. This is only needed because of the symmetry in the list of API functions. Disconnect() is the pair to Connect() function.

Note. This funtion returns before the actual connection is closed, meaning that the return value indicates merely if the starting of the command is succesfull of not. The instrument will send a separate response just before the disconnection. See interface specification of the instrument in question for more details.

Definition at line 2178 of file KFModuleDll.c.

4.2.2.8 KFM_ERROR WINAPI KFModule_DownloadProtocol (HANDLE *hConn,* LPCSTR *protocolName,* LPCSTR *path*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
protocolName	Name of the protocol to export.
path	Complete path of the .bdz file to receive the protocol.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The given protocol name must match exactly with a protocol name stored in the instrument. The dll must have write access to the given .bdz file. The file will be overwritten by the dll.

Note. This funtion returns before the actual transfer is completed, meaning that the return value indicates merely if the starting of the transfer is succesfull of not. The instrument will send a separate response after the transfer. See interface specification of the instrument in question for more details.

Definition at line 2627 of file KFModuleDll.c.

4.2.2.9 KFM_ERROR WINAPI KFModule_GetError (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Most dll functions return an error code, but an error may occur asynchronously in the receive or transmit thread of the dll. This kind of errors are reported with the KFMODULE_ERROR event. The application should then call this function to read the error code and take action. The error usually is KFM_ERROR_DISCONNECTED, in which case the application should close the current connection and try to open a new one.

Definition at line 2673 of file KFModuleDll.c.

4.2.2.10 DWORD WINAPI KFModule_ListLanDevices (LPCSTR *DeviceName,* LPCSTR *SerialNumber,* LPSTR *buf,* DWORD *bufsize*)

Parameters

instrumentName	Name of the instrument. Only instruments with a matching name are listed. May be NULL, in which case the found devices are not filtered by the name.
SerialNumber	The serial number of the instrument. Only instruments with a matching serial number are listed. May be NULL, in which case the found devices are not filtered by the serial number.
buf	The buffer to list the found devices to.
bufsize	Size of the buffer in bytes.

Returns

The size in bytes of the complete list of found devices. If the returned value is equal or smaller than the given buffer size, the buffer contains the complete list of devices found on the LAN. If the returned value is higher than the given buffer size, no data is returned and the caller must call the function again with big enough buffer. Return value 0 means that no instruments were found.

On success, the caller's buffer contains zero terminated strings with a combined length of the return value. The string terminating zeros are included in the length.

For each found instrument, the first string is the instrument IPv4 address and the TCP port number it is listening, enclosed in square brackets, e.g. [10.32.196.210:49536].

The IP address string is followed by the WS-Discovery match strings, usually 3 of them. The first one is always 'Thermo-Device', the second one is the instrument name and the third one the instrument serial number string.

If the match strings are followed by a string in angle brackets, e.g. <10.32.196.154:57403>, it means that the instrument is currently connected that IP address and TCP port, and trying to connect to that instrument with function KFModule_OpenLan() will fail. If there is no string in square brackets, the instrument will accept a connection.

Definition at line 2000 of file KFModuleDll.c.

4.2.2.11 HANDLE WINAPI KFModule_OpenLan (LPCSTR *DeviceName,* LPCSTR *SerialNumber,* UINT *timeout*)

Parameters

instrumentName	Name of the instrument. This must match the device name the instrument uses for matching a WS--Discovery Probe and Resolve.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first found KFModule instrument.
timeout	Timeout which is used to search the instrument. If 0 is given then WS-Discovery will use default 4 sec timeout.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per device is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 1935 of file KFModuleDll.c.

4.2.2.12 HANDLE WINAPI KFModule_OpenSerial (WORD *port,* DWORD *baud,* LPCSTR *DeviceName,* LPCSTR *SerialNumber*)

Parameters

port	Serial port number.
baud	Serial port baudrate.
DeviceName	Pointer to the device name string of the device. The device must report an identical name to the VER command for the connection to succeed. This parameter may be NULL, in which case the device name is ignored.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number to the VER command for the connection to succeed. This parameter may be NULL, in which case the serial number is ignored.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per serial port is allowed. If both DeviceName and SerialNumber are NULL, no VER command is sent to the instrument.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 1880 of file KFModuleDll.c.

4.2.2.13 HANDLE WINAPI KFModule_OpenSimulator (KFM_SIMULATOR_PORT *port,* WORD *productId*)

Parameters

port	The simulator port to connect to, one of KFM_SIMULATOR_USB, KFM_SIMULATOR_COM, KFM_SIMULATOR_DBG or KFM_SIMULATOR_LAN.
productId	(USB) product id of the instrument.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per simulator communication channel is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 2030 of file KFModuleDll.c.

4.2.2.14 HANDLE WINAPI KFModule_OpenUsb (LPCSTR *SerialNumber,* WORD *productId*)

Parameters

SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first device with matching VendorID and ProductID.
productId	Manufacturer product id number of the USB device.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per device is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 1827 of file KFModuleDll.c.

4.2.2.15 KFM_ERROR WINAPI KFModule_ReadEvent (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the event.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_EVENT event. The received data is a XML Event. The event is not sent until a whole XML Event is received. If the caller's buffer is not large enough to hold the whole Event, a partial Event is copied. An empty string is copied when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent events from the instrument may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_EVENT( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadEvent( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

Definition at line 2333 of file KFModuleDll.c.

4.2.2.16 KFM_ERROR WINAPI KFModule_ReadReceived (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the line.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_RECEIVE event. The received data is neither a XML Response nor a XML Event. An empty string will be copied to caller's buffer when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent strings from the instrument may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_RECEIVE( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadReceived( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

Definition at line 2239 of file KFModuleDll.c.

4.2.2.17 KFM_ERROR WINAPI KFModule_ReadResponse (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the response.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_RESPONSE event. The received data is a XML Response. The event is not sent until a whole XML Response is received. If the caller's buffer is not large enough to hold the whole response, a partial response is copied. An empty string is copied when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent responses may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_RESPONSE( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadResponse( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

Definition at line 2286 of file KFModuleDll.c.

4.2.2.18 KFM_ERROR WINAPI KFModule_Send (HANDLE *hConn,* LPCSTR *buf*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	The NUL terminated string to send.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Use this function to send commands and acknowledges to the instrument. For uploading and downloading a protocol there are dedicated functions KFModule_UploadProtocol() and KFModule_DownloadProtocol(). Also connecting/disconnecting is recommended to be done with separate KFModule_Connect() and KFModule_Disconnect() functions.

The data to be sent to the instrument is buffered in the dll. A KFMODULE_TRANSMIT event is sent to the application when all data is sent. You may call this function repeatedly without waiting for the event, but a KFM_ERROR_OUT_OF_HEAP error may be returned if you send a lot of data without waiting for the event.

Note. Remember to terminate commands with a new line character (ASCII 10). See interface specification of the instrument in question for more details.

Definition at line 2495 of file KFModuleDll.c.

4.2.2.19 KFM_ERROR WINAPI KFModule_UploadProtocol (HANDLE *hConn*, LPCSTR *protocolName*, LPCSTR *path*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
protocolName	Name of the protocol to export.
path	Complete path of the .bdz file containing the protocol to export.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The given protocol name must match exactly with a protocol name stored in the .bdz file. The .bdz file may contain several protocols, but only the requested protocol is sent to the instrument.

Note. This funtion returns before the actual transfer is completed, meaning that the return value indicates merely if the starting of the transfer is succesfull of not. The instrument will send a separate response after the transfer. See interface specification of the instrument in question for more details.

Definition at line 2600 of file KFModuleDll.c.

4.3 KFModuleDll.h File Reference

Functions exported from KFModule.dll.

Macros

- #define KFMODULE_RECEIVE 1

  Data line received.
- #define KFMODULE_TRANSMIT 2

  All data transmitted.
- #define KFMODULE_RESPONSE 4

  A response received from KFModule.

- #define KFMODULE_EVENT 8

  An event received from KFModule.
- #define KFMODULE_ERROR 16

  Fatal error.
- #define KF_PRESTO_PID 713

  Thermo Fisher Scientific KingFisher Presto product id.

Enumerations

- enum KFM_ERROR {
  KFM_ERROR_SUCCESS, KFM_ERROR_NO_DATA, KFM_ERROR_INVALID_HANDLE, KFM_ERROR_INVALID_ARGUMENT,
  KFM_ERROR_DISCONNECTED, KFM_ERROR_FILE_NOT_FOUND, KFM_ERROR_INVALID_BDZ_FILE, KFM_ERROR_PROTOCOL_NOT_FOUND,
  KFM_ERROR_TEMP_FILE_CREATE, KFM_ERROR_EXPORT_IMPORT_BUSY, KFM_ERROR_OUT_OF_HEAP
  }

  Error codes returned by KFModuleDll functions.
- enum KFM_SIMULATOR_PORT { KFM_SIMULATOR_USB, KFM_SIMULATOR_COM, KFM_SIMULATOR_DBG, KFM_SIMULATOR_LAN }

  KFModule simulator ports.

Functions

- DllExport HANDLE WINAPI KFModule_OpenUsb (LPCSTR SerialNumber, WORD productId)

  Open an USB communication channel to a KFModule instrument.
- DllExport HANDLE WINAPI KFModule_OpenSerial (WORD port, DWORD baud, LPCSTR DeviceName, LPCSTR SerialNumber)

  Open serial communication port to a KFModule instrument.
- DllExport HANDLE WINAPI KFModule_OpenLan (LPCSTR DeviceName, LPCSTR SerialNumber, UINT timeout)

  Open a LAN communication channel to a KFModule instrument.
- DllExport DWORD WINAPI KFModule_ListLanDevices (LPCSTR DeviceName, LPCSTR SerialNumber, LPSTR buf, DWORD bufsize)

  List instruments found on the LAN.
- DllExport HANDLE WINAPI KFModule_OpenSimulator (KFM_SIMULATOR_PORT port, WORD productId)

  Open communication channel to KFModule simulator.
- DllExport KFM_ERROR WINAPI KFModule_Close (HANDLE hConn)

  Close a communication channel.
- DllExport KFM_ERROR WINAPI KFModule_Connect (HANDLE hConn)

  Connect to the instrument.
- DllExport KFM_ERROR WINAPI KFModule_Disconnect (HANDLE hConn)

  Disconnect the instrument.
- DllExport KFM_ERROR WINAPI KFModule_AttachEvent (HANDLE hConn, UINT ev, HANDLE object)

  Attach an event object to a KFModule connection.
- DllExport KFM_ERROR WINAPI KFModule_AttachMsg (HANDLE hConn, HANDLE hWnd, UINT msg, UINT ev)

  Attach an event message to the KFModule connection.
- DllExport KFM_ERROR WINAPI KFModule_AttachCallback (HANDLE hConn, void(∗callback)(HANDLE conn, UINT ev))

  Attach an event callback function to the KFModule connection.
- DllExport KFM_ERROR WINAPI KFModule_Send (HANDLE hConn, LPCSTR buf)

  Send a NUL terminated ASCII string to the instrument.

- DllExport KFM_ERROR WINAPI KFModule_Abort (HANDLE hConn)

  Send Abort command to the instrument.
- DllExport KFM_ERROR WINAPI KFModule_ReadReceived (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received data line from the received data chain.
- DllExport KFM_ERROR WINAPI KFModule_ReadResponse (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received XML response from the response chain.
- DllExport KFM_ERROR WINAPI KFModule_ReadEvent (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received XML event from the event chain.
- DllExport KFM_ERROR WINAPI KFModule_UploadProtocol (HANDLE hConn, LPCSTR protocolName, LPCSTR path)

  Export the requested protocol to the instrument.
- DllExport KFM_ERROR WINAPI KFModule_DownloadProtocol (HANDLE hConn, LPCSTR protocolName, LPCSTR path)

  Import the requested protocol to a .bdz file.
- DllExport KFM_ERROR WINAPI KFModule_GetError (HANDLE hConn)

  Get the asynchronous error code.

4.3.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2016

Interface to a KingFisher module. Following instruments uses this interface:

- KingFisher Presto

Definition in file KFModuleDll.h.

4.3.2 Macro Definition Documentation

4.3.2.1 #define KF_PRESTO_PID 713

Definition at line 65 of file KFModuleDll.h.

4.3.2.2 #define KFMODULE_ERROR 16

Definition at line 28 of file KFModuleDll.h.

4.3.2.3 #define KFMODULE_EVENT 8

Definition at line 27 of file KFModuleDll.h.

4.3.2.4 #define KFMODULE_RECEIVE 1

Definition at line 24 of file KFModuleDll.h.

4.3.2.5 #define KFMODULE_RESPONSE 4

Definition at line 26 of file KFModuleDll.h.

4.3.2.6 #define KFMODULE_TRANSMIT 2

Definition at line 25 of file KFModuleDll.h.

4.3.3 Enumeration Type Documentation

4.3.3.1 enum KFM_ERROR

These are the error codes returned by the API functions.

Enumerator

KFM_ERROR_SUCCESS No error.

KFM_ERROR_NO_DATA No more data to return.

KFM_ERROR_INVALID_HANDLE The connection handle is invalid.

KFM_ERROR_INVALID_ARGUMENT Invalid function argument.

KFM_ERROR_DISCONNECTED The instrument is disconnected.

KFM_ERROR_FILE_NOT_FOUND Protocol input or output file not found.

KFM_ERROR_INVALID_BDZ_FILE The given file is not a valid bdz file.

KFM_ERROR_PROTOCOL_NOT_FOUND Requested protocol not found in the given bdz file.

KFM_ERROR_TEMP_FILE_CREATE Failed to create a temporary file.

KFM_ERROR_EXPORT_IMPORT_BUSY Can only execute one Export or Import at a time.

KFM_ERROR_OUT_OF_HEAP Out of heap memory.

Definition at line 34 of file KFModuleDll.h.

4.3.3.2 enum KFM_SIMULATOR_PORT

Enumerator

KFM_SIMULATOR_USB USB port.

KFM_SIMULATOR_COM Serial command port.

KFM_SIMULATOR_DBG Serial debug port.

KFM_SIMULATOR_LAN LAN port.

Definition at line 50 of file KFModuleDll.h.

4.3.4 Function Documentation

4.3.4.1 DllExport KFM_ERROR WINAPI KFModule_Abort (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Aborts any command(s) being executed in the instrument.

Definition at line 2553 of file KFModuleDll.c.

4.3.4.2 DllExport KFM_ERROR WINAPI KFModule_AttachCallback (HANDLE *hConn,* void(∗)(HANDLE conn, UINT ev) *callback*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
callback	Address of the callback function.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The callback function will be called whenever any of events KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT or KFMODULE_ERROR occurs. The connection handle and the event code are passed as parameters to the function.

