DEWETRON

DEWETRON
TRION SDK MANUAL

Version: 2.6
Date: 14/05/2019
Author: Gunther Laure

DEWETRON

The information contained in this document is subject to change without notice.

DEWETRON GmbH (DEWETRON) shall not be liable for any errors contained in this document. DEWETRON MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. DEWETRON SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

DEWETRON shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Technical Support

Please contact your local authorized DEWETRON representative first for any support and service questions.

For Asia and Europe, please contact:

DEWETRON GmbH
Parkring 4
8074 Grambach
AUSTRIA
Tel.: +43 316 3070
Fax: +43 316 307090
Email: support@dewetron.com
Web: http://www.dewetron.com

For America, please contact:

DEWETRON, Inc.
PO Box 1460
Charlestown, RI 02813
U.S.A.
Tel.: +1 401 364 9464
Toll-free: +1 877 431 5166
Fax: +1 401 364 8565
Email: support@dewamerica.com
Web: http://www.dewamerica.com

The telephone hotline is available Monday to Friday between 08:00 and 17:00 GST (GMT -5:00)

Restricted Rights Legend:

Use Austrian law for duplication or disclosure.

DEWETRON GmbH
Parkring 4
8074 Grambach
AUSTRIA

Printing History:

DEWETRON

Before updating your software please contact DEWETRON. Use only original software from DEWETRON.

Please find further information at www.dewetron.com.

DEWETRON

DOCUMENT HISTORY

Document Type:	Technical Document
Document Name:	TRION SDK Manual
Document Owner:	Gunther Laure

Date	Author	Changes	Rev.
2012-07-24	M. Poniz	First version of the document.	0.1
2012-07-26	J. Goetzenauer	Review of the document.	1.0
2012-08-10	M. Poniz	Synchronous channel readout clarification	1.1
2012-08-14	M. Poniz	Added section 'To be Added to Document'	1.2
2012-09-13	M. Poniz	Added command glossary section	1.3
2012-09-15	M. Poniz	Added baseline information about asynchronous data channels	1.4
2012-10-03	J. Goetzenauer	Review of the document.	1.5
2013-03-27	M. Poniz	Added Advanced Commands and Examples	1.6
2014-01-22	G. Laure	64bit Build options	1.7
2014-09-02	A. Lengauer	Scan Descriptor update	1.8
2014-11-14	M. Poniz	Added the more advanced I32 commands, updated document to reflect recent interface-changes, Added information regarding Amplifier constraints	1.9
2015-08-14	M. Tangl	Updated descriptions for Hardware SelfTest functions	2.0
2016-09-12	H. Ploner	Added section on resistances for RTD in Three-Wire Configuration	2.1
2017-06-09	M. Straka	Updated ScanDescriptor to Version 2	2.2
2018-06-21	G. Laure	Operating Systems, TRIONET Examples	2.3
2018-11-20	G. Laure	Synchronizaton chapter	2.4
2019-03-06	G. Laure	ExtTrigger Setting updated	2.5
2019-05-14	W.Gaggl	Fixed Typos in TRIONET chapter	2.6

DEWETRON

Table of Contents

DEWETRON

1 PREFACE

1.1 Used Terms and Abbreviations

Term	Description
ADC	Analogue-Digital Converter
AI	Analogue Input Channels
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
Board	A Data Acquisition Board
DI	Digital Input Channels
DMA	Direct Memory Access
Scan	The set of one sample for each channel on a board
SDK	Software Development Kit
SPS	Samples per second
XML	eXtendable Markup Language

1.2 Scope of This Document

This document describes the architecture of the DEWETRON TRION SDK in an abstract way.
It is intended to provide an overview of the interface and the used mechanisms to interact with the actual data acquisition hardware.

1.3 Supported Operating Systems

Following operating systems are supported:

- Windows 7 32/64bit
- Windows 10 64bit
- Ubuntu 1604 LTS
- Ubuntu 1804 LTS
- Redhat Enterprise Linux

Support for other Linux distributions is possible on customer request.

1.4 SUPPORTED PROGRAMMING LANGUAGES

1.4.1 NATIVELY SUPPORTED PROGRAMMING LANGUAGES

The DEWETRON TRION SDK is shipped with the necessary interface files for C/C++ (.h -Files) and Pascal-dialects like Delphi (.pas – Files) Using these shipped files, direct integration into the application can be achieved by including the interface files.

1.4.2 INDIRECTLY SUPPORTED LANGUAGES

- All public Interface functions are using stdcall – calling convention.

- All public interface functions take only following primitive data-types:
  - 32-Bit Integers and pointers to 32Bit Integers,
  - 64-Bit Integers and pointers to 64Bit Integers,
  - Pointers to zero-terminated ASCII-Strings (no Unicode/Wide string Support)

- All public interface functions are accessible by using the Windows® API function `GetProcAddress()`.

Any programming language that allows the above mentioned perquisites can exploit the functionality of the API.

2 ARCHITECTURE OVERVIEW

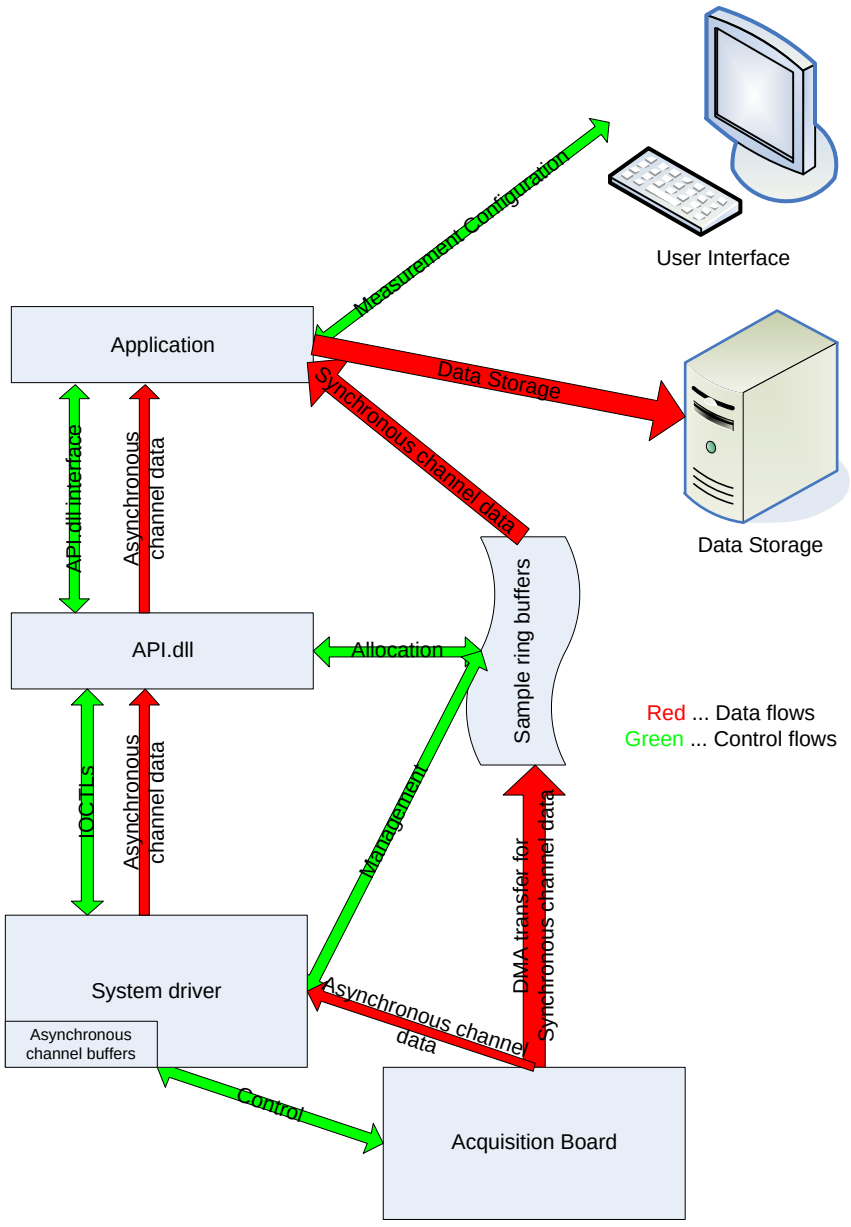

Illustration 1 - Driver Architecture

2.1 COMPONENTS

2.1.1 SYSTEM DRIVER (KERNEL MODE DRIVER)

The kernel driver is providing the actual hardware access and the DMA mechanics for data acquisition.

2.1.2 API

The API is providing the interface to the application.

2.1.3 APPLICATION

This is the actual application, using the measurement hardware, by interfacing to it via the API.

2.2 DATA ACQUISITION

2.2.1 SYNCHRONOUS DATA CHANNELS

2.2.1.1 The Acquisition Buffer

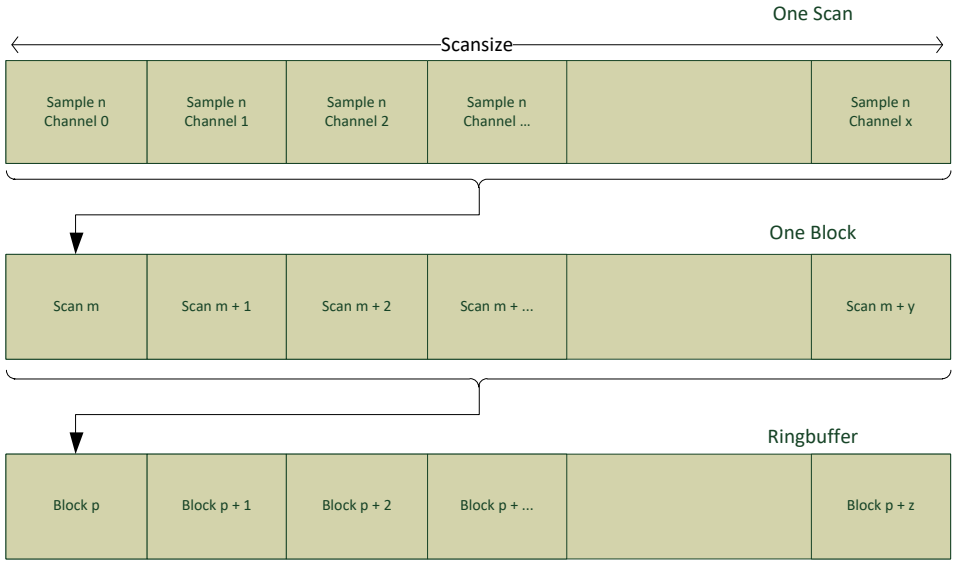

Illustration 2 - Acquisition Ring-Buffer

2.2.1.1.1 Used Terms

2.2.1.1.1.1 Scan and Scan Size

One scan is the portion of data that consists of exactly one sample for each sampled channel on a board.
So if there are 2 analogue channels and 1 counter channel active, the scan would logically hold three values. (AI0, AI1, CNT0).
The scan-size therefore directly derives from this information. It describes the memory-consumption of one scan in Bytes. In above example, when using the AI-channels in 24Bit mode (consuming 32Bit per Sample) the resulting scan size would be:

```
ScanSize := sizeof(AI0) + sizeof(AI1) + sizeof(CNT0)
Scansize := 32Bit + 32Bit + 32Bit
Scansize := 4 Byte + 4 Byte + 4 Byte
Scansize := 12 Byte
```

So one scan would have the size of 12 Byte.

The scan size cannot be directly controlled by the application as it directly depends on the number and type of activated channels.
Usually the application does not have to know very detailed about one scan and its layout inherently, as there are ways to get this information from the API in an abstracted way at runtime.

2.2.1.1.1.2 Block and Block Size

One block is a collection of *n* scans.
It is only meant as a logical unit and does not directly influence the driver in any way. Usually it is set up in accordance with the polling-interval of the application.
The block-size can be set to any arbitrary value > 0. A standard use case would set it to SampleRate * pollingIntervall. For Example:

```
BlockSize := SampleRate * pollingIntervall
BlockSize := 2000 SPS * 0.1 sec
BlockSize := 200
```

This has to be set by the application.

2.2.1.1.1.3 Block Count

This defines how many blocks the buffer shall be able to hold. This allows the application to control how big the backlog of data shall be and thus how much time the application may spend with tasks not related to the acquisition – so that peaks in computation times won't lead to lost acquisition data.
It can be set to any value > 0, and is only limited by the total available memory.
For example:

```
BlockCount := 50
```

This has to be setup by the application.

2.2.1.1.1.4 Total Buffer Size

The total buffer size is calculated based on the above described information.

```
Buffersize := ScanSize * BlockSize * BlockCount
```

In our example:

```
BufferSize := 12 Bytes * 200 * 50
BufferSize := 12 Bytes * 10000
Buffersize := 12000 Bytes
```

2.2.1.2 Acquisition of Synchronous Channels

Each sampling period produces one sample for each channel and consumes "Scan Size" amount of data in the buffer. There are currently three kinds of synchronous data in the buffer: analog channel samples, counter channel samples and digital channel samples.

The driver itself maintains a separate read- and write-pointer into this buffer. So the hardware can add new samples independent of the applications data-processing.
The driver will notify the application with an error-code if a buffer-overrun occurs. That is, if the application processes data too slow, so that the new samples have already overwritten unprocessed old ones.
The application then can freely decide how to handle this error case.

2.2.1.3 Buffer Setup and Buffer Ownership

The buffer itself is completely maintained inside the API – so the applications do not have to bother with allocation and de-allocation issues, which usually come with having a buffer.
However – to allow the application a fine granulated control over the buffer, it is able and obligated to indicate to the API the desired size of the buffer in terms of logical units, by using the integer-based functions. The application decides, how many scans one block shall hold, and how many blocks shall be allocated. The actual size in Bytes is then calculated by the API, and the buffer is allocated.

2.2.1.4 Buffer Readout from Application Point of View

The ring-buffer is exposed to the application by providing the related pointer information.
The API will provide:

- Start-pointer of the ring buffer
- End-Pointer of the ring buffer
- Pointer to the first unprocessed scan
  Together with the information how many unprocessed samples are available the application iterates directly over the ring buffer.
  This approach allows a minimal internal overhead on data-access.

2.2.1.5 Scan Layout

As mentioned in Chapter 2.2.1.1.1.1, a scan is the portion of data containing exactly one sample per used channel. Without knowledge about its internal layout, this would just be a binary stream with arbitrary length.
But the application does not need to know implicitly about the layout of the data. This would be undesirable, as the layout may change with coming driver versions or coming hardware. For example, when a new type of synchronous data will be added, inherent hardcoded knowledge within the application would immediately break the data-readout mechanism of the application.
So after setting up the acquisition environment, the API can be queried about the layout.
The detailed layout-information will be returned as an XML-string.
This information would look like:

```
<ScanDescriptor>
  <BoardId0>
    <ScanDescription version="2" scan_size="96" byte_order="little_endian"
     unit="bit">
    <Channel type="Analog" index="3" name="AI3">
      <Sample offset="32" size="24" />
    </Channel>
    </ScanDescriptor>
  </BoardId0>
</ScanDescriptor>
```

Illustration 3 - ScanDescriptor Example

For further details of the scan descriptor's document structure, please reference the following Section 2.2.1.6.

2.2.1.6 Scan Descriptor Structure

The following API string command returns the scan information for a specific Board:

```
DeweGetParamStruct_Str( "BoardId0", "ScanDescriptor_V2", Buf, sizeof(Buf));
```

The returned XML document correlates with the following hierarchy:

1. <ScanDescriptor> : XML Element. Max. Occurrences: 1.
2. <BoardID0> : XML Element. Max. Occurrences: 1.
3. <ScanDescription> : XML Element. Max. Occurrences: 1.
   - 3.1 <Channel> : XML Element. Max. Occurrences: Unbounded.
     - 3.1.1. <Sample> : XML Element. Max. Occurrences: Unbounded.

Please be aware that the scan descriptor annotates only the enabled channels for a specific Board. In case no channel is enabled, the API returns an empty scan descriptor with "scan_size" set to the value 0.
The API considers disabled channels and therefore the returned "scan_size" and "offsets" are being returned accordingly.

The following table depicts all possible XML Elements and their XML attributes and values of the returned scan descriptor XML document:

XML Element	*XML Attribute(s)*	*Value*	*Note*
ScanDescriptor	-	-	An empty Element, same as the API call command. Always “ScanDescriptor”.
BoardID	-	-	An empty Element, same as the API call Target. E.g. “BoardID0”.
ScanDescripton	version, scan_size, byte_order	-	Describes the scan for the requested Board.
ScanDescription	version	“2”	Scan descriptor’s document version. For this API version always “2” when requested with “ScanDescriptor_V2”. Requesting “ScanDescriptor” will return version “1”.
ScanDescription	scan_size	Integer value	The size of the scan expressed in the unit, described via the attribute “unit”.
ScanDescription	byte_order	“little_endian”	Describes the byte order of the scan. Only little endian is supported with this API version.
ScanDescription	unit	“bit”	Describes the unit of “scan_size” attribute for the children attributes “offset” and “size”. Always “bit” for this API version.
Channel	index, name	-	Describes an enabled channel.
Channel	index	Integer Value	The channel index on the specific Board.

