Aria Software Developement Kit

User Manual

Version 1.2.0.0 09/2023

Contents

1 | Introduction to Aria SDK

Aria Software Development Kit (SDK) allows you to fully integrate Aria device in your application; it has been declined in several languages, namely C#.NET, C++ and Python.

The aim of this document is to introduce the SDK's exposed functions which can be used to interact with your Aria.

2 | Requirements

2.1 System requirements

The Aria SDK can only run on Windows systems for the moment. Any version more recent than Windows 10 (included) are supported.

2.2 Supported instrument

By using Aria SDK, you have direct access to Aria instrument as a whole, when using a sequence, or to certain of its individual components when used "remotely". The controllable components are:

- Flow EZ™ pressure controller
- FlowUnit M or L depending on the Aria model
- Internal M-Switch (reservoir selection)
- Internal 2-Switch (stop flow)
- External valve (M-Switch or 2-Switch depending on the Aria model)

3 | SDK general philosophy

As for the Aria UI software, the Aria SDK lies on the control of Aria via sequences (or protocols). Even if some remote control of the individual components is possible, the SDK really shines when it comes to schedule some long sequences of commands. Taking all possible parameters into account, Aria SDK is able to play a sequence of injection from multiple reservoirs and to multiple output channels while minimizing the consumption of chemical products (buffer, cell culture, etc.). The current manual aims to provide technical help to setup, control and monitor your Aria in a programmatic way. For any functioning details of the Aria instrument itself, scientific applications or hardware specificities, please refer to the Aria user manual.

3.1 Pipeline

An Aria experiment typically follows this suite of actions:

1. Instrument detection (real or simulated)
2. (opt.) Check configuration (SN, etc.)
3. Calibration (calculate the internal volumes for accurate prediction)
4. Sequence edition
5. (opt.) Save to file
6. Run sequence

3.2 Status management

When called, each function returns an error ID. If the command was properly executed a -1 value ID is returned, otherwise a new ID (incremented from the last error ID) is returned. All errors are saved in a stack during a session lifetime and none is cleared by default. Only a call to ResetErrors (see below) during a session would result in an empty stack.

The lone ID does not provide any details about the error returned (other than the presence of the error). To handle error details, three specific functions can be used:

- GetErrorSeverity: Returns the error severity as an ErrorSeverity enum.
- GetErrorMessage: Returns the error details as a string.
- GetErrorTimestamp: Returns the time at which the error happened.
- ResetErrors: Cleans the error stack, next error ID will be 0.

Errors that happen in functions running asynchronously are called Async Errors. Their error ID can be retrieved from the Async Error Queue using the following functions:

- HasAsyncError: Checks if there are Async Errors in the queue.
- GetAsyncErrorCount: Returns the number of Async Errors in the queue.
- TryGetNextAsyncError: Checks if there is an Async Error in the queue and returns it by reference.

4 | Software layers

The Aria SDK is a C# library built for windows and written in C#. This library handles all communication with the Aria instrument.

It also supports two other major programming languages as additional packages : C++ and Python. They are collectively referred to as **Middleware** in this manual.

We **strongly recommend** using the **Middleware** if your programming language of choice is supported. It is open source, so you can modify it to suit your needs.

4.1 Aria SDK native shared libraries

The native shared libraries (as .DLL files) are provided in each language folder of the SDK archive.

Any language that interfaces with C# should be able to access the library functions, as demonstrated in the Middleware source code.

The library functions are generally non-blocking and return immediately, with the exception of the functions that start a sequence or a procedure or that explicitly wait for a signal, such as StartAwaitingTCPMessage and StartAwaitingTTLSignal.

These values represent expected response time both when reading and when setting values on the instrument. Calling `GetXXX` functions more frequently than these delays will simply return the same value repeatedly until it is updated by the instrument. Calling `SetXXX` functions more frequently might cause the library to block while it waits for the instrument to process the commands.

For your information, The data refresh rate of Aria is **20ms**.

4.2 Middleware

The SDK middleware is a set of packages that make it easier to use the SDK with various programming languages. So far, they mainly act as examples and can be used as they are as a coding starting point for new developments.

The following programming languages are supported:

Language	Package
C++	**main.cpp** example script **aria_sdk_example_cpp.sln** Visual Studio complete solution containing middleware and examples
C#	**Program.cs** example script **aria-sdk-example.sln** Visual Studio complete solution containing middleware and examples
Python	**aria-sdk-example.py** example script

The middleware matches the conventions of each programming language while keeping the interface as similar as possible across all supported languages.

The following sections contain installation and usage instructions for each language.

4.3 Installation

See for each language the installation specificities.

4.3.1 C++

The C++ example consists of a Visual Studio solution (aria-sdk-example.sln) containing:

- A native C++14 middleware **aria-sdk-example.vcxproj** project file
- An example script **main.cpp**
- A **Headers** folder with all *.h* files
- A **README.md** file
- **DLL files** for x86 and x64 architectures (and both 2SW and MSW versions)

Language specifics:

- Because of some limits for the C++ integration, all units will be automatically converted to **"µl/min"** for the flowrates and **"µl"** for the volumes.

4.3.2 C#

The C# middleware consists of a Visual Studio solution (aria-sdk-example.sln) containing:

- A .NET Framework 4.8 middleware **aria-sdk-example.csproj** project file
- An example script **Program.cs**
- A **README.md** file
- **DLL files** for x86 and x64 architectures (and both 2SW and MSW versions)

Simply copy the DLL library to the example folder then build and run the project.

4.3.3 Python

The Python package groups:

- An example script **example/aria-sdk-example.py**
- A Python package as an archive (**aria_sdk-X.X.X.zip**)
- A **README.md** file

The Python package is provided as a .zip file that can be installed using the pip or easy_install modules. This package can be installed in all supported operating systems and includes the necessary shared libraries. It is compatible with Python 3.1 and later:

```
python -m pip install -user aria_sdk-X.X.X.zip
python -m easy_install -user aria_sdk-X.X.X.zip
```

Language specifics:

- The Python support involves the usage of **pythonnet** package that can be found on PIP (https://pypi.org/project/pythonnet/).
- Functions that return values (such as the functions starting with `GetXXX`) returns both the desired value and the error as a tuple. You must store both value into a tuple or an exception will be raised.

```
stepProgress = Monitoring.GetProgress(currentStep)         # WRONG
stepProgress, error = Monitoring.GetProgress(currentStep) # OK
```

- For more details about how `ref` and `out` values are handled in **Python.NET**, please refer to https://pythonnet.github.io/pythonnet/python.html#out-and-ref-parameters
- Because of some limits for the Python integration, all units will be automatically converted to **"µl/min"** for the flowrates and **"µl"** for the volumes.