Definition at line 2451 of file KFModuleDll.c.

4.3.4.3 DllExport KFM_ERROR WINAPI KFModule_AttachEvent (HANDLE *hConn,* UINT *ev,* HANDLE *object*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
ev	Event(s) to signal.
object	Handle of an event object.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Parameter 'ev' may be any combination of KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT and KFMODULE_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event. If the same event object is used for more than one event, the application must check for all possible events after the event object is signalled.

Definition at line 2362 of file KFModuleDll.c.

4.3.4.4 DllExport KFM_ERROR WINAPI KFModule_AttachMsg (HANDLE *hConn,* HANDLE *hWnd,* UINT *msg,* UINT *evCode*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
evCode	Event(s) for which a message is sent.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of KFMODULE_RECEIVE, KFMODULE_TRANSMIT, KFMODULE_RESPONSE, KFMODULE_EVENT and KFMODULE_ERROR. The wParam of the sent message will be one of these event codes.

Definition at line 2417 of file KFModuleDll.c.

4.3.4.5 DllExport KFM_ERROR WINAPI KFModule_Close (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function should be called when the communication channel is no longer needed. Failing to close the channel prevents new connections to the channel as long as the dll stays loaded.

Note. When using the XML interface, function KFModule_Disconnect() must be called before this function in order to disconnect the instrument from the open communication channel. Otherwise the instrument may refuse new connections.

Definition at line 2099 of file KFModuleDll.c.

4.3.4.6 DllExport KFM_ERROR WINAPI KFModule_Connect (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function sends the "Connect" command with a current date/time setting to the instrument. It is recommended to use this function to connect to the instrument so that the calendar of the instrument is always set to the local time automatically.

Note. This funtion returns before the actual connection is done, meaning that the return value indicates merely if the starting of the command is succesfull of not. The instrument will send a separate response after the connection is finished. See interface specification of the instrument in question for more details.

Definition at line 2133 of file KFModuleDll.c.

4.3.4.7 DllExport KFM_ERROR WINAPI KFModule_Disconnect (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

This function sends the "disconnect" command to the instrument. This is only needed because of the symmetry in the list of API functions. Disconnect() is the pair to Connect() function.

Note. This funtion returns before the actual connection is closed, meaning that the return value indicates merely if

the starting of the command is succesfull of not. The instrument will send a separate response just before the disconnection. See interface specification of the instrument in question for more details.

Definition at line 2178 of file KFModuleDll.c.

4.3.4.8 DllExport KFM_ERROR WINAPI KFModule_DownloadProtocol (HANDLE *hConn,* LPCSTR *protocolName,* LPCSTR *path*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
protocolName	Name of the protocol to export.
path	Complete path of the .bdz file to receive the protocol.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The given protocol name must match exactly with a protocol name stored in the instrument. The dll must have write access to the given .bdz file. The file will be overwritten by the dll.

Note. This funtion returns before the actual transfer is completed, meaning that the return value indicates merely if the starting of the transfer is succesfull of not. The instrument will send a separate response after the transfer. See interface specification of the instrument in question for more details.

Definition at line 2627 of file KFModuleDll.c.

4.3.4.9 DllExport KFM_ERROR WINAPI KFModule_GetError (HANDLE *hConn*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Most dll functions return an error code, but an error may occur asynchronously in the receive or transmit thread of the dll. This kind of errors are reported with the KFMODULE_ERROR event. The application should then call this function to read the error code and take action. The error usually is KFM_ERROR_DISCONNECTED, in which case the application should close the current connection and try to open a new one.

Definition at line 2673 of file KFModuleDll.c.

4.3.4.10 DllExport DWORD WINAPI KFModule_ListLanDevices (LPCSTR *DeviceName,* LPCSTR *SerialNumber,* LPSTR *buf,* DWORD *bufsize*)

Parameters

instrumentName	Name of the instrument. Only instruments with a matching name are listed. May be NULL, in which case the found devices are not filtered by the name.
SerialNumber	The serial number of the instrument. Only instruments with a matching serial number are listed. May be NULL, in which case the found devices are not filtered by the serial number.
buf	The buffer to list the found devices to.
bufsize	Size of the buffer in bytes.

Returns

The size in bytes of the complete list of found devices. If the returned value is equal or smaller than the given buffer size, the buffer contains the complete list of devices found on the LAN. If the returned value is higher than the given buffer size, no data is returned and the caller must call the function again with big enough buffer. Return value 0 means that no instruments were found.

On success, the caller's buffer contains zero terminated strings with a combined length of the return value. The string terminating zeros are included in the length.

For each found instrument, the first string is the instrument IPv4 address and the TCP port number it is listening, enclosed in square brackets, e.g. [10.32.196.210:49536].

The IP address string is followed by the WS-Discovery match strings, usually 3 of them. The first one is always 'Thermo-Device', the second one is the instrument name and the third one the instrument serial number string.

If the match strings are followed by a string in angle brackets, e.g. <10.32.196.154:57403>, it means that the instrument is currently connected that IP address and TCP port, and trying to connect to that instrument with function KFModule_-OpenLan() will fail. If there is no string in square brackets, the instrument will accept a connection.

Definition at line 2000 of file KFModuleDll.c.

4.3.4.11 DllExport HANDLE WINAPI KFModule_OpenLan (LPCSTR *DeviceName,* LPCSTR *SerialNumber,* UINT *timeout*)

Parameters

instrumentName	Name of the instrument. This must match the device name the instrument uses for matching a WS--Discovery Probe and Resolve.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first found KFModule instrument.
timeout	Timeout which is used to search the instrument. If 0 is given then WS-Discovery will use default 4 sec timeout.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per device is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 1935 of file KFModuleDll.c.

4.3.4.12 DllExport HANDLE WINAPI KFModule_OpenSerial (WORD *port,* DWORD *baud,* LPCSTR *DeviceName,* LPCSTR *SerialNumber*)

Parameters

port	Serial port number.
baud	Serial port baudrate.
DeviceName	Pointer to the device name string of the device. The device must report an identical name to the VER command for the connection to succeed. This parameter may be NULL, in which case the device name is ignored.

SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number to the VER command for the connection to succeed. This parameter may be NULL, in which case the serial number is ignored.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per serial port is allowed. If both DeviceName and SerialNumber are NULL, no VER command is sent to the instrument.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 1880 of file KFModuleDll.c.

4.3.4.13 DllExport HANDLE WINAPI KFModule_OpenSimulator (KFM_SIMULATOR_PORT *port,* WORD *productId*)

Parameters

port	The simulator port to connect to, one of KFM_SIMULATOR_USB, KFM_SIMULATOR_COM, KFM_SIMULATOR_DBG or KFM_SIMULATOR_LAN.
productId	(USB) product id of the instrument.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per simulator communication channel is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 2030 of file KFModuleDll.c.

4.3.4.14 DllExport HANDLE WINAPI KFModule_OpenUsb (LPCSTR *SerialNumber,* WORD *productId*)

Parameters

SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first device with matching VendorID and ProductID.
productId	Manufacturer product id number of the USB device.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

One of the KFModule_OpenXxx() functions must be called first before using any other functions in the library. Only one connection per device is allowed.

Note. Function KFModule_Connect() must be called after this function in order to use the XML interface to communicate with the instrument.

Definition at line 1827 of file KFModuleDll.c.

4.3.4.15 DllExport KFM_ERROR WINAPI KFModule_ReadEvent (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the event.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_EVENT event. The received data is a XML Event. The event is not sent until a whole XML Event is received. If the caller's buffer is not large enough to hold the whole Event, a partial Event is copied. An empty string is copied when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent events from the instrument may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_EVENT( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadEvent( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

Definition at line 2333 of file KFModuleDll.c.

4.3.4.16 DllExport KFM_ERROR WINAPI KFModule_ReadReceived (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the line.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_RECEIVE event. The received data is neither a XML Response nor a XML Event. An empty string will be copied to caller's buffer when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent strings from the instrument may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_RECEIVE( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadReceived( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

Definition at line 2239 of file KFModuleDll.c.

4.3.4.17 DllExport KFM_ERROR WINAPI KFModule_ReadResponse (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	Caller's buffer for the response.
bufsize	Size of the caller's buffer.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS if any data was copied to caller's buffer.

Call this function in response to the KFMODULE_RESPONSE event. The received data is a XML Response. The event is not sent until a whole XML Response is received. If the caller's buffer is not large enough to hold the whole response, a partial response is copied. An empty string is copied when there is no more data to read. The returned string is always NUL terminated and does not contain the CR/LF characters sent by the instrument.

Note. Always read all data from the KFModuleDll when handling the event. Otherwise subsequent responses may be missed. This can happen especially when the user application doesn't react to the events immediately. See example below:

```
void handler_KFMODULE_RESPONSE( HANDLE connection )
{
    KFM_ERROR      err;
    char         buf[100];

    while( (err = KFModule_ReadResponse( connection, buf, sizeof( buf ))) == KFM_ERROR_SUCCESS)
    {
        // Do something with the data
```

```
    }

    if( err > KFM_ERROR_NO_DATA)
    {
        // Handle the error
    }
}
```

Definition at line 2286 of file KFModuleDll.c.

4.3.4.18 DllExport KFM_ERROR WINAPI KFModule_Send (HANDLE *hConn*, LPCSTR *buf*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
buf	The NUL terminated string to send.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

Use this function to send commands and acknowledges to the instrument. For uploading and downloading a protocol there are dedicated functions KFModule_UploadProtocol() and KFModule_DownloadProtocol(). Also connecting/disconnecting is recommended to be done with separate KFModule_Connect() and KFModule_Disconnect() functions.

The data to be sent to the instrument is buffered in the dll. A KFMODULE_TRANSMIT event is sent to the application when all data is sent. You may call this function repeatedly without waiting for the event, but a KFM_ERROR_OUT_OF_HEAP error may be returned if you send a lot of data without waiting for the event.

Note. Remember to terminate commands with a new line character (ASCII 10). See interface specification of the instrument in question for more details.

Definition at line 2495 of file KFModuleDll.c.

4.3.4.19 DllExport KFM_ERROR WINAPI KFModule_UploadProtocol (HANDLE *hConn*, LPCSTR *protocolName*, LPCSTR *path*)

Parameters

hConn	A handle returned by one of the KFModule_OpenXxx() functions.
protocolName	Name of the protocol to export.
path	Complete path of the .bdz file containing the protocol to export.

Returns

One of KFM_ERROR codes, KFM_ERROR_SUCCESS on success.

The given protocol name must match exactly with a protocol name stored in the .bdz file. The .bdz file may contain several protocols, but only the requested protocol is sent to the instrument.

Note. This funtion returns before the actual transfer is completed, meaning that the return value indicates merely if the starting of the transfer is succesfull of not. The instrument will send a separate response after the transfer. See interface specification of the instrument in question for more details.

Definition at line 2600 of file KFModuleDll.c.

4.4 mainpage.h File Reference

KFModuleDll Interface Specification.

Index

api.h, 27

KFM_ERROR_DISCONNECTED
    KFModuleDll.h, 39
KFM_ERROR_EXPORT_IMPORT_BUSY
    KFModuleDll.h, 39
KFM_ERROR_FILE_NOT_FOUND
    KFModuleDll.h, 39
KFM_ERROR_INVALID_ARGUMENT
    KFModuleDll.h, 39
KFM_ERROR_INVALID_BDZ_FILE
    KFModuleDll.h, 39
KFM_ERROR_INVALID_HANDLE
    KFModuleDll.h, 39
KFM_ERROR_NO_DATA
    KFModuleDll.h, 39
KFM_ERROR_OUT_OF_HEAP
    KFModuleDll.h, 39
KFM_ERROR_PROTOCOL_NOT_FOUND
    KFModuleDll.h, 39
KFM_ERROR_SUCCESS
    KFModuleDll.h, 39
KFM_ERROR_TEMP_FILE_CREATE
    KFModuleDll.h, 39
KFM_SIMULATOR_COM
    KFModuleDll.h, 39
KFM_SIMULATOR_DBG
    KFModuleDll.h, 39
KFM_SIMULATOR_LAN
    KFModuleDll.h, 39
KFM_SIMULATOR_USB
    KFModuleDll.h, 39
KFModuleDll.h
    KFM_ERROR_DISCONNECTED, 39
    KFM_ERROR_EXPORT_IMPORT_BUSY, 39
    KFM_ERROR_FILE_NOT_FOUND, 39
    KFM_ERROR_INVALID_ARGUMENT, 39
    KFM_ERROR_INVALID_BDZ_FILE, 39
    KFM_ERROR_INVALID_HANDLE, 39
    KFM_ERROR_NO_DATA, 39
    KFM_ERROR_OUT_OF_HEAP, 39
    KFM_ERROR_PROTOCOL_NOT_FOUND, 39
    KFM_ERROR_SUCCESS, 39
    KFM_ERROR_TEMP_FILE_CREATE, 39
    KFM_SIMULATOR_COM, 39
    KFM_SIMULATOR_DBG, 39
    KFM_SIMULATOR_LAN, 39
    KFM_SIMULATOR_USB, 39
KF_PRESTO_PID
    KFModuleDll.h, 38
KFM_ERROR
    KFModuleDll.h, 39
KFM_SIMULATOR_PORT
    KFModuleDll.h, 39
KFMODULE_ERROR
    KFModuleDll.h, 38
KFMODULE_EVENT
    KFModuleDll.h, 38
KFMODULE_RECEIVE
    KFModuleDll.h, 38
KFMODULE_RESPONSE
    KFModuleDll.h, 38
KFMODULE_TRANSMIT
    KFModuleDll.h, 38
KFModule_Abort
    KFModuleDll.c, 28
    KFModuleDll.h, 39
KFModule_AttachCallback
    KFModuleDll.c, 28
    KFModuleDll.h, 39
KFModule_AttachEvent
    KFModuleDll.c, 29
    KFModuleDll.h, 40
KFModule_AttachMsg
    KFModuleDll.c, 29
    KFModuleDll.h, 40
KFModule_Close
    KFModuleDll.c, 29
    KFModuleDll.h, 40
KFModule_Connect
    KFModuleDll.c, 30
    KFModuleDll.h, 41
KFModule_Disconnect
    KFModuleDll.c, 30
    KFModuleDll.h, 41
KFModule_DownloadProtocol
    KFModuleDll.c, 30
    KFModuleDll.h, 42
KFModule_GetError
    KFModuleDll.c, 31
    KFModuleDll.h, 42
KFModule_ListLanDevices
    KFModuleDll.c, 31
    KFModuleDll.h, 42
KFModule_OpenLan

