Irinos EC

Diese Betriebsanleitung wurde für die Darstellung in einem Webbrowser im HTML-Format optimiert. Verwenden Sie die PDF-Version nur, wenn kein Zugriff auf die Online-Hilfe möglich ist.

Overview

1 Overview

Diese Web-basierte Hilfe besteht aus 3 Handbüchern:

A. **Irinos EC Users Manual** 24

This is the **main documentation**. Start here.

B. Irinos Tool Users Manual 93

This documentation describes the operation of the setup and diagnostic software Irinos-Tool / ITool.

C. NmxDLL Referenz (Software API) 139

Description of the software interface of the NMX DLL. For software developers.

Irinos EC Users Manual

2 Irinos EC Users Manual

2.1 Introduction

This users manual explains the setup, installation and handling of the Irinos EC measuring system. Read it before using the product.

It has been optimized for viewing on electronic terminals and contains multimedia content. It is therefore recommended to use the electronic version with a PC, tablet, smartphone or similar.

At the beginning you find this introduction together with the legal notes and the safety instructions 19.

- The System Overview 24 provides a general introduction to the Irinos EC - System. It is supplemented by the product descriptions 29 and the pin assignment 37.
- This is followed by information on the assembly 39 and setup 48 of the system.
- In addition, you will find information on diagnosis 63, maintenance, cleaning and disposal 75 as well as specific application notes 77.

Imprint

Title	Irinos EC
Manufacturer	Messtechnik Sachs GmbH Siechenfeldstraße 30/1 D-73614 Schorndorf Tel. +49 / 7181 / 26935-0 post@messtechnik-sachs.de
Valid for	Measurement modules Irinos EC
Copyright Note	© 2019-2020 Messtechnik Sachs GmbH
Trademarks	All product names used in this manual are trademarks of their respective owners.
Material-No.	785-1028
Change notice	Subject to change without notice.
Release date	05.06.2020

2.1.1 Revision History

Version	Date	Changes
A	2019-08-19	First Version

2.1.2 Legal Notes

2.1.2.1 Terms and conditions of use for software & documentation

I. Protection rights and scope of use

Messtechnik Sachs provides operating instructions, manuals, documentation, and software programs - all collectively referred to as "LICENSED OBJECT" below - either on portable data storage devices (e.g. diskettes, CD ROMs, DVDs, etc.), in written (printed) form or in electronic form, for a fee and/or free of charge. The LICENSED OBJECT is subject to proprietary safeguarding provisions among other regulations. Messtechnik

Sachs or third parties have protection rights for this LICENSED OBJECT. In so far as third parties have whole or partial right of access to this LICENSED OBJECT, Messtechnik Sachs has the appropriate rights of use. Messtechnik Sachs permits the user the use of the LICENSED OBJECT under the following conditions:

1.1) Scope of use for electronic documentation

a) With the acquisition/purchase or relinquishment of a LICENSED OBJECT, you as the user acquire a simple, non-transferable right of use with regard to the respective LICENSED OBJECT. This right of use authorises the user to use the LICENSED OBJECT for the user's own, exclusively company-internal purposes on any number of machines within the user's business premises. This right of use includes exclusively the right to save the LICENSED OBJECT on the central processors (machines) used at the location.
b) Irrespective of the form in which operating instructions and/or documentation are provided, the user may furthermore print out any number of copies on a printer at the user's location, providing this printout is printed with or kept in a safe place together with these complete terms and conditions of use and other user instructions.
c) With the exception of the Messtechnik Sachs logo, the user has the right to use pictures and texts from the operating instructions/documentation for creating the user's own machine and system documentation. The use of the Messtechnik Sachs logo requires written consent from Messtechnik Sachs. The user is responsible for ensuring that the pictures and texts used match the machine/system or the product.
d) Further uses are permitted within the following framework:

Copying exclusively for use within the framework of machine and system documentation from electronic documents of all documented supplier components. Demonstrating to third parties exclusively under guarantee that no data material is stored wholly or partly in other networks or other data storage devices or can be reproduced there.
Passing on printouts to third parties not covered by the regulation in item 3, as well as any processing or other use are not permitted.

1.2) Scope of use for software products

For any type of Messtechnik Sachs software including the associated documentation, the customer shall receive a non-exclusive, non-transferable and time-unlimited right of use on a certain hardware product or on a hardware product to be determined in individual cases. Messtechnik Sachs shall remain the owner of the copyright as well as of any other industrial property rights. The customer may make copies for back-up purposes only. Any copyright notes may not be removed.

2. Copyright note

Every LICENSED OBJECT contains a copyright note. In any duplication permitted under these provisions, the corresponding copyright note of the original document concerned must be included:

Example: © 2019, Messtechnik Sachs GmbH,
D-73614 Schorndorf

3. Transferring the authorisation of use
The user can transfer the authorisation of use re. the respective LICENSED OBJECT as per these provisions in the scope and with the limitations of the conditions in accordance with items 1 and 2 completely to a third party. The third party must be made explicitly aware of these terms and conditions of use.

II. Exporting the LICENSED OBJECT
When exporting the LICENSED OBJECT or parts thereof, the user must observe the export regulations of the exporting country and those of the acquiring country.

III. Warranty
1. Messtechnik Sachs products are being further developed with regard to hardware and software. If the LICENSED OBJECT, in whatever form, is not supplied with the product, i.e. is not supplied on a data storage device as a delivery unit with the relevant product, Messtechnik Sachs does not guarantee that the electronic documentation corresponds to every hardware and software status of the product. In this case, the printed documentation from Messtechnik Sachs accompanying the product is alone decisive for ensuring that the hardware and software status of the product matches that of the electronic documentation.
2. The information contained in an item of electronic documentation can be amended by Messtechnik Sachs without prior notice and does not commit Messtechnik Sachs in any way.
3. Messtechnik Sachs guarantees that the software program it created agrees with the change description and program specification but not that the functions included in the software run entirely without interruptions and errors or that the functions included in the software can run or meet the requirements in all combinations selected by and in all conditions of use designated by the acquirer.

IV. Liability/limitations on liability
1. Messtechnik Sachs provides LICENSED OBJECTS to allow the user to use - in conformity with the contract - Messtechnik Sachs products which require software for proper operation, or to assist the user in creating the user's machine and system documentation. In the case of electronic documentation which in the form of data storage devices does not accompany a product, i.e. which is not supplied together with that product, Messtechnik Sachs does not guarantee that the electronic documentation separately available/supplied matches the product actually used by the user.
The latter applies particularly to extracts of the documents for the user's own documentation. The guarantee and liability

for separately available/supplied portable data storage devices, i.e. with the exception of electronic documentation provided on the Internet/Intranet, are limited exclusively to proper duplication of the software, whereby Messtechnik Sachs guarantees that in each case the relevant portable data storage device or software contains the latest status of the documentation. In respect of the electronic documentation on the Internet/Intranet, it is not guaranteed that this has the same version status as the last printed edition.
2. Furthermore, Messtechnik Sachs cannot be held liable for the lack of economic success or for damage or claims by third parties resulting from use of the LICENSED OBJECTS by the user, with the exception of claims arising from infringement of protection rights of third parties concerning the use of the LICENSED OBJECTS.
3. The limitations on liability as per paragraphs 1 and 2 do not apply if, in cases of intent or wanton negligence or lack of warranted quality, liability is absolutely necessary. In such a case, the liability of Messtechnik Sachs is limited to the damage recognisable by Messtechnik Sachs when the specific circumstances are made known.

V. Safety guidelines/documentation
Guarantee and liability claims in conformity with the regulations mentioned above (items III and IV) can only be made if the user has observed the safety guidelines of the documentation in conjunction with use of the machine and its safety guidelines or the terms and conditions of use of the software. The user is responsible for ensuring that the electronic documentation, which is not supplied with the product, matches the product actually used by the user.

2.1.2.2 Warning notice system

This users manual contains notes, which you must observe to ensure your personal safety, as well as to protect the product and connected equipment. Notes related to your personal safety are highlighted by a yellow exclamation mark. Notes for property / material damage are without an exclamation mark. These notices are graded according to the degree of danger.

	**Danger**
	indicates the immediate threat of danger. If it is not avoided, it will result in death or serious injury.

	**Warning**
	indicates a possibly dangerous situation. If it is not avoided, it may result in death or serious injury.

	**Caution**
	indicates a possibly dangerous situation. If it is not avoided, it may result in injury.

Attention
indicates a possibly harmful situation. If it is not avoided, the product or related / surrounding equipment may be damaged.

2.1.2.3 Qualified personnel

The product system described in this documentation must only be handled by qualified personal according to the given scope of work. All documentation relevant for the scope of work must be observed, especially the safety and warning notes. Due to its education and experience, qualified personal is able to identify risks and possible dangers when using this products / systems.

2.1.2.4 Disclaimer

The content of this documentation has been carefully reviewed to comply with the documented hard- and software. We can, however, not exclude discrepancies and do therefore not accept any liability for the exact compliance. This documentation is reviewed regularly. Corrections may be contained in newer versions.

2.1.3 Preface

	**Warning**
	Carefully read this complete users manual and all related documentation before setup and use of the Irinos-System. This applies especially to the safety instruction. Misuse may lead do death, serious injury, injury or damage of man, equipment or machine.

Purpose

This users manual contains all relevant information for setup, use and maintenance of the Irinos-System. Target groups are users and service technicians, who setup the product or who perform system diagnostics.

Scope of this users manual

This users manual is valid for the industrial measurement system Irinos EC and related accessories.

Intended use

Irinos EC is a flexible High-Speed measurement system for the industrial production measurement technology.

The measurement device is not appropriate for use in medical fields or in explosive areas, for aerospace and for home- or office use. Other fields of application, which are not mentioned but similar, are also excluded from use.

In safety critical areas, the safety in operation must be ensured by external equipment (e.g. external emergency stop).