XML Element	*XML Attribute(s)*	*Value*	*Note*
Channel	name	String Value	e.g. name=”AI1” The composite string “BoardID1/AI1” can be used as Target for further API calls. For further details regarding the API Targets please reference Chapter 3.5.
Channel	type	Possible Value: • “Analog” • “Counter” • “Discrete” • “BoardCounter”	Discrete Channels describe digital channels.
Sample	offset, size, [subChannel]	-	Describes an available sample of a specific channel.
Sample	offset	Integer Value	The sample offset within the whole scan. The unit is according the attribute “unit” of the “ScanDescription” Element.
Sample	size	Integer Value	The sample size. The unit is according the attribute “unit” of the “ScanDescription” Element.
Sample	subChannel	Possible Value: • “Counter/Sub” • “BoardCounter/Sub”	Optional Attribute: Only returned if a subchannel is enabled e.g. “BoardCounter/Sub”

For an example usage of the scan descriptor, please reference to Chapter 3.3.2.

2.2.1.7 Restrictions

When requesting a scan descriptor with command “ScanDescriptor” (Version 1), some newer board may not be able to return a valid scan descriptor for analog 24bit channels. Therefore, always use “ScanDescriptor_V2”.

2.2.2 ASYNCHRONOUS DATA CHANNELS

2.2.2.1 The Acquisition Buffer

Compared to the synchronous data channels, asynchronous data channels have a considerably lower data rate. This is true even if many asynchronous channels are used simultaneously.
Also unlike synchronous channels there is an expected amount of incoming data per time-unit. Asynchronous messages can be several seconds apart from each other or even within the same millisecond. So the data influx is non-deterministic.
This fact is exploited to reduce buffer-maintenance-overhead within the driver and to make use of a more suitable DMA approach for this kind of non-deterministic data.
Unlike the acquisition buffer for synchronous data, that is maintained dynamically depending on several input parameters, the buffer for asynchronous data is static and allocated with a reasonably big size. (Default is 64kB).

2.2.2.2 Types of supported Asynchronous Channels

Up to Driver version 0.35.3530.0:

- CAN channels
- UART (RS485) channels

2.2.2.3 Acquisition of Asynchronous Channels

The kernel-mode driver queries the hardware in a configurable time interval if asynchronous data is available in the board hardware buffers.
If a certain threshold of data is exceeded a DMA transfer is started.
As asynchronous data is already time stamped on TRION hardware, even on sub-sample-count basis, the minimal delay introduced by this approach has no negative impact on correlation between asynchronous and synchronous data.

2.2.2.4 Buffer Setup and Buffer Ownership

The buffer itself is completely maintained by the driver itself and of fixed size. The application has no direct influence on the existence or the size of the buffer itself. (TBD: Detailed information about changing size by providing system-wide configuration data)

2.2.2.5 Buffer Readout from Application Point of View

The buffer is not directly exposed to the application.
To allow data access on acquired asynchronous data a specialized set of non-blocking function is available. Each set of functions is specialized to a specific type of asynchronous data.
For standard formats like for example CAN, definitions and functions to retrieve already decoded frames are available.
To allow the application for more granulated control in data-handling also functions to retrieve undecoded frames (raw frames) are available.

2.3 MAIN INTERFACE PARTS

This part describes the main interface parts, grouped by their type.

This part assumes that the API is used in one of the natively supported programming languages and omits the steps that would be necessary to invoke the API inside a not-natively supported programming language.

2.3.1 TYPE OF FUNCTIONS

The API supports three distinct types of functions to cover all tasks needed to interface with it and the data acquisition hardware.

1.) Administrative functions
2.) String based functions
3.) Integer based functions

2.3.2 ADMINISTRATIVE FUNCTIONS

These functions focus on strictly administrative parts of the API like loading, initializing and unloading.
The function signatures are tailored towards the intended use case and are therefore not as generic as for the other two types of interface-functions.

2.3.2.1 Loading

To use the API, the DLL first has to be loaded into the process-memory.

```
BOOLEAN DewePXILoad();
```

This function returns TRUE on success.

2.3.2.2 Initializing

A call to

```
int DeWeDriverInit( [out] int* nNumOfBoards );
```

initializes the API for usage, and returns the number of detected boards.

2.3.2.3 Uninitializing

Before closing the API, the application should uninitialize the API by calling:

```
int DeweDriverDeInit();
```

The API implementation is graceful, if this call is omitted.

2.3.2.4 Unloading

A call to

```
void RTDaqUnload();
```

will unload the API from the process memory.

2.3.3 STRING BASED FUNCTIONS

This set of functions covers the complete part of the logical configuration of the data acquisition system.
These functions are mainly used in not speed-critical parts of the API, e.g. when setting up the logical configuration of the measurement system.

The set consists of the following three functions:

- `int DeWeSetParamStruct_str( [in] const char* Target, [in] const char* Command, [in] const char* Var);`

- `int DeWeGetParamStruct_str( [in] const char* Target, [in] const char* Command, [out] char* Var, size_t num );`

- `int DeWeGetParamStruct_strLEN( [in] const char* Target, [in] const char* Command, int* Len);`

2.3.4 INTEGER BASED FUNCTIONS

This set of functions serves various purposes.
One example for its usage is to trigger a specific action on the hardware, like updating the hardware settings to match the logical settings.
Another possible use case of these functions is in places, where only information that is easily expressible as integers is passed to or from the API and a low latency is desired. So taking the sidestep of string-conversion inside the API and re-conversion on application level is avoided.
An example of this use case would be the polling for the actual number of acquired samples. This information clearly can be expressed as an integer value. As it is frequently used during acquisition, it should be as fast as possible.

The set consists of the following two functions:

- `DeWeGetParam_i32( [in] int nBoardNo, [in] int nCommandId, [out] int* pVal);`

- `DeWeSetParam_i32( [in] int nBoardNo, [in] int nCommandId, [in] int nVal);`

2.4 BASIC API USAGE SEQUENCE

This chapter shows how the API is basically intended to be used.
Some of the steps shown are optional and thus will be marked indicating this fact. Each step will be explained with a few sentences.
More detailed information can be found in Chapter 3.

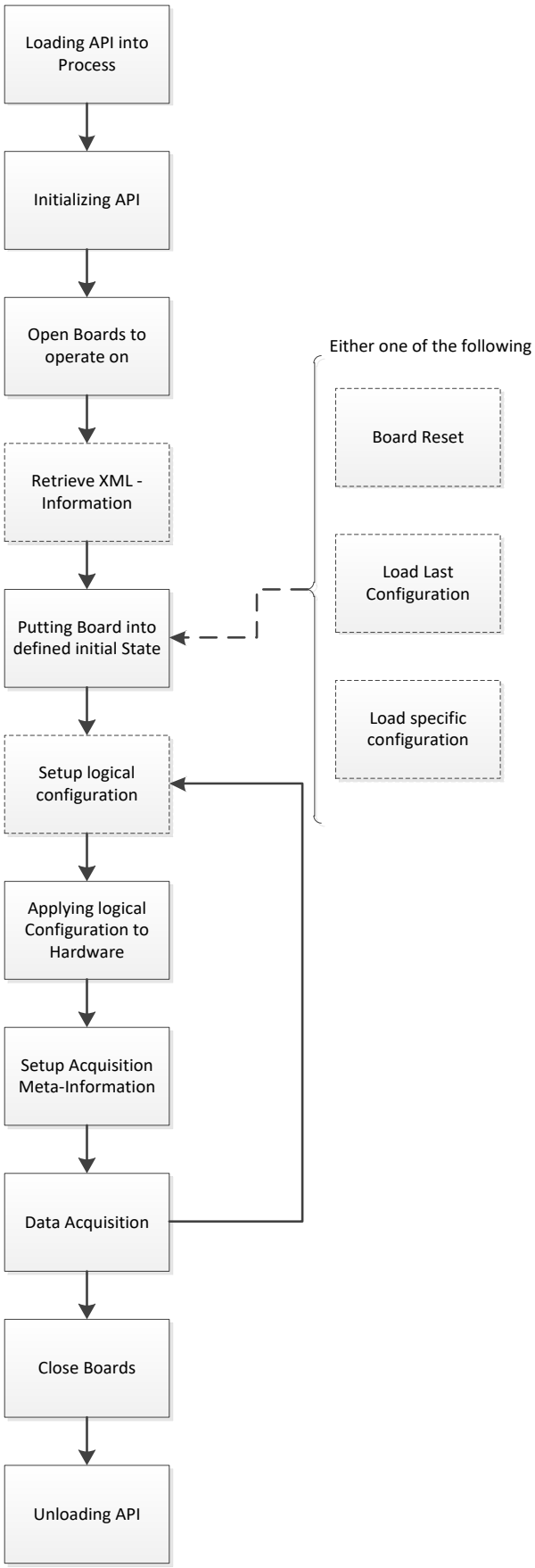
Loading API into Process
Initializing API
Open Boards to operate on
Retrieve XML - Information
Putting Board into defined initial State
Setup logical configuration
Applying logical Configuration to Hardware
Setup Acquisition Meta-Information
Data Acquisition
Close Boards
Unloading API
Either one of the following
Board Reset
Load Last Configuration
Load specific configuration

Illustration 4 - Basic API Usage Sequence

2.4.1 LOADING API

Requirement: Mandatory

Loading the API into the process memory space.

2.4.2 INITIALIZING THE API

Requirement: Mandatory

- Initialize the internal states of the API
- Enumerate available hardware (boards and enclosures)
- Extract and generate the baseline XML-Information for the found hardware

This step may be called several times. Only the first call will actually perform any action. All subsequent calls will notice that initialization already has been performed within this process and therefore omit further actions.
It is not necessary to keep the Initialize/DeInitialize – calls balanced.

2.4.3 LOGICAL BOARD-OPEN

Requirement: Mandatory

Board open has to be called, for each board that will be operated during the API usage.

- Generates detailed XML-Information about the hardware
- Basic hardware initialization

2.4.4 RETRIEVE XML-INFORMATION

Requirement: Optional but recommended

For each board a descriptive XML-File can be retrieved. This XML-File holds all valid configuration information for this specific type of board. This file serves as documentation about the board capabilities. So by looking at the specific XML-File it is obvious what configuration settings are valid on which properties. For a more detailed example see Chapter 3.9.

2.4.5 PUTTING THE BOARD INTO A DEFINED INITIAL STATE

Requirement: Mandatory – but either one of the following steps is sufficient

2.4.5.1 Board-Reset

Requirement: Optional

Loads the ‘default’ settings- as specified in the Boardproperties XML-File into the logical layer, and immediately apply the logical settings to the hardware. See Chapter 3.9 for details.
This will put the board into a ‘factory’-default state.

2.4.5.2 Load Last Configuration

Requirement: Optional

This will load the last used logical configuration into the logical layer and will immediately apply these settings to the hardware.
If no (valid) last setting is retrievable the default settings will be loaded and the result will be the same as if performing a Board-Reset.

2.4.5.3 Load a Specific Configuration

Requirement: Optional

Any previously stored configuration can be loaded into the logical layer of the board.
In this case it is the responsibility of the application to trigger the hardware update for this board.

2.4.6 Setting up the Logical Configuration

Requirement: Optional - After setting the board to a defined initial state this step could be omitted.

During this phase the various necessary parameters to set up the measurement system are applied to the logical configuration.

These parameters include for example:

- Desired sample-rate
- Detailed configuration for the various channels:
  Like for analogue channels the mode (Voltage, Resistance ...) or the range (1000 Ohm, 15 Volt …)
- Enabling and disabling of single channels for acquisition

All these settings are applied by using the string-based functions. Each function call will evaluate the passed information against the XML-definition for this board, and will indicate to the application if the information could be processed, was auto-adjusted, or erroneous.
The hardware is not yet affected during this step

2.4.7 Applying the Configuration to the actual Hardware

Requirement: Mandatory when logical changes have been performed

This will reflect the logical setting down to the actual acquisition hardware.

2.4.8 Setting up the Acquisition Meta-Information

Requirement: Mandatory

Setting up information not directly related to the hardware like

- Desired size of DMA-Buffer in samples

After this step, the acquisition system is ready to perform the data acquisition. From this point on, the application can request information about the DMA-buffer-layout from the API.
This information describes the layout of a single scan, and thus tells the application where within a single scan the acquired data of a specific channel can be found.

2.4.9 ACQUISITION-PHASE

The acquisition phase starts with the command `Start Acquisition` and ends with `Stop Acquisition`.
During this phase the application has to poll the API to obtain the number of available samples. As the application sets up the DMA-buffer-size according its own needs (see Chapter 2.4.8) the application can poll as frequent or infrequent as it sees the fit.
The application just has to make sure to be able to process acquired data quick enough so that the own selected buffer is never exceeded.

2.4.10 UNLOADING THE API

Requirement: Mandatory

When finished with using the API, the API has to be unloaded by the application.
This will trigger several internal operations like storing the current configuration to be re-used later on (see Chapter 2.4.5.2).

3 USING THE API

This section explains the steps described in Chapter 2.4 in more detail, and provides some basic examples.

3.1 RETURN CODE CONVENTION

A return code of 0 indicates on all API-interface-functions success.
A negative error code indicates a warning. As a general rule it is safe to assume that the operation requested has been performed.
A typical example of a warning would be for example requesting a low-pass-filter-frequency which cannot be set exactly to hardware. In this example the next lower possible frequency would be chosen by the API and the API would inform the application by issuing a corresponding warning.
A positive error code indicates an error. As a general rule this means that the requested operation could not be performed.
A typical example would be trying to set a property of a non-existing channel on a board.

3.2 PREPROCESSOR DEFINES

A small set of preprocessor defines can be used to select the correct API build. These should be set for files that include the dewepxi_load.h header file.

3.2.1 BUILD_X64

If BUILD_X64 is defined the API file name is set to its 64bit variant. The DeWePxiLoad function tries to load an API build for 64bit applications.

3.2.2 BUILD_X86

This is the default setting. If BUILD_X86 is defined DeWePxiLoad tries to load an API build for 32bit applications.

3.3 A SHORT EXAMPLE APPLICATION

This simple example application assumes that one board with counter-channels is installed. For sake of simplicity, error-handling is omitted in this example.

```
1   static int SampleCode()
2   {
3       int     nNoOfBoards;
4       int     nErrorCode = 0;
5
6       //Load the API
7       RTdaqLoad();
8
9       //Initial Driver, retrieve number of Boards
10      nErrorCode = DeweDriverInit( &nNoOfBoards );
11
12      //Open & Reset Board
13      nErrorCode = DeWeSetParam_i32( 0, CMD_OPEN_BOARD, 0 );
14      nErrorCode = DeWeSetParam_i32( 0, CMD_RESET_BOARD, 0 );
15
16      //By default, all channels are disable
17      //So enable 1 CNT-Channel
```

```
18      //And set its input to ACQ-Clock, so that we actually can see some Data
19      nErrorCode = DeWeSetParamStruct_str( "BoardID0/CNT0", "Used", "True" );
20      nErrorCode = DeWeSetParamStruct_str( "BoardID0/CNT0", "Source_A", "Acq_Clk" );
21
22      //Setup the Acquisition Buffer
23      //for the Default SampleRate of 2kSPS this would be a Block-size of 0.1 sec
24      nErrorCode = DeWeSetParam_i32( 0, CMD_BUFFER_BLOCK_SIZE, 200 );
25      //Ring-buffer is 50 Blocks large (so 5 secs in this sample)
26      nErrorCode = DeWeSetParam_i32( 0, CMD_BUFFER_BLOCK_COUNT, 50 );
27
28      //Update the Hardware and prepare Acquisition
29      nErrorCode = DeWeSetParam_i32( 0, CMD_UPDATE_PARAM_ALL, 0 );
30
31      //Data-Acquisition
32      //until we hit a Keyboard - Key
33      nErrorCode = DeWeSetParam_i32( 0, CMD_START_ACQUISITION, 0 );
34      if ( nErrorCode <= 0 ) {
35          int     nBufEndPos;         //Wrap around of Ring-Buffer
36          int     nBufSize;           //Total Size of Buffer
37
38          //Get Information, to be able to handle Buffer-Wrap-Around
39          nErrorCode = DeWeGetParam_i32( 0, CMD_BUFFER_END_POINTER, &nBufEndPos );
40          nErrorCode = DeWeGetParam_i32( 0, CMD_BUFFER_TOTAL_MEM_SIZE, &nBufSize );
41
42          while( !kbhit() ){
43              ::Sleep(100);               //wait for ~100ms of Samples
44              int     nCurPos;            //actually a pointer to Ringbuffer ReadPos
45              int     nAvailSamples;
46              //Get No of Samples, Current Read-Pointer- Position
47              nErrorCode = DeWeGetParam_i32( 0, CMD_BUFFER_AVAIL_NO_SAMPLE, &nAvailSamples );
48              nErrorCode = DeWeGetParam_i32( 0, CMD_BUFFER_ACT_SAMPLE_POS, &nCurPos );
49
50              for ( int i = 0; i < nAvailSamples; i++ ){
51              //Only one channel active, so we can Advance sizeof(DWORD) on each step hard-coded
52                  DWORD           dwRawData = *(DWORD*)nCurPos;
53                  cout << dwRawData << endl;
54                  //move along in Buffer
55                  nCurPos += sizeof(DWORD);
56                  //Handle Wrap around
57                  if ( nCurPos > nBufEndPos ){
58                      nCurPos -= nBufSize;
59                  }
60              }
61              //And allow the Data to be freed
62              nErrorCode = DeWeSetParam_i32( 0, CMD_BUFFER_FREE_NO_SAMPLE, nAvailSamples );
63          }
64      }
65      nErrorCode = DeWeSetParam_i32( 0, CMD_STOP_ACQUISITION, 0 );
66      //Finished - Unload the Driver
67      nErrorCode = DeWeSetParam_i32( 0, CMD_CLOSE_BOARD, 0 );
68      RTdaqUnload();
69      return 0;
70  }
```

Sample Code 1 - Simple Acquisition Application

3.3.1 WHAT DOES THE SAMPLE-CODE ACTUALLY DO?

This code is fully functional and will write the raw counter values of one active channel into the console. Of course for a real application a more thorough error handling would be obligate.
The following paragraphs will provide a step-by-step description of the code.