KFModuleDll.c, 32
KFModuleDll.h, 43
KFModule_OpenSerial
KFModuleDll.c, 32
KFModuleDll.h, 43
KFModule_OpenSimulator
KFModuleDll.c, 33
KFModuleDll.h, 44
KFModule_OpenUsb
KFModuleDll.c, 33
KFModuleDll.h, 44
KFModule_ReadEvent
KFModuleDll.c, 33
KFModuleDll.h, 45
KFModule_ReadReceived
KFModuleDll.c, 34
KFModuleDll.h, 45
KFModule_ReadResponse
KFModuleDll.c, 35
KFModuleDll.h, 46
KFModule_Send
KFModuleDll.c, 35
KFModuleDll.h, 47
KFModule_UploadProtocol
KFModuleDll.c, 36
KFModuleDll.h, 47
KFModuleDll.c, 27
KFModule_Abort, 28
KFModule_AttachCallback, 28
KFModule_AttachEvent, 29
KFModule_AttachMsg, 29
KFModule_Close, 29
KFModule_Connect, 30
KFModule_Disconnect, 30
KFModule_DownloadProtocol, 30
KFModule_GetError, 31
KFModule_ListLanDevices, 31
KFModule_OpenLan, 32
KFModule_OpenSerial, 32
KFModule_OpenSimulator, 33
KFModule_OpenUsb, 33
KFModule_ReadEvent, 33
KFModule_ReadReceived, 34
KFModule_ReadResponse, 35
KFModule_Send, 35
KFModule_UploadProtocol, 36
KFModuleDll.h, 36
KF_PRESTO_PID, 38
KFM_ERROR, 39
KFMODULE_ERROR, 38
KFMODULE_EVENT, 38
KFMODULE_RECEIVE, 38
KFMODULE_RESPONSE, 38
KFMODULE_TRANSMIT, 38
KFModule_Abort, 39
KFModule_AttachCallback, 39
KFModule_AttachEvent, 40
KFModule_AttachMsg, 40
KFModule_Close, 40
KFModule_Connect, 41
KFModule_Disconnect, 41
KFModule_DownloadProtocol, 42
KFModule_GetError, 42
KFModule_ListLanDevices, 42
KFModule_OpenLan, 43
KFModule_OpenSerial, 43
KFModule_OpenSimulator, 44
KFModule_OpenUsb, 44
KFModule_ReadEvent, 45
KFModule_ReadReceived, 45
KFModule_ReadResponse, 46
KFModule_Send, 47
KFModule_UploadProtocol, 47

mainpage.h, 48

Thermo Scientific

ThermoUSB.dll

Interface Specification

Thermo SCIENTIFIC

Contents

1 ThermoUSB dll Interface Specification 1

2 Using ThermoUSB 3
- 2.1 About . . . 3
- 2.2 Exported functions . . . 4
  - 2.2.1 ThermoUSBOpen . . . 5
  - 2.2.2 ThermoUSBClose . . . 6
  - 2.2.3 ThermoUSBAttachEvent . . . 7
  - 2.2.4 ThermoUSBAttachMsg . . . 8
  - 2.2.5 ThermoUSBRead . . . 9
  - 2.2.6 ThermoUSBReadBinary . . . 10
  - 2.2.7 ThermoUSBWrite . . . 11
  - 2.2.8 ThermoUSBWriteBinary . . . 12
  - 2.2.9 ThermoUSBGetError . . . 13
  - 2.2.10 ThermoUSBAbort . . . 14
  - 2.2.11 ThermoUSBGetThermoVendorId . . . 15

3 File Index 17
- 3.1 File List . . . 17

4 File Documentation 19
- 4.1 api.h File Reference . . . 19
- 4.2 mainpage.h File Reference . . . 19
- 4.3 ThermoUSB.c File Reference . . . 19
  - 4.3.1 Detailed Description . . . 20
  - 4.3.2 Function Documentation . . . 20
    - 4.3.2.1 ThermoUSBAbort . . . 20
    - 4.3.2.2 ThermoUSBAttachEvent . . . 20
    - 4.3.2.3 ThermoUSBAttachMsg . . . 20
    - 4.3.2.4 ThermoUSBClose . . . 21
    - 4.3.2.5 ThermoUSBGetError . . . 21
    - 4.3.2.6 ThermoUSBGetThermoVendorId . . . 21
    - 4.3.2.7 ThermoUSBOpen . . . 21
    - 4.3.2.8 ThermoUSBRead . . . 22
    - 4.3.2.9 ThermoUSBReadBinary . . . 22
    - 4.3.2.10 ThermoUSBWrite . . . 23
    - 4.3.2.11 ThermoUSBWriteBinary . . . 23
- 4.4 ThermoUSB.h File Reference . . . 23
  - 4.4.1 Detailed Description . . . 24
  - 4.4.2 Macro Definition Documentation . . . 24
    - 4.4.2.1 THERMOUSB_ERROR . . . 24
    - 4.4.2.2 THERMOUSB_RECEIVE . . . 24
    - 4.4.2.3 THERMOUSB_TRANSMIT . . . 24
  - 4.4.3 Function Documentation . . . 25
    - 4.4.3.1 ThermoUSBAbort . . . 25
    - 4.4.3.2 ThermoUSBAttachEvent . . . 25

4.4.3.3 ThermoUSBAttachMsg . . . 25
4.4.3.4 ThermoUSBClose . . . 26
4.4.3.5 ThermoUSBGetError . . . 26
4.4.3.6 ThermoUSBGetThermoVendorId . . . 26
4.4.3.7 ThermoUSBOpen . . . 26
4.4.3.8 ThermoUSBRead . . . 27
4.4.3.9 ThermoUSBReadBinary . . . 27
4.4.3.10 ThermoUSBWrite . . . 27
4.4.3.11 ThermoUSBWriteBinary . . . 27

Chapter 1

ThermoUSB dll Interface Specification

About

The purpose of the ThermoUSB.dll dynamic link library is to make interface to the Thermo microplate instruments easy. The user of the ThermoUSB.dll does not have to know the details of USB communication. Knowing the USB Product Id of the instrument is enough.

Confidential

Chapter 2

Using ThermoUSB

2.1 About

Because it is not trivial to write PC software from scratch to communicate with the instrument, the ThermoUSB Dynamic Link Library is provided. It hides much of the complexity of the communication.

ThermoUSB.dll is a generic library for communicating with several different Thermo Scientific microplate instruments. To communicate with an instrument you use the exported functions of ThermoUSB.dll.

Depending on your project setup, you may find useful a couple of other files which are also provided. The header file ThermoUSB.h contains the prototypes of the exported functions and definitions of constant values used by the dll. File ThermoUSB.lib contains information about the dll the linker uses to add references to the library in the executable. This way the dll is automatically loaded and the exported functions of the library can be called as easy as the functions in the code using the dll.

2.2 Exported functions

- ThermoUSBOpen
- ThermoUSBClose
- ThermoUSBAttachEvent
- ThermoUSBAttachMsg
- ThermoUSBRead
- ThermoUSBReadBinary
- ThermoUSBWrite
- ThermoUSBWriteBinary
- ThermoUSBGetError
- ThermoUSBAbort
- ThermoUSBGetThermoVendorId

2.2.1 ThermoUSBOpen

Search for the requested USB device and open a communication channel to it. Full declaration: ThermoUSBOpen().

Parameters

VendorID	The USB vendor id of the device manufacturer, 0x0AB6 for Thermo Fisher Scientific Oy.
ProductID	Product id number of the device.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first device with matching VendorID and ProductID.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function (or ThermoUSBOpenSimulator) must be called first before using any other functions in the library. Only one connection per device is allowed.

2.2.2 ThermoUSBClose

Close an USB connection. Full declaration: ThermoUSBClose().

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen()

Closes the communication channel to the instrument. This function should be called when the application has finished communicating with the instrument. It is not possible to open a communication channel to the instrument while the previous channel is still open. When the dll is unloaded from the memory, all channels still open are automatically closed.

2.2.3 ThermoUSBAttachEvent

Attach an event object to an USB connection. Full declaration: ThermoUSBAttachEvent().

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen() (pointer to a USB_CONNECTION structure).
evCode	Event(s) to signal.
object	Handle of an event object.

Parameter 'evCode' may be any combination of THERMOUSB_RECEIVE, THERMOUSB_TRANSMIT and THERMOUSB_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOUSB_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoUSBRead() or ThermoUSBReadBinary as long as they return data.

The THERMOUSB_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOUSB_TRANSMIT event is received, functions ThermoUSBWrite() and ThermoUSBWriteBinary() will never fail.

2.2.4 ThermoUSBAttachMsg

Attach an event message to an USB connection. Full declaration: ThermoUSBAttachMsg().

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen() (pointer to a USB_CONNECTION structure).
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
event	Event(s) for which a message is sent.

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of THERMOUSB_RECEIVE, THERMOUSB_TRANSMIT and THERMOUSB_ERROR. The event code is passed in the wParam member of the sent Windows message.

The THERMOUSB_RECEIVE event is sent whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. Therefore upon receiving this event the application should call ThermoUSBRead() or ThermoUSBReadBinary as long as they return data.

The THERMOUSB_TRANSMIT event is sent when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOUSB_TRANSMIT event is received, functions ThermoUSBWrite() and ThermoUSBWriteBinary() will never fail.

2.2.5 ThermoUSBRead

Read a received response line. Full declaration: ThermoUSBRead().

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Pointer to a buffer to receive the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line was retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

2.2.6 ThermoUSBReadBinary

Read data received from the USB device. Full declaration: ThermoUSBReadBinary().

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Pointer to buffer receiving the data.
bufsize	Size of the buffer in bytes.

Returns

Number of bytes copied to the user buffer.

Unlike function ThermoUSBRead(), this function returns all data received from the instrument without doing any interpretation on it. You can control the number of bytes returned with the bufsize parameter.

If the return value is less than the given bufsize parameter, there is no more data.

2.2.7 ThermoUSBWrite

Write a string to the transmit buffer. Full declaration: ThermoUSBWrite().

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Buffer containing the ASCIIZ string to send.

Returns

TRUE if the string was written to the transmit buffer, else FALSE.

If the whole string does not fit in the transmit buffer, writes nothing and returns FALSE. The application may retry at a later time.

2.2.8 ThermoUSBWriteBinary

Write binary data to the transmit buffer. Full declaration: ThermoUSBWriteBinary().

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Buffer containing the data to send.
count	Number of bytes to send.

Returns

TRUE if the data was written to the transmit buffer, else FALSE.

If all data does not fit in the transmit buffer, writes nothing and returns FALSE. The application may retry at a later time.

2.2.9 ThermoUSBGetError

Return the error code stored to the connection structure. Full declaration: ThermoUSBGetError().

2.2.10 ThermoUSBAbort

Send Abort command to the device. Full declaration: ThermoUSBAbort().

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().

2.2.11 ThermoUSBGetThermoVendorId

Return Thermo Fisher Scientific Oy USB vendor id. Full declaration: ThermoUSBGetThermoVendorId().

Returns

Thermo Fisher Scientific Oy USB vendor id, 0x0AB6.

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

api.h
: Part of ThermoUSB dll documentation 19

mainpage.h
: Start page of ThermoUSB.dll Design Description 19

ThermoUSB.c
: Implements USB interface to Thermo microplate instruments using either Windows HID driver or the libusb-win32 driver from http://libusb-win32.sourceforge.net 19

ThermoUSB.h
: Functions exported from ThermoUSB.dll 23

Chapter 4

File Documentation

4.1 api.h File Reference

Part of ThermoUSB dll documentation.

4.2 mainpage.h File Reference

Start page of ThermoUSB.dll Design Description.

4.3 ThermoUSB.c File Reference

Implements USB interface to Thermo microplate instruments using either Windows HID driver or the libusb-win32 driver from http://libusb-win32.sourceforge.net.

Functions

- HANDLE WINAPI ThermoUSBOpen (WORD VendorID, WORD ProductID, LPCSTR SerialNumber)

  Search for the requested USB device and open a communication channel to it.
- void WINAPI ThermoUSBClose (HANDLE hConn)

  Close an USB connection.
- BOOL WINAPI ThermoUSBAttachEvent (HANDLE hConn, UINT evCode, HANDLE object)

  Attach an event object to an USB connection.
- BOOL WINAPI ThermoUSBAttachMsg (HANDLE hConn, HANDLE hWnd, UINT msg, UINT evCode)

  Attach an event message to an USB connection.
- BOOL WINAPI ThermoUSBRead (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received response line.
- DWORD WINAPI ThermoUSBReadBinary (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read data received from the USB device.
- BOOL WINAPI ThermoUSBWrite (HANDLE hConn, LPCSTR buf)

  Write a string to the transmit buffer.
- BOOL WINAPI ThermoUSBWriteBinary (HANDLE hConn, LPCSTR buf, DWORD count)

  Write binary data to the transmit buffer.
- void WINAPI ThermoUSBAbort (HANDLE hConn)

  Send Abort command to the device.

- DWORD WINAPI ThermoUSBGetError (HANDLE hConn)

  Return the error code stored to the connection structure.
- WORD WINAPI ThermoUSBGetThermoVendorId (void)

  Return Thermo Fisher Scientific Oy USB vendor id.

4.3.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

Definition in file ThermoUSB.c.

4.3.2 Function Documentation

4.3.2.1 void WINAPI ThermoUSBAbort (HANDLE *hConn*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().

Definition at line 2241 of file ThermoUSB.c.

4.3.2.2 BOOL WINAPI ThermoUSBAttachEvent (HANDLE *hConn*, UINT *evCode*, HANDLE *object*)

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen() (pointer to a USB_CONNECTION structure).
evCode	Event(s) to signal.
object	Handle of an event object.

Parameter 'evCode' may be any combination of THERMOUSB_RECEIVE, THERMOUSB_TRANSMIT and THERMOUSB_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOUSB_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoUSBRead() or ThermoUSBReadBinary as long as they return data.

The THERMOUSB_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOUSB_TRANSMIT event is received, functions ThermoUSBWrite() and ThermoUSBWriteBinary() will never fail.

Definition at line 1853 of file ThermoUSB.c.

4.3.2.3 BOOL WINAPI ThermoUSBAttachMsg (HANDLE *hConn*, HANDLE *hWnd*, UINT *msg*, UINT *evCode*)

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen() (pointer to a USB_CONNECTION structure).
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
event	Event(s) for which a message is sent.