Please note:

	Warning
	Products from Messtechnik Sachs GmbH must only be used for applications, which are mentioned in the datasheet or in the related documentation. If third party products are used, these must be recommended or permitted by Messtechnik Sachs GmbH. Proper and safe operation of the products require appropriate transportation, storage, mounting, usage and maintenance. Environmental conditions stated in the specification must be observed as well as notes in the related documentation.

Required knowledge

For the mechanical integration and mounting, solid knowledge and skills in mechanics and machinery are required.

For the electrical installation and the setup, solid knowledge and skills in electrics and electrical safety are required.

For the setup of the measurement application, profund knowledge in industrial measurement technology is required as well as PC skills.

Further documentation

Please note the short booklet, which is delivered with each Irinos module. This applies especially to the safety warnings, which are mentioned in it. The specifications of the Irinos-Boxes can be found in the respective datasheet.

A separate documentation is available for the setup tool "ITool".

For the integration of the software library used for attaching the Irinos to PC software, a separate reference manual is available.

Firmware version

This users manual is related to firmware version 2.

(Version 2 is the first version. Version number 1 has been skipped to provide more clarity in regard to the Irinos IR system.)

2.1.4 Safety Instructions

Attention
Damage by opening the device Do not open the Irinos components. They are designed for use without the necessity to open them. The measurement box and/or the measurement system may be damaged. Malfunction or destruction are possible results. Opening the Irinos components will void the warranty.

Attention
Unintended operating situation High frequency radiation, e.g. from a mobile phone, can interrupt the device operation and may lead to malfunction of the Irinos-System. People or material can be injured or damaged. Avoid high frequency radiation: ○ Do not place sources of radiation next to the Irinos measurement system. ○ Turn off devices, which are a source of radiation. ○ Reduce the radio performance of radiation emitting devices. Ensure the compliance regarding electromagnetic compatibility.

	Warning
	**Electric shock** An insufficient earth grounding and/or electrical separation from the mains can lead to injury or damage of people or machine. Please note: ○ Only use PELV supply circuits according to IEC60204-1 (Protective Extra-Low Voltage, PELV). ○ Observe the additional requirements for PELV supply circuits according to IEC60204-1. Only use power supplies, which allow a safe separation of the operating voltage and the load voltage according to IEC60204-1.

	**Warning**
	Danger at unprotected machinery For the operation of machinery, the following has to be observed: At unprotected machinery danger may exist according to the results of a risk analysis. This danger may lead to personal injury. Personal injury can be avoided according to a risk analysis by the following actions: ○ Additional protection equipment at the machinery. Thereby especially programming, parametrization and wiring of the peripherals must comply with the safety performance (SIL, PL or Cat.), which has been assessed in a risk analysis. ○ Appropriate use of the Irinos-System, which is verified by a functional test at the machinery. This allows identifying programming, parametrization or wiring mistakes. ○ Documentation of the test results in the relevant safety documentation.

Attention
Electrostatic sensitive devices An Irinos-Box contains electrostatic sensitive devices. It is possible that electrostatically sensitive equipment is destroyed by energies and voltages that are far less than the human threshold of perception. Do not open the Irinos-Box. Thereby you avoid touching the sensitive devices.

Attention
Damage of the Irinos-System by transport and storage If the Irinos-System is transported or stored without packaging, shocks, vibrations, pressure or moisture may harm the Irinos components. Damaged packaging signals, that environmental conditions have already affected the Irinos components. The Irinos components and/or the Irinos-System may be damaged. Do not dispose the original packaging. Proper packaging is required during transport and storage.

Attention
Damage due to condensation If the Irinos components or the Irinos-System are exposed to low temperatures or high temperature changes, moisture may cover the Irinos-System or the components. Moisture leads to short circuit and damages the Irinos-System. To avoid damage, please observe the following advices: o Wait before use, until the temperature of the Irinos-System has adjusted to the surrounding temperature. o Avoid direct heat radiation next to the Irinos-System. o If moisture is present, wait until the Irinos-System has completely dried (approximately 8 hours).

Attention

Environmental conditions and chemical resistance

Environments, which are not appropriate for the Irinos-System, may lead to malfunction. Chemical substances (e.g. cleaning agent) can change the colour, form or structure of the device.

The Irinos-components may be damaged. This may result in malfunction.

Please note:

- Only use the Irinos-System in closed rooms.
- Only use the Irinos-System according to the environmental conditions given in the specifications.
- Protect the Irinos-System against dust, moisture and heat.
- Do not place the Irinos-System into direct sunlight or other strong sources of light.
- Without additional actions, the Irinos-System must not be used in surroundings where caustic vapours or gases are used.
- Only use appropriate cleaning agents.

Any and all warranty or liability claims are excluded if these regulations are violated.

Inappropriate cleaning agents may damage the device.

Only use washing-up liquid for cleaning. Do not use:

- Aggressive solvents and abrasive cleaner
- Steam jet
- Compressed air
- Vacuum cleaner

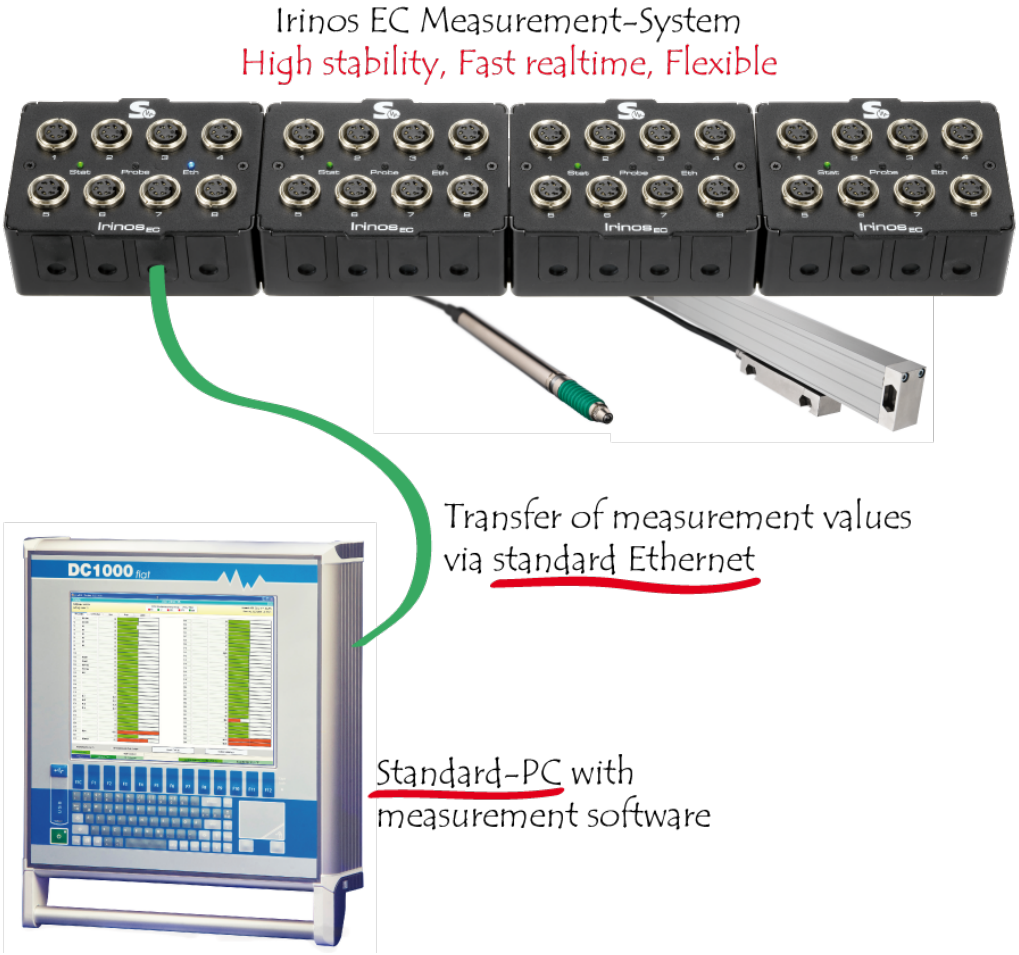
	**Caution**
	Unexpected / unintended reaction while cleaning the Irinos-System If the Irinos-System is in operation while cleaning, this may result in unintended actions. This may lead to personal injury or damage at machinery. Always turn off the Irinos-System before cleaning.

2.2 System Overview

As part of an industrial measurement system, the Irinos-System covers all functionality, which is time-critical and hardware dependant.

It allows for an easy connection of various probe and sensor types, e.g. inductive probes. Via a directly attachable foot-switches or push-button and via digital I/Os, control functions can be integrated.

The measurement values and the I/O data is exchanged via Standard-Ethernet to the PC, where it is processed in a measurement software:



For time-critical applications, measurement data can be sampled synchronously in realtime. The measurement values are sampled and buffered simultaneously by the Irinos-System. The Irinos-System sorts them similar to a table and transfers them to the PC.

This real-time sampling is independent of the time-response of the PC! As an example, roundness and form measurements can even be done using a standard laptop without special hardware or realtime extensions. Integration on the PC side 56 is done via a Windows DLL with a simple but powerful API.

Measurement hardware, software and PC are available from Messtechnik Sachs as a complete system. However, due to the open concept, the Irinos can also be used with measurement software and PCs from other manufacturers. Alternatively the DLL can be integrated into your own measurement software.

2.2.1 Modularity

Due to its flexible design, the Irinos-System can be used for a wide range of measurement applications.

An Irinos system consists of a minimum of 1 und up to 8 Irinos-Boxes. Each Irinos-Box has a fixed number of measurement inputs (e.g. 2, 4 or 8). The

Irinos boxes are connected in line topology via the EC-Link interface, i.e. there is always a EC-Link cable between two Irinos boxes.