Line 7 to 10 handle loading and initializing the API as mentioned in Chapter 2.4.1 and 2.4.2.

Line 12 and 13 open the board for logic operations and put it into a defined default-state (see Chapter 2.4.3 and 2.4.5).

After this step, the logical configuration is put into a default-state. The actual default-values for any given board can be examined by looking at the XML-Description. For more details see Chapter 3.9.
The default-state for all channels is to be turned off. So to be able to actually acquire data and process it at least one channel has to be turned on.

Line 19 and 20 show how this is achieved. While line number 19 simply sets the channel to a logical used-state, line number 20 sets the input-source for this counter-channel to the acquisition clock – to make the counter actually counting. Otherwise the input would be routed by default to the input-pin, and an external signal would be needed to make the counter count. This probably would be the default-use-case, but for the example using an internal source is easier.
No hardware is set yet. All these changes are only related to the logical layer.

Line 24 to 26 deals with setting up the ring-buffer for the data-acquisition. To do so, the driver has to know how big we want the buffer to be. This is usually depending on the applications desired polling-interval and its capability to process the data. This allows for a fine granulated control for the application and can for example be used to setup the block-size and polling interval in a way to exploit the CPU-internal caches most efficient, when it comes to more than simple data-acquisition (like when complicated mathematical evaluation has to be applied to the acquired samples). See Chapter 2.2 for more details.

Line 29 finally reflects all logical settings to the hardware and sets up the acquisition environment, like allocating the ring-buffer.
From this point on it is possible to query the API for detailed information about the ring-buffer necessary to process the acquired data (see Chapter 3.3.2 to learn in detail what this simple example application does not show)

Line 39 and 40 retrieve the end-pointer of the ring-buffer and its total memory size – so the application is able to detect the wrap-around during readout and handles it.

Line 33 starts the actual acquisition of data. From that point on the application is able to see data in the acquisition buffer. After starting the acquisition the example code polls the API roughly all 100ms for new data.

Line 47 queries the currently number of available new samples in the acquisition buffer.

Line 48 queries for the current position of the read-pointer. So the application does not necessarily need to keep track of the read-pointer by itself.

The loop from **line 50 to line 60** deals with actually reading out the acquired data, and processing of the data (in this case, simply writing the read data to the console).
Reading out data is as simple as dereferencing the read-pointer and incrementing the pointer by scan-size. As there is only one active 32Bit channel in the example, incrementing the pointer by a fixed size is easy and sufficient to show the working principle. For a more advanced usage see Chapter 3.3.2 and 2.2.1.5.

After processing the currently acquire data is finished, the driver has to be notified that the buffer may now be reused for acquisition.

Line 62 notifies the Driver that the application has finished processing `nAvailSamples` of Samples.

Lines 42 to 64 are repeated until a key is hit on the keyboard. After the acquisition is finished, the acquisition process is stopped.

Line 65 sends the command to stop the current acquisition.

Line 67 and 68 deal with logically closing the board and unloading the driver

3.3.2 WHAT IS MISSING IN THE EXAMPLE CODE?

From point of most basic operation the example code is pretty complete. More sophisticated channel-setups or more sophisticated system setups, involving more boards, and even multi-enclosure synchronization scenarios are not that different. All of these tasks can be achieved by setting the desired properties with the same string-base-functions as shown in line 19/20.

The biggest simplification in the sample-code surly is the hardcoded assumption about the scan-layout. As long as only one channel of known data-width is used, this is easy. See Chapter 2.2.1.5 about more details about the scan-layout. Real production code would have to process this API-provided information to navigate within the buffer. There are several ways the application could use this meta-information for navigation. Depending on its own internal needs, the application may choose to process the acquired data scan-by-scan or channel-by-channel. Even multi-threaded approaches are easily possible. The application would request the information between line 29 and 33 – so after setting up the environment but before starting the acquisition.

The information is requested with a simple string-command:

```
DeweGetParamStruct_Str( "BoardId0", "ScanDescriptor_V2", Buf, sizeof(Buf));
```

In the sample-code, the returned information would look like this:

```
<?xml version=”1.0”>
<ScanDescriptor>
  <BoardID0>
    <ScanDescription version="2" scan_size="160" byte_order="little_endian"
     unit="bit">
      <Channel index="1" name="CNT1" type="Counter">
        <Sample offset="0" size="32" />
      </Channel>
      <Channel index="2" name="CNT2" type="Counter">
        <Sample offset="32" size="32" subchannel="Counter/Sub" />
      </Channel>
      <Channel index="3" name="CNT3" type="Counter">
        <Sample offset="64" size="32" />
        <Sample offset="96" size="32" subchannel="Counter/Sub" />
       </Channel>
       <Channel index="5" name="CNT5" type="Counter">
         <Sample offset="128" size="32" />
       </Channel>
     </ScanDescription>
  </BoardID0>
</ScanDescriptor>
```

With this information the application would be able to know:

- One scan has the size of 20 Bytes (attribute ScanDescription/scan_size, which is 160 Bit)
- The data for channel CNT1 is found at position 0 within one scan (Channel/Sample/offset, which is 0 Bit)
- The data acquired has a logical size of 32 Bit. This means that really 32 Bits are used for data. (Channel/Sample/size)
  For example, a 24bit ADC would result in a data size of 24 but may consume 32 Bit in the buffer. New boards can pack 24bit continuously in the scan.
- 4 channels are enabled: Therefore 4 Channel Elements are returned.

3.4 ADDRESSING TARGETS WITH _I32 COMMANDS

- `DeWeGetParam_i32( [in] int nBoardNo, [in] int nCommandId, [out] int* pVal);`

- `DeWeSetParam_i32( [in] int nBoardNo, [in] int nCommandId, [in] int nVal);`

`nBoardNo` is the logical board-number. This parameter is used in all commands as the target is always a board.

`nCommandID` is the command to be executed like:

- `CMD_OPEN_BOARD`
- `CMD_START_ACQUISITION`
- `CMD_RESET_BOARD`

An exhaustive list of commands can be found in the interface definition files.

`nVal` is depending on the Command-ID. Some commands don't take a parameter at all. In this case the parameter is N/A.

3.5 ADDRESSING TARGETS WITH _STR COMMANDS

- `int DeWeSetParamStruct_str( [in] const char* Target, [in] const char* Command, [in] const char* Var)`

Target:
Usually either a board or a specific subsystem of a board is addressed with these types of commands. The target is expressed as a string like "BoardID0"

Command (or Item):
This indicates the item that is intended to be set or get. Most valid items can be identified by looking at the XML description. See Chapter 3.8 for more details.
For some special items it is possible to use a shorthand (Alias) target.
A typical item would be for example "BoardName" for human readable name of the board-type. (e.g. TRION-2402-dACC.6.BN)

Var:
For the set-function this is the actual value to set expressed as string.
For the get-function this is a pointer to the buffer where the result will be returned.

Typical targets (see Chapter 3.8 for a better understanding, how to derive these):

Target	***Description***
"BoardID0"	Accessing items of the logical board 0
"BoardID0/AI1"	Accessing properties for the analogue channel 1 on board 0
"BoardID1/CNT0"	Accessing counter channel 0 on board 1
"BoardID0/AcqProp"	Accessing the acquisition properties for board 0
"BoardID1/AcqProp/Timing"	Accessing the timing-properties for board 1

Typical combination of Target and Item (see Chapter 3.8 for a better understanding, how to derive these):

Target	*Item*	*Description*
"BoardID1/AI1"	"Mode"	Operation mode for analogue channel 1 on board 1
"BoardID0/AcqProp"	"SampleRate"	Sample rate for board 0
"BoardID1/CN2"	"Used"	Enabled status for counter channel 2 on board 1

3.6 Initializing the API

```
int DeweDriverInit( [out] int* nNumOfboards );
```

This function will return the number of detected boards to the application.

3.7 Open the Board

```
int DeweSetParm_i32( nBoardNo, CMD_OPEN_BOARD, 0 );
```

3.8 Retrieve XML Description

The XML properties can either be retrieved as a string or can be written to a file.
While retrieving the description as a string is usually faster, dumping the file to disk may help in understanding the concepts.

To retrieve the XML-description into a buffer for application internal processing:

```
int DeweGetParmaStruct_Str( "BoardID0", "BoardProperties", Buf, sizeof(Buf) );
```

To request the XML-description as a physical file on the hard-disk:

```
int DeweGetParmaStruct_Str( "BoardID0", "BoardPropertiesFile", "C:\Properties.XML",
1 );
```

The location can be any arbitrary valid path. Please make sure that the application actually has write access to the requested location.

3.9 BoardProperties XML-File

3.9.1 Purpose of the File

The Boardproperties XML-file serves several purposes:

- It holds information specific to a single board, like
  - Board-type and Board-name
  - The serial number
  - Information in which enclosure and at which slot within the enclosure this board can be found in the physical system
- It is the implicit documentation of the capabilities that one specific board offers
- It is the implicit documentation of all settable properties of a single board together with the definition set for each property.

So instead of having the risk of a potentially outdated paper-documentation about the logical capabilities of a single board, this XML-File can and should be used as a look-up reference while developing an application.

This file is generated at runtime and will therefore always reflect the actual capabilities of the current API-version in conjunction with the current capabilities of one specific board. So for example if either the API and/or the firmware of a specific board are enhanced with new functionality that is externally accessible, this fact will be immediately visible in the generated file.

This file is internally used by the API for parameter-checking. Using this file as base for the application guarantees using valid settings that will be accepted by the API.

Not all information within this file is strictly necessary for operating the board. Some of the information only serves documentation or maintenance purposes.

3.9.2 Relevant Sections of the File

The top-level nodes of the XMLfile are:

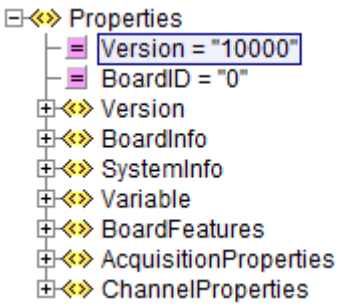

Illustration 5 - BoardProperties Root Node

3.9.2.1 Version

This node holds detailed information about the API and the input-source used for generating the properties file. Its main use is for documentation purposes.

3.9.2.2 BoardInfo

This node holds detailed information about the board itself like

- Serial number
- Firmware version information
- Administrative information about the calibration

3.9.2.3 SystemInfo

This node holds detailed information about the enclosure the board is residing in. For example the PXI-slot-number could be extracted from here to be shown in the application User Interface.

3.9.2.4 Variable

This node only serves documentation purposes and is not meant to be used by the application directly.

3.9.2.5 BoardFeatures

This section roughly describes the acquisition capabilities of the board at hand. This example shows a TRION-2402-dACC.6.BN (analogue sampling board with six analogue channels)

```
<BoardFeatures>
        <AI>
            <Resolution Count = "2" Default = "0">
                <ID0>24</ID0>
                <ID1>16</ID1>
            </Resolution>
            <Channels>6</Channels>
        </AI>
        <ARef>
            <Channels>1</Channels>
        </ARef>
        <CNT>
            <Resolution>32</Resolution>
            <TimeBase Unit = "MHz">80</TimeBase>
            <Channels>2</Channels>
        </CNT>
        <BoardCNT>
            <Resolution>32</Resolution>
            <TimeBase Unit = "MHz">80</TimeBase>
            <Channels>1</Channels>
        </BoardCNT>
</BoardFeatures>
```

Illustration 6 - BoardProperties - BoardFeatures Node

From this information the application can deduce:

- The board has six analogue input channels
- The analogue input channels can be used with 16 and 24 Bit resolution
- The default resolution is 24 Bit
- 2 counter channels are available with a 80MHz resolution

- 1 internal counter channel is available - a so called BoardCounter
- No digital or CAN channels are available

3.9.2.6 AcquisitionProperties

This node holds very detailed information about the various settings necessary for the synchronization capabilities of the board and general settings affecting the acquisition itself (e.g. the sample-rate).

This node is very elaborate and needs only to be considered in detail for more advanced setup. Discussing these nodes in detail is beyond the scope of this document. So its contents are not fully shown here.

One more general sub-node within this node is AcqProp

3.9.2.6.1 AcqProp

This node holds generic setup information about the acquisition parameters for this board.

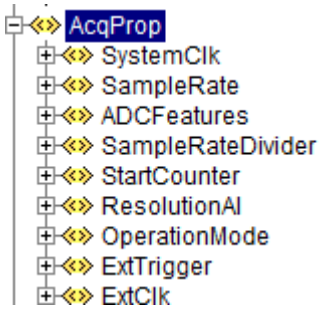

Illustration 7 - BoardProperties - AcqProp Node

The most interesting sub-elements within here probably are:

- SampleRate
- OperationMode
- ResolutionAI

3.9.2.6.2 SampleRate

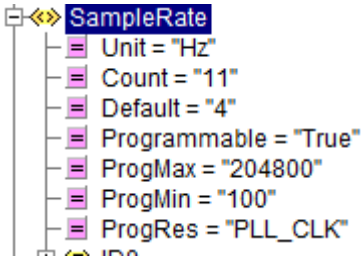

Illustration 8 - BoardProperties - SampleRate Node

This allows the application to know the upper and lower limits of the available sample-rates. In this case the range goes from 100 Samples/second up to 200 kSamples/second

3.9.2.6.3 OperationMode

```
<OperationMode Count = "3" Default = "0">
      <ID0>Slave</ID0>
      <ID1>Master</ID1>
      <ID2>Custom</ID2>
</OperationMode>
```

Illustration 9 - BoardProperties - OperationMode Node

This property allows selecting the predefined roles of the board within a multi-board system. Together with the information about external clocking and external triggering this will automatically set up the routing for the internal trigger- and clock-lines to a predefined state that is suited to make the board fulfill its desired role.

3.9.2.6.4 ResolutionAI

```
<ResolutionAI Count = "2" Default = "0">
       <ID0>24</ID0>
       <ID1>16</ID1>
</ResolutionAI>
```

Illustration 10 - BoardProperties - ResolutionAI Node

This property allows setting the analogue channels to a desired ADC-resolution.

3.9.2.7 ChannelProperties

This node gives exhaustive information about all available acquisition channels and all their settable properties.

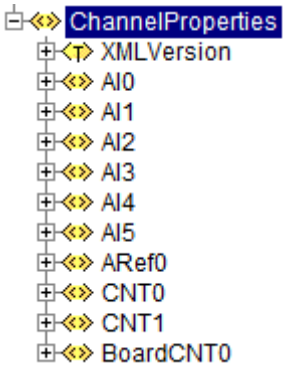

Illustration 11 - BoardProperties - ChannelProperties Node

In this case, the XML-File shows that:

- There are six analogue channels, labeled AI0 to AI5
- Two counter channels, labeled CNT0 and CNT 1

- One internal counter (the Board-Counter), labeled BoardCNT0

The basic layout for all the channel-types is always the same and allows for easy initial navigation within the node. The analogue channel 0 is used as example to explain this in more depth.

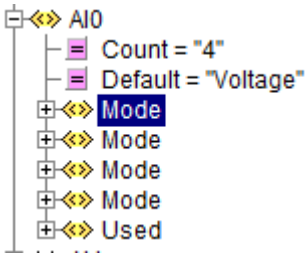

Illustration 12 - BoardProperties - AI0 Channel Node

The first layer always holds:

- a list of supported operation modes
- the Used-flag itself as it is independent of the chosen mode

In this example the list of modes is:

- `<Mode Mode = "Calibration">`
- `<Mode Mode = "Voltage">`
- `<Mode Mode = "Resistance">`
- `<Mode Mode = "IEPE">`

Looking at Illustration 12 - BoardProperties - AI0 Channel Node, the default indicates that the mode “Voltage” is set as default.