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of THERMOUSB_RECEIVE, THERMOUSB_TRANSMIT and THERMOUSB_ERROR. The event code is passed in the wParam member of the sent Windows message.

The THERMOUSB_RECEIVE event is sent whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. Therefore upon receiving this event the application should call ThermoUSBRead() or ThermoUSBReadBinary as long as they return data.

The THERMOUSB_TRANSMIT event is sent when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOUSB_TRANSMIT event is received, functions ThermoUSBWrite() and ThermoUSBWriteBinary() will never fail.

Definition at line 1908 of file ThermoUSB.c.

4.3.2.4 void WINAPI ThermoUSBClose (HANDLE *hConn*)

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen()

Closes the communication channel to the instrument. This function should be called when the application has finished communicating with the instrument. It is not possible to open a communication channel to the instrument while the previous channel is still open. When the dll is unloaded from the memory, all channels still open are automatically closed.

Definition at line 1767 of file ThermoUSB.c.

4.3.2.5 DWORD WINAPI ThermoUSBGetError (HANDLE *hConn*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().

Returns

The current Windows error code stored for the connection.

Error conditions may be encountered asynchronously and not just as a direct result of a function call to the dll. Therefore the application should enable error reporting with ThermoUSBAttachEvent() or ThermoUSBAttachMsg(). When an error is reported to the application, it may call this function to get the Windows error code, which may or may not be helpful in resolving the problem. In general, if an error is reported then some receive or transmit data is lost, and the best way of action is to close the port and then retry opening it again.

A call to ThermoUSBGetError() resets the stored error code to ERROR_SUCCESS.

Definition at line 2308 of file ThermoUSB.c.

4.3.2.6 WORD WINAPI ThermoUSBGetThermoVendorId (void)

Returns

Thermo Fisher Scientific Oy USB vendor id, 0x0AB6.

Definition at line 2332 of file ThermoUSB.c.

4.3.2.7 HANDLE WINAPI ThermoUSBOpen (WORD *VendorID,* WORD *ProductID,* LPCSTR *SerialNumber*)

Parameters

VendorID	The USB vendor id of the device manufacturer, 0x0AB6 for Thermo Fisher Scientific Oy.

ProductID	Product id number of the device.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first device with matching VendorID and ProductID.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function (or ThermoUSBOpenSimulator) must be called first before using any other functions in the library. Only one connection per device is allowed.

Definition at line 1738 of file ThermoUSB.c.

4.3.2.8 BOOL WINAPI ThermoUSBRead (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Pointer to a buffer to receive the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line was retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

Definition at line 1948 of file ThermoUSB.c.

4.3.2.9 DWORD WINAPI ThermoUSBReadBinary (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Pointer to buffer receiving the data.
bufsize	Size of the buffer in bytes.

Returns

Number of bytes copied to the user buffer.

Unlike function ThermoUSBRead(), this function returns all data received from the instrument without doing any interpretation on it. You can control the number of bytes returned with the bufsize parameter.

If the return value is less than the given bufsize parameter, there is no more data.

Definition at line 2123 of file ThermoUSB.c.

4.3.2.10 BOOL WINAPI ThermoUSBWrite (HANDLE *hConn,* LPCSTR *buf*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Buffer containing the ASCIIZ string to send.

Returns

TRUE if the string was written to the transmit buffer, else FALSE.

If the whole string does not fit in the transmit buffer, writes nothing and returns FALSE. The application may retry at a later time.

Definition at line 2169 of file ThermoUSB.c.

4.3.2.11 BOOL WINAPI ThermoUSBWriteBinary (HANDLE *hConn,* LPCSTR *buf,* DWORD *count*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Buffer containing the data to send.
count	Number of bytes to send.

Returns

TRUE if the data was written to the transmit buffer, else FALSE.

If all data does not fit in the transmit buffer, writes nothing and returns FALSE. The application may retry at a later time.

Definition at line 2190 of file ThermoUSB.c.

4.4 ThermoUSB.h File Reference

Functions exported from ThermoUSB.dll.

Macros

- #define THERMOUSB_RECEIVE 1

  Data received.
- #define THERMOUSB_TRANSMIT 2

  All data transmitted.
- #define THERMOUSB_ERROR 4

  Fatal error event.

Functions

- DllExport HANDLE WINAPI ThermoUSBOpen (WORD VendorID, WORD ProductID, LPCSTR SerialNumber)

  Search for the requested USB device and open a communication channel to it.
- DllExport void WINAPI ThermoUSBClose (HANDLE hConn)

  Close an USB connection.
- DllExport BOOL WINAPI ThermoUSBAttachEvent (HANDLE hConn, UINT evCode, HANDLE object)

  Attach an event object to an USB connection.

- DllExport BOOL WINAPI ThermoUSBAttachMsg (HANDLE hConn, HANDLE hWnd, UINT msg, UINT event)

  Attach an event message to an USB connection.
- DllExport BOOL WINAPI ThermoUSBRead (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received response line.
- DllExport DWORD WINAPI ThermoUSBReadBinary (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read data received from the USB device.
- DllExport BOOL WINAPI ThermoUSBWrite (HANDLE hConn, LPCSTR buf)

  Write a string to the transmit buffer.
- DllExport BOOL WINAPI ThermoUSBWriteBinary (HANDLE hConn, LPCSTR buf, DWORD count)

  Write binary data to the transmit buffer.
- DllExport DWORD WINAPI ThermoUSBGetError (HANDLE hConn)

  Return the error code stored to the connection structure.
- DllExport void WINAPI ThermoUSBAbort (HANDLE hConn)

  Send Abort command to the device.
- DllExport WORD WINAPI ThermoUSBGetThermoVendorId (void)

  Return Thermo Fisher Scientific Oy USB vendor id.

4.4.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

USB interface to Thermo microplate instruments.

Definition in file ThermoUSB.h.

4.4.2 Macro Definition Documentation

4.4.2.1 #define THERMOUSB_ERROR 4

In case of error the application receives an error event or an error message. The application may query the error code with function ThermoUSBGetError(). If the device is a HID device, the error code may be any of the Windows error codes. In case of a libusb-win32 device, the error code from libusb driver is translated to one of the following Windows error codes:

Windows error	Code	Description
ERROR_SUCCESS	0	No error.
ERROR_ACCESS_DENIED	5	I/O error.
ERROR_NOT_ENOUGH_MEMORY	12	Not enough memory.
ERROR_BAD_COMMAND	22	Invalid parameter.
WAIT_TIMEOUT	116	A transfer timed out.

What the application can do in case of an error is to first call ThermoUSBClose() and then try to reopen with ThermoUSB-Open().

Definition at line 47 of file ThermoUSB.h.

4.4.2.2 #define THERMOUSB_RECEIVE 1

Definition at line 18 of file ThermoUSB.h.

4.4.2.3 #define THERMOUSB_TRANSMIT 2

Definition at line 19 of file ThermoUSB.h.

4.4.3 Function Documentation

4.4.3.1 DllExport void WINAPI ThermoUSBAbort (HANDLE *hConn*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().

Definition at line 2241 of file ThermoUSB.c.

4.4.3.2 DllExport BOOL WINAPI ThermoUSBAttachEvent (HANDLE *hConn,* UINT *evCode,* HANDLE *object*)

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen() (pointer to a USB_CONNECTION structure).
evCode	Event(s) to signal.
object	Handle of an event object.

Parameter 'evCode' may be any combination of THERMOUSB_RECEIVE, THERMOUSB_TRANSMIT and THERMOUSB_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOUSB_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoUSBRead() or ThermoUSBReadBinary as long as they return data.

The THERMOUSB_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOUSB_TRANSMIT event is received, functions ThermoUSBWrite() and ThermoUSBWriteBinary() will never fail.

Definition at line 1853 of file ThermoUSB.c.

4.4.3.3 DllExport BOOL WINAPI ThermoUSBAttachMsg (HANDLE *hConn,* HANDLE *hWnd,* UINT *msg,* UINT *evCode*)

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen() (pointer to a USB_CONNECTION structure).
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
event	Event(s) for which a message is sent.

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of THERMOUSB_RECEIVE, THERMOUSB_TRANSMIT and THERMOUSB_ERROR. The event code is passed in the wParam member of the sent Windows message.

The THERMOUSB_RECEIVE event is sent whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. Therefore upon receiving this event the application should call ThermoUSBRead() or ThermoUSBReadBinary as long as they return data.

The THERMOUSB_TRANSMIT event is sent when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOUSB_TRANSMIT event is received, functions ThermoUSBWrite() and ThermoUSBWriteBinary() will never fail.

Definition at line 1908 of file ThermoUSB.c.

4.4.3.4 DllExport void WINAPI ThermoUSBClose (HANDLE *hConn*)

Parameters

hConn	Connection handle returned form a call to ThermoUSBOpen()

Closes the communication channel to the instrument. This function should be called when the application has finished communicating with the instrument. It is not possible to open a communication channel to the instrument while the previous channel is still open. When the dll is unloaded from the memory, all channels still open are automatically closed.

Definition at line 1767 of file ThermoUSB.c.

4.4.3.5 DllExport DWORD WINAPI ThermoUSBGetError (HANDLE *hConn*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().

Returns

The current Windows error code stored for the connection.

Error conditions may be encountered asynchronously and not just as a direct result of a function call to the dll. Therefore the application should enable error reporting with ThermoUSBAttachEvent() or ThermoUSBAttachMsg(). When an error is reported to the application, it may call this function to get the Windows error code, which may or may not be helpful in resolving the problem. In general, if an error is reported then some receive or transmit data is lost, and the best way of action is to close the port and then retry opening it again.

A call to ThermoUSBGetError() resets the stored error code to ERROR_SUCCESS.

Definition at line 2308 of file ThermoUSB.c.

4.4.3.6 DllExport WORD WINAPI ThermoUSBGetThermoVendorId (void)

Returns

Thermo Fisher Scientific Oy USB vendor id, 0x0AB6.

Definition at line 2332 of file ThermoUSB.c.

4.4.3.7 DllExport HANDLE WINAPI ThermoUSBOpen (WORD *VendorID,* WORD *ProductID,* LPCSTR *SerialNumber*)

Parameters

VendorID	The USB vendor id of the device manufacturer, 0x0AB6 for Thermo Fisher Scientific Oy.
ProductID	Product id number of the device.
SerialNumber	Pointer to the serial number string of the device. The device must report an identical serial number for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first device with matching VendorID and ProductID.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function (or ThermoUSBOpenSimulator) must be called first before using any other functions in the library. Only one connection per device is allowed.

Definition at line 1738 of file ThermoUSB.c.

4.4.3.8 DllExport BOOL WINAPI ThermoUSBRead (HANDLE *hConn*, LPSTR *buf*, DWORD *bufsize*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Pointer to a buffer to receive the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line was retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

Definition at line 1948 of file ThermoUSB.c.

4.4.3.9 DllExport DWORD WINAPI ThermoUSBReadBinary (HANDLE *hConn*, LPSTR *buf*, DWORD *bufsize*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Pointer to buffer receiving the data.
bufsize	Size of the buffer in bytes.

Returns

Number of bytes copied to the user buffer.

Unlike function ThermoUSBRead(), this function returns all data received from the instrument without doing any interpretation on it. You can control the number of bytes returned with the bufsize parameter.

If the return value is less than the given bufsize parameter, there is no more data.

Definition at line 2123 of file ThermoUSB.c.

4.4.3.10 DllExport BOOL WINAPI ThermoUSBWrite (HANDLE *hConn*, LPCSTR *buf*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().
buf	Buffer containing the ASCIIZ string to send.

Returns

TRUE if the string was written to the transmit buffer, else FALSE.

If the whole string does not fit in the transmit buffer, writes nothing and returns FALSE. The application may retry at a later time.

Definition at line 2169 of file ThermoUSB.c.

4.4.3.11 DllExport BOOL WINAPI ThermoUSBWriteBinary (HANDLE *hConn*, LPCSTR *buf*, DWORD *count*)

Parameters

hConn	Handle of the USB connection. This must be a handle returned from ThermoUSBOpen().

buf	Buffer containing the data to send.
count	Number of bytes to send.

Returns

TRUE if the data was written to the transmit buffer, else FALSE.

If all data does not fit in the transmit buffer, writes nothing and returns FALSE. The application may retry at a later time.

Definition at line 2190 of file ThermoUSB.c.

Index

api.h, 19

mainpage.h, 19

THERMOUSB_ERROR
    ThermoUSB.h, 24
THERMOUSB_RECEIVE
    ThermoUSB.h, 24
THERMOUSB_TRANSMIT
    ThermoUSB.h, 24
ThermoUSB.c, 19
    ThermoUSBAbort, 20
    ThermoUSBAttachEvent, 20
    ThermoUSBAttachMsg, 20
    ThermoUSBClose, 21
    ThermoUSBGetError, 21
    ThermoUSBGetThermoVendorId, 21
    ThermoUSBOpen, 21
    ThermoUSBRead, 22
    ThermoUSBReadBinary, 22
    ThermoUSBWrite, 22
    ThermoUSBWriteBinary, 23
ThermoUSB.h, 23
    THERMOUSB_ERROR, 24
    THERMOUSB_RECEIVE, 24
    ThermoUSBAbort, 25
    ThermoUSBAttachEvent, 25
    ThermoUSBAttachMsg, 25
    ThermoUSBClose, 25
    ThermoUSBGetError, 26
    ThermoUSBGetThermoVendorId, 26
    ThermoUSBOpen, 26
    ThermoUSBRead, 26
    ThermoUSBReadBinary, 27
    ThermoUSBWrite, 27
    ThermoUSBWriteBinary, 27
ThermoUSBAbort
    ThermoUSB.c, 20
    ThermoUSB.h, 25
ThermoUSBAttachEvent
    ThermoUSB.c, 20
    ThermoUSB.h, 25
ThermoUSBAttachMsg
    ThermoUSB.c, 20
    ThermoUSB.h, 25
ThermoUSBClose
    ThermoUSB.c, 21
    ThermoUSB.h, 25
ThermoUSBGetError
    ThermoUSB.c, 21
    ThermoUSB.h, 26
ThermoUSBGetThermoVendorId
    ThermoUSB.c, 21
    ThermoUSB.h, 26
ThermoUSBOpen
    ThermoUSB.c, 21
    ThermoUSB.h, 26
ThermoUSBRead
    ThermoUSB.c, 22
    ThermoUSB.h, 26
ThermoUSBReadBinary
    ThermoUSB.c, 22
    ThermoUSB.h, 27
ThermoUSBWrite
    ThermoUSB.c, 22
    ThermoUSB.h, 27
ThermoUSBWriteBinary
    ThermoUSB.c, 23
    ThermoUSB.h, 27