5 | Fluigent SDK Functions

5.1 Types definition

1. ErrorSeverity

   Returned error severity when requested by GetErrorSeverity.

Value	Enum	Description
0	Info	No error
1	Warning	Report non-blocking warning
2	Error	Report failed action

2. FlowUnitType

   Type of the internal FlowUnit. Only FlowUnits **M** and **L** are currently available for Aria.

Value	Enum	Description
0	UnknownFlowUnit	Cannot get the FlowUnit type
1	*XS*	*FlowUnit XS (NA)*
2	*S*	*FlowUnit S (NA)*
4	M	FlowUnit M
8	L	FlowUnit L
16	*XL*	*FlowUnit XL (NA)*
32	*MPLUS*	*FlowUnit M+ (NA)*
64	*LPLUS*	*FlowUnit L+ (NA)*

3. SignalType

   Aria allows to send/receive TTL binary signals and TCP/IP messages. Those signals can be sent at the beginning and/or end of any sequence function. See Sequence edition section for more details.

Value	Enum	Description
0	TTL	TTL binary signal
1	TCP	TCP/IP message

4. FlowRatePreset

   It is possible to tune the flowrate used for the prefill step of a sequence. The FlowRatePreset indicates the balance between precision and speed to be used in the SetPrefillAndPreloadFlowRate function.

Value	Enum	Description
0	Precision	FlowUnit M: 30 µl/min | FlowUnit L: 50 µl/min
1	Balanced	FlowUnit M: 55 µl/min | FlowUnit L: 250 µl/min
2	Fast	FlowUnit M: 80 µl/min | FlowUnit L: 500 µl/min
3	Max	FlowUnit M: 80 µl/min | FlowUnit L: 1000 µl/min

5. SwitchType

   Type of the external Switch returned by GetExternalSwitchType.

Value	Enum	Description
0	UnknownSwitch	Cannot get the Switch type
1	TwoSwitch	2-Switch (3-port/2-way valve)
2	MSwitch	M-Switch (11-port/10-position valve)

6. StepType

   Type of a sequence step.

Value	Enum	Description
0	Flush	Flushed the liquid remaining in the tubing to the waste
1	TimeInjection	Injection based on time and flowrate
2	VolumeInjection	Injection based on volume and flowrate
3	Wait	Wait for a certain time
4	WaitForUser	Wait until user input
5	WaitForExternalSignal	Wait for an external signal of type SignalType before proceeding
6	SendExternalSignal	Send a signal of type SignalType

7. StepParameter

   Type of a step parameter. Used in SetParameter and all Get*Parameter functions.

Value	Enum	Description
0	PRE_SIGNAL	bool
1	PRE_SIGNAL_TYPE	SignalType
2	POST_SIGNAL	bool
3	POST_SIGNAL_TYPE	SignalType
4	INPUT_RESERVOIR	int (1 -> 10)
5	OUTPUT_DESTINATION	int (1 -> 2 with 2-Switch, 1 -> 10 with M-Switch)
6	FLOWRATE	float
7	VOLUME	float
8	DURATION	int
9	SIGNAL_MESSAGE	string
10	AWAITED_SIGNAL_TYPE	SignalType
11	BACKTRACK	bool

8. CalibrationState

   State of the Calibration phase.

Value	Enum	Description
0	NotRunning	No Calibration phase currently running
1	Flushing	Flushing in progress
2	SettingUp	Setting up in progress
3	Calibrating	Calibration in progress

5.2 SDK Wrapper

Errors

5.2.1 GetErrorMessage

```
string GetErrorMessage(int errorId);
```

Returns the message associated to the error with ID *errorId*

Parameters

errorId	int	Error ID

Returns

errorMsg	string	Error message

5.2.2 GetErrorSeverity

```
ErrorSeverity GetErrorSeverity(int errorId);
```

Returns the error severity (as ErrorSeverity) associated to the error with ID *errorId*

Parameters

errorId	int	Error ID

Returns

errorSeverity	ErrorSeverity	Error severity

5.2.3 GetErrorTimestamp

```
string GetErrorTimestamp(int errorId);
```

Returns the timestamp (with the following format: yyyy/MM/dd-HH:mm:ss) associated to the error with ID *errorId*

Parameters

errorId	int	Error ID

Returns

errorTimestamp	string	Error timestamp with format (as yyyy/MM/dd-HH:mm:ss)

5.2.4 ResetErrors

```
void ResetErrors();
```

Clears the error stack of all previous errors. Next error ID will be then **0**.

5.2.5 HasAsyncError

```
bool HasAsyncError();
```

Returns true if the error stack has one error or more.

Returns

hasError	bool	Error stack has one error or more

5.2.6 GetAsyncErrorCount

```
int GetAsyncErrorCount();
```

Returns the number of errors in the error stack.

Returns

nbError	int	Number of errors in the error stack

5.2.7 TryGetNextAsyncError

```
bool TryGetNextAsyncError(int* errorId);
```

Gets the ID of the first (oldest) error from the error stack.

Output

errorId	int	ID or the first (oldest) error of the error stack

Returns

success	bool	Returns true if an Async Error was returned in Output, false if the Async Error queue was empty.

5.2.8 LoadPhysicalInstrument

```
bool LoadPhysicalInstrument(int* error)
```

Searches for a connected Aria instrument and loads it. Returns true if a connected Aria instrument was detected, false otherwise.

Output

errorId	int	Error ID (-1 if none)

Returns

success	bool	Has the instrument be successfully loaded or not.

5.2.9 LoadSimulatedInstrument

```
void LoadSimulatedInstrument(FlowUnitType flowUnit, SwitchType externalSwitch,
    int* errorId);
```

Loads a Simulated Instrument with the given flowUnit and externalSwitch types.

Parameters

flowUnit	FlowUnitType	Type of the FlowUnit to be simulated
externalSwitch	SwitchType	Type of the external Switch to be simulated

Output

errorId	int	Error ID (-1 if none)

5.2.10 IsInstrumentSimulated

```
bool IsInstrumentSimulated(int* errorId);
```

Reports if the current instrument is simulated.

Output

errorId	int	Error ID (-1 if none)

Returns

isSimulated	bool	Is the instrument simulated or not.