Each of the modes lists its own associated properties.

The list of applicable properties may vary between the modes. Properties which are only mentioned in some modes but not in others simply indicate that they would have no actual meaning in the modes where they are not listed.

Here within the analogue channels this is not the case. An example would be counter-channels that have a mode called “Simple Event Counting” - that only takes one input signal - and therefore have only one source mentioned in this mode but also support a “gated event counting” - that will take two distinct input-signals - and therefore has two separate sources settable.

Taking the Voltage mode as an example:

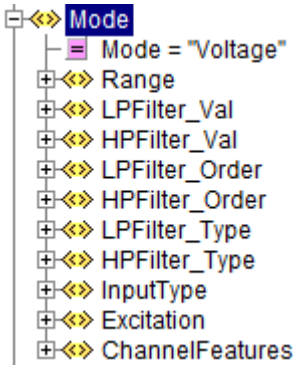

Illustration 13 - BoardProperties - Voltage Mode

This is the exhaustive list of supported properties for an analogue channel in voltage mode.

The most obvious property here is Range:

```
<Range Unit = "V" Count = "8" Default = "0" Programmable = "True"
MinInputOffset = "-200" MaxInputOffset = "200" MinOutputOffset = "-150"
MaxOutputOffset = "150" MinTotalOffset = "-300" MaxTotalOffset = "300"
ProgMax = "200" ProgMin = "0.03">
                    <ID0>200</ID0>
                    <ID1>100</ID1>
                    <ID2>30</ID2>
                    <ID3>10</ID3>
                    <ID4>3</ID4>
                    <ID5>1</ID5>
                    <ID6>0.3</ID6>
                    <ID7>0.1</ID7>
                    <ID8>0.03</ID8>
</Range>
```

Illustration 14 - BoardProperties - Range Node

From this information the application can deduce:

- The analogue input supports ranges from 0.03V to 200V
- The default input range is 200V

- It is freely programmable. So any value between min and max can be set and the hardware is not limited to the values presented in the list
- Various information about offsets. Explaining these in detail is beyond the scope of this overview document

3.9.3 USING THE BOARDPROPERTIES XML-FILE

With the overview provided in Chapter 3.9.2 in mind one obvious use case for this document is to allow the application to perform some preliminary evaluation of settings before trying to apply them to the API.
One other use case would be that it easily allows the application to decide about setup-information shown in the user-interface if needed. It would be easy to just offer such options in the UI that are actually supported by the board for the given mode.
One less obvious information the document provides is information needed for the string based functions – namely the target string and the item-identifier. See Chapters 2.3.3 and 3.5 for more details.

3.9.4 DERIVING TARGET STRINGS AND ITEM IDS FROM THE DOCUMENT

As shown in Chapter 3.5 all logical properties are addressed by strings.
As a rule of thumb one can assume that the target string matches the path within the XML-File starting from the root but omitting the first node-level.
The last element identifies the Items. Anything that remains in before is part of the target-string.
This is entirely true for the acquisition properties and slightly different for the channel properties. The detailed rational for this approach is provided within this chapter along with a couple of examples.

3.9.4.1 Acquisition Properties

While the examples provided here will look awfully complicated on first sight, the procedure itself is generic enough to become natural to the application developer very quickly.

Example: How to setup/retrieve the SampleRate with the string based functions
Path within the XML-File: `\\AcquisitionProperties\AcqProp\SampleRate`
Derived target and item-string for the sting based functions:

- Applying the rule of omitting the first node-level: `\AcqProp\SampleRate`
- Last part is the item: "`SampleRate`"
- Remainder for the target string: `AcqProp`
- As the target-string always needs the `BoardID` as part of it the complete Target string is: "`BoardID0/AcqProp`"

So the final function call looks like:

```
DeweSetParmaStruct_Str( "BoardID0/AcqProp", "SampleRate", "20000" );
```

Example: How to setup the ResolutionAI
Path within XML-File: `\\AcquisitionProperties\AcqProp\ResolutionAI`
Derived target- and item-string:
Target: "`BoardID0/AcqProp`"
Item: "`ResolutionAI`"
Final function call:

```
DeweSetParmaStruct_Str( "BoardID0/AcqProp", "ResolutionAI", "16" );
```

3.9.4.2 Channel Properties

As discussed in Chapter 3.9.2.7 the first level on a single channel within the XML-File is a list of various modes grouping the properties meaningful for the given mode.

This has two consequences when deriving target and item information:

1.) Mode itself is a valid Item
2.) When addressing any property the mode-node is omitted within the target

Example: How to set the mode of an analogue channel to Resistance
Path within XML-File: `\\ChannelProperties\AI0\Mode`
Derived target- and item-string:
Target: "`BoardID0/AI0`"
Item: "`Mode`"

Final function call:

```
DeweSetParmaStruct_Str( "BoardID0/AI0", "Mode", "Resistance" );
```

Example: How to set the Range of an analogue channel in Resistance-mode
Path within XML-File: `\\ChannelProperties\AI0\Mode\Range`
Derived target- and item-string:
Target: "`BoardID0/AI0`"
Item: "`Range`"

Final function call:

```
DeweSetParmaStruct_Str( "BoardID0/AI0", "Range", "3000" );
```

4 COMMAND GLOSSARY

4.1 ADMINISTRATIVE DIRECT CALL FUNCTIONS

This section gives a description of the available direct callable functions that serve administrative purposes.

4.1.1 DEWEPXILOAD

Signature:

```
int DeWePxiLoad(void)
function DeWePxiLoad: integer;
```

This function loads the API into the process memory and sets up the API-Interface-functions for use.
It is safe to call this function more than once.

This function returns 0 on failure.
Any return value greater than 0 indicates the loaded interface-version of the API.
The current interface version is 3.

4.1.2 DEWEPXIFREE

Signature:

```
void RTdaqUnload(void)
procedure DeWePxiFree;
```

This function unloads the API from process memory.
While it is not strictly necessary to call this function it is highly recommended to do so, especially if unloading the API is not the logical last step performed by the application.
Calling this function more than once may have undesired side-effects. The first call will perform the clean-up and unload.
Any subsequent calls to any API-function other than `RtdaqLoad` or `DeWePxiLoad` will fail.

4.1.3 DEWEDRIVERINIT

Signature:

```
int DeweDriverInit(int *nNumOfBoard)
function DeweDriverInit(var nNumOfBoard: integer): integer; stdcall;
```

This function initializes the API internal state and will report the number of detected boards.
Note: To allow the application to detect whether the boards are simulated hardware or real hardware the `nNumOfBoards` will have a negative sign when simulated hardware is used. (e.g. `nNumOfboards = -2` indicates 2 simulated boards)

4.1.4 DEWEDRIVERDEINIT

Signature

```
int RT_EXPORT DeweDriverDeInit( void )
```

```
function DeweDriverDeInit :integer; stdcall;
```

This function is used to trigger a clean shutdown of API internal data structures.
Calling this function is not strictly necessary as calling `RtdaqUnload / DeWePxiFree` will implicitly perform a call to this function.

4.2 INTEGER BASED COMMANDS

This section gives a description of the available command-IDs (**nCommandId**) for the integer based commands.

There are two distinct integer based commands.

- One to set data, or trigger an action
- One to retrieve data

Another distinct for the integer command is, whether it is an atomic command, or a composite command.
Composite commands are a collection of atomic commands executed within the context of the API. Hardware-updates in general are more time-consuming, as they involve hardware-access, and board-specific guard-times between those accesses, determined by the components used on the board itself. Composite commands allow the API to perform order-optimization, and thereby cut the time spent in the actual function, compared to using the atomic-commands on application level.
When invoking a composite command, API will try to perform the underlying atomic commands with a best-effort-policy. So every atomic command will be tried to execute, even if any of the prior executed atomic operation returned an error. (Except if there are internal hard-dependencies, which would render an operation as potentially harmful to the hardware).
As the i32-invokers use a single integer-return-value for the error-code, the return-value of the composite commands will always reflect the 'worst' error-code encountered during the execution. Therefore it is possible to lose detailed information, as it is possible that more than one of the underlying atomic commands would indicate an error-state.

4.2.1 USAGE OF INTEGER BASED COMMANDS

4.2.1.1 DeWeGetParam_i32

- `DeWeGetParam_i32( [in] int nBoardNo, [in] int nCommandId, [out] int* pVal);`

Parameters:
- `nBoardNo`: Either the logical Board number of a natural Board (0,1,2,…) or in more special cases `UPDATE_ALL_BOARDS` to address all Boards in the system. Within the command-glossary it will be explicitly stated for each command if this command allows only for natural Boards or the 'All Boards' parameter.
- `nCommandId`: The ID of the command to be executed. This glossary holds a list of all allowed commands
- `pVal`: a pointer to an integer32 sized variable to take the result of the operation.

Return codes:
- A return code of 0 (`ERR_NONE`), indicates execution without failure. The content of `pVal` is reliable.
- A positive return code indicates a hard error. The value returned in `pVal` is not reliable, and should not be considered for further usage.
- A negative return code indicates a warning. The value returned in `pVal` can be taken as reliable.

4.2.1.2 DeWeSetParam_i32

- `DeWeSetParam_i32( [in] int nBoardNo, [in] int nCommandId, [in] int nVal);`

Parameters:

- `nBoardNo`: Either the logical Board number of a natural Board (0,1,2,…) or in more special cases `UPDATE_ALL_BOARDS` to address all Boards in the system. Within the command-glossary it will be explicitly stated for each command if this command allows only for natural Boards or the 'All Boards' parameter.
- `nCommandId`: The ID of the command to be executed. This glossary holds a list of all allowed commands
- `nVal`: Depending on the Command. Not all commands take an actual value. For example `CMD_OPEN_BOARD` takes no further parameter to operate. Within the command-glossary it will be explicitly stated for each command if this command supports this parameter and what the legal values for this parameter would be.

Return codes:

- A return code of 0 (`ERR_NONE`), indicates execution without failure. The content of `pVal` is reliable.
- A positive return code indicates a hard error. The operation cannot be considered as successfully executed.
- A negative return code indicates a warning. The operation can be considered as executed.

4.2.2 BASIC SET OF I32 COMMANDS

This section covers the most basic set of i32-commands to be able to perform data-acquisition with via the API with a TRION™-Board.
Most SDK Examples use this basic set.
Any application that does not intend to change amplifier-settings on analogue cards during a running acquisition can rely on this basic interface.
The main difference between this basic interface and the advanced interface-functions is the level of granularity of the commands.

4.2.2.1 CMD_OPEN_BOARD

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

Up to Driver Version 0.35.3530.0: Logically opens a Board for operations. This step is mandatory before executing any other command onto the given Board.
This logic will be changed in the upcoming driver version: On first `CMD_OPEN_BOARD` performed on a given board the board will not only be opened logically but also the logical board-setting will be reset to the values given as default within the BoardProperties.xml file. (See also `CMD_RESET_BOARD`)

4.2.2.2 CMD_CLOSE_BOARD

Type: composite

- Issues the String-Set-Command "CommandFile", to store away current logical configuration in the SystemXML-Folder
- Persists changed XML-data stored on board (like amplifier-balancing-values) towards the e2prom. (equivalent to CMD_BOARD_BASEEEPROM_WRITE and CMD_BOARD_CONEEPROM_WRITE, but with prior checking of document-change)
- Finally closes the board logically
  Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

Logically closes a Board for operations.
This step is not strictly mandatory as all boards that remained open at Driver unload will be implicitly closed

4.2.2.3 CMD_RESET_BOARD

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

Up to Driver Version 0.35.3530.0: Puts both the logical layer and the hardware in a well-defined default state. The logical Layer will be set to the values given as default within the BoardProperties.xml file. (See also `CMD_OPEN_BOARD`)
This logic will be changed in the upcoming driver version: The default setting will be applied to the board on the initial `CMD_OPEN_BOARD`. `CMD_RESET_BOARD` will by then only reflect these settings to the hardware of the given board, and therefore will act like `CMD_UPDATE_PARAM_ALL`.
To clear out the ADC-pipes if present, a short (around 100 ms) measurement is started during this operation.

4.2.2.4 CMD_START_ACQUISITION

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	ACQ_STATE_XXXX	Aliased by CMD_ACQ_STATE. Should not be used directly, as support for the not-aliased command may be dropped in upcoming versions

Set	Only natural boards	N/A	

This command starts the acquisition on the given board. To allow this command to actually execute and start acquisition on the hardware, it is mandatory, that the hardware setup (both, on the logical layer and on the hardware) has been executed and finished.

4.2.2.5 CMD_STOP_ACQUISITION

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command stops the acquisition for the given board.

4.2.2.6 CMD_BUFFER_BLOCK_SIZE

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Currently set blocksize [in samples]	Not supported
Set	Only natural boards	Blocksize to set [in samples]	

This command queries or sets the block-size to be used during acquisition. Please refer to the chapter “The Acquisition Buffer” for details

4.2.2.7 CMD_BUFFER_BLOCK_COUNT

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Currently set block-count	Not supported
Set	Only natural boards	Block-count to set	

This command queries or sets the block-count to be used during acquisition. Please refer to the chapter “The Acquisition Buffer” for details

4.2.2.8 CMD_UPDATE_PARAM_ALL

Type: composite, issues

- CMD_UPDATE_PARAM_CHN_ALL
- CMD_UPDATE_PARAM_ACQ_ALL

Usable on running acquisition: yes (but not recommended)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This function updates the whole hardware to match the logical settings.
It allocates the sample ring buffer according to the specified parameters and inherent parameters like samplerate, number of used channels and channel data width.
This is equivalent to calling CMD_UPDATE_PARAM_CHN_ALL followed by CMD_UPDATE_PARAM_ACQ_ALL.
During setup this is typically the last set-command issued to the driver before starting the acquisition.
Typically after this command only get-commands are applied to the driver. An exception to this rule would be the command CMD_BUFFER_FREE_NO_SAMPLE as this is part of a typical data-readout-loop and is used as a set-command.
As this command potentially changes the DMA layout, it is not advised to issue this command during a running acquisition.
During a running acquisition this command will return ERR_COMMAND_NOT_ALLOWED.
But as this command is a composite command, updating all channels, the trigger-line-MUX and the acquisition parameters parts of changed configuration will take effect, using an best-effort-policy.
This makes the command somewhat nondeterministic from point of view of an application, as the single integer-return-value cannot provide a more granulated information, which parameters where updated, and which didn't.

4.2.2.9 CMD_BUFFER_START_POINTER

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	Pointer to the start of the sample ring buffer	
Set	N/A	N/A	Not supported

This command retrieves the start pointer of the sample ring buffer.
This value can be queried after applying all necessary parameters for acquisition setup (for example by calling CMD_UPDATE_PARAM_ALL).

4.2.2.10 CMD_BUFFER_END_POINTER

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*

Get	Only natural boards	Pointer to the last address within the sample ring buffer	
Set	N/A	N/A	Not supported

This command retrieves the end pointer of the sample ring buffer (this is the last valid pointer within the ring buffer).
A typical usage for this information is the wrap around handling when reading out the sample ring buffer.
This value can be queried after applying all necessary parameters for acquisition setup (for example by calling `CMD_UPDATE_PARAM_ALL`).

4.2.2.11 CMD_BUFFER_TOTAL_MEM_SIZE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Total size of the ring buffer in bytes	
Set	N/A	N/A	Not supported

This command retrieves the total size of the allocated ring buffer in bytes.
A typical usage for this information is the wrap around handling when reading out the sample ring buffer
This value can be queried after applying all necessary parameters for acquisition setup (for example by calling `CMD_UPDATE_PARAM_ALL`).

4.2.2.12 CMD_BUFFER_AVAIL_NO_SAMPLE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Number of yet unprocessed scans	
Set	N/A	N/A	Not supported

This command retrieves the number of unprocessed scans within the ring buffer.
This command is non-blocking so `*pVal` may have the value of 0 if there are no not yet processed samples available. This function will indicate if a buffer-overflow in the ring buffer has occurred by returning `ERR_BUFFER_OVERWRITE`. This can happen if the actual data processing is too slow. To clear this error the acquisition should be stopped on all boards and restarted. This value can be queried at any time during a running acquisition. Calling this command on a stopped board will result in an error code indicating that no acquisition is running. (`ERR_DAQ_NOT_STARTED`)

4.2.2.13 CMD_BUFFER_ACT_SAMPLE_POS

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	Pointer to the first unprocessed scan	
Set	N/A	N/A	Not supported

This command retrieves the address of the start of the first unprocessed scan.
This value can be queried at any time during a running acquisition.
Calling this command on a stopped board will result in an error code indicating that no acquisition is running. (`ERR_DAQ_NOT_STARTED`)

4.2.2.14 CMD_BUFFER_FREE_NO_SAMPLE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural Boards	Number of scans that shall be considered as consumed	

This command indicates to the driver that `nVal` numbers of scans have been processed by the application.
In a typical data-readout-loop the sequence would be:

- `CMD_BUFFER_AVAIL_NO_SAMPL`E returning *pVal = x
- `CMD_BUFFER_ACT_SAMPLE_POS`
- Actual data processing of y scans
- `CMD_BUFFER_FREE_NO_SAMPLE` setting nVal = y
  So the application does not have necessarily to process the same amount of data as reported back by `CMD_BUFFER_AVAIL_NO_SAMPLE`. The application may process less data (provided enough data is processed per loop, to ensure, that no buffer overrun occurs).
  If the application now issues the `CMD_BUFFER_FREE_NO_SAMPLE` command indicating this 'less-consumption' the remaining samples will be presented again to the application in the next run.