Thermo Scientific

ThermoLAN.dll

Interface Specification

Thermo SCIENTIFIC

Contents

1 ThermoLAN dll Interface Specification 1

2 Using ThermoLAN 3
- 2.1 About . . . 3
- 2.2 Exported functions . . . 4
  - 2.2.1 ThermoLANListDevices . . . 5
  - 2.2.2 ThermoLANOpen . . . 6
  - 2.2.3 ThermoLANClose . . . 7
  - 2.2.4 ThermoLANWrite . . . 8
  - 2.2.5 ThermoLANRead . . . 9
  - 2.2.6 ThermoLANAttachEvent . . . 10
  - 2.2.7 ThermoLANAbort . . . 11
  - 2.2.8 ThermoLANGetError . . . 12

3 File Index 13
- 3.1 File List . . . 13

4 File Documentation 15
- 4.1 api.h File Reference . . . 15
- 4.2 mainpage.h File Reference . . . 15
- 4.3 ThermoLANWrapper.cpp File Reference . . . 15
  - 4.3.1 Detailed Description . . . 16
  - 4.3.2 Function Documentation . . . 16
    - 4.3.2.1 ThermoLANAbort . . . 16
    - 4.3.2.2 ThermoLANAttachEvent . . . 16
    - 4.3.2.3 ThermoLANClose . . . 17
    - 4.3.2.4 ThermoLANGetError . . . 17
    - 4.3.2.5 ThermoLANListDevices . . . 17
    - 4.3.2.6 ThermoLANOpen . . . 18
    - 4.3.2.7 ThermoLANRead . . . 18
    - 4.3.2.8 ThermoLANWrite . . . 18
- 4.4 ThermoLANWrapper.h File Reference . . . 19
  - 4.4.1 Detailed Description . . . 19
  - 4.4.2 Function Documentation . . . 19
    - 4.4.2.1 ThermoLANAbort . . . 19
    - 4.4.2.2 ThermoLANAttachEvent . . . 20
    - 4.4.2.3 ThermoLANClose . . . 20
    - 4.4.2.4 ThermoLANGetError . . . 20
    - 4.4.2.5 ThermoLANListDevices . . . 21
    - 4.4.2.6 ThermoLANOpen . . . 21
    - 4.4.2.7 ThermoLANRead . . . 22
    - 4.4.2.8 ThermoLANWrite . . . 22

Chapter 1

ThermoLAN dll Interface Specification

About

The purpose of the ThermoLAN.dll dynamic link library is to make interface to the Thermo microplate instruments easy and to offer a similar interface to the instrument as the ThermoUSB.dll library offers. The user of the ThermoLAN.dll does not have to know the details of serial port communication.

Confidential

Chapter 2

Using ThermoLAN

2.1 About

ThermoLAN.dll is a generic library for communicating with several different Thermo Scientific microplate instruments via a Ethernet port. To communicate with an instrument you use the exported functions of ThermoLAN.dll.

Depending on your project setup, you may find useful a couple of other files which are also provided. The header file ThermoLAN.h contains the prototypes of the exported functions and definitions of constant values used by the dll. File ThermoLAN.lib contains information about the dll the linker uses to add references to the library in the executable. This way the dll is automatically loaded and the exported functions of the library can be called as easy as the functions in the code using the dll.

2.2 Exported functions

- ThermoLANListDevices
- ThermoLANOpen
- ThermoLANClose
- ThermoLANWrite
- ThermoLANRead
- ThermoLANAttachEvent
- ThermoLANAbort
- ThermoLANGetError

2.2.1 ThermoLANListDevices

Search for existing instruments. Full declaration: ThermoLANListDevices().

Parameters

instrumentName	Name of the instrument. Only instruments with a matching name are returned in the list. May be NULL, which matches all instruments.
serialNumber	Pointer to the serial number string of the instrument. Only instruments with a matching serial number are returned. May be NULL, which matches any serial number.
buf	Buffer to list found instruments to.
bufSize	Size of the buffer.

Returns

The size in bytes of the complete list of found devices. If the returned value is equal or smaller than the given buffer size, the buffer contains the complete list of devices found on the LAN. If the returned value is higher than the given buffer size, no data is returned and the caller must call the function again with big enough buffer. Return value 0 means that no instruments were found.

On success, the caller's buffer contains zero terminated strings with a combined length of the return value. The string terminating zeros are included in the length.

For each found instrument, the first string is the instrument IPv4 address and the TCP port number it is listening, enclosed in square brackets, e.g. [10.32.196.210:49536].

The IP address string is followed by the WS-Discovery match strings, usually 3 of them. The first one is always 'Thermo-Device', the second one is the instrument name and the third one the instrument serial number string.

If the match strings are followed by a string in angle brackets, e.g. <10.32.196.154:57403>, it means that the instrument is currently connected that IP address and TCP port, and trying to connect to that instrument with function ThermoLANOpen() will fail. If there is no string in square brackets, the instrument will accept a connection.

2.2.2 ThermoLANOpen

Search for the requested instrument and open a communication channel to it. Full declaration: ThermoLANOpen().

Parameters

instrumentName	Name of the instrument.
serialNumber	Pointer to the serial number string of the instrument. This must match with the actual serial number of the instrument for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first devicee with matching instrument name.
timeout	Timeout which is used to search the instrument. If 0 is given then WS-Discovery will use default 4 sec timeout.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function must be called first before using any other functions in the library. Only one connection per instrument is allowed.

To find the requested instrument on the LAN, the WS-Discovery protocol is used. The instrument responds with a Probe Match to a Probe message if all Target Service strings of the <Types> element of the Probe match the Target Service strings of the instrument. One of the strings, "ThermoDevice", is common to all instruments. Other strings are the instrument name string and the serial number string. Due to Windows limitations, a "SN_" prefix is added to the serial number string.

The <XAddrs> element of the Probe Match response contains the IP address of the instrument and the TCP port number it is listening to.

2.2.3 ThermoLANClose

Close an LAN connection. Full declaration: ThermoLANClose().

Parameters

hConn	Connection handle returned form a call to ThermoLANOpen()

Closes the communication channel to the instrument. This function should be called when the application has finished communicating with the instrument. It is not possible to open a communication channel to the instrument while the previous channel is still open. When the dll is unloaded from the memory, all channels still open are automatically closed.

2.2.4 ThermoLANWrite

Write a string to the transmit buffer. Full declaration: ThermoLANWrite().

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().
buf	Buffer containing the ASCIIZ string to send.

Returns

TRUE if the string was written to the transmit buffer, else FALSE.

If the whole string does not fit in the transmit buffer, writes nothing and returns FALSE. If FALSE is returned, the application can call function ThermoLANGetError() to determine if the transmit buffer was full or is some other error occurred. In case of transmit buffer full, ThermoLANGetError() returns ERROR_SUCCESS and the application may retry sending at a later time.

2.2.5 ThermoLANRead

Read a received response line. Full declaration: ThermoLANRead().

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().
buf	Pointer to a buffer to receive the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line was retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

2.2.6 ThermoLANAttachEvent

Attach an event object to a LAN connection. Full declaration: ThermoLANAttachEvent().

Parameters

hConn	Connection handle returned form a call to ThermoLANOpen().
evCode	Event(s) to signal.
object	Handle of an event object.

Returns

TRUE on success, else FALSE.

Parameter 'evCode' may be any combination of THERMOLAN_RECEIVE, THERMOLAN_TRANSMIT and THERMOLAN-_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOLAN_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. Therefore upon receiving this event the application should repeatedly call ThermoLANRead() as long it returns TRUE.

The THERMOLAN_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOLAN_TRANSMIT event is received, function ThermoLANWrite() will never fail.

The THERMOLAN_ERROR event is signaled when the dll detects an error, for example trying to send data when the TCP connection has been closed by the instrument side. If the application wants to continue communication with the instrument after an error, it should close and reopen the connection.

2.2.7 ThermoLANAbort

Send Abort command to the instrument. Full declaration: ThermoLANAbort().

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().

The LAN Abort sequence is this:

- Send an UDP message containing text "Abort" (without quotes or newline) to the same UDP port number as is used for the TCP and wait a moment for an identical response from the instrument. The response is sent over the same UDP port.
- Retry two times more at half second interval if no response.
- If a response received or no response after retries, send Abort character (0x1B) to the TCP port.

2.2.8 ThermoLANGetError

Get the last error code recorded for the communication channel. Full declaration: ThermoLANGetError().

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().

Returns

One of the Windows system error codes, ERROR_SUCCESS if no error has occurred.

The recorded error code is automatically cleared when this function is called. ERROR_INVALID_HANDLE is returned if the connection handle is invalid.

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

api.h
: Part of ThermoLAN dll documentation 15

mainpage.h
: Start page of ThermoLAN.dll Design Description 15

ThermoLANWrapper.cpp
: Implements ethernet interface to Thermo microplate instruments 15

ThermoLANWrapper.h
: Functions exported from ThermoLAN.dll 19

Chapter 4

File Documentation

4.1 api.h File Reference

Part of ThermoLAN dll documentation.

4.2 mainpage.h File Reference

Start page of ThermoLAN.dll Design Description.

4.3 ThermoLANWrapper.cpp File Reference

Implements ethernet interface to Thermo microplate instruments.

Functions

- void ∗ ThermoLANOpen (char ∗instrumentName, char ∗serialNumber, int timeout)

  Search for the requested instrument and open a communication channel to it.
- DWORD ThermoLANListDevices (char ∗instrumentName, char ∗serialNumber, char ∗buf, unsigned long bufSize)

  Search for existing instruments.
- void ThermoLANClose (void ∗hConn)

  Close an LAN connection.
- BOOL __stdcall ThermoLANWrite (void ∗hConn, const char ∗str)

  Write a string to the transmit buffer.
- BOOL __stdcall ThermoLANRead (void ∗hConn, char ∗str, unsigned long bufSize)

  Read a received response line.
- BOOL __stdcall ThermoLANAttachEvent (void ∗hConn, unsigned int evCode, void ∗receiver)

  Attach an event object to a LAN connection.
- void ThermoLANAbort (void ∗hConn)

  Send Abort command to the instrument.
- DWORD ThermoLANGetError (void ∗hConn)

  Get the last error code recorded for the communication channel.

4.3.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

Definition in file ThermoLANWrapper.cpp.

4.3.2 Function Documentation

4.3.2.1 void ThermoLANAbort (void * *hConn*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().

The LAN Abort sequence is this:

- Send an UDP message containing text "Abort" (without quotes or newline) to the same UDP port number as is used for the TCP and wait a moment for an identical response from the instrument. The response is sent over the same UDP port.
- Retry two times more at half second interval if no response.
- If a response received or no response after retries, send Abort character (0x1B) to the TCP port.

Definition at line 379 of file ThermoLANWrapper.cpp.

4.3.2.2 BOOL __stdcall ThermoLANAttachEvent (void * *hConn,* unsigned int *evCode,* void * *receiver*)

Parameters

hConn	Connection handle returned form a call to ThermoLANOpen().
evCode	Event(s) to signal.
object	Handle of an event object.

Returns

TRUE on success, else FALSE.

Parameter 'evCode' may be any combination of THERMOLAN_RECEIVE, THERMOLAN_TRANSMIT and THERMOLAN-_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOLAN_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. Therefore upon receiving this event the application should repeatedly call ThermoLANRead() as long it returns TRUE.

The THERMOLAN_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOLAN_TRANSMIT event is received, function ThermoLANWrite() will never fail.

The THERMOLAN_ERROR event is signaled when the dll detects an error, for example trying to send data when the TCP connection has been closed by the instrument side. If the application wants to continue communication with the instrument after an error, it should close and reopen the connection.

Definition at line 354 of file ThermoLANWrapper.cpp.

4.3.2.3 void ThermoLANClose (void * *hConn*)

Parameters

hConn	Connection handle returned form a call to ThermoLANOpen()

Closes the communication channel to the instrument. This function should be called when the application has finished communicating with the instrument. It is not possible to open a communication channel to the instrument while the previous channel is still open. When the dll is unloaded from the memory, all channels still open are automatically closed.

Definition at line 234 of file ThermoLANWrapper.cpp.

4.3.2.4 DWORD ThermoLANGetError (void * *hConn*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().

Returns

One of the Windows system error codes, ERROR_SUCCESS if no error has occurred.

The recorded error code is automatically cleared when this function is called. ERROR_INVALID_HANDLE is returned if the connection handle is invalid.

Definition at line 400 of file ThermoLANWrapper.cpp.

4.3.2.5 DWORD ThermoLANListDevices (char * *instrumentName,* char * *serialNumber,* char * *buf,* unsigned long *bufSize*)

Parameters

instrumentName	Name of the instrument. Only instruments with a matching name are returned in the list. May be NULL, which matches all instruments.
serialNumber	Pointer to the serial number string of the instrument. Only instruments with a matching serial number are returned. May be NULL, which matches any serial number.
buf	Buffer to list found instruments to.
bufSize	Size of the buffer.

Returns

The size in bytes of the complete list of found devices. If the returned value is equal or smaller than the given buffer size, the buffer contains the complete list of devices found on the LAN. If the returned value is higher than the given buffer size, no data is returned and the caller must call the function again with big enough buffer. Return value 0 means that no instruments were found.

On success, the caller's buffer contains zero terminated strings with a combined length of the return value. The string terminating zeros are included in the length.

For each found instrument, the first string is the instrument IPv4 address and the TCP port number it is listening, enclosed in square brackets, e.g. [10.32.196.210:49536].

The IP address string is followed by the WS-Discovery match strings, usually 3 of them. The first one is always 'Thermo-Device', the second one is the instrument name and the third one the instrument serial number string.

If the match strings are followed by a string in angle brackets, e.g. <10.32.196.154:57403>, it means that the instrument is currently connected that IP address and TCP port, and trying to connect to that instrument with function ThermoLANOpen() will fail. If there is no string in square brackets, the instrument will accept a connection.

Definition at line 213 of file ThermoLANWrapper.cpp.

4.3.2.6 void* ThermoLANOpen (char * *instrumentName,* char * *serialNumber,* int *timeout*)

Parameters

instrumentName	Name of the instrument.
serialNumber	Pointer to the serial number string of the instrument. This must match with the actual serial number of the instrument for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first devicee with matching instrument name.
timeout	Timeout which is used to search the instrument. If 0 is given then WS-Discovery will use default 4 sec timeout.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function must be called first before using any other functions in the library. Only one connection per instrument is allowed.