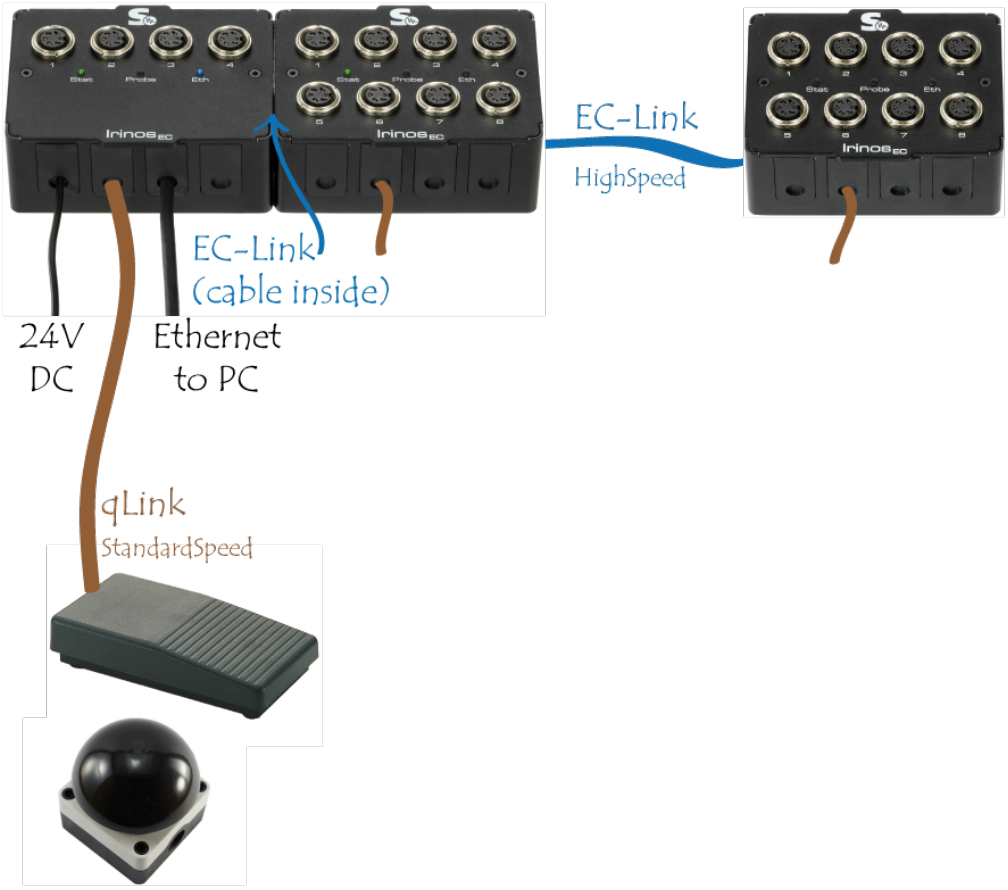
If 2 boxes are placed directly next to each other, the EC-Link cable can be placed on the inside, i.e. it is invisible. Longer distances can be bridged by external cables, whereby the total distance between the first and the last Irinos box can be 10m.

The EC-Link interface combines three tasks in one:

a) Data exchange between the Irinos Boxes.

b) Time-Synchronisation of all measurement channels.

c) Forwarding the power supply to all Irinos-Boxes.



Irinos-Boxes with different types of measurement inputs can be combined without restrictions.

For the PC respectively the measurement software, the number of Irinos-Boxes is irrelevant. It always "sees" one Irinos-System, whose number of measurement channels is determined by the Irinos-Boxes available.

The measurement values are transferred automatically between the Irinos-Boxes via the EC-Link interface. Via Ethernet they are transferred as a coherent block to the PC.

In addition, each Irinos-Box has a "qLink"-Interface, which allows for connecting simple extensions. One example is the direct connection of a foot-switch or push-button.

2.2.2 Syncronization & Speed

The simplest form of measurement is the so-called **static measurement**, in which the measured values of all measurement channels are updated asynchronously with an update rate of approx. 30 Hz. It is always active and requires no parameterization. This is completely sufficient for numerous measurement tasks.

In addition to static measurement, the Irinos system also offers the possibility of fast **real-time measurement sampling** up to 4000 measured values/s. All measuring channels connected via the EC-Link interface can be included at full speed. In addition, all digital inputs and outputs can be included.

The real-time measurement requires the connection of the Irinos system to the measurement software via the NmxDLL 58.

All Irinos boxes on the EC-Link interface have the same system time called "EC-Link-time" (unit: µs). Technical deviations of the individual Irinos boxes from the EC-Link time are continuously corrected. This does not have to be parameterized or controlled by the user. In practice, the deviations are in the range of a few microseconds (µs).

The measured value sampling is triggered simultaneously for all Irinos boxes based on the EC-Link-time. The measured values are then temporarily stored in the internal buffer of the Irinos box and then merged with the other measured values by the master box. This allows the measured values of all channels to be recorded simultaneously at very high speed.

For a **time-limited real-time measurement**, the following applies: The real-time capability is independent of the number of probes, since each Irinos box has its own measurement value buffer. For example, all measurement channels can be samples simultaneously with a sampling rate of 4000

measurement values/s. The memory is dimensioned so that the measured values can be recorded for at least 10 seconds at the maximum sampling rate. If the sampling rate is lower, this time increases accordingly.

Only the transmission time to the PC depends on the number of channels and the measuring rate. Depending on the data type of the connected probes, between 80.000 and 200.000 measured values/s can be transmitted as a rough guide value.

For an **endless real-time measurement**, the following applies: The maximum sampling rate depends on the number of channels. The following maximum sampling rates are recommended:

Recommended maximum sample rate (for endless measurement)	Number of measurement channels
4000	1-16
2000	17-32
1000	33-64

2.2.3 Master vs. Slave

Each Irinos system has exactly one master box. This is the Irinos box that is connected to the PC via Ethernet. All other (optional) Irinos boxes are called slave boxes.

After power on, each Irinos box first acts as a slave box. It then checks whether a network cable is connected (i.e. Ethernet link active). If this is the case, it automatically becomes the master box and communicates this to the other boxes.

The use of several Ethernet connections in one system is not permitted.

Information for users who already know the "Irinos IR" system:

The "Irinos EC" system differs in this respect from the classic "Irinos IR" system, where each box is either a master or slave box when delivered.

2.2.4 Power Supply

It is imperative that you observe the safety instructions 19 with regard to the electrical voltage! It is particularly important to ensure that a power supply unit with functional extra-low voltage (PELV) is used.

All Irinos boxes are supplied via a common 24 V voltage. This can be supplied at any point in the system at a socket or terminal provided 43 for this purpose. It is distributed over the entire system via the EC-Link- and qLink- cables.

An exception are the digital inputs/outputs, which must be fed in separately depending on the box type. This also makes it possible to include the digital inputs/outputs in an emergency stop circuit without completely switching off the Irinos system.

The voltages required for measurement and communication are generated internally via galvanically isolated DC/DC converters and, if necessary, downstream linear regulators. This means that the internal voltages of several Irinos boxes are completely separated from each other. This increases the interference immunity. This is a prerequisite for precise measurement results. In addition, the formation of ground loops is prevented.

Please note, however, that a stable and low-noise 24 V supply is a prerequisite for proper operation of the Irinos system. Therefore use a separate power supply unit with functional extra-low voltage (PELV) for the Irinos system.

2.3 Product Descriptions

The product descriptions give an overview of the individual Irinos boxes as well as various accessories. The pin assignment and technical details of the individual connections can be found in the chapter Pin Assignments 37.

Available measurement boxes (cascadable via EC-Link - Interface)

Order Number	Name	Number of measurement channels or in-/outputs	Short description

828-6050	EC-TFV 35-8-TESA-M16-EPI	8	Measurement box for inductive probes type **TESA HalfBridge** or compatible ones
828-6051	EC-TFV 35-4-TESA-M16-EPI	4	

Accessories EC-Link cables

Order Number	Length	Short description
828-6200	ca. 0.15m for invisible cabling inside	EC-Link cable for cascading multiple Irinos-Boxes via the EC-Link interface 25
828-6201	0.5m	
828-6202	1.5m	
828-6203	3m	

Accessories desktop power supply

Order Number	Included mains cable	Power
828-6130	For EU: Schuko connector, Type F	Input: 115/230V, 50/60Hz Output: 24V DC, max. 60W
828-6131	For USA/Mexico: NEMA 5 connector, Type B	
828-6132	For India: Connector BS546, Type D	

828-6133	For Switzerland: Connector SEV1011, Type J	
828-6134	For UK: Connector BS1363, Type G	