5.2.11 CheckHardware

```
bool CheckHardware(int* errorId);
```

Checks if all hardware components are detected.

Output

errorId	int	Error ID (-1 if none)

Returns

hardwareStatus	bool	True if no hardware issue was detected, false otherwise.

5.2.12 GetFlowUnitType

```
FlowUnitType GetFlowUnitType(int* errorId);
```

Returns the current Instrument FlowUnit Type.

Output

errorId	int	Error ID (-1 if none)

Returns

type	FlowUnitType	Type of the current FlowUnit.

5.2.13 GetExternalSwitchType

```
SwitchType GetExternalSwitchType(int* errorId);
```

Returns the current instrument external Switch type.

Output

errorId	int	Error ID (-1 if none)

Returns

type	SwitchType	Type of the external Switch type.

5.2.14 GetAriaSerialNumber

```
int GetAriaSerialNumber(int* errorId);
```

Returns Aria instrument Serial Number.

Output

errorId	int	Error ID (-1 if none)

Returns

serialNumber	int	Aria instrument SN.

5.2.15 GetFirmwareVersion

```
int GetFirmwareVersion(int* errorId);
```

Returns Aria instrument firmware version.

Output

errorId	int	Error ID (-1 if none)

Returns

firmwareVersion	int	Aria instrument firmware version.

5.2.16 GetFlowEZFirmwareVersion

```
int GetFlowEZFirmwareVersion(int* errorId);
```

Returns Aria FlowEZ firmware version.

Output

errorId	int	Error ID (-1 if none)

Returns

firmwareVersion	int	Aria FlowEZ firmware version.

5.2.17 GetMinFlowRate

```
float GetMinFlowRate(int* errorId);
```

Returns the minimum flowrate order allowed by the current instrument.

Output

errorId	int	Error ID (-1 if none)

Returns

flowRate	float	Minimum flowrate possible with the current instrument.

5.2.18 GetMaxFlowRate

```
float GetMaxFlowRate(int* errorId);
```

Returns the maximum flowrate order allowed by the current instrument.

Output

errorId	int	Error ID (-1 if none)

Returns

flowRate	float	Maximum flowrate possible with the current instrument.

5.2.19 GetMinPressure

```
float GetMinPressure(int* errorId);
```

Returns the minimum pressure order allowed by the current instrument.

Output

errorId	int	Error ID (-1 if none)

Returns

pressure	float	Minimum pressure possible with the current instrument.

5.2.20 GetMaxPressure

```
float GetMaxPressure(int* errorId);
```

Returns the maximum pressure order allowed by the current instrument.

Output

errorId	int	Error ID (-1 if none)

Returns

pressure	float	Maximum pressure possible with the current instrument.

5.2.21 GetDateTimeFormat

```
string GetDateTimeFormat();
```

Returns the DateTime format used in Aria SDK functions.

Returns

timeFormat	string	DateTime format used in SDK functions.

5.2.22 SetPrefillAndPreloadFlowRate

```
void SetPrefillAndPreloadFlowRate(FlowratePreset flowratePreset, int* errorId);
```

Defines the prefill and preload flowrate from the given FlowratePreset. This impacts the precision vs speed balance for those two steps.

Parameters

flowratePreset	FlowRatePreset	Flowrate preset to use for the prefill and preload steps

Output

errorId	int	Error ID (-1 if none)

5.2.23 GetPrefillAndPreloadFlowRatePreset

```
FlowratePreset GetPrefillAndPreloadFlowRatePreset(int* errorId);
```

Returns the current prefill and preload flowrate preset as FlowratePreset.

Output

errorId	int	Error ID (-1 if none)

Returns

flowratePreset	FlowRatePreset	Current flowrate preset.

5.2.24 GetPrefillAndPreloadFlowRate

```
float GetPrefillAndPreloadFlowRate(int* errorId);
```

Returns the current prefill and preload flowrate (in µl/min).

Output

errorId	int	Error ID (-1 if none)

Returns

flowrate	float	Current prefill and preload flowrate (in µl/min).

5.2.25 SetCalibrationValue

```
void SetCalibrationValue(int stepId, float volume, int* errorId);
```

Sets the internal volume **volume** (in µl) for step **stepId**.

Parameters

stepId	int	Step ID
volume	float	Internal volume for step *step*

Output

errorId	int	Error ID (-1 if none)

5.2.26 SetStep3CalibrationValue

```
void SetStep3CalibrationValue(int step3PortId, float volume, int* errorId);
```

Sets the internal volume **volume** (in µl) for port **step3PortId** of step 3 (2-Switch: 1 -> 2, M-Switch: 1 -> 10).

Parameters

step3PortId	int	Step 3 port ID (2-Switch: 1 -> 2, M-Switch: 1 -> 10)
volume	float	Internal volume for step *step*

Output

errorId	int	Error ID (-1 if none)

5.2.27 SetCalibrationValues

```
void SetCalibrationValues(float step1Volume, float step2Volume, float[]
   step3Volumes, int* errorId);
```

Sets the internal volumes **step1Volume** (in µl), **step2Volume** (in µl) and **step3Volumes** (in µl) for all Calibration steps.

Parameters

stepId	int	Step ID
volume	float	Internal volume for step *step* (in µl)

Output

errorId	int	Error ID (-1 if none)

5.2.28 GetCalibrationValues

```
void GetCalibrationValues(float *step1Volume, float *step2Volume, int* errorId);
```

Returns the internal volumes (in µl) for Calibration steps 1 and 2.

Output

step1Volume	float	Internal volume of Calibration step 1 (in µl).
step2Volume	float	Internal volume of Calibration step 2 (in µl).
errorId	int	Error ID (-1 if none)

5.2.29 GetStep3CalibrationValues

```
float[] GetStep3CalibrationValues(int* errorId);
```

Returns the internal volumes (in µl) for Calibration steps 1 and 2.

Output

errorId	int	Error ID (-1 if none)

Returns

step3Volumes	float[]	Internal volumes of Calibration step 3 (in µl).

5.2.30 GetMaxStep3CalibrationValueCount

```
int GetMaxStep3CalibrationValueCount();
```

Returns the maximum number of calibration values in the Calibration step 3 table.

Returns

nbMaxValues	int	Maximum number of calibration values in Calibration step 3 table.

Sequence configuration

5.2.31 StartSequence

```
void StartSequence(int* errorId);
```

Start the current sequence.