To find the requested instrument on the LAN, the WS-Discovery protocol is used. The instrument responds with a Probe Match to a Probe message if all Target Service strings of the <Types> element of the Probe match the Target Service strings of the instrument. One of the strings, "ThermoDevice", is common to all instruments. Other strings are the instrument name string and the serial number string. Due to Windows limitations, a "SN_" prefix is added to the serial number string.

The <XAddrs> element of the Probe Match response contains the IP address of the instrument and the TCP port number it is listening to.

Definition at line 161 of file ThermoLANWrapper.cpp.

4.3.2.7 BOOL __stdcall ThermoLANRead (void * *hConn,* char * *str,* unsigned long *bufSize*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().
buf	Pointer to a buffer to receive the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line was retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

Definition at line 299 of file ThermoLANWrapper.cpp.

4.3.2.8 BOOL __stdcall ThermoLANWrite (void * *hConn,* const char * *str*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().
buf	Buffer containing the ASCIIZ string to send.

Returns

TRUE if the string was written to the transmit buffer, else FALSE.

If the whole string does not fit in the transmit buffer, writes nothing and returns FALSE. If FALSE is returned, the application can call function ThermoLANGetError() to determine if the transmit buffer was full or is some other error occurred. In case of transmit buffer full, ThermoLANGetError() returns ERROR_SUCCESS and the application may retry sending at a later time.

Definition at line 260 of file ThermoLANWrapper.cpp.

4.4 ThermoLANWrapper.h File Reference

Functions exported from ThermoLAN.dll.

Functions

- void * ThermoLANOpen (char *instrumentName, char *serialNumber, int timeout)

  Search for the requested instrument and open a communication channel to it.
- DWORD ThermoLANListDevices (char *instrumentName, char *serialNumber, char *buf, unsigned long bufSize)

  Search for existing instruments.
- void ThermoLANClose (void *hConn)

  Close an LAN connection.
- BOOL __stdcall ThermoLANWrite (void *hConn, const char *str)

  Write a string to the transmit buffer.
- BOOL __stdcall ThermoLANRead (void *hConn, char *str, unsigned long bufSize)

  Read a received response line.
- BOOL __stdcall ThermoLANAttachEvent (void *hConn, unsigned int evCode, void *receiver)

  Attach an event object to a LAN connection.
- void ThermoLANAbort (void *hConn)

  Send Abort command to the instrument.
- DWORD ThermoLANGetError (void *hConn)

  Get the last error code recorded for the communication channel.

4.4.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

Ethernet interface to Thermo microplate instruments.

Definition in file ThermoLANWrapper.h.

4.4.2 Function Documentation

4.4.2.1 void ThermoLANAbort (void * *hConn*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().

The LAN Abort sequence is this:

- Send an UDP message containing text "Abort" (without quotes or newline) to the same UDP port number as is used for the TCP and wait a moment for an identical response from the instrument. The response is sent over the same

UDP port.

- Retry two times more at half second interval if no response.
- If a response received or no response after retries, send Abort character (0x1B) to the TCP port.

Definition at line 379 of file ThermoLANWrapper.cpp.

4.4.2.2 BOOL __stdcall ThermoLANAttachEvent (void * *hConn*, unsigned int *evCode*, void * *receiver*)

Parameters

hConn	Connection handle returned form a call to ThermoLANOpen().
evCode	Event(s) to signal.
object	Handle of an event object.

Returns

TRUE on success, else FALSE.

Parameter 'evCode' may be any combination of THERMOLAN_RECEIVE, THERMOLAN_TRANSMIT and THERMOLAN-_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOLAN_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. Therefore upon receiving this event the application should repeatedly call ThermoLANRead() as long it returns TRUE.

The THERMOLAN_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOLAN_TRANSMIT event is received, function ThermoLANWrite() will never fail.

The THERMOLAN_ERROR event is signaled when the dll detects an error, for example trying to send data when the TCP connection has been closed by the instrument side. If the application wants to continue communication with the instrument after an error, it should close and reopen the connection.

Definition at line 354 of file ThermoLANWrapper.cpp.

4.4.2.3 void ThermoLANClose (void * *hConn*)

Parameters

hConn	Connection handle returned form a call to ThermoLANOpen()

Closes the communication channel to the instrument. This function should be called when the application has finished communicating with the instrument. It is not possible to open a communication channel to the instrument while the previous channel is still open. When the dll is unloaded from the memory, all channels still open are automatically closed.

Definition at line 234 of file ThermoLANWrapper.cpp.

4.4.2.4 DWORD ThermoLANGetError (void * *hConn*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().

Returns

One of the Windows system error codes, ERROR_SUCCESS if no error has occurred.

The recorded error code is automatically cleared when this function is called. ERROR_INVALID_HANDLE is returned if the connection handle is invalid.

Definition at line 400 of file ThermoLANWrapper.cpp.

4.4.2.5 DWORD ThermoLANListDevices (char * *instrumentName,* char * *serialNumber,* char * *buf,* unsigned long *bufSize*)

Parameters

instrumentName	Name of the instrument. Only instruments with a matching name are returned in the list. May be NULL, which matches all instruments.
serialNumber	Pointer to the serial number string of the instrument. Only instruments with a matching serial number are returned. May be NULL, which matches any serial number.
buf	Buffer to list found instruments to.
bufSize	Size of the buffer.

Returns

The size in bytes of the complete list of found devices. If the returned value is equal or smaller than the given buffer size, the buffer contains the complete list of devices found on the LAN. If the returned value is higher than the given buffer size, no data is returned and the caller must call the function again with big enough buffer. Return value 0 means that no instruments were found.

On success, the caller's buffer contains zero terminated strings with a combined length of the return value. The string terminating zeros are included in the length.

For each found instrument, the first string is the instrument IPv4 address and the TCP port number it is listening, enclosed in square brackets, e.g. [10.32.196.210:49536].

The IP address string is followed by the WS-Discovery match strings, usually 3 of them. The first one is always 'Thermo-Device', the second one is the instrument name and the third one the instrument serial number string.

If the match strings are followed by a string in angle brackets, e.g. <10.32.196.154:57403>, it means that the instrument is currently connected that IP address and TCP port, and trying to connect to that instrument with function ThermoLANOpen() will fail. If there is no string in square brackets, the instrument will accept a connection.

Definition at line 213 of file ThermoLANWrapper.cpp.

4.4.2.6 void* ThermoLANOpen (char * *instrumentName,* char * *serialNumber,* int *timeout*)

Parameters

instrumentName	Name of the instrument.
serialNumber	Pointer to the serial number string of the instrument. This must match with the actual serial number of the instrument for the connection to succeed. This parameter may be NULL, in which case the connection is made to the first devicee with matching instrument name.
timeout	Timeout which is used to search the instrument. If 0 is given then WS-Discovery will use default 4 sec timeout.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function must be called first before using any other functions in the library. Only one connection per instrument is allowed.

To find the requested instrument on the LAN, the WS-Discovery protocol is used. The instrument responds with a Probe Match to a Probe message if all Target Service strings of the <Types> element of the Probe match the Target Service strings of the instrument. One of the strings, "ThermoDevice", is common to all instruments. Other strings are the instrument name string and the serial number string. Due to Windows limitations, a "SN_" prefix is added to the serial number string.

The <XAddrs> element of the Probe Match response contains the IP address of the instrument and the TCP port number it is listening to.

Definition at line 161 of file ThermoLANWrapper.cpp.

4.4.2.7 BOOL __stdcall ThermoLANRead (void * *hConn*, char * *str*, unsigned long *bufSize*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().
buf	Pointer to a buffer to receive the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line was retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

Definition at line 299 of file ThermoLANWrapper.cpp.

4.4.2.8 BOOL __stdcall ThermoLANWrite (void * *hConn*, const char * *str*)

Parameters

hConn	Handle of the LAN connection. This must be a handle returned from ThermoLANOpen().
buf	Buffer containing the ASCIIZ string to send.

Returns

TRUE if the string was written to the transmit buffer, else FALSE.

If the whole string does not fit in the transmit buffer, writes nothing and returns FALSE. If FALSE is returned, the application can call function ThermoLANGetError() to determine if the transmit buffer was full or is some other error occurred. In case of transmit buffer full, ThermoLANGetError() returns ERROR_SUCCESS and the application may retry sending at a later time.

Definition at line 260 of file ThermoLANWrapper.cpp.

Index

api.h, 15

mainpage.h, 15

ThermoLANAbort
    ThermoLANWrapper.cpp, 16
    ThermoLANWrapper.h, 19
ThermoLANAttachEvent
    ThermoLANWrapper.cpp, 16
    ThermoLANWrapper.h, 20
ThermoLANClose
    ThermoLANWrapper.cpp, 16
    ThermoLANWrapper.h, 20
ThermoLANGetError
    ThermoLANWrapper.cpp, 17
    ThermoLANWrapper.h, 20
ThermoLANListDevices
    ThermoLANWrapper.cpp, 17
    ThermoLANWrapper.h, 21
ThermoLANOpen
    ThermoLANWrapper.cpp, 17
    ThermoLANWrapper.h, 21
ThermoLANRead
    ThermoLANWrapper.cpp, 18
    ThermoLANWrapper.h, 22
ThermoLANWrapper.cpp, 15
    ThermoLANAbort, 16
    ThermoLANAttachEvent, 16
    ThermoLANClose, 16
    ThermoLANGetError, 17
    ThermoLANListDevices, 17
    ThermoLANOpen, 17
    ThermoLANRead, 18
    ThermoLANWrite, 18
ThermoLANWrapper.h, 19
    ThermoLANAbort, 19
    ThermoLANAttachEvent, 20
    ThermoLANClose, 20
    ThermoLANGetError, 20
    ThermoLANListDevices, 21
    ThermoLANOpen, 21
    ThermoLANRead, 22
    ThermoLANWrite, 22
ThermoLANWrite
    ThermoLANWrapper.cpp, 18
    ThermoLANWrapper.h, 22

Thermo Scientific

ThermoCOM.dll

Interface Specification

Thermo SCIENTIFIC

Contents

1 ThermoCOM dll Interface Specification 1

2 Using ThermoCOM 3
2.1 About . . . 3
2.2 Exported functions . . . 4
2.2.1 ThermoCOMOpen . . . 5
2.2.2 ThermoCOMOpenSimulator . . . 6
2.2.3 ThermoCOMClose . . . 7
2.2.4 ThermoCOMSetParam . . . 8
2.2.5 ThermoCOMAttachEvent . . . 9
2.2.6 ThermoCOMAttachMsg . . . 10
2.2.7 ThermoCOMRead . . . 11
2.2.8 ThermoCOMReadBinary . . . 12
2.2.9 ThermoCOMWrite . . . 13
2.2.10 ThermoCOMWriteBinary . . . 14
2.2.11 ThermoCOMGetError . . . 15
2.2.12 ThermoCOMAbort . . . 16

3 File Index 17
3.1 File List . . . 17

4 File Documentation 19
4.1 api.h File Reference . . . 19
4.2 mainpage.h File Reference . . . 19
4.3 ThermoCOM.c File Reference . . . 19
4.3.1 Detailed Description . . . 20
4.3.2 Function Documentation . . . 20
4.3.2.1 ThermoCOMAbort . . . 20
4.3.2.2 ThermoCOMAttachEvent . . . 20
4.3.2.3 ThermoCOMAttachMsg . . . 21
4.3.2.4 ThermoCOMClose . . . 21
4.3.2.5 ThermoCOMGetError . . . 21
4.3.2.6 ThermoCOMOpen . . . 22
4.3.2.7 ThermoCOMOpenSimulator . . . 22
4.3.2.8 ThermoCOMRead . . . 22
4.3.2.9 ThermoCOMReadBinary . . . 23
4.3.2.10 ThermoCOMSetParam . . . 23
4.3.2.11 ThermoCOMWrite . . . 23
4.3.2.12 ThermoCOMWriteBinary . . . 24
4.4 ThermoCOM.h File Reference . . . 24
4.4.1 Detailed Description . . . 25
4.4.2 Macro Definition Documentation . . . 25
4.4.2.1 HSK_NONE . . . 25
4.4.2.2 HSK_RTSCTS . . . 25
4.4.2.3 HSK_XONXOFF . . . 25
4.4.2.4 THERMOCOM_ERROR . . . 26

4.4.2.5 THERMOCOM_RECEIVE . . . 26
4.4.2.6 THERMOCOM_TRANSMIT . . . 26
4.4.3 Function Documentation . . . 26
4.4.3.1 ThermoCOMAbort . . . 26
4.4.3.2 ThermoCOMAttachEvent . . . 26
4.4.3.3 ThermoCOMAttachMsg . . . 27
4.4.3.4 ThermoCOMClose . . . 27
4.4.3.5 ThermoCOMGetError . . . 27
4.4.3.6 ThermoCOMOpen . . . 28
4.4.3.7 ThermoCOMOpenSimulator . . . 28
4.4.3.8 ThermoCOMRead . . . 28
4.4.3.9 ThermoCOMReadBinary . . . 29
4.4.3.10 ThermoCOMSetParam . . . 29
4.4.3.11 ThermoCOMWrite . . . 29
4.4.3.12 ThermoCOMWriteBinary . . . 30

Chapter 1

ThermoCOM dll Interface Specification

About

The purpose of the ThermoCOM.dll dynamic link library is to make interface to the Thermo microplate instruments easy and to offer a similar interface to the instrument as the ThermoUSB.dll library offers. The user of the ThermoCOM.dll does not have to know the details of serial port communication.

Confidential

Chapter 2

Using ThermoCOM

2.1 About

ThermoCOM.dll is a generic library for communicating with several different Thermo Scientific microplate instruments via a PC serial port. To communicate with an instrument you use the exported functions of ThermoCOM.dll.

Depending on your project setup, you may find useful a couple of other files which are also provided. The header file ThermoCOM.h contains the prototypes of the exported functions and definitions of constant values used by the dll. File ThermoCOM.lib contains information about the dll the linker uses to add references to the library in the executable. This way the dll is automatically loaded and the exported functions of the library can be called as easy as the functions in the code using the dll.

2.2 Exported functions

- ThermoCOMOpen
- ThermoCOMOpenSimulator
- ThermoCOMClose
- ThermoCOMSetParam
- ThermoCOMAttachEvent
- ThermoCOMAttachMsg
- ThermoCOMRead
- ThermoCOMReadBinary
- ThermoCOMWrite
- ThermoCOMWriteBinary
- ThermoCOMGetError
- ThermoCOMAbort

2.2.1 ThermoCOMOpen

Open the requested serial port and check whether an instrument with the requested serial number is connected to the port. Full declaration: ThermoCOMOpen().