Accessories for mounting

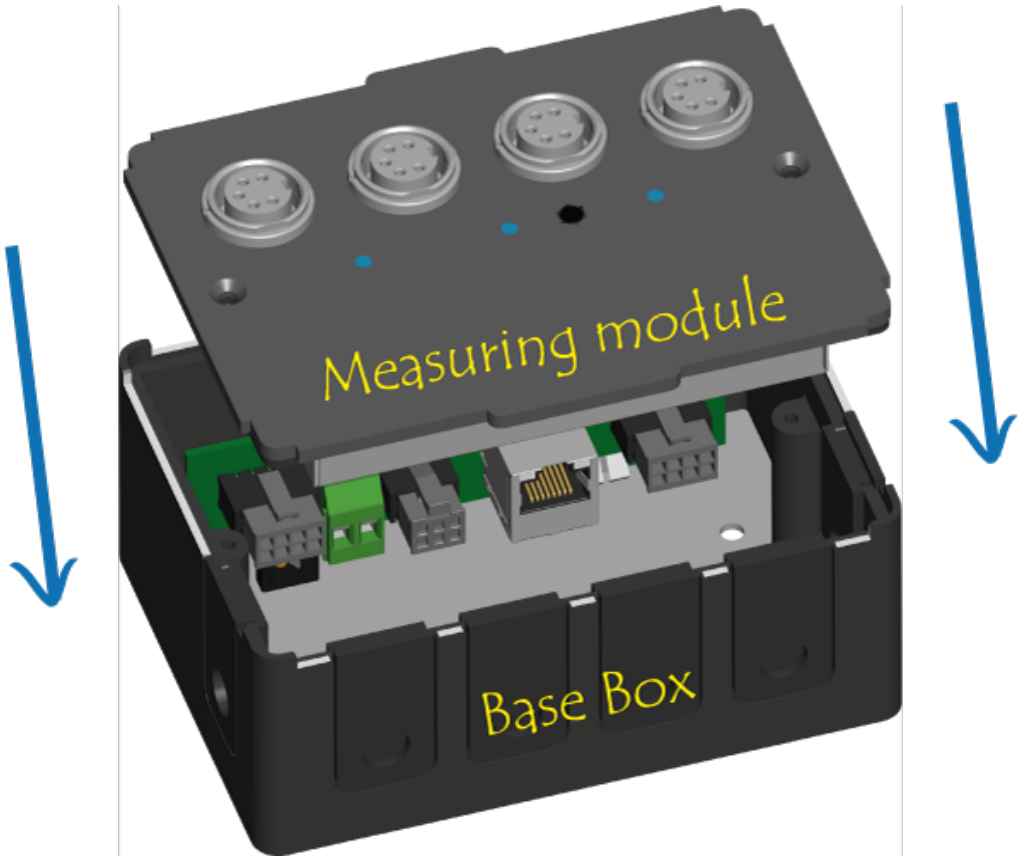
Order Number	Name	Short description
828-6150	EC-MHRM-1	Adapter for mounting an Irinos EC - Box on a DIN-Rail / H-Rail / Hat-Rail

Accessories network cables

Order Number	Length	Short description
828-6181	2m	Irinos EC network cable, 2 x RJ45
828-6182	5m	
828-6183	10m	
828-6184	15m	

2.3.1 Basic Composition Irinos-EC - Box with EC-Link Interface

An Irinos-EC - box with EC-Link - interface consists of 2 parts, the base box and the measuring module:

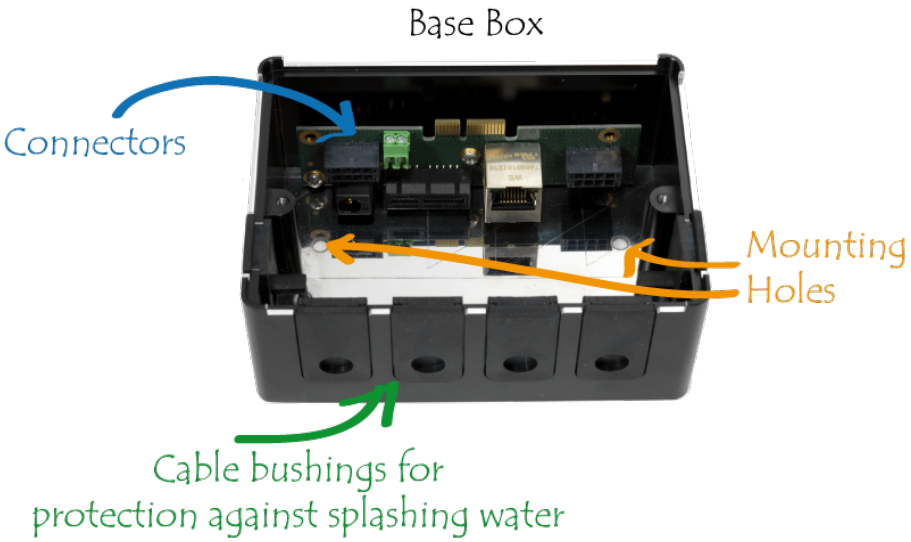


Base Box / EC-Link

The base box is identical for all Irinos EC boxes with EC-Link interface. It does not contain any active electronics.

A special feature of the housing concept is that all connections are internal (except measurement inputs and digital inputs/outputs). This has several advantages:

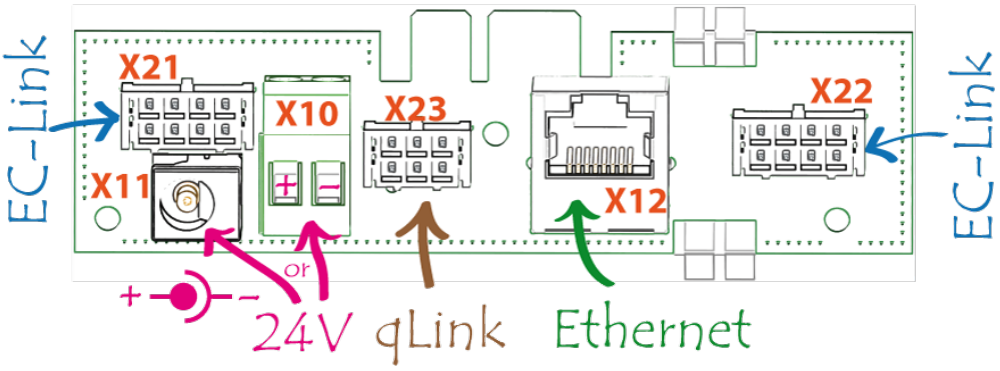
- Optically appealing, as the connectors are not visible from the outside.
- Use of inexpensive standard cables possible, no industrial connectors required. The required dust/fluid protection is nevertheless achieved by suitable cable grommets.
- Compact yet very flexible system design possible.
- Quick exchange of the measuring module possible.



On the bottom side there are 2 holes for screws M4. This allows the basic box to be fastened directly. Alternatively, the optional DIN rail adapter can also be attached here.

The base box contains **connectors** for:

- X10 / X11: Power supply 29 24V DC (Input)
- X12: Ethernet-Interface for the connection to the PC
- X21 / X22: EC-Link - Interface for cascading multiple Irinos Boxes
- X23: qLink - Interface for expanding the Box with simple extensions (e.g. foot switch)



The PCB with the connectors may move slightly. This is intended!

Measuring module

The measuring module is mounted in the basic box and contains the external connections for the actual measuring or control function, such as the connections for inductive probes. It contains the complete electronics.

Measuring Module

Front View

Rear View

Each measuring module has at least 2 **LEDs** at the front, one for the box status and one for the Ethernet connection.

LED "Stat" / green: Box Status	
Flashing: lange on, short off	Everything ok
Fast flashing	An error 63 has occurred. The error type can be read out for example with the ITool 51 or via the DLL interface 56.

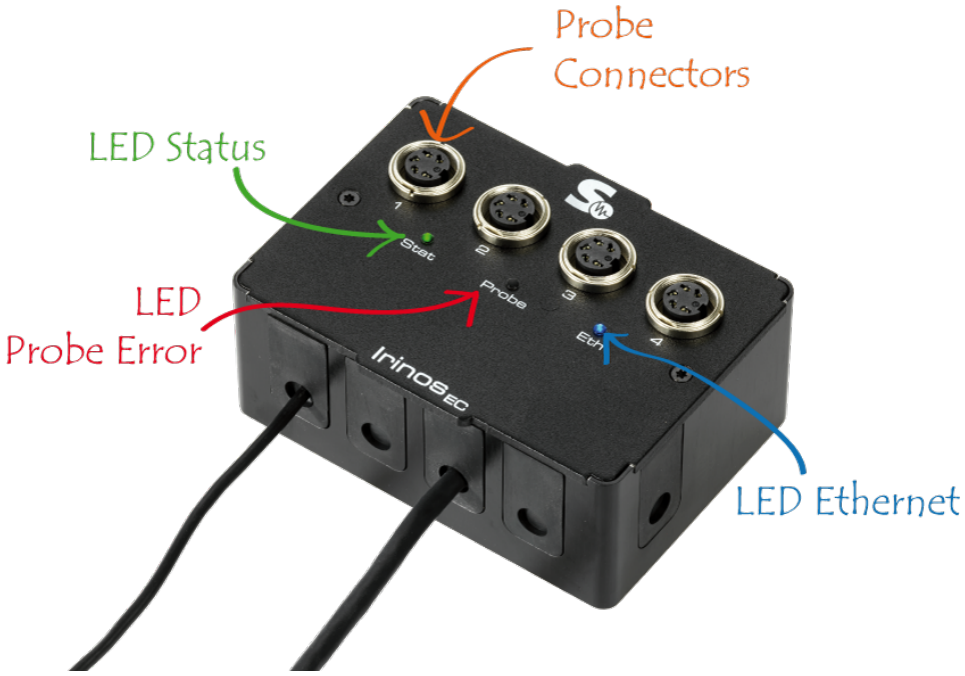
LED "Eth" / blue: Network Status	
Off	There is no network connection. This is the normal case for slave boxes 28.
On	A network connection has been established, but no data is transferred. This is the normal case if the physical connection to the PC has been established but no measurement software has been started.

Flashing	A network connection exists and data is transferred. This is the normal case if the connection between the measurement software and the Irinos system has been established.

2.3.2 EC-TFV for inductive probes

The measuring box EC-TFV is suitable for the connection of inductive probes. The supported probe type can be found on the type plate.

The connections for the inductive probes are located on the front of the measuring box.



In addition to the Standard LEDs 34, this Box has a "Probe"-LED:

LED "Probe" / red: Signalling a probe error	
Off	Everything ok or no error recognizable.

On	Short circuit of the probe supply.

Data acquisition

All measuring inputs are **sampled synchronously** (no multiplexing). The measurement method takes the complete measurement signal into account (i.e. integrating measurement). Compared to the frequently used 1-point or 2-point sampling, this method has the advantage that its sensitivity to interference is much better. The analog filter used in this measuring method is designed for the mechanical properties of the respective probe.