4.2.3 ADVANCED SET OF I32 COMMANDS

The advanced set of i32 commands allows a finer granulated control over the used hardware.

4.2.3.1 CMD_ACT_SAMPLE_COUNT

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	Number of samples acquired since start of acquisition.	
Set	N/A	N/A	Not supported

This function retrieves the number of already sampled values on the given board. This is a low latency function and allows for example for software time stamping of asynchronous Non-TRION data sources.
Asynchronous TRION data channels are usually hardware time stamped as this is the more accurate way for time stamping. This function is meant to be used in conjunction with third party hardware that does not provide reliable timestamps on its own.

Note: Not to be confused with `CMD_BUFFER_AVAIL_NO_SAMPLE` or `CMD_BUFFER_ACT_SAMPLE_POS`. This function does not provide any information regarding the already transferred samples. It cannot be used for any conclusion regarding any sample buffer information.

4.2.3.2 CMD_UPDATE_PARAM_ACQ_ALL

Type: composite, issues:

- `CMD_UPDATE_PARAM_ACQ`
- `CMD_UPDATE_PARAM_MUX`
- `CMD_UPDATE_PARAM_INTSIG0`
- `CMD_UPDATE_PARAM_INTSIG1`

Usable on running acquisition: no

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This function updates the hardware regarding the pure data-acquisition parameters. This includes, but is not limited to setting up the DAM-characteristics.
This is equivalent to call `CMD_UPDATE_PARAM_ACQ`, `CMD_UPDATE_PARAM_MUX`, `CMD_UPDATE_PARAM_INTSIG0` and `CMD_UPDATE_PARAM_INTSIG1` in this order.
As this command potentially changes the DMA layout, it is not advised to issue this command during a running acquisition.
During a running acquisition this command will return ERR_COMMAND_NOT_ALLOWED.

4.2.3.3 CMD_ UPDATE_PARAM_CHN_ALL

Type: composite, issues

- `CMD_UPDATE_PARAM_AI with parameter UPDATE_ALL_CHANNELS`
- `CMD_UPDATE_PARAM_AREF with parameter UPDATE_ALL_CHANNELS`
- `CMD_UPDATE_PARAM_CNT with parameter UPDATE_ALL_CHANNELS`
- `CMD_UPDATE_PARAM_DI with parameter UPDATE_ALL_CHANNELS`
- `CMD_UPDATE_PARAM_BOARD_CNT with parameter UPDATE_ALL_CHANNELS`
- `CMD_UPDATE_PARAM_CAN with parameter UPDATE_ALL_CHANNELS`

- `CMD_UPDATE_PARAM_UART with parameter UPDATE_ALL_CHANNELS`
  Usable on running acquisition: yes (but not recommended)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command updates the hardware-settings of all channels featured on the given board to reflect the latest logical seting.

4.2.3.4 CMD_UPDATE_PARAM_ACQ

Type: atomic
Usable on running acquisition: no

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command updates the board-hardware to reflect all acquisition related logical settings.
This includes the signal-routing on the PXI-plane (eg for synchronization purposes), configuring the hardware to the selected samplingrate and preparing the DMA-ring-buffer.

4.2.3.5 CMD_UPDATE_PARAM_ACQ_ROUTE

Type: atomic
Usable on running acquisition: yes (but only recommended in limited cases)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command updates the signal-route-multiplexer for the signals directly relevant to the acquisition to their latest logical values.
In detail these are the settings:

- AcqClk
- AcqSync
- AcqStart
  Usually those parameters are set prior to acquisition.
  A possible use case, when to issue this command during a running acquisition would be, to route AcqStart to High after an acquisition using external trigger has been started successfully, to prevent the acquisition from being stopped, if the external-trigger returns to low.

4.2.3.6 CMD_UPDATE_PARAM_MUX

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command applies the logical settings of the signal-line multiplexers to the hardware.
This includes the PXI-line settings like TRIG0-TRIG7, the LBL and LBR lines, Start Trigger lines, etc.
Usually this command is invoked implicitly by calling one of the PARAM_UPDATE_ACQ commands.
This command is only useful, if an application needs to change MUX-settings during a running acquisition and can be ignored otherwise.

4.2.3.7 CMD_UPDATE_PARAM_INTSIG0, CMD_UPDATE_PRAM_INTSIG1

Type: atomic
Usable on running acquisition: yes (but with considerations)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command updates the hardware settings for the IntSig0 and IntSig1 line to reflect the latest logical settings.
It can safely be used during a running acquisition.

4.2.3.8 CMD_UPDATE_PARAM_AI

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	Possible during a running acquisition, unless the Used-State of the channel has been changed

This command updates the analogue amplifier chain to reflect the latest logic values.
It is safe to issue this command during a running acquisition, unless the Used – state of the channel in question has been changed.

4.2.3.9 CMD_UPDATE_PARAM_CNT

Type: atomic
Usable on running acquisition: no

	nBoardNo	**pVal / nVal*	*Remark*

Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	

This command updates the counter properties on the FPGA to reflect the latest logical configuration values.
This includes the input-multiplexer as well, as the filter settings and super-counter parameters.
As the super-counter parameters change the logical math used on the channel it makes no sense, to make changes to those settings during a running acquisition.

4.2.3.10 CMD_UPDATE_PARAM_DI

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	

This command updates the digital input settings on the board to reflect the latest logical configuration values.
As this is most likely to change scan-layout it is not possible to issue this command on a running acquisition.

4.2.3.11 CMD_UPDATE_PARAM_BOARD_CNT

Type: atomic
Usable on running acquisition: no

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	

This command updates the board-counter properties on the FPGA to reflect the latest logical configuration values.
This includes the input-multiplexer as well, as the filter settings and super-counter parameters.
As the super-counter parameters change the logical math used on the channel it makes no sense, to make changes to those settings during a running acquisition.

4.2.3.12 CMD_UPDATE_PARAM_CAN

Type: atomic

Usable on running acquisition: yes (synchronous acquisition), no (CAN acquisition)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	

This command updates the CAN-controller settings to reflect the latest logical configuration values.
The CAN-acquisition has to be in a stopped stated, to accept this command.

4.2.3.13 CMD_UPDATE_PARAM_UART

Type: atomic
Usable on running acquisition: yes (synchronous acquisition), no (UART acquisition)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	

This command updates the UART-controller settings to reflect the latest logical configuration values.
The UART-acquisition has to be in a stopped stated, to accept this command.

4.2.3.14 CMD_ ASYNC_POLLING_TIME

Type: atomic
Usable on running acquisition: yes (synchronous acquisition), no (UART or CAN acquisition)

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	Currently set polling time [ms]	Not supported
Set	Only natural boards	Polling time to set to [ms]	

Sets or gets the used data-polling time for asynchronous channels.
Asynchronous data usually should only be transferred in OSI-Layer-7-complete chunks. Depending on the used Application, the OSI-Layer-7-data may span over multiple 32-Bit values.
To prevent tearing this Data over several DMA-transfers, and to prevent to burden the BUS with unnecessary small-chunk-dma transfers, the transfer for asynchronous data is not interrupt driven, but uses a 2-step-polling policy.

The driver polls the number of available asynchronous data-chunks using a low-latency, low-bandwidth call to an FPGA-register, and initiates a dma-transfer only if at least 1 sufficient-sized data-packet is available.

4.2.3.15 CMD_ ASYNC_FRAME_SIZE

Type: atomic
Usable on running acquisition: yes (synchronous acquisition), no (UART or CAN acquisition)

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Currently set async frame-size [in multiples of 32Bit words]. Eg.: 2 means 64 bit	
Set	Only natural boards	Frame size to set to [in multiples of 32Bit words]. Eg.: 2 means 64 bit	

Sets or gets the expected size of one atomic asynchronous data-element.

4.2.3.16 CMD_UPDATE_PARAM_ACQ_TIMING

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

Updates the hardware to reflect the currently set logical configuration related to the various timing-modes (eg.: IRIG, GPS-SYNC, PPS-Sync).

4.2.3.17 CMD_TIMING_STATE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Current timing-state	
Set	N/A	N/A	Not supported

This command queries the board for the current state of the FPGA-internal state machine.
The possible reported values are:

- TIMINGSTATE_LOCKED
- TIMINGSTATE_NOTRESYNCED
- TIMINGSTATE_UNLOCKED

- TIMINGSTATE_LOCKEDOOR
- TIMINGSTATE_TIMEERROR
- TIMINGSTATE_RELOCKOOR
- TIMINGSTATE_NOTIMINGMODE
  For a detailed description of those values please refer to the section Timing-Modes

4.2.3.18 CMD_BUFFER_CLEAR_ERROR

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command is used in more advanced error-control of buffer-overruns during acquisition.
When a buffer-overrun occurs, the driver will stay in the overrun-error-occurred state, until either the acquisition is stopped or the command CMD_BUFFER_CLEAR_ERROR is issued.
The basic and recommended error-handling for buffer-overruns is to stop the running acquisition, and restart it.
For advanced applications this might be a undesired error-handling-policy, especially in multi-board-systems. With the stop-start approach even boards that are currently error-free would be stopped (in a master-slave-environment).
With the command CMD_ BUFFER_CLEAR_ERROR it is possible to inform the driver that the application did acknowledge the overrun-condition, and handled it, and normal acquisition can occur from now on.
For details about the usage of this command, please refer to the related SDK example

4.2.3.19 CMD_GET_UART_STATUS

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	Current UART-FIFO-state	
Set	N/A	N/A	Not supported

This command retrieves the current FIFO-status of the on-board-UART (UART capable boards only).
The possible reported values are:

- `WARNING_UART_FIFO_BUSY`
- `WARNING_UART_FIFO_FULL`
- `WARNING_UART_FIFO_ERROR`
- ERR_NONE

4.2.3.20 CMD_BUFFER_WAIT_AVAIL_NO_SAMPLE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*

Get	Only natural boards	Number of samples to wait for	
Set	N/A	N/A	Not supported

This command is the blocking sibling of CMD_BUFFER_AVAIL_NO_SAMPLE.
When issued, the API will block until the required number of samples has been acquired.
This command frees the application of the need to poll for available samples.
The subsequent processing of acquired data is the same, as it would be with CMD_BUFFER_AVAIL_NO_SAMPLE.

4.2.3.21 CMD_ACQ_STATE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	Current acquisition state	
Set	N/A	N/A	Not supported

This command retrieves the current acquisition state of the board.
The possible reported values:

- `ACQ_STATE_IDLE`
- `ACQ_STATE_RUNNING`
- `ACQ_STATE_SYNCED`
- `ACQ_STATE_ERROR`
  This way an application can always see, in what detailed state the given data-acquisition hardware currently is.
  This command is for example useful on external-trigger-setups, as the command CMD_START_ACQUISTION would just arm the given board, but as long, as the external trigger did not happen, the board itself would stay in ACQ_STATE_IDLE.

4.2.3.22 CMD_UPDATE_PARAM_AREF

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	Natural channel number, `UPDATE_ALL_CHANNELS`, `UPDATE_GROUP_CHANNELS`	

This command updates the current hardware-settings of the internal analogue reference source to reflect the latest logical values.

4.2.3.23 CMD_TIMING_TIME

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	N/A	
Set	N/A	N/A	Not supported

This low-latency command updates the logical data-elements below /AcqProp/Timing/SystemTime to hold the current hardware-values. Typical usage examples would be a GPS or IRIG time-source connected to a IRIG or GPS capable TRION™-board.
After issuing this command the string-getters for the single time-elements can be issued to obtain the information.

4.2.3.24 CMD_GPS_RECEIVER_RESET

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command issues the command-sequence, that will perform a low-level-reset on the receiver of the GPS-capable TRION™-board.

4.2.3.25 CMD_PXI_LINE_STATE

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	Only natural boards	Current state of the PXI-lines	
Set	N/A	N/A	Not supported

This command retrieves the current state of the PXI-Lines.
For decoding use the filter-bits PXI_LINE_STATE_TRIG0 to PXI_LINE_STATE_LBL12.

4.2.3.26 CMD_DISCRET_STATE_SET

Type: atomic
Usable on running acquisition: yes

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported

Set	Only natural boards	Logical channel number of discreet state to set	

This command is used to set the given logical discreet (DIO/DO) channel to a logic '1' value. (= High)
As only natural channel number can be passed, only one DIO/DO channel (bit) can be set at the same time

4.2.3.27 CMD_DISCRET_STATE_CLEAR

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	Logical channel number of discreet state to set	

This command is used to set the given logical discreet (DIO/DO) channel to a logic '0' value. (= Low)
As only natural channel number can be passed, only one DIO/DO channel (bit) can be set at the same time

4.2.3.28 CMD_DISCRET_GROUP32_SET

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	Bit-pattern to set on group	

Discreet asynchronous output channels (DIO/DO) are organized in 32Bit groups on TRION™. This command is used to update a whole 32-bit group at once.
This command by nature operates on a less hardware-abstract level than the CMD_DISCRET_STATE_SET/CMD_DISCRET_STATE_CLEAR commands. Therefore an application has to have a higher awareness about the detailed hardware-capabilities of the given board.

Note: the second group of 32-Discreets can be addressed by incrementing the command ID by 1. (CMD_DISCRET_GROUP32_Set +1)

4.2.3.29 CMD_IDLED_BOARD_ON

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	Color to set LED to	

Specific TRION™-boards feature a single LED that can be SW-controlled.
Currently valid colors to set the LED to are:

- IDLED_COL_RED
- IDLED_COL_GREEN
- IDLED_COL_ORANGE
- IDLED_COL_OFF
  Whether this command is supported, and details about the allowed values are depending on the specific TRION™-board.

4.2.3.30 CMD_IDLED_BOARD_OFF

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

Specific TRION™-boards feature a single LED that can be SW-controlled.
This command will turn the LED of the given board off.
The command is equivalent to issue CMD_IDLED_BOARD_ON with the parameter IDLED_COL_OFF.

4.2.3.31 CMD_IDLED_CHANEL_ON

This command is currently not supported.

4.2.3.32 CMD_IDLED_CHANNEL_OFF

This command is currently not supported.

4.2.3.33 CMD_BOARD_ADC_DELAY

Type: atomic
Usable on running acquisition: yes

	nBoardNo	****pVal / nVal***	***Remark***
Get	Only natural boards	ADC delay, with current settings applied	
Set	N/A	N/A	Not supported

This command retrieves the ADC delay, using the current settings.
Various AD-converter-types have different runtime of the samples from analogue side to the final converted value. Therefore analogue samples may appear in later scans, than time-wise corresponding digital channels. The ADC-delay is used to allow the application to re-associate samples of different channel-types.
Please refer to the SDK-example “ADCDelay” for correct usage.

4.2.3.34 CMD_BOARD_BASEEEPROM_WRITE

Type: atomic

Usable on running acquisition: yes (but not recommended)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command writes the content of the base-e2rpom-xml-file back to the e2prom.
Unless the application willingly writes directly to the baseeeprom-xml-file, there is no need to issue this command.

4.2.3.35 CMD_BOARD_CONEEPROM_WRITE

Type: atomic
Usable on running acquisition: yes (but not recommended)

	nBoardNo	**pVal / nVal*	*Remark*
Get	N/A	N/A	Not supported
Set	Only natural boards	N/A	

This command writes the content of the connector-e2rpom-xml-file back to the e2prom.
Unless the application willingly writes directly to the coneeprom-xml-file, there is no need to issue this command.

DEWETRON

5 ADVANCED COMMANDS

This chapter provides an overview over more advanced commands. These commands usually deal with a very specific aspect, either directly related to the hardware- or driver-architecture, or very specific to common tasks for measurement-systems in general.
Opposed to the property-string-commands, that can be deduced from the BoardProperties XML-file by applying the simple rules described in 3.9.4, these commands are not evident by themselves.

5.1 BOARD-WIDE COMMANDS

This chapter provides an overview over the commands that affect whole boards.