Parameters

PortNumber	The Windows serial port number to open.
baud	The baudrate to use.
InstrumentName	The name of the instrument. If not NULL, the name the instrument returns to a version query must match this string.
SerialNumber	The serial number of the instrument. If not NULL, the serial number the instrument returns to a version query must match this string.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function (or ThermoCOMOpenSimulator) must be called first before using any other functions in the library. Only one connection per communication port is allowed.

If either InstrumentName or SerialNumber, or both, are defined, a VER command is sent to the serial port, and these arguments are checked against the returned response. If there is a mismatch, the port is closed and NULL is returned.

If both InstrumentName and SerialNumber are NULL, the VER command is not sent. This makes it possible to open a channel to an instrument which does not support the VER command. In this case it is the responsibility of the application to check what if anything is connected to the serial port.

2.2.2 ThermoCOMOpenSimulator

Open a pipe to a communication port of an instrument simulator. Full declaration: ThermoCOMOpenSimulator().

Parameters

pipeName	Name of the named pipe used for simulating the communication port. The pipe must have been created by the simulated port driver of the instrument simulator.

Returns

Communication handle to the simulator, NULL if the function fails.

This function (or ThermoCOMOpen) must be called first before using any other functions in the library. Only one connection per device is allowed. Standard names of the pipes used for simulating communication ports are:

- Serial port: \0xXXXXCOM
- Debug port: \0xXXXXDBG
- USB port: \0xXXXXUSB
- LAN port: \0xXXXXLAN

The XXXX in all pipe names is the ProductID of the device in hex.

2.2.3 ThermoCOMClose

Close a serial port connection. Full declaration: ThermoCOMClose().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

None.

All open connections are automatically closed when the dll is unloaded, but it is good programming practise to close them explicitly when no longer used.

2.2.4 ThermoCOMSetParam

Set the baud rate of the serial port. Full declaration: ThermoCOMSetParam().

Parameters

hConn	Connection handle returned from a call to ThermoCOMOpen().
baud	The baud rate to set.
handshake	The handshake to set.

Returns

TRUE on success, else FALSE.

It is recommended to use on of the standard baudrates from 110 to 256000. Using a non standard baudrate may lead to too high bit time error for the communication to work.

The handshake must be one of HSK_NONE, HSK_XONXOFF and HSK_RTSCTS. Thermo microplate instruments use the HSK_XONXOFF handshake with the exception of some very old models.

2.2.5 ThermoCOMAttachEvent

Attach an event object to a serial port connection. Full declaration: ThermoCOMAttachEvent().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
evCode	Event(s) to signal.
object	Handle of an event object.

Returns

None

Parameter 'evCode' may be any combination of THERMOCOM_RECEIVE, THERMOCOM_TRANSMIT and THERMOCOM_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOCOM_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoCOMRead() or ThermoCOMReadBinary as long as they return data.

The THERMOCOM_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOCOM_TRANSMIT event is received, functions ThermoCOMWrite() and ThermoCOMWriteBinary() will never fail.

2.2.6 ThermoCOMAttachMsg

Attach an event message to a serial port connection. Full declaration: ThermoCOMAttachMsg().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
evCode	Event(s) for which a message is sent.

Returns

None

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of THERMOCOM_RECEIVE, THERMOCOM_TRANSMIT and THERMOCOM_ERROR. The event code is passed to the application in the wParam member of the message structure.

The THERMOCOM_RECEIVE event is sent whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoCOMRead() or ThermoCOMReadBinary as long as they return data.

The THERMOCOM_TRANSMIT event is sent when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOCOM_TRANSMIT event is received, functions ThermoCOMWrite() and ThermoCOMWriteBinary() will never fail.

2.2.7 ThermoCOMRead

Read a received response line. Full declaration: ThermoCOMRead().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Pointer to buffer receiving the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

2.2.8 ThermoCOMReadBinary

Read received data. Full declaration: ThermoCOMReadBinary().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Pointer to buffer receiving the data.
bufsize	Size of the buffer in bytes.

Returns

Number of bytes copied to the user buffer.

Unlike function ThermoCOMRead(), this function returns all data received from the instrument without doing any interpretation on it. Note however that the XON and XOFF characters are filtered by the serial port driver when XON/XOFF flow control is used. You can control the number of bytes returned with the bufsize parameter.

If the return value is less than the given bufsize parameter, there is no more data.

2.2.9 ThermoCOMWrite

Write a string to the transmit buffer. Full declaration: ThermoCOMWrite().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Buffer containing the NUL terminated string to send.

Returns

TRUE if string written to the transmit buffer, else FALSE.

When this function returns, the string is not yet sent to the instrument but is queued for sending. Memory for transmit data is dynamically allocated and the only reason for this function returning FALSE is that no more memory is available. If FALSE is returned, the string is not even partially written to transmit buffer and the application should retry with the whole string at a later time.

2.2.10 ThermoCOMWriteBinary

Write binary data to the transmit buffer. Full declaration: ThermoCOMWriteBinary().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Buffer containing the data to send.
count	Number of bytes to send from the buffer.

Returns

TRUE if data written to the buffer, else FALSE.

When this function returns, the data is not yet sent to the instrument but is queued for sending. Memory for transmit data is dynamically allocated and the only reason for this function returning FALSE is that no more memory is available. If FALSE is returned, the data is not even partially written to transmit buffer and the application should retry with all data at a later time.

2.2.11 ThermoCOMGetError

Return the error code stored to the connection structure. Full declaration: ThermoCOMGetError().

2.2.12 ThermoCOMAbort

Send Abort command to instrument. Full declaration: ThermoCOMAbort().

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

None

This function first flushes the transmit buffer and also cancels transmission of any data already in the serial port driver. It then sends characters ESC (Abort) and XON to the serial port.

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

api.h
: Part of ThermoCOM dll documentation 19

mainpage.h
: Start page of ThermoCOM.dll Design Description 19

ThermoCOM.c
: Implements serial port interface to Thermo microplate instruments 19

ThermoCOM.h
: Functions exported from ThermoCOM.dll 24

Chapter 4

File Documentation

4.1 api.h File Reference

Part of ThermoCOM dll documentation.

4.2 mainpage.h File Reference

Start page of ThermoCOM.dll Design Description.

4.3 ThermoCOM.c File Reference

Implements serial port interface to Thermo microplate instruments.

Functions

- HANDLE WINAPI ThermoCOMOpen (DWORD PortNumber, DWORD baud, LPCSTR InstrumentName, LPCSTR SerialNumber)

  Open the requested serial port and check whether an instrument with the requested serial number is connected to the port.
- HANDLE WINAPI ThermoCOMOpenSimulator (LPCSTR pipeName)

  Open a pipe to a communication port of an instrument simulator.
- BOOL WINAPI ThermoCOMSetParam (HANDLE hConn, DWORD baud, DWORD handshake)

  Set the baud rate of the serial port.
- void WINAPI ThermoCOMClose (HANDLE hComm)

  Close a serial port connection.
- BOOL WINAPI ThermoCOMAttachEvent (HANDLE hConn, UINT evCode, HANDLE object)

  Attach an event object to a serial port connection.
- BOOL WINAPI ThermoCOMAttachMsg (HANDLE hConn, HWND hWnd, UINT msg, UINT evCode)

  Attach an event message to a serial port connection.
- BOOL WINAPI ThermoCOMRead (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received response line.
- DWORD WINAPI ThermoCOMReadBinary (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read received data.
- BOOL WINAPI ThermoCOMWrite (HANDLE hConn, LPCSTR buf)

  Write a string to the transmit buffer.

- BOOL WINAPI ThermoCOMWriteBinary (HANDLE hConn, LPCSTR buf, DWORD count)

  Write binary data to the transmit buffer.
- void WINAPI ThermoCOMAbort (HANDLE hConn)

  Send Abort command to instrument.
- DWORD WINAPI ThermoCOMGetError (HANDLE hConn)

  Return the error code stored to the connection structure.

4.3.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

Definition in file ThermoCOM.c.

4.3.2 Function Documentation

4.3.2.1 void WINAPI ThermoCOMAbort (HANDLE *hConn*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

None

This function first flushes the transmit buffer and also cancels transmission of any data already in the serial port driver. It then sends characters ESC (Abort) and XON to the serial port.

Definition at line 1843 of file ThermoCOM.c.

4.3.2.2 BOOL WINAPI ThermoCOMAttachEvent (HANDLE *hConn,* UINT *evCode,* HANDLE *object*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
evCode	Event(s) to signal.
object	Handle of an event object.

Returns

None

Parameter 'evCode' may be any combination of THERMOCOM_RECEIVE, THERMOCOM_TRANSMIT and THERMOCOM_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOCOM_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoCOMRead() or ThermoCOMReadBinary as long as they return data.

The THERMOCOM_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOCOM_TRANSMIT event is received, functions ThermoCOMWrite() and ThermoCOMWriteBinary() will never fail.

Definition at line 1464 of file ThermoCOM.c.

4.3.2.3 BOOL WINAPI ThermoCOMAttachMsg (HANDLE *hConn*, HWND *hWnd*, UINT *msg*, UINT *evCode*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
evCode	Event(s) for which a message is sent.

Returns

None

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of THERMOCOM_RECEIVE, THERMOCOM_TRANSMIT and THERMOCOM_ERROR. The event code is passed to the application in the wParam member of the message structure.

The THERMOCOM_RECEIVE event is sent whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoCOMRead() or ThermoCOMReadBinary as long as they return data.

The THERMOCOM_TRANSMIT event is sent when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOCOM_TRANSMIT event is received, functions ThermoCOMWrite() and ThermoCOMWriteBinary() will never fail.

Definition at line 1518 of file ThermoCOM.c.

4.3.2.4 void WINAPI ThermoCOMClose (HANDLE *hComm*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

None.

All open connections are automatically closed when the dll is unloaded, but it is good programming practise to close them explicitly when no longer used.

Definition at line 1377 of file ThermoCOM.c.

4.3.2.5 DWORD WINAPI ThermoCOMGetError (HANDLE *hConn*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

Windows system error code.

The application may call this function after receiving the THERMOCOM_ERROR event. The returned error code may or may not give a clue about what the actual problem is. If an error is reported, data is already lost and the best action for the application to do is to close the connection and then try to reopen it.

A call to ThermoCOMGetError() resets the stored error code to ERROR_SUCCESS.

Definition at line 1890 of file ThermoCOM.c.

4.3.2.6 HANDLE WINAPI ThermoCOMOpen (DWORD *PortNumber,* DWORD *baud,* LPCSTR *InstrumentName,* LPCSTR *SerialNumber*)

Parameters

PortNumber	The Windows serial port number to open.
baud	The baudrate to use.
InstrumentName	The name of the instrument. If not NULL, the name the instrument returns to a version query must match this string.
SerialNumber	The serial number of the instrument. If not NULL, the serial number the instrument returns to a version query must match this string.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function (or ThermoCOMOpenSimulator) must be called first before using any other functions in the library. Only one connection per communication port is allowed.

If either InstrumentName or SerialNumber, or both, are defined, a VER command is sent to the serial port, and these arguments are checked against the returned response. If there is a mismatch, the port is closed and NULL is returned.

If both InstrumentName and SerialNumber are NULL, the VER command is not sent. This makes it possible to open a channel to an instrument which does not support the VER command. In this case it is the responsibility of the application to check what if anything is connected to the serial port.

Definition at line 1237 of file ThermoCOM.c.

4.3.2.7 HANDLE WINAPI ThermoCOMOpenSimulator (LPCSTR *pipeName*)

Parameters

pipeName	Name of the named pipe used for simulating the communication port. The pipe must have been created by the simulated port driver of the instrument simulator.

Returns

Communication handle to the simulator, NULL if the function fails.

This function (or ThermoCOMOpen) must be called first before using any other functions in the library. Only one connection per device is allowed. Standard names of the pipes used for simulating communication ports are:

- Serial port: \0xXXXXCOM
- Debug port: \0xXXXXDBG
- USB port: \0xXXXXUSB
- LAN port: \0xXXXXLAN

The XXXX in all pipe names is the ProductID of the device in hex.

Definition at line 1288 of file ThermoCOM.c.

4.3.2.8 BOOL WINAPI ThermoCOMRead (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Pointer to buffer receiving the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

Definition at line 1557 of file ThermoCOM.c.

4.3.2.9 DWORD WINAPI ThermoCOMReadBinary (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Pointer to buffer receiving the data.
bufsize	Size of the buffer in bytes.

Returns

Number of bytes copied to the user buffer.

Unlike function ThermoCOMRead(), this function returns all data received from the instrument without doing any interpretation on it. Note however that the XON and XOFF characters are filtered by the serial port driver when XON/XOFF flow control is used. You can control the number of bytes returned with the bufsize parameter.

If the return value is less than the given bufsize parameter, there is no more data.

Definition at line 1733 of file ThermoCOM.c.

4.3.2.10 BOOL WINAPI ThermoCOMSetParam (HANDLE *hConn,* DWORD *baud,* DWORD *handshake*)

Parameters

hConn	Connection handle returned from a call to ThermoCOMOpen().
baud	The baud rate to set.
handshake	The handshake to set.

Returns

TRUE on success, else FALSE.

It is recommended to use on of the standard baudrates from 110 to 256000. Using a non standard baudrate may lead to too high bit time error for the communication to work.

The handshake must be one of HSK_NONE, HSK_XONXOFF and HSK_RTSCTS. Thermo microplate instruments use the HSK_XONXOFF handshake with the exception of some very old models.

Definition at line 1347 of file ThermoCOM.c.

4.3.2.11 BOOL WINAPI ThermoCOMWrite (HANDLE *hConn,* LPCSTR *buf*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Buffer containing the NUL terminated string to send.

Returns

TRUE if string written to the transmit buffer, else FALSE.

When this function returns, the string is not yet sent to the instrument but is queued for sending. Memory for transmit data is dynamically allocated and the only reason for this function returning FALSE is that no more memory is available. If FALSE is returned, the string is not even partially written to transmit buffer and the application should retry with the whole string at a later time.

Definition at line 1781 of file ThermoCOM.c.

4.3.2.12 BOOL WINAPI ThermoCOMWriteBinary (HANDLE *hConn*, LPCSTR *buf*, DWORD *count*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Buffer containing the data to send.
count	Number of bytes to send from the buffer.

Returns

TRUE if data written to the buffer, else FALSE.

When this function returns, the data is not yet sent to the instrument but is queued for sending. Memory for transmit data is dynamically allocated and the only reason for this function returning FALSE is that no more memory is available. If FALSE is returned, the data is not even partially written to transmit buffer and the application should retry with all data at a later time.