For technical reasons, different input channels never have exactly the same measured value with an identical input signal. In order to enable a trouble-free change of the input channel or an exchange of the Irinos box, the measuring inputs are digitally pre-calibrated at the factory. The adjustment is made in such a way that the measured value -32,000 is supplied for the maximum negative nominal deflection, the measured value 0 for the middle of the probe and +32,000 for the maximum positive nominal deflection.

The internal data type is "16 Bit signed integer".

Digital value	**Probe deflection Tesa HalfBridge GT21** **Sensitivity: 73.75mV/V/µm**	**Probe deflection Tesa HalfBridge GT61** **Sensitivity: 29.5mV/V/µm**
- 32,000	- 2000 µm	- 5000 µm
0	0 µm	0 µm
+32,000	+ 2000 µm	+ 5000 µm
+32,767	Probe error detected	

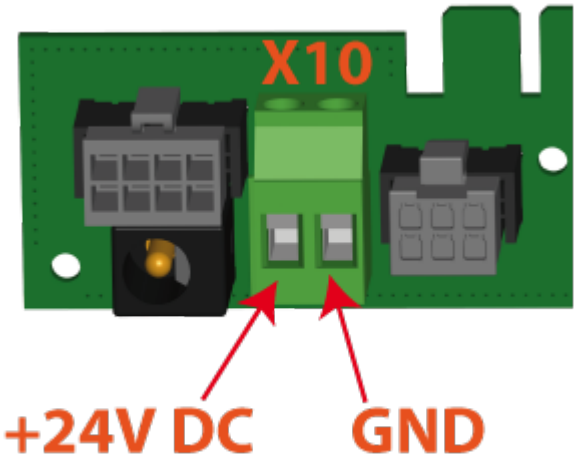
2.4 Pin assignments

Attention
Always use ready-made cables to ensure proper operation. Otherwise, there is a risk that components of the Irinos system or associated components may be damaged. This does not apply to terminal connections, e.g. for power supply with 24V DC via connector X10 31.

2.4.1 Power supply 24V

It is imperative that you observe the safety instructions 19 with regard to the electrical voltage! It is particularly important to ensure that a power supply unit with functional extra-low voltage (PELV) is used.

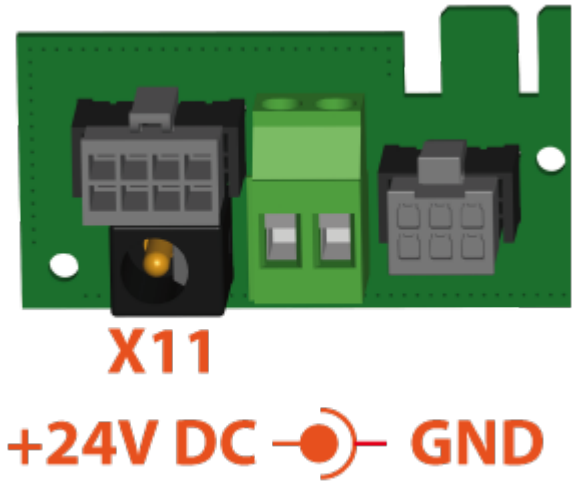
Alternative 1 via Connector X10 31 (Terminal Block):



Before connection, provide the cable cores with ferrules with collars. The maximum conductor cross-section is 1.5mm² / AWG16.

A shielded cable is required to ensure trouble-free operation even under extremely unfavourable ambient conditions with complex cabling conditions. Connect the cable shield on the Irinos box side to the FE bolt on the base plate. This is marked accordingly inside the base box.

Alternative 2 via Connector X11 33 (Power Supply Connector):



The connection is made via a 2.5mm jack plug, which is de facto standard for PELV-based desktop power supplies. GND is external, 24V is internal.

2.4.2 Ethernet

The Ethernet connection is established via a standard patch cable with RJ45 connector according to EIA/TIA 568B. Always use a shielded cable according to Cat-6 or better.

The Ethernet interface of the Irinos system has crossover detection. Therefore it does not matter whether a standard Ethernet cable or a crossover cable is used.

The data rate is 100 MBit/s.

Pin	Name	Description
1	TX+	Transmission Data+
2	TX-	Transmission Data-
3	RX+	Receive Data+
4		unused
5		unused
6	RX-	Receive Data-
7		unused
8		unused

2.5 Assembly

➔ It is essential that you read the safety instructions 19 before system assembly!

2.5.1 Checking the delivery

Checking the delivery

- When you receive the delivery, check the packaging for visible transport damage.
- If there is damage in transit, complain the delivery to the responsible carrier. Have the carrier confirm the transport damage immediately.
- Unpack the Irinos components at their destination.
- Keep the original packaging for future transport.
- Check the contents of the packaging and your specially ordered accessories for completeness and damage. If the contents of the packaging are

incomplete or damaged or do not correspond to your order, inform the supplier immediately.

- Also keep the supplied documents. They are part of the Irinos system.

Scope of delivery: Irinos-Box

- 1 Base box with connectors
- 1 Measuring- or I/O-module
- 5 Blind cable grommets to cover openings in the housing that are not required
- 1 cable grommet for power supply
- 1 Cable grommet for a commercially available network cable
- 2 plastic washers for mounting
- Safety instructions leaflet
- Warning note DHCP-Server

Scope of delivery: accessories

Item	**Scope of delivery**
Desktop power supply	• Power Supply • Mains cable
Ethernet cable	• Ethernet cable
EC-MHRM-1 DIN-Rail Adapter	• DIN-Rail Adapter • 2 matching Allen screws M4x6 for fixing the adapter to the Irinos box
EC-Link - connection cable	Connecting cable with matching cable grommets
qLink - connection cable	Connecting cable with matching cable grommets

2.5.2 Mounting location

As field devices, the Irinos boxes are suitable for use in closed enclosures, e.g. in switch cabinets, and for placement on or near the measuring device.

Especially in the case of larger systems, placement close to the measuring device is preferable. It offers two important advantages:

- The cables of the measuring probes, incremental encoders and other sensors can be very short. The quality of the analog signals at the measuring input is therefore particularly good. In addition, this simplifies cable routing far from possible sources of interference.
- Replacing a probe, e.g. in case of a defect, is easier.

For trouble-free operation of the Irinos system, take this into account:

Always place the Irinos boxes away from possible sources of interference, such as converters or motor cables.

Protection against dust and water is an important factor when choosing a suitable location. The Irinos boxes are designed in such a way that they are impermeable to the usual soiling. In order to comply with this, however, suitable connectors for the connected cables are required. However, most of the standard probes and incremental encoders on the market have connectors with a low degree of protection. These connectors would have to be replaced in order to achieve the degree of protection.

It is therefore advisable to place the Irinos boxes in such a way that little protection is sufficient or no protection at all is required.

The Irinos boxes have a low inherent heat development and are designed for industrial ambient temperatures. In addition, the integrated measuring electronics are particularly temperature-stable.

Nevertheless, choose a location with moderate ambient temperatures. The permissible temperature range is specified in the respective data sheet. In particular, avoid placing the device near heat sources such as heat sinks of other devices or heating elements.

2.5.3 Mounting

Irinos boxes can be mounted in 2 different ways:

1. By means of 2 M4 screws, e.g. on a sheet metal plate or on an aluminium profile (Item or similar).
2. Using the EC-MHRM-1 DIN rail adapter on a DIN rail (also known as H rail).

In any case, a very fast exchange of an Irinos box is possible.

Mounting with 2 screws M4

In the bottom of the base box 31 there are 2 holes for screws M4. These are closed with rubber plugs when delivered.

For the attachment are needed:

- 2 screws M4
- 2 plastic washers M4 (included in scope of delivery)
- Suitable screwdriver

The procedure is described in the following tutorial:

Direct Mounting

Mounting vai DIN rail adapter EC-MHRM-1

The DIN rail adapter is fastened to the bottom of the base box 31 with 2 screws. In the bottom there are 2 holes for screws M4. These are closed with rubber plugs in the delivery condition.

For the attachment are needed:

- DIN rail adapter EC-MHRM-1

- 2 screws M4x6 (delivered with DIN rail adapter)
- 2 plastic washers M4 (delivered with base box)
- Allen key, size 2.5

The procedure is described in the following tutorial:

DIN rail mounting

2.5.4 Wiring

Proper cabling is essential for trouble-free operation of the Irinos system. Please observe the following rules:

- Place all cables spatially separated from possible sources of interference, such as inverters or motor cables.
- Avoid unnecessarily long cables. Avoid "cable loops" in particular.
- All measurement cables and the EC-Link- and qLink-cables must be shielded.
- Use the locking mechanisms of the connectors.
- Avoid mechanical stress that can affect the cables.

When used in drag chains, ensure that suitable cables are used.

Provide all unused external connectors with a protective cap.

The PCB with the connectors in the base box 31 may move slightly. This is intended!

First mount the Irinos boxes. Then connect the cables in the following order:

1. If more than 1 Irinos Box is used: Connect EC-Link 44 - cables.
2. If extensions are used, which are connected via qLink: Connect qLink cables.
3. Connect Ethernet 45 cable to the Master Box.
4. Connect power 46 supply.