Output

errorId	int	Error ID (-1 if none)

5.2.32 GenerateSequenceJSON

```
string GenerateSequenceJSON(int* errorId);
```

Returns the sequence saved as a JSON string.

Output

errorId	int	Error ID (-1 if none)

Returns

sequenceAsJSON	string	JSON string representation of the current sequence

5.2.33 LoadSequenceFromJSON

```
void LoadSequenceFromJSON(string jsonString, int* errorId);
```

Load a sequence from a JSON String.

Parameter

jsonString	string	JSON string representation of the current sequence

Output

errorId	int	Error ID (-1 if none)

5.2.34 LoadSequence

```
void LoadSequence(string filePath, int* errorId);
```

Load sequence from the file at **filePath**.

Parameter

filePath	string	Path of a JSON file containing a sequence information

Output

errorId	int	Error ID (-1 if none)

5.2.35 SetBufferReservoir

```
void SetBufferReservoir(int reservoirNumber, int* errorId);
```

Defines the reservoir that will be used to inject Buffer during the sequence execution. Only reservoirs 9 and 10 can be used as Buffer Reservoirs.

Parameter

reservoirNumber	int	New Buffer Reservoir ID (9 or 10)

Output

errorId	int	Error ID (-1 if none)

5.2.36 GetBufferReservoir

```
int GetBufferReservoir(int* errorId);
```

Returns the number of the reservoir used to inject Buffer during the sequence execution.

Output

errorId	int	Error ID (-1 if none)

Returns

reservoirNumber	int	Buffer reservoir number.

5.2.37 GetReservoirEstimatedRequiredVolume

```
float GetReservoirEstimatedRequiredVolume(int reservoirNumber, int* errorId);
```

Returns the estimated required volume with which to fill the given reservoir before starting the sequence.

Parameter

reservoirNumber	int	Reservoir ID (1 -> 10) to be considered.

Output

errorId	int	Error ID (-1 if none)

Returns

requiredVolume	float	Estimated required volume for the reservoir considered (in microliters).

5.2.38 IsReservoirEstimatedOverCapacity

```
bool IsReservoirEstimatedOverCapacity(int reservoirNumber, int* errorId);
```

Returns true if the required volume for the given reservoir is above that reservoir capacity, false otherwise. Reservoirs over capacity will require to refill them during the sequence execution.

Parameter

reservoirNumber	int	Reservoir ID (1 -> 10) to be considered.

Output

errorId	int	Error ID (-1 if none)

Returns

overCapacity	bool	Is the reservoir over capacity?

5.2.39 EnablePrefill

```
void EnablePrefill(bool enabled, int* errorId);
```

Enables or disables the prefill phase at the begining of the sequence. The prefill fills the tubing between the reservoir and the internal M-Switch. Prefill can be safely disabled only if the tubing is already filled with the correct content.

Parameter

enabled	bool	Enables or not the prefill.

Output

errorId	int	Error ID (-1 if none)

5.2.40 IsPrefillEnabled

```
bool IsPrefillEnabled(int* errorId);
```

Returns true if Prefill is enabled, false otherwise.

Output

errorId	int	Error ID (-1 if none)

Returns

enabled	bool	Is the prefill enabled (true) or not (false)

5.2.41 EnableZeroPessureMode

```
void EnableZeroPessureMode(bool enabled, int* errorId);
```

Enables or disables Zero Pressure mode. Zero Pressure mode forces the pressure to reset to 0 every time a Switch is moved to avoid flow rate spikes, irregularities, etc.

Parameter

enabled	bool	Enables or not the Zero Pressure mode.

Output

errorId	int	Error ID (-1 if none)

5.2.42 IsZeroPressureModeEnabled

```
bool IsZeroPressureModeEnabled(int* errorId);
```

Returns true if Zero Pressure Mode is enabled, false otherwise.

Output

errorId	int	Error ID (-1 if none)

Returns

enabled	bool	True if Zero Pressure mode is enabled.

5.2.43 SetSequenceStartASAP

```
void SetSequenceStartASAP(bool startASAP, int* errorId);
```

Defines if the sequence must start as soon as possible, or with a delay.

Parameter

startASAP	bool	Start the sequence as soon as possible or not.

Output

errorId	int	Error ID (-1 if none)

5.2.44 IsSequenceStartingASAP

```
bool IsSequenceStartingASAP(int* errorId);
```

Returns true if *startASAP* is enabled, false otherwise.

Output

errorId	int	Error ID (-1 if none)

Returns

startingASAP	bool	Will the sequence start ASAP?

5.2.45 SetSequenceStartTime

```
void SetSequenceStartTime(string dateTime, int* errorId);
```

Defines the time at which the sequence will be executed if *startASAP* is disabled. Time must be in the following format: yyyy/MM/dd-HH:mm:ss

Parameter

dateTime	string	Time at which the sequence will be started.

Output

errorId	int	Error ID (-1 if none)

5.2.46 GetSequenceStartTime

```
string GetSequenceStartTime(int* errorId);
```

Returns the estimated start time of the first step of the sequence.

Output

errorId	int	Error ID (-1 if none)

Returns

dateTime	string	Time at which the sequence will be started.

5.2.47 GetTotalDuration

```
int GetTotalDuration(int* errorId);
```

Returns the estimated total duration of the sequence.

Output

errorId	int	Error ID (-1 if none)

Returns

duration	int	Sequence estimated total duration (in seconds).

Sequence edition

5.2.48 GetSequenceStepCount

```
int GetSequenceStepCount(int* errorId);
```

Returns the total number of steps in the current sequence.

Output

errorId	int	Error ID (-1 if none)

Returns

nbSteps	int	Number of current sequence step

5.2.49 RemoveStep

```
bool RemoveStep(int index, int* errorId);
```

Removes the step at **index** from the sequence.

Parameter

index	int	Index of the step to be removed

Output

errorId	int	Error ID (-1 if none)

Returns

success	bool	True if the step was removed

5.2.50 InsertFlushStep

```
void InsertFlushStep(int index, int inputReservoir, float flowRate, bool
    preSignal, SignalType preSignalType, bool postSignal, SignalType
    postSignalType, int* errorId);
```

Inserts a step of flushing in the current sequence at **index**. Reservoir **inputReservoir** will be flushed at **flowRate** µl/min.

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
inputReservoir	int	Input reservoir to be used (1 -> 10).
flowRate	float	Flowrate order to reach for this step (in µl/min).
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.51 InsertSendSignalStep

```
void InsertSendSignalStep(int index, string message, bool preSignal, SignalType
    preSignalType, bool postSignal, SignalType postSignalType, int* errorId);
```

Inserts a step of TCP/IP signal sending (with message **message**) in the current sequence at **index**.