5.1.1 STORE AND RETRIEVE BOARD CONFIGURATION

The API offers the possibility to either export the complete logical configuration of a board as XML-Document, or to import a XML-Document that represents a valid Board configuration. This logical configuration consists of the sub-tree “Acquisition” and the sub-tree “Channel” within the configuration-document.
The configuration can be imported and exported either to a file in the file-system, or the string-buffer.
The import of a configuration-file is performed ‘greedy’. So each setting inside the configuration-document will be tried to be applied to the given board. As each single setting may end in an error or warning, a result-XML-document containing each WARNING or ERROR-state encountered during this process.
The over-all-worst-case result will be returned as error-codes. If the most severe encountered problem was a WARING, the first WARNING encountered will be returned, if it was an ERROR, the corresponding error code will be returned. The safest approach for usage would be to examine the result-XML-file if an error-code unequal ERR_NONE is returned.

5.1.1.1 Auto-Persisted configuration

On each CMD_CLOSE_BOARD call, the API will auto-persist the current logical configuration to the SystemXML-folder. A auto-generated filename with the following pattern will be used:
BoardX_Config.xml
This auto-persisted configuration will not be re-applied automatically. It can be loaded manually by either specifying the path/filename or by issuing the “lastconfig” command.

5.1.1.2 XML-Document overview

5.1.1.2.1 Configuration XML Document

The detailed layout of the configuration-file is boar specific. It can be derived from the content of the BoardProperties XML-File.
The file has three major sections:

`\\BoardInfo`

This section holds basic information about the board from which this configuration was generated from. This includes the BaseModel-Number and the human readable BoardName (BoardType, eg. TRION-BASE).

`\\Acquisition`

This is the counterpart to the Node `\\AcquisitionProperties` in the BoardProperties-XML-Document.

Each property from the BoardProperties-XML-Document, that is not marked with the attribute "`Config = 'False'`" is also present in the `\\Acquisition` Node in the configuration document.
This includes some obvious information like for example the "SampleRate" as well, as for example detailed routing information for the PXI-trigger-lines or the Star-Hub.

`\\Channel`

This is the counterpart to the Node `\\ChannelProperties` in the BoardProperties-XML-document.
Each single channel is present within this node, with each of the settable properties, that is not marked with the attribute "`Config = 'False'`" inside the BoardProperties-MXL-document.

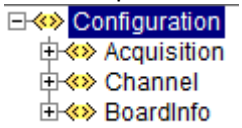

Illustration 15 Configuration-XML major sections

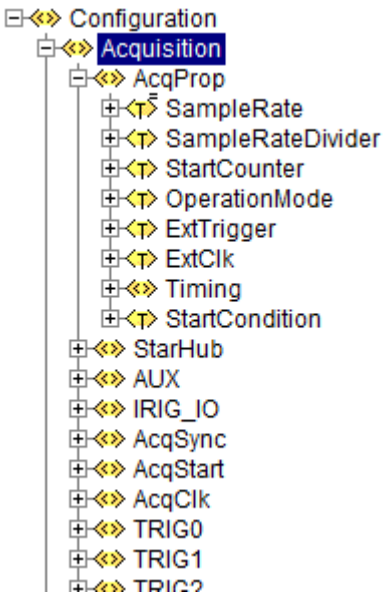

Illustration 16 Configuration XML Section: Acquisition

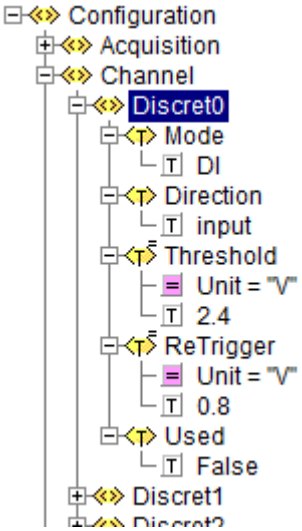

Illustration 17 Configuration XML Section: Channel

5.1.1.2.2 Result XML Document

The result xml-document has the same layout, as the configuration xml-document.
It's main purpose is to provide a fine granulated feedback about any set-configuration command. When loading (setting) a configuration, each property is applies one after the other. Every single setting-command (this means every set single property) ends up in a defined result state. Ideally the operation succeeds with ERR_NONE. However - as any arbitrary configuration can be presented to any board, this isnot guaranteed at all.
To make diagnostics easier, this result-file offers the possibility to see the exact result of each single property.

Depending on the API-configuration itself, either all results are returned, or only results <> ERR_NONE.
The following figure shows an example for a result-file, when trying to apply a 8-AI-Channel-board configuration to a board, that only supports 6 AI-channels.
All settings are applied ok, except for the non-existing channels AI6 and AI7.

```
<?xml version="1.0"?>
<Results>
  <Acquisition>
    <AcqProp>
      <StartCondition>Warning -190910, WARNING_STARTCONDITION_NOT_USED (-190910)</StartCondition>
    </AcqProp>
  </Acquisition>
  <Channel>
    <AI6>
      <Mode>Error 120012, ERROR_AI_CHANNEL_NOT_VALID (120012)</Mode>
    </AI6>
    <AI7>
      <Mode>Error 120012, ERROR_AI_CHANNEL_NOT_VALID (120012)</Mode>
    </AI7>
  </Channel>
</Results>
```

Illustration 18 – configuration load result XML example

5.1.1.3 Command set

Related Command	***Purpose***
DeWeGetParamStruc_Str	Export the configuration-XML-document, or retrieve the result-XML-document
DeWeSetParamStruct_str	Import the configuration-XML-document

Table 1 - DeWeGetParamStruct_Str commands for board configuration

Target	*Item*	*Parameter*	*Note*
"BoardIDx"	"config"	Buffer to hold the resulting XML-document as string	Retrieve the current logical board-configuration into a string
"BoardIDx"	"configfile"	String, containing a valid path(optional) and filename, where the XML-document shall be stored to	Retrieves the current logical board-configuration into a file in the file-system
"BoardIDx"	"configresult"	Buffer to hold the resulting XML-document as string	Retrieve a elaborate xml-document, that will indicate the error-code for each single setting imported in the last "config" or "configfile" –command.
"BoardIDx"	"configresultfile"	String, containing a valid path(optional) and filename, where the XML-document shall be stored to	Retrieve the elaborate xml-result-document into a file in the filesystem
"BoardIDx"	"defaultconfig"	Buffer to hold the resulting XML-document as string	Retrieve the logical default configuration for the given Board. Issuing this command will not alter current logical setting of the board in any way. The settings retrieved are basically the same, the board would hold by itself after issuing a CMD_RESET_BOARD command
"BoardIDx"	"defaultconfigfile"	String, containing a valid path(optional) and filename, where the XML-document shall be stored to	Retrieve the logical default configuration for the given board, and store it to the file specified.
"BoardIDx"	"lastconfig"	Buffer to hold the resulting XML-document as string	Retrieve the last auto-persisted configuration.
"BoardIDx"	"lastconfigfile"	String, containing a valid path(optional) and filename, where the XML-document shall be stored to	Retrieves the last auto-persisted configuration, and stores the resulting xml-document to the file specified

Table 2 - DeWeSetParamStruct_Str commands for board configuration

Target	*Item*	*Parameter*	*Note*
"BoardIDx"	"config"	Buffer, that holds a valid xml-configuration document for the specific board as string	Apply the logical configuration passed in the xml-document to the board
"BoardIDx"	"configfile"	String, containing a valid path(optional) and filename, where the XML-document shall be loaded from	Apply the logical configuration passed in the xml-document found at the given location to the board
"BoardIDx"	"defaultconfig"	N/A	Apply the logical default-configuration to the board. As the default-configuration is defined by the BoardProperties XML-document, no further parameter is needed.
"BoardIDx"	"lastconfig"	N/A	Apply the last auto-persisted configuration to the given board.

5.1.1.4 Usage Examples

Please refer to 8.1 and its corresponding example-source-code to see a more detailed simple usage example within a simple program.

Example: Load a configuration file

```
nErrorCode = DeWeSetParamStruct_Str( "BoardID0", "configfile", "exampleconfig.xml");
```

Example: retrieve detailed status operation of last operation

```
nErrorCode = DeWeGetParamStruct_Str( "BoardID0", "configresult", Buffer,
sizeof(Buffer));
```

Example: store the current configuration to a string-bufffer

```
nErrorCode = DeweGetParamStruct_str( "BoardID0", "config", Buffer, sizeof(Buffer));
```

Example: store the current configuration to a file

```
nErrorCode = DeweGetParamStruct_Str( "BoadID0", "configfile", "exampleconfig.xml",
1);
```

Please note: although this is executed as a DeweGetParamStruct_Str – command, the third parameter, that usually holds the pointer to the buffer for the result is used as an in-parameter in this case. Therefore the size-parameter is N/A.

5.1.2 QUERY ONBOARD SENSORS

TRION boards offer temperature sensors both on the board itself, as well as on the connector-panel. Specific TRION models may also offer additional, internal sensors. Refer to the Board specific documentation for details.

5.1.2.1 Command set

Table 3 - DeWeGetParamStruct_Str commands for onboard sensors

Target	*Item*	*Parameter*	*Note*
"BoardIDx"	"basetemp"	Buffer to hold the resulting information	Retrieve the current board temperature in °C by querying the onboard sensor
"BoardIDx"	"contemp"	Buffer to hold the resulting information	Retrieve the current connector panel temperature in °C by querying the onboard sensor

5.1.3 DEFINING CHANNEL GROUPS

To allow applying operations with just one function-call to any arbitrary number of specific channels of a given board, the API offers the possibility to freely define and clear Groups of channels.
One possible (but not necessarily useful-real-world) use-case would be to set the range on all Odd-numbered AI-channels.
This could be achieved by the following small code-snippet

```
nErrorCode = DeWeSetParamStruct_Str("Board0", "groupai", "1;3;5;7");
nErrorCode = DeWeSetParamStruct_Str("Board0/AIGrp", "Range", "5V");
```

The first command defines the AI-Group for Board0, including the members 1,3,5,7.
The subsequent command applies a setting to the defined group of channels.
A group stays defined, until redefined. To completely clear a group-definition, an empty parameter has to be passed. A group may consist of 0 to max-channel-count members. If all available channels of a given type are member of the group, the group-target behaves equivalent to the /XXall command. (like AIAll, or CNTAll)

5.1.3.1 Command set

Table 4 - DeWeSetParamStruct_Str commands channel group definition

Target	*Item*	*Parameter*	*Note*
"BoardIDx"	"groupai"	A semicolon-separated list of channel – indices, or an empty string to clear the definition at all.	
"BoardIDx"	"grouparef"	A semicolon-separated list of channel – indices, or an empty string to clear the definition at all.	
"BoardIDx"	"groupcnt"	A semicolon-separated list of channel – indices, or an empty string to clear the definition at all.	
"BoardIDx"	"groupbdcnt"	A semicolon-separated list of channel – indices, or an empty string to clear the definition at all.	
"BoardIDx"	"groupdiscret"	A semicolon-separated list of channel – indices, or an empty string to clear the definition at all.	

Table 5 - DeWeGetParamStruct_Str commands channel group definition

Target	*Item*	*Parameter*	*Note*
"BoardIDx"	"groupai"	Buffer to hold the resulting information	
"BoardIDx"	"grouparef"	Buffer to hold the resulting information	
"BoardIDx"	"groupcnt"	Buffer to hold the resulting information	
"BoardIDx"	"groupbdcnt"	Buffer to hold the resulting information	
"BoardIDx"	"groupdiscret"	Buffer to hold the resulting information	

5.2 CHANNEL-SPECIFIC COMMANDS

5.2.1 AI SPECIFIC COMMANDS

5.2.1.1 Compensating for amplifier offset

All TRION-Boards are shipped adjusted and calibrated. Due to changes in environmental conditions or due to aging, the amplifier may show a small but noticeable offset over time.
The API offers a simple command, to update the board-internal adjustment tables to compensate for this error.
The command can either be applied to an idle board (that is a board, that is not currently running an active acquisition), or on a board, that is running an acquisition.

There are small differences between these two use-cases that need to be considered:

- As it is not possible to enable or disable channels during a running acquisition, using this command on a non-idle-board is limited to the currently enabled channels.
  For example: An application enables all odd AI-channels (1, 3, 5, and 7) and starts an acquisition. During this acquisition it issues the command to compensate amplifier offset on all AI-channels.
  Obviously this operation cannot be executed on all channels, as this would require enabling the currently disabled channels 0, 2, 4 and 6.
  As a consequence the API will only compensate on channels 1, 3, 5 and 7, leaving 0, 2, 4, 6 with their original, old compensation values.
- Also it is not possible to change the sampling-rate during a running acquisition.
  When performing this command on an idle board, the API will select the highest possible sampling-rate for determining the current offset. When executing on a board that is currently running an acquisition, the sampling rate will be the one that has been chosen by the application. In corner case circumstances (for example, when the application picks a very low sampling rate like 100 samples per second), the results may

vary – as the API has less samples to calculate the offset error from. The deviation will not be dramatically – but comparable less optimal.

- When executed on an idle board, the command behaves blocking. So the function will not return, unless the operation is finished.
- When executed on a non-idle board, the command behaves non-blocking. So the function returns immediately. To allow the operation to finish, the application has to process the acquired samples (actually just freeing the acquired samples will pump the amplifier-offset-correction function). As with a normal running acquisition this leaves the application responsible for preventing data-loss-conditions.

The command “amplifieroffset” involves several actions performed on the analogous channels, like setting the input to ‘short’ and iterating over all natural amplifier ranges (that are those ranges, that are determined only by the resistor and amplifier chain itself, and no mathematical scaling)
Due to this, it is not possible to obtain sensible measurement values during this operation. So there are only rare use-cases where this command would not be used preferably on an idle board by the application.

After Execution of this command, the new determined offset-correction values are stored on the board itself, and will be automatically applied with the next started acquisition.
If the application wants to manually query the determined values, it can retrieve a xml-document containing detailed information.

5.2.1.1.1 Command set

Table 6 - DeWeSetParamStruct_Str commands compensate for amplifier offset

Target	*Item*	*Parameter*	*Note*
“BoardIDx/AIy” **“BoardIDx/AiAll”** **“BoardIDx/AIgrp”**	“amplifieroffset”	Desired duration. Eg. “100 msec”	The desired duration is the averaging-duration per range. So the overall duration depends on the very specific board-type – as different board-types support different amount of natural ranges. Also this time does not include the time needed for hardware setup (in the low ms range), and the time needed for the internal reference signals to stabilize to get optimal results. As a rule of thumb this time should be take *3 for each range.

Table 7 - DeWeGetParamStruct_Str commands compensate for amplifier offset

Target	*Item*	*Parameter*	*Note*
"BoardIDx/AIy" **"BoardIDx/AiAll"** **"BoardIDx/AIgrp"**	"amplifieroffset"	Buffer to hold the resulting information	The result set will only contain the channels, where an amplifier-offset-correction has been performed on during this load/unload-cycle of the API, and that are queried. So if an amplifieroffset has been performed on channels 1, 3, 5, and 7, and a result for ALL is queried, only the channels 1, 3, 5, and 7 will be present in the result-set. If for example channels 0, 2, 4, and 6 are queried, the result-set will be empty, as the operation has only be performed on channels 1, 3, 5, and 7.

In asynchronous operation mode (that is, when the operation is performed during an running acquisition) the string getter will return a rough estimation of the expected remaining runtime.
A typical implementation pattern for this use case would be to periodically poll the result-string, and test for the presence of an XML-attribute called "EstDuration". Please refer to the SDK-examples for further details.