2.5.4.1 EC-Link Wiring

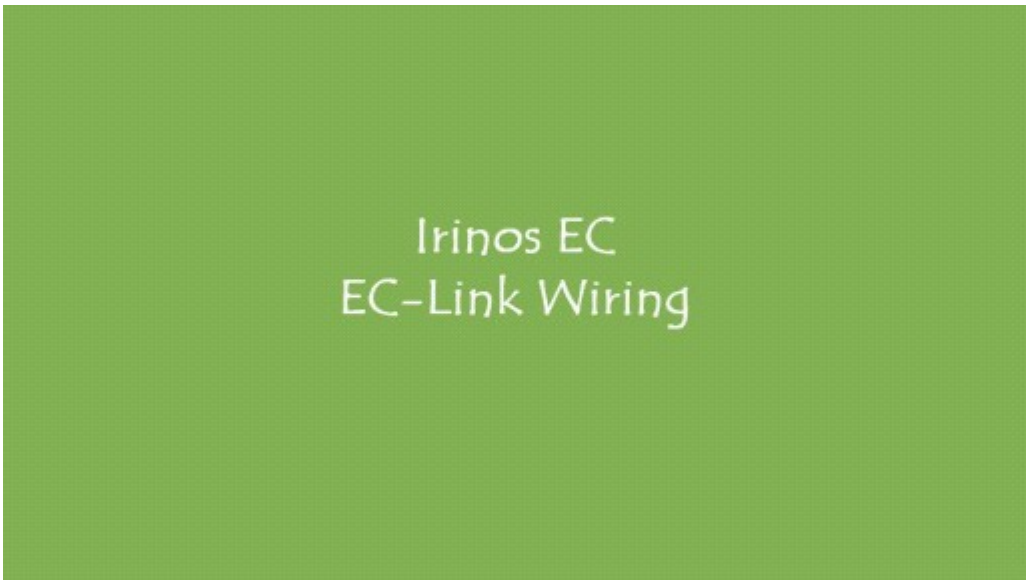
EC-Link - wiring is only required when 2 or more Irinos boxes are used.

Basics

- The EC-Link interface is a bus system in line topology. Each Irinos box has 2 EC-Link connectors 31.
- Two Irinos boxes are always connected to each other via a EC-Link line. One EC-Link connector remains unused at the first and the last Irinos Box.
- Manual termination is not required. The Irinos system automatically activates the required termination resistors, when it is switched on.
- The maximum permissible total length of the EC-Link> cabling is 10m.
- An Irinos system may consist of a maximum of 8 Irinos boxes (including master box).

Procedure

The procedure is described in the following tutorial:



EC-Link Wiring

2.5.4.2 Connecting Ethernet

In a system with several Irinos boxes, the Ethernet connection may only be established on one box. This is then referred to as the Master-Box 28.

The Ethernet interface of the Irinos system is a standard Ethernet interface, which is also used, for example, for IT networking. The Irinos system can therefore also be used with standard Ethernet switches.

Data communication between the Irinos system and the PC is fault-tolerant, so that if a packet is lost, a transmission repeat is automatically performed. However, this repetition always leads to a significant delay in the availability of the measured values. A transmission repetition should therefore be an exception in practice.

In order to minimize the number of transmission repetitions, **a direct connection between the Irinos system and the PC is strongly recommended**. To do this, connect the Ethernet interface of the Irinos system to a free Ethernet interface of the PC. Experience shows that transmission repetitions practically never occur.

Operation of the Irinos system via router, VPN connections, wireless connections (WLAN) or similar is not intended.

The Ethernet interface of the Irinos system has an automatic "cross-over detection". It does not matter whether a 1:1 Ethernet cable or a crossed Ethernet cable is used.

The DHCP server of the Irinos system is activated on delivery. This is the ideal setting for a direct connection to the PC.

Before the Irinos Box is operated in an IT network, the DHCP server must be deactivated via a direct connection. This is done via the Irinos tool. Please refer to the Irinos Tool 51 documentation for more information.

Procedure

The procedure for the Ethernet cabling is described in the following tutorial:

Connecting Ethernet

2.5.4.3 Connecting the power supply

It is imperative that you observe the safety instructions 19 with regard to the electrical voltage! It is particularly important to ensure that a power supply unit with functional extra-low voltage (PELV) is used. Observe the general notes on power supply 29.

A 24V DC power supply is required.

This can either be done via a 2.5mm DC plug connector (-> desktop power supply), or via terminals.

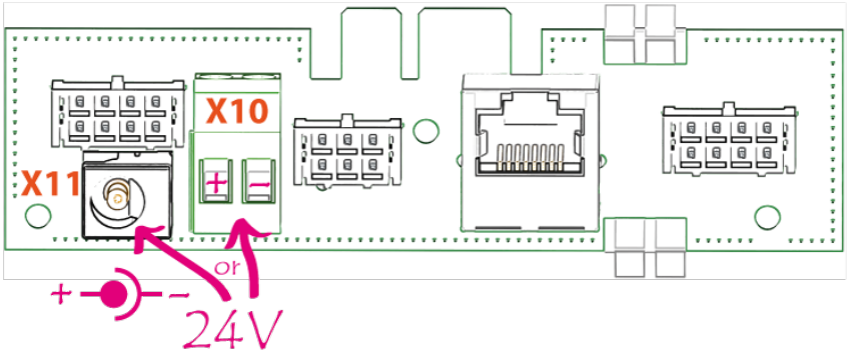
Procedure / Desktop Power Supply / Connector X11

The procedure for connecting the power supply via a desktop power supply unit is described in the following tutorial:

Connecting the power supply

Procedure / Terminals X10

If, for example, a switch cabinet power supply is used instead of a desktop power supply, the 24V connection can also be made via the green terminals (X10). The terminal assignment is shown in the sticker inside the Irinos box:



Always use ferrules with collars for the cable cores. The terminals are designed for conductor cross-sections from 0.25mm² to 1.5mm² (AWG 24 to AWG 14).

A shielded cable must be used and connected to the functional earth bolt in order to guarantee optimum EMC immunity in accordance with the requirements of the CE marking even under extremely unfavourable environmental conditions. The bolt is available at the bottom of the housing (M3 thread).

2.5.5 Insert Measuring Modules

Attention
Damage due to improper installation When installing the measuring module, make sure that it does not jam. Make sure that all cables are connected in such a way that there is sufficient space for the measuring module. Make sure in particular that the plug connector of the measuring module can be plugged onto the connector board.

Procedure

The procedure for installing the measuring module (front unit) is described in the following tutorial:

Inserting the Measurement Module

2.6 Setup & First Steps

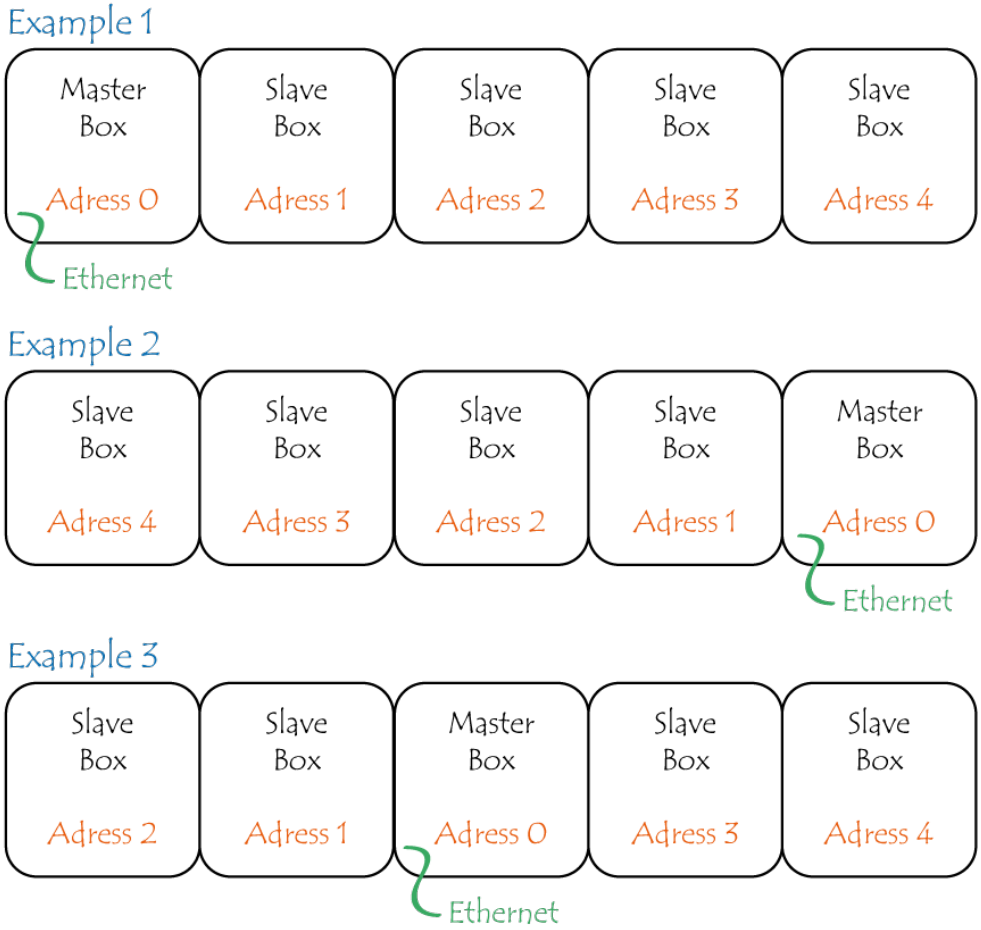
The Irinos system is designed so that no configuration of the system or components is required. The only exception to this is the network settings (IP address), which can be reconfigured via the ITool 51.

Once the mounting and wiring 39 is complete, the Irinos system can be turned on immediately by supplying power to the power supply.

2.6.1 Box addressing

All Irinos boxes are automatically addressed after switching on. The master box always has the address 0. The slave boxes are numbered consecutively in the order in which they are connected. The box sequence is also decisive for the initial numbering of the measurement inputs or the digital inputs/outputs.

The following figure shows some examples of box addressing:



In order not to get a shift of the addressing in case of a system extension, it is recommended to use the first or the last box as master box (examples 1 and 2).

The **duration** of the addressing process depends on the number of connected boxes. Typically, it only takes a few seconds.

The first and last boxes are also determined during box addressing. Bus **termination** is automatically activated for both boxes.

During setup, it should be checked after switching on whether all boxes are correctly connected and thus correctly addressed. This can be done, for example, via the webserver 52.