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
message	string	TCP/IP message to be sent.
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.52 InsertTimedInjectionStep

```
void InsertTimedInjectionStep(int index, int inputReservoir, int destination,
    float flowRate, int duration_s, bool preSignal, SignalType preSignalType,
    bool postSignal, SignalType postSignalType, int* errorId);
```

Inserts a step of timed injection in the current sequence at **index**. Input reservoir **inputReservoir** will be injected into output port **destination** at **flowRate** µl/min for **duration_s** seconds.

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
inputReservoir	int	Input reservoir to be used (1 -> 10).
destination	int	External Switch port to be used (1 -> 2 for 2-Switch, 1 -> 10 for M-Switch).
flowRate	float	Flowrate order to reach for this step (in µl/min).
duration_s	int	Time of the injection (in seconds).
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.53 InsertVolumeInjectionStep

```
void InsertVolumeInjectionStep(int index, int inputReservoir, int destination,
    float flowRate, int volume, bool preSignal, SignalType preSignalType, bool
    postSignal, SignalType postSignalType, int* errorId);
```

Inserts a step of volume injection in the current sequence at **index**. A volume of **volume** µl will be injected from input reservoir **inputReservoir** to output port **destination** at **flowRate** µl/min.

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
inputReservoir	int	Input reservoir to be used (1 -> 10).
destination	int	External Switch port to be used (1 -> 2 for 2-Switch, 1 -> 10 for M-Switch).
flowRate	float	Flowrate order to reach for this step (in µl/min).
volume	float	Volume to be injected (in µl).
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.54 InsertWaitStep

```
void InsertFlushStep(int index, int duration_s, bool preSignal, SignalType
    preSignalType, bool postSignal, SignalType postSignalType, int* errorId);
```

Inserts a step of waiting in the current sequence at **index**. Step will wait for **duration_s** seconds before proceeding to the next step.

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
duration_s	int	Waiting time (in seconds).
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.55 InsertWaitUserStep

```
void InsertWaitUserStep(int index, int timeout_s, bool preSignal, SignalType
    preSignalType, bool postSignal, SignalType postSignalType, int* errorId);
```

Inserts a step of waiting for user in the current sequence at **index**. Step will wait for the execution of ResumeSequenceExecution before proceeding to the next step.

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
timeout_s	int	Waiting timeout if no signal has been received (in seconds).
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.56 InsertWaitSignalStep

```
void InsertWaitSignalStep(int index, int timeout_s, SignalType signalType, bool
    enableBacktrack, bool preSignal, SignalType preSignalType, bool postSignal,
    SignalType postSignalType, int* errorId);
```

Inserts a step of waiting for signal in the current sequence at **index**. Step will wait for a signal of type SignalType before proceeding to the next step. When **enableBacktrack** is true

Parameters

index	int	Position in the sequence where the step will be inserted. **0** to insert it at the beginning, **-1** to insert it at the end.
timeout_s	int	Waiting timeout if no signal has been received (in seconds).
signalType	SignalType	Type of the signal to be waiting for.
preSignal	bool	Send a pre-signal.
preSignalType	SignalType	Pre-signal type.
postSignal	bool	Send a post-signal.
postSignalType	SignalType	Post-signal type.

Output

errorId	int	Error ID (-1 if none)

5.2.57 GetEstimatedStepStartTime

```
int GetEstimatedStepStartTime(int index, int* errorId);
```

Returns the estimated delay (in seconds) before the step at **index** is executed.

Parameter

index	int	Index of the selected step.

Output

errorId	int	Error ID (-1 if none)

Returns

delay	int	Number of seconds before the selected step will be executed.

5.2.58 GetEstimatedStepDuration

```
int GetEstimatedStepDuration(int index, int* errorId);
```

Returns the estimated duration (in seconds) of the Step at **index**.

Parameter

index	int	Index of the selected step.

Output

errorId	int	Error ID (-1 if none)

Returns

duration	int	Estimated duration of the selected step (in seconds).

5.2.59 GetStepType

```
StepType GetStepType(int index, int* errorId);
```

Returns the Step type (as StepType) of the step at **index**.

Parameter

index	int	Index of the selected step.

Output

errorId	int	Error ID (-1 if none)

Returns

stepType	StepType	Step type of the selected step.

5.2.60 SetParameter

```
int SetParameter(int index, StepParameter parameterType, int value);
```

Assigns value **value** of the parameter **parameterType** (StepParameter) of the step at **index**. Check that the value type corresponds to the type of the step parameter (see StepParameter).

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter to edit.
value	int	New value of the step parameter.

Returns

errorId	int	Error ID (-1 if none)

5.2.61 SetParameter

```
void SetParameter(int index, StepParameter parameterType, bool value, int*
    errorId);
```

Assigns value **value** of the parameter **parameterType** (StepParameter) of the step at **index**. Check that the value type corresponds to the type of the step parameter (see StepParameter).

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter to edit.
value	bool	New value of the step parameter.

Output

errorId	int	Error ID (-1 if none)

5.2.62 SetParameter

```
void SetParameter(int index, StepParameter parameterType, float value, int*
    errorId);
```

Assigns value **value** of the parameter **parameterType** (StepParameter) of the step at **index**. Check that the value type corresponds to the type of the step parameter (see StepParameter).

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter to edit.
value	float	New value of the step parameter.

Output

errorId	int	Error ID (-1 if none)

5.2.63 SetParameter

```
void SetParameter(int index, StepParameter parameterType, string value, int*
    errorId);
```

Assigns value **value** of the parameter **parameterType** (StepParameter) of the step at **index**. Check that the value type corresponds to the type of the step parameter (see StepParameter).

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter to edit.
value	string	New value of the step parameter.

Output

errorId	int	Error ID (-1 if none)

5.2.64 SetParameter

```
void SetParameter(int index, StepParameter parameterType, SignalType value,
    int* errorId);
```

Assigns value **value** of the parameter **parameterType** (StepParameter) of the step at **index**. One must check that the value type corresponds to the type of the step parameter (see StepParameter).

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter to edit.
value	SignalType	New value of the step parameter.

Output

errorId	int	Error ID (-1 if none)

5.2.65 GetIntParameter

```
int GetIntParameter(int index, StepParameter parameterType, int* errorId);
```

Returns the parameter value (int) according to the **parameterType** for the step at **index**.