5.2.1.1.2 Structure of returned data

The information returned by the DeWeGetParamStruct_Str function is a small XML-document.

```
<?xml version="1.0"?>
 <TestResults TestCount="18" EstDuration="1800"/>
```

Illustration 19 - Example amplifieroffset pending result

```
<?xml version="1.0"?>
<AmplifierOffset>
  <BoardId0 ID = "BoardId0" BrdName = "TRION-2402-dSTG-8-RJ" Slot = "1" SerialNumber = "A01200E3" Passed = "true">
    <Check Target = "AI0" Type = "Amplifier Balance Test" Passed = "True">
      <Averaging>200msec</Averaging>
      <Date>02.07.2015</Date>
      <Time>11:41:54</Time>
      <BaseBoardTemp>47</BaseBoardTemp>
      <ConPanelTemp>43.5</ConPanelTemp>
      <ID0 Device = "AmplifierBalance" Range = "10 V" Limit = "0.200000" Passed = "True" Coupling = "AC">
```

```
      <Offset Unit = "V" Passed = "True">-0.000069</Offset>
    </ID0>
    <ID1 Device = "AmplifierBalance" Range = "10 V" Limit = "0.200000" Passed = "True" Coupling = "DC">
      <Offset Unit = "V" Passed = "True">0.000215</Offset>
    </ID1>
    <ID2 Device = "AmplifierBalance" Range = "3 V" Limit = "0.060000" Passed = "True" Coupling = "AC">
      <Offset Unit = "V" Passed = "True">-0.000019</Offset>
    </ID2>
    <ID3 Device = "AmplifierBalance" Range = "3 V" Limit = "0.060000" Passed = "True" Coupling = "DC">
      <Offset Unit = "V" Passed = "True">0.000084</Offset>
    </ID3>
    <ID4 Device = "AmplifierBalance" Range = "1 V" Limit = "0.020000" Passed = "True" Coupling = "AC">
      <Offset Unit = "V" Passed = "True">-0.000005</Offset>
    </ID4>
    <ID5 Device = "AmplifierBalance" Range = "1 V" Limit = "0.020000" Passed = "True" Coupling = "DC">
      <Offset Unit = "V" Passed = "True">0.000014</Offset>
    </ID5>
  </Check>
 </BoardId0>
</AmplifierOffset>
```

Illustration 20 - Example amplifier-offset final result (shortened)

5.2.1.2 Determining sensor offset

If the output signal is not zero when the measured property is zero, the sensor has an offset or bias. This is defined as the output of the sensor at zero input.
Analogue input channels on TRION-boards feature a property called “inputoffset”. By setting this property, the TRION-Board will automatically subtract this value from the measured signal.
For easier handling, the unit of this property is always the same, as the unit for the range. So if the channel is in bridge-measurement-mode, the range, the scaled measured value and the input-offset all are in either mV/V or mV/mA.
Compensating for sensor-offset usually involves averaging the measured value over a specified time, while a physical zero-value is applied to the sensor.
The API allows to perform this by a simple single command.
However – To avoid confusion by auto-applying this information to the channel-properties, it is the responsibility of the application to set the “inputoffset”
While the amplifier offset compensation iterates over all ranges to determine the adjustment values, the sensor offset test operates with the current amplifier settings. This includes (but is not limited to) current range, filter settings, and excitation. Though there is a small exception: If the sensor offset is >100% of the selected range, the

amplifier settings are escalated to allow for this. (This means a lower gain is chosen, and the scaling-factors on the board are adjusted to still show the desired range full scale)
Like the amplifier offset compensation this command can be either executed on an idle board, or on a board that is running an acquisition. Refer to 5.2.1.1 and 5.2.1.1.2 for details about the differences between these two operations modes.

5.2.1.2.1 Command set

Table 8 - DeWeSetParamStruct_Str commands compensate for sensor offset

Target	*Item*	*Parameter*	*Note*
"BoardIDx/AIy" **"BoardIDx/AiAll"** **"BoardIDx/AIgrp"**	"sensoroffset"	Desired duration. Eg. "100 msec"	The desired duration is the averaging-duration. This time does not include the duration that is needed for necessary internal operations, and the times that are needed if a range escalation needs to be performed. As a rule of thumb this time should be take *3 for estimating the overall time needed.

Table 9 - DeWeGetParamStruct_Str commands compensate for sensor offset

Target	*Item*	*Parameter*	*Note*
"BoardIDx/AIy" **"BoardIDx/AiAll"** **"BoardIDx/AIgrp"**	"sensoroffset"	Buffer to hold the resulting information	The result set will only contain the channels, where a sensor-offset-measurement has been performed on during this load/unload-cycle of the API, and that are queried. So if a sensoroffset command has been performed on channels 1, 3, 5, and 7, and a result for ALL is queried, only the channels 1, 3, 5, and 7 will be present in the result-set. If for example channels 0, 2, 4, and 6 are queried, the result-set will be empty, as the operation has only be performed on channels 1, 3, 5, and 7.

In asynchronous operation mode (that is, when the operation is performed during an running acquisition) the string getter will return a rough estimation of the expected remaining runtime.
A typical implementation pattern for this use case would be to periodically poll the result-string, and test for the presence of an XML-attribute called "EstDuration". Please refer to the SDK-examples for further details.

After obtaining the results, an application typically would parse the resulting xml-document for the determined offset-values, and apply them to the channel-property “inputoffset”.

5.2.1.2.2 Structure of returned data

```
<?xml version="1.0"?>
<TestResults TestCount="3" EstDuration="300"/>
```

Illustration 21- Example sensor offset pending result

```
<?xml version="1.0"?>
<SensorOffset>
  <BoardId0 ID = "BoardId0" BrdName = "TRION-2402-dSTG-8-RJ" Slot = "1" SerialNumber = "A01200E3" Passed = "true">
    <Check Target = "AI0" Type = "Sensor Balance Test" Passed = "true">
      <Averaging>1000msec</Averaging>
      <Date>07.07.2015</Date>
      <Time>08:57:45</Time>
      <BaseBoardTemp>45</BaseBoardTemp>
      <ConPanelTemp>42</ConPanelTemp>
      <ID0 Device = "SensorBalance" Range = "1 mV/mA" Passed = "True">
        <Offset Unit = "mV/mA" Passed = "True">0.000000</Offset>
      </ID0>
    </Check>
    <Check Target = "AI1" Type = "Sensor Balance Test" Passed = "true">
      <Averaging>1000msec</Averaging>
      <Date>07.07.2015</Date>
      <Time>08:57:45</Time>
      <BaseBoardTemp>45</BaseBoardTemp>
      <ConPanelTemp>42</ConPanelTemp>
      <ID0 Device = "SensorBalance" Range = "1 mV/mA" Passed = "True">
        <Offset Unit = "mV/mA" Passed = "True">0.000000</Offset>
      </ID0>
    </Check>
    <Check Target = "AI2" Type = "Sensor Balance Test" Passed = "true">
      <Averaging>1000msec</Averaging>
      <Date>07.07.2015</Date>
```

```
        <Time>08:57:45</Time>
        <BaseBoardTemp>45</BaseBoardTemp>
        <ConPanelTemp>42</ConPanelTemp>
        <ID0 Device = "SensorBalance" Range = "1 mV/mA" Passed = "True">
          <Offset Unit = "mV/mA" Passed = "True">0.000000</Offset>
        </ID0>
      </Check>
    </BoardId0>
  </SensorOffset>
```

Illustration 22 – Example sensor-offset final result (shortened)

5.2.1.3 Basic measurement mode self-check (Mode-check)

Often it is very hard, to find the source of unexpected measurements-results in more complex measurement setups. As a special diagnostic functionality TRION-boards offer a command, which allows testing the basic measurement-parameters with the currently applied property-settings.
The reported results include:

- Detailed information about the amplifier-settings
- Offset error
- Gain error
- Excitation error, if excitation is used
- Current and maximum drive for excitation (so how many current can be supplied with the chosen voltage, or what voltage can be sustained with the given current.)
  All this information will also provide details about the expected ‘good’-limit. This is calculated from the values provided with the board-specification.

Also every test-result is decorated with a Passed-attribute, to allow for a quick check, if all parameters are within limits. While the ‘hardwaretest’-command iterates over all ‘natural’ amplifier ranges to determine offset and gain error, the ‘modecheck’ command will use the currently set properties.
This specialty allows also identifying problems that may occur by choosing less optimal logical settings.
For example: High amounts of sensor-offset (>>100%), together with a custom-range close to the theoretical limit (eg. Range = ‘5..10mV/V’ may cause the driver to escalate the amplifier-settings to an unexpected high range, and therefore have negative influence on gain-errors.

5.2.1.3.1 Command Set

Table 10 - DeWeSetParamStruct_Str commands for modecheck

Target	*Item*	*Parameter*	*Note*
"BoardIDx/AIy" **"BoardIDx/AiAll"** **"BoardIDx/AIgrp"**	"modecheck"	None	The desired duration is the averaging-duration. So the overall duration depends on the very specific board-type – as different board-types support different amount of natural ranges. Also this time does not include the time needed for hardware setup (in the low ms range), and the time needed for the internal reference signals to stabilize to get optimal results. As a rule of thumb this time should be take *3 for each range.

Table 11- DeWeGetParamStruct_Str commands for modecheck

Target	*Item*	*Parameter*	*Note*
"BoardIDx/AIy" **"BoardIDx/AiAll"** **"BoardIDx/AIgrp"**	"modecheck"	Buffer to hold the resulting information	The result set will only contain the channels, where a sensor-offset-measurement has been performed on during this load/unload-cycle of the API, and that are queried. So if a modecheck command has been performed on channels 1, 3, 5, and 7, and a result for ALL is queried, only the channels 1, 3, 5, and 7 will be present in the result-set. If for example channels 0, 2, 4, and 6 are queried, the result-set will be empty, as the operation has only be performed on channels 1, 3, 5, and 7.

5.2.1.3.2 Structure of returned data

```
<?xml version="1.0"?>
<ModeCheck>
  <BoardId0 ID = "BoardId0" BrdName = "TRION-2402-dSTG-8-RJ" Slot = "1" SerialNumber = "A01200E3" Passed = "True">

    <Check Target = "AI0" Type = "Mode Check" Passed = "True">
      <Date>09.07.2015</Date>
      <Time>08:32:11</Time>
      <ConPanelTemp>33.00</ConPanelTemp>
      <BaseBoardTemp>38.25</BaseBoardTemp>
      <ID0 Device = "ADC" HWRange = "10V" Range = "500 mV/V" Passed = "True">
        <Offset Limit = "0.00303 V" RawMeas = "-0.000091 V" Unit = "V" Averaging = "100 msec">-0.000091 V</Offset>
        <Gain Limit = "0.034 %" RawMeas = "9.607286 V" RefValRaw = "9.60748 V" Unit = "%" Averaging = "100 msec">-       0.001072 %</Gain>
      </ID0>
      <ID1 Device = "Excitation" Range = "7V" Passed = "True">
        <ExcitationVoltage Limit = "0.013 V" RawMeas = "6.999259 V" Unit = "V" Averaging = "100 msec">-0.000741 V</ExcitationVoltage>
        <ExcitationVoltage-Current Limit = "NA" RawMeas = "3.663074 mA" Unit = "NA" Averaging = "100 msec">NA</ExcitationVoltage-Current>
      </ID1>
    </Check>
    <Check Target = "AI1" Type = "Mode Check" Passed = "OverLimit">
      <Date>09.07.2015</Date>
      <Time>08:32:11</Time>
      <ConPanelTemp>33.00</ConPanelTemp>
      <BaseBoardTemp>38.25</BaseBoardTemp>
      <ID0 Device = "ADC" HWRange = "3V" Range = "500 mV/V" Passed = "True">
        <Offset Limit = "0.00093 V" RawMeas = "0.000034 V" Unit = "V">0.000034 V</Offset>
        <Gain Limit = "0.035 %" RawMeas = "2.87971 V" RefValRaw = "2.87982 V" Unit = "%">-0.005 %</Gain>
      </ID0>
      <ID1 Device = "Excitation " Range = "5 mA" Passed = "True">
        <ExcitationVoltage-Current Limit = "0.1 mA" RawMeas = "4.995 mA" Unit = "mA">0.005 mA</ExcitationVoltage-Current>
        <ExcitationVoltage Limit = "NA" RawMeas = "9.990055 V" Unit = "">0</ExcitationVoltage>
      </ID1>
    </Check>
  </BoardId0>
</ModeCheck>
```

Table 12 Example Modecheck

Note the different content of the attribute ‘HWRange’ for the two shown channels. This attribute shows the chosen amplifier-chain setting. So 10 mV/V @1V Excitation will be measured in the 10mV range of the amplifier-chain. The 15mV/V @ 1V range will cause an escalation to the 30mV range.

5.2.1.4 Determining the Line Resistance for Resistive Temperature Devices (RTD) in Three-Wire Configuration

Due to component-spill on the TRION-module different channels on different boards may show a noticeable resistance-offset in 3Wire-Type2 input-configuration.
Usually those offsets are corrected by values stored on the TRION-module and determined during factory-calibration.
To allow users to update those values in case of noticeable deviation this command can be used.

It is mandatory to disconnect all sensors from the module, and to plug a special 3Wire-Short adapter prior execution. Failure to do so, will either result in values that are rejected for storage as they exceed the defined limits, or in wrong values, if the result happens by chance to fall within plausibility bounds.
In the latter case a re-execution of the command with the short-connector plugged in, will yield correct results again, and overwrite the old wrong values.
As an alternative, the i32-command CMD_RESET_SELFCAL can be issued. Be aware, that this command also resets all values determined by the “AmplifierOffset” or “SelfJustageAI”-command.

Using the “ThreeWireInternalLineResistance”-command, API will perform resistance measurements at multiple excitations to determine at least 2 points for linear interpolation for later correction. Those values are then stored on the TRION-Module, and will be used on any subsequent 3Wire-Type measurement for correction.

Like the amplifier offset compensation, this command can be either executed on an idle board, or on a board that is running an acquisition. Refer to 5.2.1.1 and 5.2.1.1.2 for details about the differences between these two modes of operation.

5.2.1.4.1 Command Set

Table 13 - DeWeSetParamStruct_Str commands to determine the line resistance for RTD in three-wire configuration

Target	*Item*	*Parameter*	*Note*
“BoardIDx/AIy” **“BoardIDx/AiAll”** **“BoardIDx/AIgrp”**	“ThreeWireInternalLineResistance”	(none)	

Table 14 - DeWeGetParamStruct_Str commands to determine the line resistance for RTD in three-wire configuration

Target	*Item*	*Parameter*	*Note*
"BoardIDx/AIy" **"BoardIDx/AiAll"** **"BoardIDx/AIgrp"**	"ThreeWireInternalLineResistance"	Buffer to hold the resulting information	The result set will only contain the channels where a measurement of the line resistance has been performed during this load/unload-cycle of the API, and that are queried. So if a ThreeWireInternalLineResistance command has been performed on channels 1, 3, 5, and 7, and a result for ALL is queried, only the channels 1, 3, 5, and 7 will be present in the result set. If for example channels 0, 2, 4, and 6 are queried, the result set will be empty, as the operation has only be performed on channels 1, 3, 5, and 7.

In asynchronous operation mode (that is, when the operation is performed during a running acquisition) the string getter will return a rough estimation of the expected remaining runtime.
A typical implementation pattern for this use case would be to periodically poll the result-string, and test for the presence of an XML-attribute called "EstDuration". Please refer to the SDK-examples for further details.

After obtaining the results, an application typically would parse the resulting XML-document for the determined values of the line resistance. From the point of view of the user, the values are for reference only.

5.2.1.4.2 Structure of returned data

```
<?xml version="1.0"?>
< ThreeWireInternalLineResistance EstDuration="300"/>
```

Illustration 23 - Example line resistance pending result

<?xml version="1.0"?>

```
<ThreeWireInternalLineResistance>
  <BoardId0 BrdName = "TRION-2402-MULTI-S1" Slot = "1" SerialNumber = "1234567890" Passed = "OverLimit">
    <Check Target = "AI0" Type = "ThreeWireInternalLineResistance" Passed = "True">
      <Date>06.07.2016</Date>
```

```
    <Time>08:14:44</Time>
    <ConPanelTemp>False</ConPanelTemp>
    <BaseBoardTemp>0.00</BaseBoardTemp>
    <ID0 Device = "ADC" Range = "100 Ohm" Passed = "True">
      <ThreeWireInternalLineResistance Limit = "3.5 Ohm" Excitation = "4 mA">
          0.002649 Ohm
       </ThreeWireInternalLineResistance>
    </ID0>
    <ID1 Device = "ADC" Range = "300 Ohm" Passed = "True">
      <ThreeWireInternalLineResistance Limit = "3.5 Ohm" Excitation = "1 mA">
          0.026916 Ohm
       </ThreeWireInternalLineResistance>
    </ID1>
  </Check>
  <Check Target = "AI1" Type = "ThreeWireInternalLineResistance" Passed = "True">
    <Date>06.07.2016</Date>
    <Time>08:14:44</Time>
    <ConPanelTemp>False</ConPanelTemp>
    <BaseBoardTemp>0.00</BaseBoardTemp>
    <ID0 Device = "ADC" Range = "100 Ohm" Passed = "True">
      <ThreeWireInternalLineResistance Limit = "3.5 Ohm" Excitation = "4 mA">
          0.018667 Ohm
       </ThreeWireInternalLineResistance>
    </ID0>
    <ID1 Device = "ADC" Range = "300 Ohm" Passed = "True">
      <ThreeWireInternalLineResistance Limit = "3.5 Ohm" Excitation = "1 mA">
          0.027093 Ohm
      </ThreeWireInternalLineResistance>
```

```
        </ID1>
      </Check>
    </BoardId0>
  </ThreeWireInternalLineResistance>
```

Illustration 24 – Example line resistance final result (abridged)

For all “Check”-nodes, that have the attribute “Passed” set to “True”, the node-values of the “ThreeWireLineResistance”-Nodes are stored, together with associated Excitation on the TRION-module, and used for later correction.

DEWETRON

6 SYNCHRONISATION

Multiple TRION (DEWE2) enclosures can be synchronized using various synchronization methods.
Supported methods are:

- TRION-SYNC-BUS
- IRIG A/B (TRION-TIMING, TRION-VGPS)
- GPS (TRION-TIMING, TRION-VPS)
- PTP (TRION-TIMING-V3, TRION-VGPS-V3)

TRION-SYNC-BUS is part of all enclosures. All other options need dedicated enclosures or TRION-TIMING boards in the enclosures first slot.