Definition at line 1805 of file ThermoCOM.c.

4.4 ThermoCOM.h File Reference

Functions exported from ThermoCOM.dll.

Macros

- #define THERMOCOM_RECEIVE 1

  Data received event.
- #define THERMOCOM_TRANSMIT 2

  Data transmitted event.
- #define THERMOCOM_ERROR 4

  Fatal error event.
- #define HSK_NONE 0

  Do not use any handshake on serial port.
- #define HSK_XONXOFF 1
- #define HSK_RTSCTS 2

  Use hardware handshake on serial port.

Functions

- DllExport HANDLE WINAPI ThermoCOMOpen (DWORD PortNumber, DWORD baud, LPCSTR InstrumentName, LPCSTR SerialNumber)

  Open the requested serial port and check whether an instrument with the requested serial number is connected to the port.

- DllExport HANDLE WINAPI ThermoCOMOpenSimulator (LPCSTR pipeName)

  Open a pipe to a communication port of an instrument simulator.
- DllExport void WINAPI ThermoCOMClose (HANDLE hComm)

  Close a serial port connection.
- DllExport BOOL WINAPI ThermoCOMSetParam (HANDLE hConn, DWORD baud, DWORD handshake)

  Set the baud rate of the serial port.
- DllExport BOOL WINAPI ThermoCOMAttachEvent (HANDLE hConn, UINT evCode, HANDLE object)

  Attach an event object to a serial port connection.
- DllExport BOOL WINAPI ThermoCOMAttachMsg (HANDLE hConn, HWND hWnd, UINT msg, UINT evCode)

  Attach an event message to a serial port connection.
- DllExport BOOL WINAPI ThermoCOMRead (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read a received response line.
- DllExport DWORD WINAPI ThermoCOMReadBinary (HANDLE hConn, LPSTR buf, DWORD bufsize)

  Read received data.
- DllExport BOOL WINAPI ThermoCOMWrite (HANDLE hConn, LPCSTR buf)

  Write a string to the transmit buffer.
- DllExport BOOL WINAPI ThermoCOMWriteBinary (HANDLE hConn, LPCSTR buf, DWORD count)

  Write binary data to the transmit buffer.
- DllExport DWORD WINAPI ThermoCOMGetError (HANDLE hConn)

  Return the error code stored to the connection structure.
- DllExport void WINAPI ThermoCOMAbort (HANDLE hConn)

  Send Abort command to instrument.

4.4.1 Detailed Description

Note

Copyright by Thermo Fisher Scientific Oy 2015

Serial port interface to Thermo microplate instruments.

Definition in file ThermoCOM.h.

4.4.2 Macro Definition Documentation

4.4.2.1 #define HSK_NONE 0

Definition at line 47 of file ThermoCOM.h.

4.4.2.2 #define HSK_RTSCTS 2

Definition at line 65 of file ThermoCOM.h.

4.4.2.3 #define HSK_XONXOFF 1

Definition at line 58 of file ThermoCOM.h.

4.4.2.4 #define THERMOCOM_ERROR 4

What the application can do in case of an error is to first call ThermoCOMClose() and then try to reopen with ThermoCOMOpen().

Definition at line 40 of file ThermoCOM.h.

4.4.2.5 #define THERMOCOM_RECEIVE 1

Definition at line 22 of file ThermoCOM.h.

4.4.2.6 #define THERMOCOM_TRANSMIT 2

Definition at line 29 of file ThermoCOM.h.

4.4.3 Function Documentation

4.4.3.1 DllExport void WINAPI ThermoCOMAbort (HANDLE *hConn*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

None

This function first flushes the transmit buffer and also cancels transmission of any data already in the serial port driver. It then sends characters ESC (Abort) and XON to the serial port.

Definition at line 1843 of file ThermoCOM.c.

4.4.3.2 DllExport BOOL WINAPI ThermoCOMAttachEvent (HANDLE *hConn,* UINT *evCode,* HANDLE *object*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
evCode	Event(s) to signal.
object	Handle of an event object.

Returns

None

Parameter 'evCode' may be any combination of THERMOCOM_RECEIVE, THERMOCOM_TRANSMIT and THERMOCOM_ERROR. The event object is set to signaled state whenever any of the selected event(s) occurs.

Parameter 'object' is a handle of a Windows event object. If NULL, the selected events will not be signaled.

This function may be called repeatedly to set up a different event object for each event.

The THERMOCOM_RECEIVE event is signaled whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoCOMRead() or ThermoCOMReadBinary as long as they return data.

The THERMOCOM_TRANSMIT event is signaled when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOCOM_TRANSMIT event is received, functions ThermoCOMWrite() and ThermoCOMWriteBinary() will never fail.

Definition at line 1464 of file ThermoCOM.c.

4.4.3.3 DllExport BOOL WINAPI ThermoCOMAttachMsg (HANDLE *hConn,* HWND *hWnd,* UINT *msg,* UINT *evCode*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
hWnd	Handle of the window to receive the message.
msg	Message id of the message to send.
evCode	Event(s) for which a message is sent.

Returns

None

A message with message id 'msg' is sent to window 'hWnd' whenever any of the event(s) selected with parameter 'evCode' occurs. Parameter 'evCode' may be any combination of THERMOCOM_RECEIVE, THERMOCOM_TRANSMIT and THERMOCOM_ERROR. The event code is passed to the application in the wParam member of the message structure.

The THERMOCOM_RECEIVE event is sent whenever something is written to the receive buffer. There is no quarantee that a whole response line is received or that only one response line is received. therefore upon receiving this event the application should call ThermoCOMRead() or ThermoCOMReadBinary as long as they return data.

The THERMOCOM_TRANSMIT event is sent when the last data from the transmit buffer is sent. It can be used as flow control: If new data is not sent until THERMOCOM_TRANSMIT event is received, functions ThermoCOMWrite() and ThermoCOMWriteBinary() will never fail.

Definition at line 1518 of file ThermoCOM.c.

4.4.3.4 DllExport void WINAPI ThermoCOMClose (HANDLE *hComm*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

None.

All open connections are automatically closed when the dll is unloaded, but it is good programming practise to close them explicitly when no longer used.

Definition at line 1377 of file ThermoCOM.c.

4.4.3.5 DllExport DWORD WINAPI ThermoCOMGetError (HANDLE *hConn*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().

Returns

Windows system error code.

The application may call this function after receiving the THERMOCOM_ERROR event. The returned error code may or may not give a clue about what the actual problem is. If an error is reported, data is already lost and the best action for the application to do is to close the connection and then try to reopen it.

A call to ThermoCOMGetError() resets the stored error code to ERROR_SUCCESS.

Definition at line 1890 of file ThermoCOM.c.

4.4.3.6 DllExport HANDLE WINAPI ThermoCOMOpen (DWORD *PortNumber,* DWORD *baud,* LPCSTR *InstrumentName,* LPCSTR *SerialNumber*)

Parameters

PortNumber	The Windows serial port number to open.
baud	The baudrate to use.
InstrumentName	The name of the instrument. If not NULL, the name the instrument returns to a version query must match this string.
SerialNumber	The serial number of the instrument. If not NULL, the serial number the instrument returns to a version query must match this string.

Returns

A handle to the opened communication channel. This handle must be passed to subsequent calls to the other functions in this library. If the channel could not be opened, NULL is returned.

This function (or ThermoCOMOpenSimulator) must be called first before using any other functions in the library. Only one connection per communication port is allowed.

If either InstrumentName or SerialNumber, or both, are defined, a VER command is sent to the serial port, and these arguments are checked against the returned response. If there is a mismatch, the port is closed and NULL is returned.

If both InstrumentName and SerialNumber are NULL, the VER command is not sent. This makes it possible to open a channel to an instrument which does not support the VER command. In this case it is the responsibility of the application to check what if anything is connected to the serial port.

Definition at line 1237 of file ThermoCOM.c.

4.4.3.7 DllExport HANDLE WINAPI ThermoCOMOpenSimulator (LPCSTR *pipeName*)

Parameters

pipeName	Name of the named pipe used for simulating the communication port. The pipe must have been created by the simulated port driver of the instrument simulator.

Returns

Communication handle to the simulator, NULL if the function fails.

This function (or ThermoCOMOpen) must be called first before using any other functions in the library. Only one connection per device is allowed. Standard names of the pipes used for simulating communication ports are:

- Serial port: \0xXXXXCOM
- Debug port: \0xXXXXDBG
- USB port: \0xXXXXUSB
- LAN port: \0xXXXXLAN

The XXXX in all pipe names is the ProductID of the device in hex.

Definition at line 1288 of file ThermoCOM.c.

4.4.3.8 DllExport BOOL WINAPI ThermoCOMRead (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Pointer to buffer receiving the response.
bufsize	Size of the buffer in bytes.

Returns

TRUE if a response line retrieved, else FALSE.

Use this function to read full response lines returned from the instrument. The response line is returned without the terminating CRLF. Therefore TRUE may be returned even if the resulting line is empty. The returned line always ends to the NUL character even if FALSE is returned.

If the caller's buffer is too small to hold the whole response line, as much as fits to the buffer is returned and the rest is returned on subsequent call(s).

Definition at line 1557 of file ThermoCOM.c.

4.4.3.9 DllExport DWORD WINAPI ThermoCOMReadBinary (HANDLE *hConn,* LPSTR *buf,* DWORD *bufsize*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Pointer to buffer receiving the data.
bufsize	Size of the buffer in bytes.

Returns

Number of bytes copied to the user buffer.

Unlike function ThermoCOMRead(), this function returns all data received from the instrument without doing any interpretation on it. Note however that the XON and XOFF characters are filtered by the serial port driver when XON/XOFF flow control is used. You can control the number of bytes returned with the bufsize parameter.

If the return value is less than the given bufsize parameter, there is no more data.

Definition at line 1733 of file ThermoCOM.c.

4.4.3.10 DllExport BOOL WINAPI ThermoCOMSetParam (HANDLE *hConn,* DWORD *baud,* DWORD *handshake*)

Parameters

hConn	Connection handle returned from a call to ThermoCOMOpen().
baud	The baud rate to set.
handshake	The handshake to set.

Returns

TRUE on success, else FALSE.

It is recommended to use on of the standard baudrates from 110 to 256000. Using a non standard baudrate may lead to too high bit time error for the communication to work.

The handshake must be one of HSK_NONE, HSK_XONXOFF and HSK_RTSCTS. Thermo microplate instruments use the HSK_XONXOFF handshake with the exception of some very old models.

Definition at line 1347 of file ThermoCOM.c.

4.4.3.11 DllExport BOOL WINAPI ThermoCOMWrite (HANDLE *hConn,* LPCSTR *buf*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Buffer containing the NUL terminated string to send.

Returns

TRUE if string written to the transmit buffer, else FALSE.

When this function returns, the string is not yet sent to the instrument but is queued for sending. Memory for transmit data is dynamically allocated and the only reason for this function returning FALSE is that no more memory is available. If FALSE is returned, the string is not even partially written to transmit buffer and the application should retry with the whole string at a later time.

Definition at line 1781 of file ThermoCOM.c.

4.4.3.12 DllExport BOOL WINAPI ThermoCOMWriteBinary (HANDLE *hConn,* LPCSTR *buf,* DWORD *count*)

Parameters

hComm	Connection handle returned from a call to ThermoCOMOpen().
buf	Buffer containing the data to send.
count	Number of bytes to send from the buffer.

Returns

TRUE if data written to the buffer, else FALSE.

When this function returns, the data is not yet sent to the instrument but is queued for sending. Memory for transmit data is dynamically allocated and the only reason for this function returning FALSE is that no more memory is available. If FALSE is returned, the data is not even partially written to transmit buffer and the application should retry with all data at a later time.

Definition at line 1805 of file ThermoCOM.c.

Index

api.h, 19

HSK_NONE
    ThermoCOM.h, 25
HSK_RTSCTS
    ThermoCOM.h, 25
HSK_XONXOFF
    ThermoCOM.h, 25

mainpage.h, 19

THERMOCOM_ERROR
    ThermoCOM.h, 25
THERMOCOM_RECEIVE
    ThermoCOM.h, 26
THERMOCOM_TRANSMIT
    ThermoCOM.h, 26
ThermoCOM.c, 19
    ThermoCOMAbort, 20
    ThermoCOMAttachEvent, 20
    ThermoCOMAttachMsg, 21
    ThermoCOMClose, 21
    ThermoCOMGetError, 21
    ThermoCOMOpen, 21
    ThermoCOMOpenSimulator, 22
    ThermoCOMRead, 22
    ThermoCOMReadBinary, 23
    ThermoCOMSetParam, 23
    ThermoCOMWrite, 23
    ThermoCOMWriteBinary, 24
ThermoCOM.h, 24
    HSK_NONE, 25
    HSK_RTSCTS, 25
    HSK_XONXOFF, 25
    THERMOCOM_ERROR, 25
    THERMOCOM_RECEIVE, 26
    ThermoCOMAbort, 26
    ThermoCOMAttachEvent, 26
    ThermoCOMAttachMsg, 27
    ThermoCOMClose, 27
    ThermoCOMGetError, 27
    ThermoCOMOpen, 27
    ThermoCOMOpenSimulator, 28
    ThermoCOMRead, 28
    ThermoCOMReadBinary, 29
    ThermoCOMSetParam, 29
    ThermoCOMWrite, 29
    ThermoCOMWriteBinary, 30
ThermoCOMAbort
    ThermoCOM.c, 20
    ThermoCOM.h, 26
ThermoCOMAttachEvent
    ThermoCOM.c, 20
    ThermoCOM.h, 26
ThermoCOMAttachMsg
    ThermoCOM.c, 21
    ThermoCOM.h, 27
ThermoCOMClose
    ThermoCOM.c, 21
    ThermoCOM.h, 27
ThermoCOMGetError
    ThermoCOM.c, 21
    ThermoCOM.h, 27
ThermoCOMOpen
    ThermoCOM.c, 21
    ThermoCOM.h, 27
ThermoCOMOpenSimulator
    ThermoCOM.c, 22
    ThermoCOM.h, 28
ThermoCOMRead
    ThermoCOM.c, 22
    ThermoCOM.h, 28
ThermoCOMReadBinary
    ThermoCOM.c, 23
    ThermoCOM.h, 29
ThermoCOMSetParam
    ThermoCOM.c, 23
    ThermoCOM.h, 29
ThermoCOMWrite
    ThermoCOM.c, 23
    ThermoCOM.h, 29
ThermoCOMWriteBinary
    ThermoCOM.c, 24
    ThermoCOM.h, 30