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter.

Output

errorId	int	Error ID (-1 if none)

Returns

parameterValue	int	Value of the selected parameter.

5.2.66 GetBoolParameter

```
bool GetBoolParameter(int index, StepParameter parameterType, int* errorId);
```

Returns the parameter value (bool) according to the **parameterType** for the step at **index**.

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter.

Output

errorId	int	Error ID (-1 if none)

Returns

parameterValue	bool	Value of the selected parameter.

5.2.67 GetFloatParameter

```
float GetFloatParameter(int index, StepParameter parameterType, int* errorId);
```

Returns the parameter value (float) according to the **parameterType** for the step at **index**.

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter.

Output

errorId	int	Error ID (-1 if none)

Returns

parameterValue	float	Value of the selected parameter.

5.2.68 GetStringParameter

```
string GetStringParameter(int index, StepParameter parameterType, int* errorId);
```

Returns the parameter value (string) according to the **parameterType** for the step at **index**.

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter.

Output

errorId	int	Error ID (-1 if none)

Returns

parameterValue	string	Value of the selected parameter.

5.2.69 GetSignalTypeParameter

```
SignalType GetSignalTypeParameter(int index, StepParameter parameterType, int*
    errorId);
```

Returns the parameter value (SignalType) according to the **parameterType** for the step at **index**.

Parameters

index	int	Index of the selected step.
parameterType	StepParameter	Type of the step parameter.

Output

errorId	int	Error ID (-1 if none)

Returns

parameterValue	SignalType	Value of the selected parameter.

Direct control

Note that it is necessary to run a SetPressureOrder(0) to unlock the direct control of the Aria instrument components. Without it, it is NOT possible to control the switches or the flowrate outside a standard sequence.

5.2.70 SetFlowRateOrder

```
void SetFlowRateOrder(float flowrate, int* errorId);
```

Set the flowrate order of the Aria instrument to **flowrate** (in µl/min).

Parameter		
flowrate	float	Flowrate order to be reached (in µl/min)
Output		
errorId	int	Error ID (-1 if none)

5.2.71 GetFlowRateOrder

```
float GetFlowRateOrder(int* errorId);
```

Returns the last flowrate order sent to the Aria instrument (in µl/min).

Output		
errorId	int	Error ID (-1 if none)
Returns		
flowrate	float	Last flowrate order sent (in µl/min)

5.2.72 GetMeasuredFlowRate

```
float GetMeasuredFlowRate(int* errorId);
```

Returns the current flowrate value measured by the Aria instrument (in µl/min).

Output		
errorId	int	Error ID (-1 if none)
Returns		
flowrate	float	Current flowrate (in µl/min)

5.2.73 SetPressureOrder

```
void SetPressureOrder(float pressure, int* errorId);
```

Set the pressure order of the Aria instrument to **pressure** (in mbar).

Parameter

pressure	float	Pressure order to be reached (in mbar)

Output

errorId	int	Error ID (-1 if none)

5.2.74 GetPressureOrder

```
float GetPressureOrder(int* errorId);
```

Returns the last pressure order sent to the Aria instrument (in mbar).

Output

errorId	int	Error ID (-1 if none)

Returns

pressure	float	Last pressure order sent (in mbar)

5.2.75 GetMeasuredPressure

```
float GetMeasuredPressure(int* errorId);
```

Returns the current pressure value measured by the Aria instrument (in mbar).

Output

errorId	int	Error ID (-1 if none)

Returns

pressure	float	Current pressure (in mbar)

5.2.76 SelectReservoir

```
void SelectReservoir(int reservoirId, int* errorId);
```

(Remote Control) Switch the internal M-Switch to connect to the given **reservoirId** (1 -> 10).

Parameter

reservoirId	int	Selected reservoir ID (1 -> 10)

Output

errorId	int	Error ID (-1 if none)

5.2.77 GetSelectedReservoir

```
int GetSelectedReservoir(int* errorId);
```

Returns the current selected reservoir.

Output

errorId	int	Error ID (-1 if none)

Returns

reservoirId	int	ID of the current selected reservoir

5.2.78 StopFlow

```
void StopFlow(bool stop, int* errorId);
```

(Remote Control) Switch the Internal 2-Switch to stop (true) or allow (false) flow.

Parameter

stop	bool	Stop (true) or allow (false) flow.

Output

errorId	int	Error ID (-1 if none)

5.2.79 IsFlowStopped

```
bool IsFlowStopped(int* errorId);
```

Returns true if the flow is stopped by the Internal 2-Switch, false if it is open.

Output

errorId	int	Error ID (-1 if none)

Returns

stopped	bool	Is flow stopped (true) or not (false)

5.2.80 GetExternalSwitchMaxReachablePort

```
int GetExternalSwitchMaxReachablePort(int* errorId);
```

Returns the maximum Port number of the external Switch (2-Switch: 2 | M-Switch: 10)

Output

errorId	int	Error ID (-1 if none)

Returns

maxNumberPorts	int	Maximum number of ports for the current external Switch

5.2.81 SetExternalSwitchPort

```
void SetExternalSwitchPort(int port, int* errorId);
```

(Remote Control) Switch the external Switch to the given chip **port**.

Parameter

port	int	Selected chip (external Switch) port.

Output

errorId	int	Error ID (-1 if none)

5.2.82 GetCurrentExternalSwitchPort

```
int GetCurrentExternalSwitchPort(int* errorId);
```

Returns the current chip port of the external Switch.

Output

errorId	int	Error ID (-1 if none)

Returns

portId	int	ID of the current selected chip port (external Switch port)

5.2.83 SetEnabledPort

```
void SetEnabledPort(int port, bool enabled, int* errorId);
```

Enables or disables the given external Switch **port**. Enabled ports can be used in sequences as well as Calibration and Cleaning procedures.

Parameters

port	int	Selected chip (external Switch) port.
enabled	bool	Enables (true) or disables (false) selected port.

Output

errorId	int	Error ID (-1 if none)

5.2.84 IsPortEnabled

```
bool IsPortEnabled(int portId, int* errorId);
```

Returns true if the external Switch **port** is enabled, false otherwise.

Output

errorId	int	Error ID (-1 if none)

Returns

enabled	bool	Is external Switch port enabled (true) or not (false)

5.2.85 GetWastePort

```
int GetWastePort(int* errorId);
```

Returns the external Switch port number of the Waste port. (2-Switch: 2 | M-Switch: 10).