6.1 TRION-SYNC-BUS

Using TRION-SYNC-BUS needs special setup for two different enclosure roles. There has to be one MASTER instrument and one or more SLAVE instruments.

6.1.1 CABLING

Connect the SYNC cable to the sync-out plug at the master instrument to the sync-in plug of the first slave instrument. For further slave instruments follow the pattern and connect the slave's sync-out plug to the next slaves' sync-in plug.

6.1.2 MASTER INSTRUMENT

The board setup is the same when multiple TRION boards are used. The first board has to be set to Master mode, all others to Slave:

Master board setup "BoardID0":
DeWeSetParamStruct_str("BoardID0/AcqProp", "OperationMode", "Master");
DeWeSetParamStruct_str("BoardID0/AcqProp", "ExtTrigger", "False");
DeWeSetParamStruct_str("BoardID0/AcqProp", "ExtClk", "False");

Slave board setup "BoardIDX" [for X from 1 to *NrOfAvailableBoards*]:
DeWeSetParamStruct_str("BoardIDX/AcqProp", "OperationMode", "Slave");
DeWeSetParamStruct_str("BoardIDX/AcqProp", "ExtTrigger", "PosEdge");
DeWeSetParamStruct_str("BoardIDX/AcqProp", "ExtClk", "False");

SYNC-OUT has to be configured:

- Set the trigger line TRIG7, Source to low
- Set the trigger line TRIG7, Inverted to false

These are the appropriate TRION-API commands:

DeWeSetParamStruct_str("BoardID0/Trig7", "Source", "Low");
DeWeSetParamStruct_str("BoardID0/Trig7", "Inverted", "False");

Then apply the settings using:
DeWeSetParam_i32(0, CMD_UPDATE_PARAM_ALL, 0);

6.1.3 SLAVE INSTRUMENT

On slave devices using TRION-SYNC-BUS has to be configured too.
All boards have to be configured to slave mode!

Slave board setup "BoardIDX" [for X from 0 to *NrOfAvailableBoards*]:
DeWeSetParamStruct_str("BoardIDX/AcqProp", "OperationMode", "Slave");
// Usually "PosEdge"
DeWeSetParamStruct_str("BoardIDX/AcqProp", "ExtTrigger", "PosEdge");
DeWeSetParamStruct_str("BoardIDX/AcqProp", "ExtClk", "False");

SYNC-OUT has to be configured:

- Set the trigger line TRIG7, Source to high
- Set the trigger line TRIG7, Inverted to false

These are the appropriate TRION-API commands:
DeWeSetParamStruct_str("BoardID0/Trig7", "Source", "High");
DeWeSetParamStruct_str("BoardID0/Trig7", "Inverted", "False");

Then apply the settings using:
DeWeSetParam_i32(0, CMD_UPDATE_PARAM_ALL, 0);

6.1.4 ACQUISITION ON THE MASTER INSTRUMENT

Acquisition on the master instrument has to be started using:

For each (slave) board of the instrument start:

```
for (int BoardID = 1; BoardID < NrOfAvailableBoards; ++BoardID)
{
        DeWeSetParam_i32(BoardID, CMD_START_ACQUISITION, 0);
}

// Then start acquisition on the master board
DeWeSetParam_i32(0, CMD_START_ACQUISITION, 0);
```

Please keep in mind:

- Acquisition on the slave boards has to be started before starting acquisition on the master board.

- Acquisition on slave instruments has to be started before starting acquisition on the master instrument.

6.1.5 ACQUISITION ON THE SLAVE INSTRUMENTS

Acquisition on the slave instruments has to be started using:

For each board of the instrument start:

```
for (int BoardID = 0; BoardID < NrOfAvailableBoards; ++BoardID)
{
        DeWeSetParam_i32(BoardID, CMD_START_ACQUISITION, 0);
}
```

6.1.6 SYNC CABLING CHECK

It is possible to check if the sync cables are plugged in correctly.

On each slave instrument use the following commands:

```
int state = 0;
DeWeGetParam_i32(0, CMD_PXI_LINE_STATE, &state);
if ((state & PXI_LINE_STATE_TRIG6) == 0)
{
        // no TRION-SYNC-BUS plugged in on slave instrument
}
```

DEWETRON

7 WORKING WITH CHANNELS

7.1 ANALOGUE CHANNELS (AI)

Technically the path for analogue measurement data consist of three distinct parts on TRION™-boards.

1. The analogue input-path performing the signal-conditioning
2. The A/D conversion
3. Digital data post-processing

The TRION-API however encapsulates the exact details of this chain in a way, so that the various differences in implementation depending on the exact board-type are not visible above the interface. This allows an application to choose a rather generic approach toward analogue channels in general, and frees the application developer from the need to develop against a specific board-type.
The property-set for analogues channels basically describes the whole chain from signal-conditioning to post-processing in an abstract way.

7.1.1 LOGICAL PROPERTY SET

The logical property set for analogue channels is organized beneath various measurement modes.
Under each mode a selected set of configurable parameters exists.
Not all properties are available for all TRION™-boards.
But for each mode a minimum-set of obvious common configuration items can be enumerated.
This chapter will provide an overview over all currently used properties, sorted by the currently supported measurement modes, split into the parameters available on all TRION-board-analogue channels, and those available on specific boards only.
Each property has a list of potential allowed settings. The list has a minimal size of one entry, if the property has a use within the given mode.
For non-trivial measurement modes some of the properties have non-trivial constraints.
Those constraints are derivable from the board-properties-xml-document.
The application does not strictly need to pre-validate properties against those constraints.
The API will usually adjust set property-values to satisfy those constraints, and will issue a WRNING-Level error-code to indicate this to the application.
In such a case, it would be a viable strategy to invoke the property-getter to retrieve the adjusted value for further application-processing.
However: As this approach might not be suitable for all types of applications an exhaustive overview over those property-constraints, and how to validate them on application level.

7.1.1.1 General used attributes

7.1.1.1.1 Default

This indicates the index of the default-setting for the property.
The API will set all settings to their default-values, when the mode is switched.

7.1.1.1.2 ProgMin, ProgMax

Some properties can be programmable in a given interval. If this is the case for a given Property, it is indicated by presence of the two attributes ProgMin and ProgMax.
Both attributes are always in the same unit as the underlying property.

The interval between ProgMin and ProgMax is open – So ProgMin and ProgMax are included.

7.1.1.1.3 Unit

Generally indicates the Unit used with the given property. This includes all fixed list-entries of the list, as well as the unit for ProgMin and ProgMax if given.
In certain modes like Bridge for example, the attribute unit can also work a distinction-predicate, if one property with all its definition may exist multiple times. In bridge-mode this would be for example the case for the property "Range", which exists once with unit = "mV/V" and once with the unit = "mV/mA".

7.1.1.2 Voltage Mode, Calibration Mode

On most TRION™-boards the modes "Voltage" and "Calibration" are very similar.
The Calibration mode usually is more restrictive on the Range-property, but less restrictive on the Input-Types.
The Calibration Mode usually allows for signal routing to onboard calibration-sources that have barely a use in normal measurement.
On the range-side it usually does not allow to use a free programmable value.

7.1.1.2.1 Range

Type: minimum-set
Unit: V (or prefixed V)
Sets the input-range of the amplifier and post processing chain, usually in V.
In terms of Non-TRION™-signal conditioners this is closely related to the used gain.

7.1.1.2.2 InputOffset

Type minimum-set
Unit: V (or prefixed V)
This property is often used synonymous to "Sensor-Offset". It's main use is to shift the virtual 0 V by a given value. Due to various physical effects any non-ideal sensor usually has a bias. With the property input-offset API can be setup to compensate for this bias.

7.1.1.2.3 InputType

Type minimum-set
Unit: N/A
This property indicates the possible input-type-configurations.
For example: Single-Ended, Differential
Note: some TRION-boards only support one non-switchable input type. In this case the property still will be present, but only feature one entry.

7.1.1.2.4 Excitation

Type: board-type-specific
Unit: either V, mA or both
This property allows to configure or disable the excitation (e.g. for sensor-supply).

7.1.1.3 Resistance

7.1.1.4 Bridge

7.1.1.5 Potentiometer

7.1.1.6 RTD-Temperature

7.1.1.7 Current

7.1.1.8 ExcCurrentMonitor

7.1.1.9 ExcVoltageMonitor

7.1.2 ADVANCED CONSTRAINS RULE-SET

In Voltage-measurement mode, the exact amplifier-setting only depends on the set range-property and the set input-offset-attribute.
In the non-trivial measurement modes the amplifier-setting are affected by more than those two logical parameters. A typical example would be bridge-mode, where the amplifier settings are affected by logical range, input-offset and excitation.
While it would be possible to limit each property in a way, so that all possible combination would yield a legal amplifier setup, it would hurt the versatility of the single properties.
This chapter will reveal the dependencies of the various parameters in the different modes, as well as the formulas used to evaluate versus the given constraints.

7.1.2.1 XML-Metainformation used for constraints

Almost all constraints affect the range-property.
Each range-property-node therefor holds several attributes relevant for constraints checking:

7.1.2.1.1 AmplRangeMax, AmplRangeMin, AmplRangeUnit

These attributes indicate the legal maximum and minimum values for the final amplifier-setup. The AmplRangeUnit is always in volt [V].

7.1.2.1.2 MaxInputOffset

Maximum allowed input-offset. This is always given in %-of-range.
On most TRION™-boards this is +/-200%, unless already in the highest possible range, where usually no further input-offset is allowed.

7.1.2.1.3 MaxOutputOffset

The output-offset is the virtual offset introduce by asymmetrical custom ranges. For example a custom range of 0..10V would yield a output-offset of -100%
The limit for the output-offset usually is +/-150%

7.1.2.2 Testing for constraints

7.1.2.2.1 Definitions

7.1.2.2.1.1 Range

As the TRION-API supports asymmetrical custom ranges, the range is split into RangeMin and RangeMax.
RangeMin is the lower value of a given range-span, whereby RangeMax is the upper value.
Examples:

Logical Range	*RangeMin*	*RangeMax*

10 V (= -10V .. 10V)	-10 V	10 V
-5..10V	-5V	10 V
0..10V	0 V	10 V
3..10V	3 V	10 V
-10..5V	-10 V	5 V
-10..0V	-10 V	0 V

7.1.2.2.1.2 AmplifierRange

This is the range (in [V]), the amplifier-path has to be set to, to satisfy the promise, that the interval RangeMin..RangeMax is covered by the raw-value-full-scale.

7.1.2.2.1.3 HWRangeMin, HWRangeMax, HWInputOffset

As the properties Range (RangeMin..RangeMx) and InputOffset are always in logical units (eg Ohms for resistance mode), a intermediate step of conversion is necessary, to translate them to the underlying voltage-measurements. The HWRangeMin/Max and InputOffset are used subsequentially to calculate the AmplifierRange.
The main-purpose of those values is to keep the calculation comprehensible.

7.1.2.2.2 Amplifier range

The result of the calculated AmplifierRange must always satisfy following condition:

$$AmplRangeMin \leq AmplifierRange \leq AmplRangeMax$$

7.1.2.2.2.1 Voltage mode, Calibration mode

Depending on properties: Range, InputOffset

$$HWRangeMin\ [V] = RangeMin\ [V]$$
$$HWRangeMax\ [V] = RangeMax\ [V]$$
$$HWInputOffset\ [V] = InputOffset\ [V]$$

$$AmplifierRange\ [V] = \max(abs(HWRangeMin + HWInputOffset), abs(HWRangeMax + HWInputOffset))$$

7.1.2.2.2.2 Resistance mode

Depending on properties: Range, InputOffset, Excitation

$$HWRangeMin\ [V] = RangeMin\ [Ohm] * Excitation\ [A]$$
$$HWRangeMax\ [V] = RangeMax[Ohm] * Excitation\ [A]$$
$$HWInputOffset\ [V] = InputIOffset\ [Ohm] * Excitation\ [A]$$

$$AmplifierRange\ [V] = \max(abs(HWRangeMin + HWInputOffset), abs(HWRangeMax + HWInputOffset))$$

7.1.2.2.2.3 Bridge mode

Depending on properties: Range, InputOffset, Excitation

Note: Excitation and Range are related.
The calculation is shown for mA-unit. Formulas also apply for V-excitations.

Excitation Unit	***Range Unit***
mA	mV/mA
V	mV/V

$$HWRangeMin\ [V] = \frac{RangeMin\left[\frac{mV}{mA}\right] * Excitation[mA]}{1000}$$
$$HWRangeMax\ [V] = \frac{RangeMax\left[\frac{mV}{mA}\right] * Excitation[mA]}{1000}$$
$$HWInputOffset[V] = \frac{InputOffset\left[\frac{mV}{mA}\right] * Excitation[mA]}{1000}$$

$$AmplifierRange\ [V] = \max(abs(HWRangeMin + HWInputOffset), abs(HWRangeMax + HWInputOffset))$$

7.1.2.2.2.4 Potentiometer

Depending on properties: Range, InputOffset, Excitation

$$HWRangeMin\ [V] = \frac{RangeMin\ [\%] * Excitation\ [V]}{100} - \frac{Excitation\ [V]}{2}$$
$$HWRangeMax\ [V] = \frac{RangeMax\ [\%] * Excitation\ [V]}{100} - \frac{Excitation\ [V]}{2}$$
$$HWInputOffset = InputOffset\ [\%] * Excitation\ [V]$$

$$AmplifierRange\ [V] = \max(abs(HWRangeMin + HWInputOffset), abs(HWRangeMax + HWInputOffset))$$

7.1.2.2.2.5 RTD-Temperature mode

TODO

7.1.2.2.2.6 Current mode, ExcCurrentMonitor

Depending on properties: Range, ShuntRes

$$HWRangeMin\ [V] = RangeMin\ [A] * ShuntRes\ [Ohm]$$
$$HWRangeMax\ [V] = RangeMax\ [A] * ShuntRes\ [Ohm]$$
$$HWInputOffset\ [V] = InputOffset\ [A] * ShuntRes\ [Ohm]$$

$$AmplifierRange\ [V] = \max(abs(HWRangeMin + HWInputOffset), abs(HWRangeMax + HWInputOffset))$$

8 SDK EXAMPLES

Programming examples are part of the normal TRION installer package. Just use manual installation and select the TRION SDK packackage.

8.1 LoadConfiguration / LoadConfigurationMultiBoard

8.1.1 Location:

- Folder LoadConfiguration
- Folder LoadConfigurationMultiBoard

8.1.2 Purpose

This example shows how a previously stored logical board-configuration can be loaded from a file, and thereby be re-applied to a board.

8.2 AmplifierBalanceAutonomous / AmpflifierBalanceDuringMeasurement

8.2.1 Location

- Folder AmplifierBalanceAutonomous
- Folder AmplifierBalanceDuringMeasurment

8.2.2 Purpose

8.3 Enhanced Analog Measurement

8.3.1 Location

- Folder EnhancedAnalogMeasurement

8.3.2 Purpose

Demonstrate the usage of the `CMD_UPDATE_PARAM_AI string command during a running measurement. This is very helpful, if the user wants to update analogue channel properties during a measurement, without closing the measurement.` The example updates the HWRange and HPFilter properties every three seconds and prints the measured data to the console. For sure also all other channel properties belonging to an analog channel, could be taken into account.

9 TRIONET SDK EXAMPLES

Currently there are two examples explaining the differences between programming local installed TRION modules and modules accesses using TRIONET.

The examples are:

- TRIONET_ListBoards
- TRIONET_OneAnalogChannel

99% of the source code is essentially the same. The difference is that the header “dewepxinet_load.h” should be used instead of “dewepxi_load.h”.

That way the TRIONET wrapper API is used completely hiding the network management and communication to TRIONET devices.
The programmer has only to select the network interface to use and is ready to measure.

```
void configureNetwork()
{
   int nErrorCode;

   // Optional: Prints available network interfaces
   // ListNetworkInterfaces();

   char* address = "127.0.0.1";
   char* netmask = "255.255.255.0";

   printf("Example is listening for TRIONET devices on: %s (%s)\n", address, netmask);

   // TODO: Configure the network interface to access TRIONET devices
   nErrorCode = DeWeSetParamStruct_str("trionetapi/config", "Network/IPV4/LocalIP", address);
   CheckError(nErrorCode);
   nErrorCode = DeWeSetParamStruct_str("trionetapi/config", "Network/IPV4/NetMask", netmask);
   CheckError(nErrorCode);
}
```

10 LIST OF ILLUSTRATIONS AND TABLES

11 FUTURE CHAPTERS

This section holds a list of ideas and recommendations from users about chapters which will to be added to this document in the future:

- Detailed information about setting up asynchronous channels
- Explanation of hardware time stamping and correlating asynchronous data to synchronous data
- Add integer based commands to glossary
- Add a list of aliases for string-commands to the glossary
- Short overview of logging mechanisms of the API to support developers
- Reference to documentation of synchronizing several systems to each other