2.6.2 Network configuration

An Irinos box has an integrated DHCP server. It is enabled by factory default. The IP address of the Irinos system is 192.168.3.99, the subnet mask 255.255.255.0. If the Ethernet interface of the PC is configured as a "DHCP client", it is automatically assigned an IP address from the address range 192.168.3.100 to 192.168.3.254 when the connection is established. Then no network configuration is necessary. Nevertheless it is recommended to assign a fixed network address on the PC side (e.g. 192.168.3.98).

The DHCP server of the Irinos system is enabled on delivery. This is the ideal setting for a direct connection to the PC.

Before the Irinos Box is operated in an IT network, the DHCP server must be deactivated via a direct connection. This is done via the Irinos Tool 51. Please refer to the Irinos tool documentation for more information.

If the use of the DHCP function is not desired, there are 2 possibilities:

a) The DHCP server on the master box remains enabled. However, the PC has a fixed IP configuration. To do this, use the following network settings on the PC, for example:
IP address: 192.168.3.98
Subnet mask: 255.255.255.0

b) The DHCP server on the master box is deactivated via the Irinos tool. The IP address of the Irinos Box can be freely assigned. The PC receives a fixed IP configuration.
Please refer to the documentation of the Irinos Tool 51 for the exact procedure.

The easiest way to test whether the network connection works is via the webserver 52 of the Irinos system. Open a web browser and enter the IP address of the Irinos system in the address bar. If the network connection is working, the web page with the measured value display will now appear. Under "First aid: Network connection 73" the typical procedure for connection problems is described.

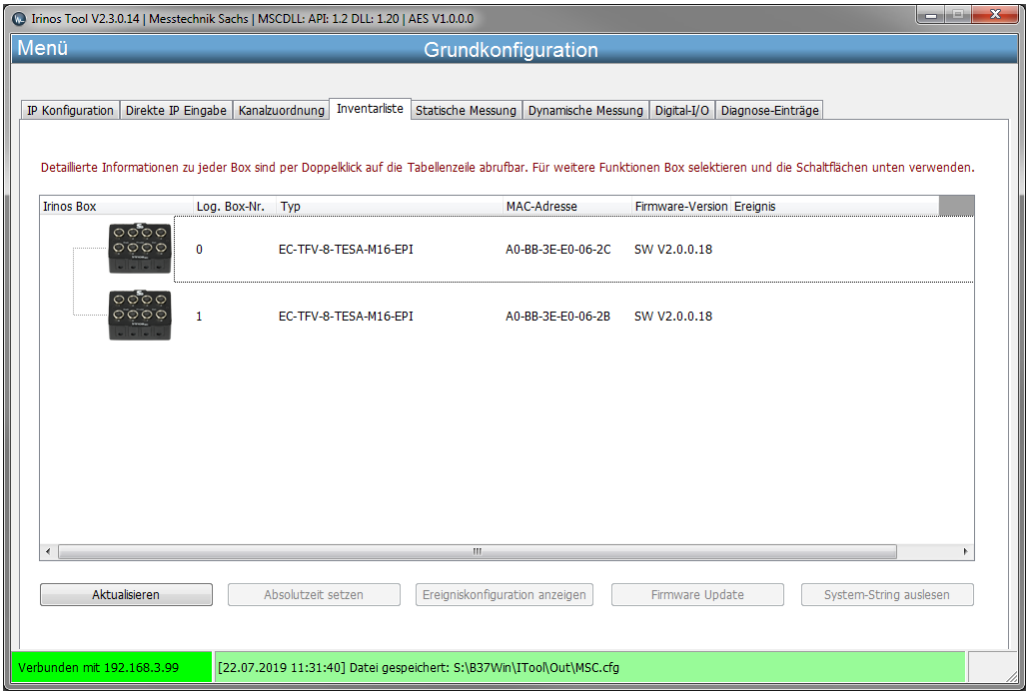
2.6.3 Irinos-Tool

The Irinos tool is a configuration and test tool for the Irinos system. Its features include:

- Change the network configuration of the Irinos system.
- Overview of the available measurement channels.
- Diagnosis of the incremental encoder signals (1Vpp).
- Display of static measurement values.
- Overview of the Irinos boxes connected to an Irinos system.
- Perform firmware updates.
- Readout and storage of the diagnostic memory contents.

For more information, refer to the Irinos tool documentation available separately.

It is recommended to keep the Irinos-Tool available on the measuring computer connected to the Irinos system, so that it can be used quickly as an aid in case of diagnosis. There are no license fees for the Irinos tool as long as it is used exclusively in connection with the Irinos system.



2.6.4 Web-Server

The Irinos system has an integrated web server that serves as a setup and diagnostic tool. The web server is accessed from a web browser such as Internet Explorer, Firefox, Chrome or Opera. Enter the IP address of the Irinos system in the address line of the web browser (default 192.168.3.99).

The presentation of the web pages was successfully tested with the web browsers InternetExplorer 11, Firefox and Chrome. Due to the different interpretation of standards, a flawless function cannot be guaranteed with all browsers and browser versions.

There are 3 different websites available:

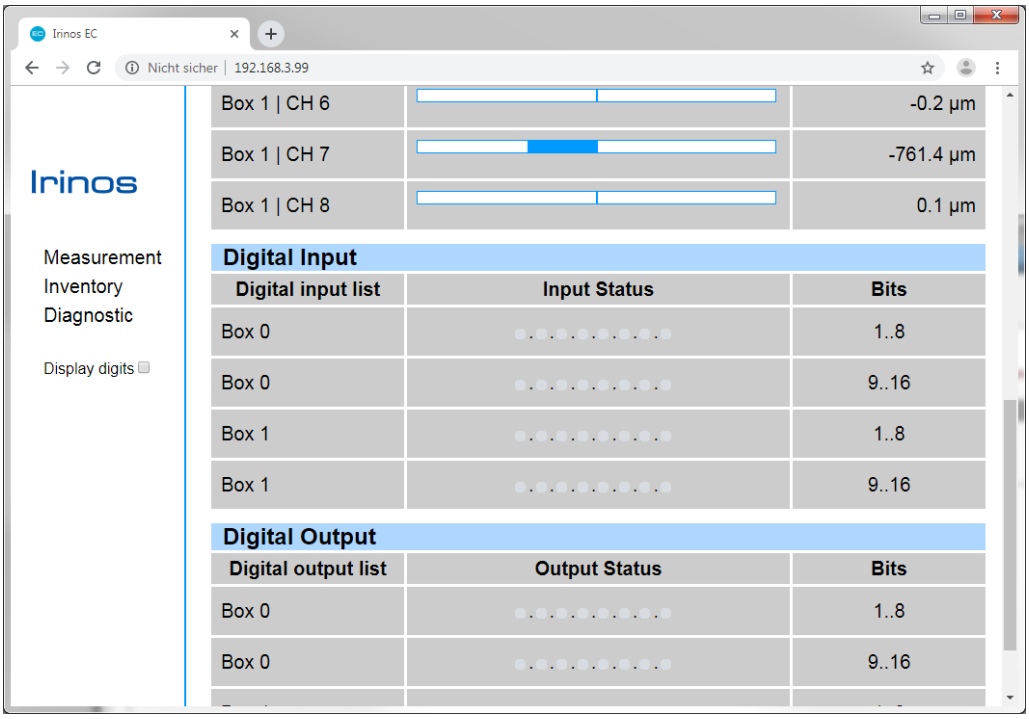
Measurement (current measurement values)

On the "Measurement" web page, the current measured values of the measurement inputs and the current status of the digital inputs/outputs are displayed live (update rate approx. 4 Hz).

This allows:

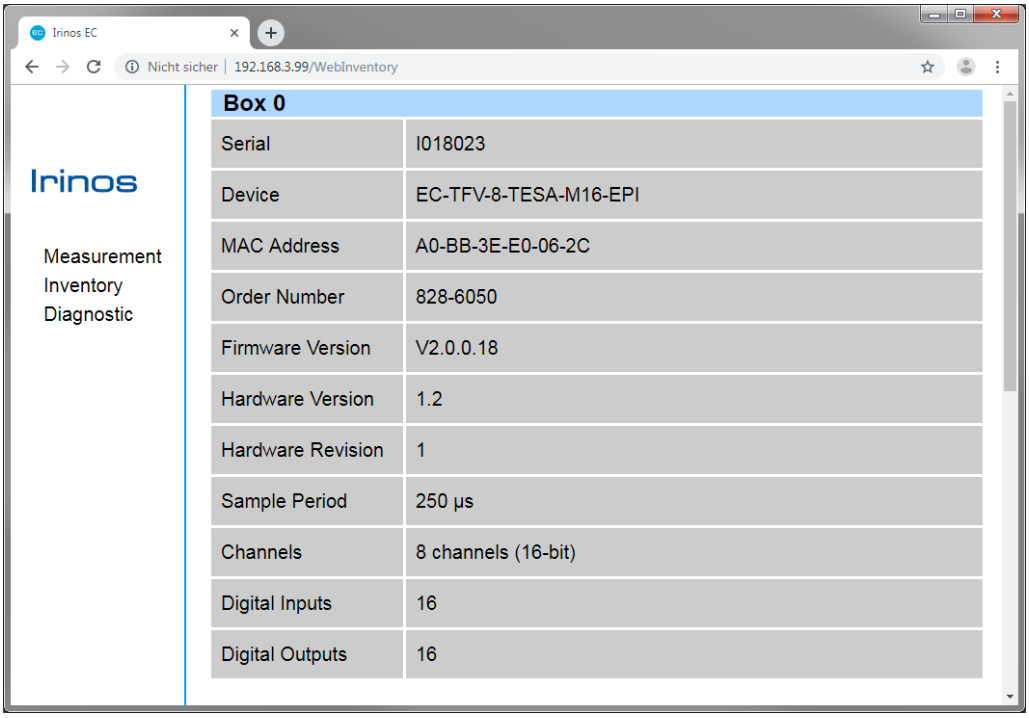
- The measurement probes can be adjusted and testes without the availability of the measurement software.
- The measurement values shown on the webserver can be compared to those, which are shown by the measurement software.