Output

errorId	int	Error ID (-1 if none)

Returns

portId	int	ID of the external Switch port used for the Waste.

5.2.86 GetDefaultOutputPort

```
int GetDefaultOutputPort(int* errorId);
```

Returns the external Switch port number of the default Output port. (2-Switch: 1 | M-Switch: 1).

Output

errorId	int	Error ID (-1 if none)

Returns

portId	int	ID of the external Switch port used for the default Output.

Sequence monitoring

5.2.87 IsSequenceRunning

```
bool IsSequenceRunning(int* errorId);
```

Returns true if the current sequence is in progress, false otherwise. A paused sequence is still considered as running.

Output

errorId	int	Error ID (-1 if none)

Returns

running	bool	Is the sequence running (true) or not (false)

5.2.88 PauseSequence

```
void PauseSequence(int* errorId);
```

Pauses the current sequence execution.

Output

errorId	int	Error ID (-1 if none)

5.2.89 IsSequencePaused

```
bool IsSequencePaused(int* errorId);
```

Returns true if the sequence is paused, false otherwise.

Output

errorId	int	Error ID (-1 if none)

Returns

paused	bool	Is the sequence paused (true) or not (false)

5.2.90 ResumeSequenceExecution

```
void ResumeSequenceExecution(int* errorId);
```

Resumes the execution of a paused sequence.

Output

errorId	int	Error ID (-1 if none)

5.2.91 Cancel

```
void Cancel(int* errorId);
```

Cancels the current procedure or sequence.

Output

errorId	int	Error ID (-1 if none)

5.2.92 GetPrefillStepNumber

```
int GetPrefillStepNumber();
```

Returns the step index of the Prefill phase of the sequence (base 1).

Returns

stepId	int	Step ID of the Prefill phase for the current sequence.

5.2.93 GetPreloadStepNumber

```
int GetPreloadStepNumber();
```

Returns the step index of the Preload phase of the sequence (base 1).

Returns

stepId	int	Step ID of the Preload phase for the current sequence.

5.2.94 GetCurrentStep

```
int GetCurrentStep(int* errorId);
```

Returns the step index of the current step of the sequence (base 1).

Output

errorId	int	Error ID (-1 if none)

Returns

stepId	int	Step ID of the current step of the running sequence

5.2.95 GetProgress

```
float GetProgress(int stepId, int* errorId);
```

Returns the progress level (%) of the step at index **stepId**. (base 1).

Parameter

stepId	int	Step to be considered.

Output

errorId	int	Error ID (-1 if none)

Returns

progress	float	Progress level of the selected step

5.2.96 GetPrefillAndPreloadProgress

```
float GetPrefillAndPreloadProgress(int* errorId);
```

Returns the cumulated progress for the Prefill and Preload phases.

Output

errorId	int	Error ID (-1 if none)

Returns

progress	float	Cumulated progress for the Prefill and Preload phases

5.2.97 HasSequenceEnded

```
bool HasSequenceEnded(int* errorId);
```

Returns true if the sequence execution ended.

Output

errorId	int	Error ID (-1 if none)

Returns

ended	bool	Has the current sequence ended (true) or not (false)

Procedures

5.2.98 GetLastMeasuredCalibrationVolume

```
float GetLastMeasuredCalibrationVolume(int* errorId);
```

Returns the last internal volume calculated during the Calibration phase.

Output

errorId	int	Error ID (-1 if none)

Returns

volume	float	Last internal volume calculated during the Calibration phase

5.2.99 GetCalibrationState

```
CalibrationState GetCalibrationState(int* errorId);
```

Returns the current state of the Calibration phase (as CalibrationState).

Output

errorId	int	Error ID (-1 if none)

Returns

state	CalibrationState	Current state of the Calibration phase

5.2.100 StartCalibrationStep1

```
void StartCalibrationStep1(int* errorId);
```

Starts the 1st step of the Calibration phase.

Output

errorId	int	Error ID (-1 if none)

5.2.101 StartCalibrationStep2

```
void StartCalibrationStep2(int* errorId);
```

Starts the 2nd step of the Calibration phase.

Output

errorId	int	Error ID (-1 if none)

5.2.102 StartCalibrationStep3_2Switch

```
void StartCalibrationStep3_2Switch(int* errorId);
```

Starts the 3rd step of the Calibration phase for Aria instrument with external 2-Switch.

Output

errorId	int	Error ID (-1 if none)

5.2.103 StartCalibrationStep3_MSwitch

```
void StartCalibrationStep3_MSwitch(int* errorId);
```

Starts the 3rd step of the Calibration phase for Aria instrument with external M-Switch.

Returns

errorId	int	Error ID (-1 if none)

5.2.104 ValidateCalibration

```
void ValidateCalibration(int* errorId);
```

Validates the current Calibration phase (results in saving the calibration volumes calculated and stopping the phase).

Output

errorId	int	Error ID (-1 if none)

5.2.105 CancelCalibration

```
void CancelCalibration(int* errorId);
```

Cancels the current Calibration phase (results in discarding the calibration volumes calculated and stopping the phase).

Output

errorId	int	Error ID (-1 if none)

5.2.106 StartCleaning1_Water

```
void StartCleaning1_Water(int[] reservoirNumbers, int bufferReservoirNumber,
    int* errorId);
```

Starts the step 1 of the Cleaning procedure : the cleaned **reservoirNumbers** should be filled with WATER, as well as the **bufferReservoirNumber** (9 - 10).

Parameters

reservoirNumbers	int[]	IDs of the reservoirs to be cleaned
bufferReservoirNumber	int	ID of the buffer reservoir (9 or 10)

Output

errorId	int	Error ID (-1 if none)

5.2.107 StartCleaning2_Tergazyme

```
void StartCleaning2_Tergazyme(int[] reservoirNumbers, int
    bufferReservoirNumber, int* errorId);
```

Starts the step 2 of the Cleaning procedure : the cleaned **reservoirNumbers** should be filled with TERGAZYME, as well as the **bufferReservoirNumber** (9 - 10).

Parameters

reservoirNumbers	int[]	IDs of the reservoirs to be cleaned
bufferReservoirNumber	int	ID of the buffer reservoir (9 or 10)

Output

errorId	int	Error ID (-1 if none)

5.2.108 StartCleaning3_Air

```
void StartCleaning3_Air(int[] reservoirNumbers, int bufferReservoirNumber, int*
    errorId);
```

Starts the step 3 of the Cleaning procedure : the cleaned **reservoirNumbers** should be EMPTY, as well as the **bufferReservoirNumber** (9 - 10).

Parameters

reservoirNumbers	int[]	IDs of the reservoirs to be cleaned
bufferReservoirNumber	int	ID of the buffer reservoir (9 or 10)

Output

errorId	int	Error ID (-1 if none)

5.2.109 StartCleaning4_IPA

```
void StartCleaning4_IPA(int[] reservoirNumbers, int bufferReservoirNumber, int*
    errorId);
```

Starts the step 4 of the Cleaning procedure : the cleaned **reservoirNumbers** should be filled with IPA, as well as the **bufferReservoirNumber** (9 - 10).

Parameters

reservoirNumbers	int[]	IDs of the reservoirs to be cleaned
bufferReservoirNumber	int	ID of the buffer reservoir (9 or 10)

Output

errorId	int	Error ID (-1 if none)

5.2.110 StartCleaning5_Air

```
void StartCleaning5_Air(int[] reservoirNumbers, int bufferReservoirNumber, int*
    errorId);
```

Starts the step 5 of the Cleaning procedure : the cleaned **reservoirNumbers** should be EMPTY, as well as the **bufferReservoirNumber** (9 - 10).

Parameters

reservoirNumbers	int[]	IDs of the reservoirs to be cleaned
bufferReservoirNumber	int	ID of the buffer reservoir (9 or 10)

Output

errorId	int	Error ID (-1 if none)

5.2.111 CancelCleaning

```
void CancelCleaning(int* errorId);
```

Cancels the current Cleaning phase.

Output

errorId	int	Error ID (-1 if none)

5.2.112 SendTTLSignal

```
void SendTTLSignal(int* errorId);
```

Sends a TTL signal through the Aria instrument.

Output

errorId	int	Error ID (-1 if none)

5.2.113 StartAwaitingTTLSignal

```
void StartAwaitingTTLSignal(int period, int* errorId);
```

Starts waiting for a TTL signal. TTL check is done every **period** milliseconds (adapt with respect to incoming TTL pulse duration). Must be stopped by StopAwaitingTTL

Parameters

period	int	Check frequency (in ms)

Output

errorId	int	Error ID (-1 if none)

5.2.114 StartAwaitingTTLSignal

```
void StartAwaitingTTLSignal(int period, int timeout, int* errorId);
```

Starts waiting for a TTL signal. TTL check is done every **period** milliseconds (adapt with respect to incoming TTL pulse duration). Stops awaiting after **timeout** milliseconds.

Parameters

period	int	Check frequency (in ms)
timeout	int	Timeout duration (in ms)

Output

errorId	int	Error ID (-1 if none)

5.2.115 CheckTTLSignal

```
bool CheckTTLSignal(int* errorId);
```

Checks if a TTL signal was received. If it was, stops the current TTL waiting process.

Output

errorId	int	Error ID (-1 if none)

Returns

signalReceived	bool	Has a signal been received (true) or not (false)

5.2.116 StopAwaitingTLL

```
void StopAwaitingTLL(int* errorId);
```

Stops the current TTL waiting process.

Returns

errorId	int	Error ID (-1 if none)

5.2.117 SetTTLPulseDuration

```
void SetTTLPulseDuration(int duration, int* errorId);
```

Defines the duration of the TTL pulse sent by the Aria instrument (TTL pulse = **duration** * 100ms).

Parameter

duration	int	Number of 100ms periods for a single TTL pulse signal

Output

errorId	int	Error ID (-1 if none)

5.2.118 GetTTLPulseDuration

```
int GetTTLPulseDuration(int* errorId);
```

Gets the current duration of the TTL pulse sent by the Aria instrument. (TTL pulse = **duration** * 100ms).

Output

errorId	int	Error ID (-1 if none)

Returns

duration	int	Number of 100ms periods for a single TTL pulse signal

5.2.119 SendTCPMessage

```
void SendTCPMessage(string message, int* errorId);
```

Sends a TCP text message.

Parameter

message	string	Message to be sent

Output

errorId	int	Error ID (-1 if none)

5.2.120 StartAwaitingTCPMessage

```
void StartAwaitingTCPMessage(string awaitedMessage, int* errorId);
```

Start waiting for a TCP message with the content **awaitedMessage**.

Parameter

awaitedMessage	string	Content of the message to be waited for

Output

errorId	int	Error ID (-1 if none)

5.2.121 CheckTCPMessage

```
bool CheckTCPMessage(string awaitedMessage, int* errorId);
```

Checks if a TCP message with the content **awaitedMessage** was received. If it was, stops the process waiting for this message.

Parameter

awaitedMessage	string	Content of the message to be waited for

Output

errorId	int	Error ID (-1 if none)

Returns

signalReceived	bool	Has a signal been received (true) or not (false)

5.2.122 StopAwaitingTCPMessage

```
void StopAwaitingTCPMessage(string awaitedMessage, int* errorId);
```

Stops waiting for a TCP message with the content **awaitedMessage**.

Output

errorId	int	Error ID (-1 if none)

5.2.123 SetTCPMode

```
void SetTCPMode(bool enableServer, int* errorId);
```

Set TCP Mode : Servor or Client

Parameter

enableServer	bool	Enables server (true) or client (false) mode

Returns

errorId	int	Error ID (-1 if none)

5.2.124 IsTCPServerMode

```
bool IsTCPServerMode(int* errorId);
```

Checks if Aria is in TCP Server mode.

Output

errorId	int	Error ID (-1 if none)

Returns

serverMode	bool	Is Aria in TCP server mode (true) or not (false)

5.2.125 SetTCPPort

```
void SetTCPPort(int port, int* errorId);
```

Define the TCP port used for the TCP client and server.

Parameter

port	int	TCP client and server port number

Output

errorId	int	Error ID (-1 if none)

5.2.126 GetTCPPort

```
int GetTCPPort(int* errorId);
```

Stops the current TTL waiting process.

Output

errorId	int	Error ID (-1 if none)

Returns

port	int	TCP client and server port number

FLUIGENT

O'kabé bureaux

57-77, avenue de Fontainebleau

94270 Le Kremlin-Bicêtre

FRANCE

Phone: +331 77 01 82 68

Fax: +331 77 01 82 70

www.fluigent.com

Technical support:

support@fluigent.com

Phone : +331 77 01 82 65

General information:

contact@fluigent.com