Please note for inductive probes: The displayed unit is valid for standard probes. For probes with longer travel (e.g. ±5mm) and different sensitivity, a manual conversion is required.



Inventory (Box-Overview)

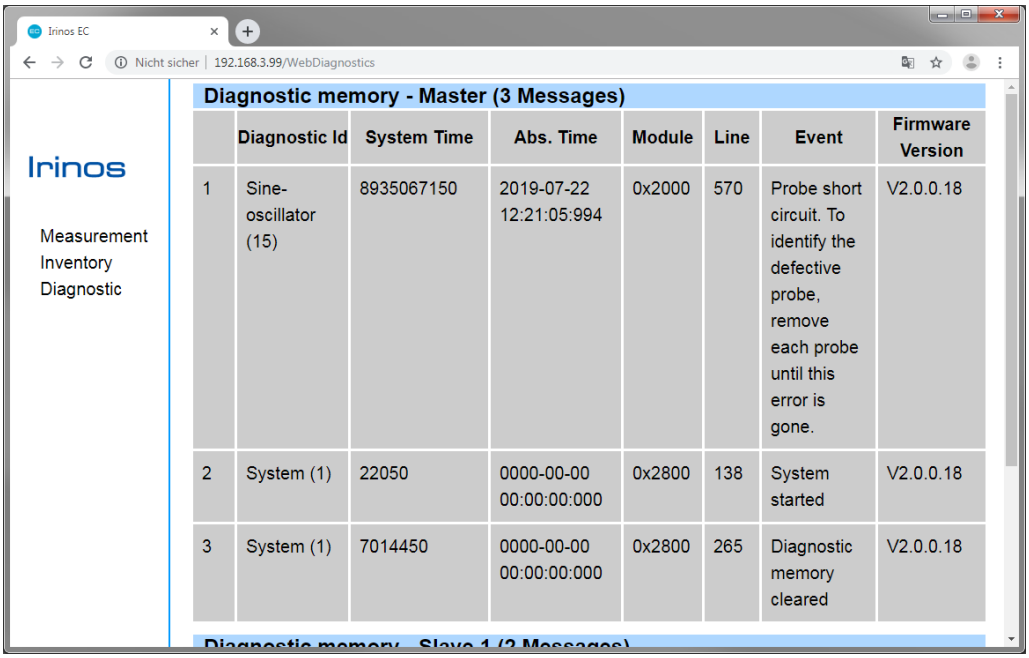
The website "Inventory" displays an overview of all Irinos boxes available in the Irinos system with the most important box information:



Information	Example	Description
Serial	I018023	Serial number of the Irinos-Box
Device	EC-TFV-8-TESA-M16-EPI	Description of the Irinos-Box
MAC Address	A0-BB-3E-E0-06-2C	Unique MAC address of the Irinos-Box
Order Number	828-6050	Order number. It always begins with 8.
Firmware Version	V2.0.0.18	Firmware version
Hardware Version	V1.2	Hardware version
Hardware Revision	1	Hardware compatibility code for firmware update
Sample Period	250 µs	Internal sampling period in µs
Channels	8 channels (16-bit)	Number of measurement channels and their internal data type
Digital Inputs	16	Number of digital input bits
Digital Outputs	16	Number of digital output bits

Diagnostic (Diagnostic memory)

The Diagnostic web page displays the contents of the diagnostic memory of the individual Irinos boxes:



	Diagnostic Id	System Time	Abs. Time	Module	Line	Event	Firmware Version
1	Sine-oscillator (15)	8935067150	2019-07-22 12:21:05:994	0x2000	570	Probe short circuit. To identify the defective probe, remove each probe until this error is gone.	V2.0.0.18
2	System (1)	22050	0000-00-00 00:00:00:000	0x2800	138	System started	V2.0.0.18
3	System (1)	7014450	0000-00-00 00:00:00:000	0x2800	265	Diagnostic memory cleared	V2.0.0.18

Column	Example	Explanation
Diagnostic-Id	Sine-oscillator (15)	Type of diagnostic event 63.
System Time	8935067150	Link-Zeit 27 in µs since power-on of the Irinos-Systems. (This internal time is identical for all Irinos Boxes)
Abs. Time	2019-07-22 12:21:05:994	Date and time when the event occurred. Year-Month-Day Hour:Minute:Second:Millisecond
Module	0x2000	Additional Information for support requests.
Line	570	
Event	Probe short circuit. To identify the defective probe, remove each probe until this error is gone.	Help text for the event.
Firmware-Version	V2.0.0.18	Firmware version with which the event occurred.

2.7 Software Interface

The Irinos EC - System offers 3 different possibilities for integration into the measuring software, whereby the NmxDLL is the preferred one:

	NmxDLL 58	ASCII / Telnet 59	MscDLL 62

Type	Windows based DLL interface (Win 7 / 8 / 10)	ASCII based readout of measurement values via Telnet of UDP	Windows based DLL interface (Win XP / Vista / 7 / 8 / 10)
Static Measurement	Yes	Yes	Yes
Realtime measurement	Yes	No	No
Exchange of digital I/O data	Yes	Yes	Yes
Readout of diagnostic information	Yes	No	Yes
Parametrization of measurement channels	Yes	No	Yes
Supported by other measurement systems	○ Irinos IR, using Firmware-Version 2 or greater	○ Irinos IR, using Firmware-Version 2 or greater	○ Irinos IR ○ Older systems
Note	Preferred interface for standard applications	Quick implementation for simple applications	Compatibility with existing integration of the MscDLL into measurement software (**Attention: dynamic measurement not supported!**).

			If you start the DLL implementation from scratch, use the NmxDLL.

The NmxDLL and the MscDLL are described in detail in a separate reference manual. Consult the respective reference manual for detailed information.

Common to all software interfaces is that the software implementation is independent of the number of measurement channels.

For example, for software implementation it does not matter whether a system has 4 or 64 measurement channels. Only one connection to the Irinos system needs to be established and managed at a time. Only the number of available measuring channels depends on the number of channels.

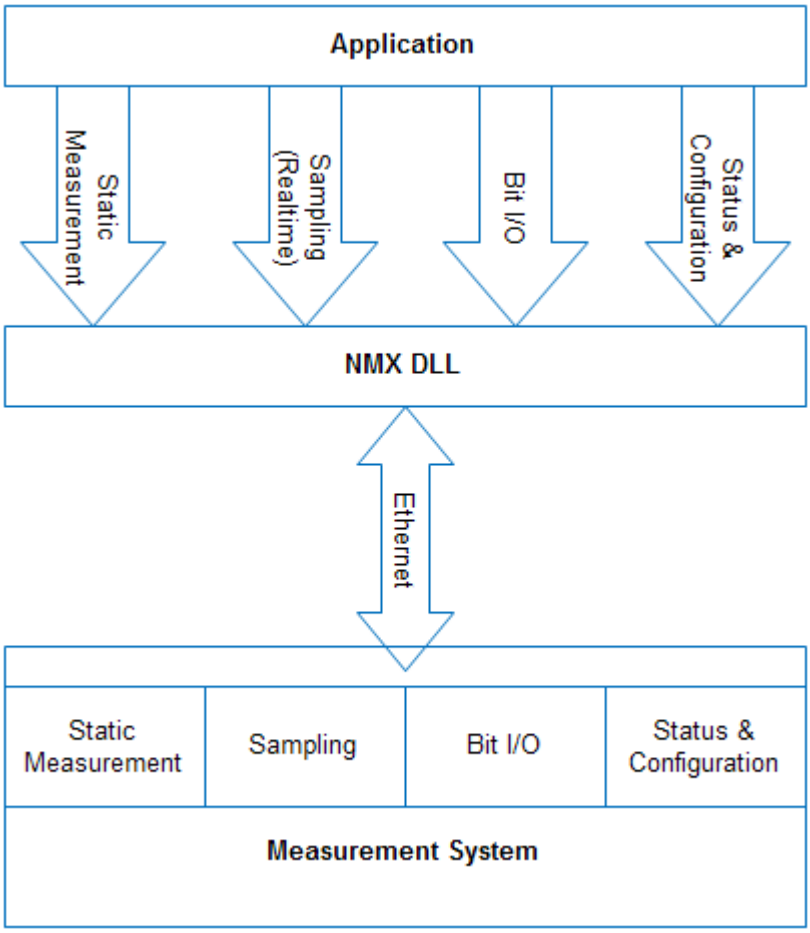
2.7.1 NmxDLL Quick Overview

This is only a short overview!

Detailed information can be found in the reference manual of the NmxDLL.

The NmxDLL is the link between the measuring software and the Irinos system. It offers the following possibilities:

- Reading the measurement values from the Irinos system (static and/or in real time).
- Read status information (e.g. current diagnostic event 63).
- Reading the state of the digital inputs / Setting the state of the digital outputs
- Parameterization of the Irinos system



For small applications with a small range of functions, the use of a few function calls of the NmxDLL is sufficient.

Others also enable the simple implementation of complex applications.

Various sample programs are available for integration into measurement software.

2.7.2 ASCII- / Telnet-Interface

The Irinos EC offers an ASCII-based text interface for the simplest applications. This is comparable to the widely used connection of measurement hardware via RS232.

The ASCII interface can be accessed either via TCP-based Telnet or via UDP. While the ASCII protocol itself is identical, the two options differ as follows:

	Telnet / TCP	UDP

Port-Number	TCP 22515	UDP 22515
Packet loss	Automatic repetition of data transmission at protocol level (TCP) in the event of packet loss.	In case of packet loss at UDP level, a manual repetition of the data exchange is required.
Latency	Theoretically very long latency time possible. With a 1:1 Ethernet connection, however, this is negligible in practice.	Very short latency times possible.
Max. Text-Length	16000 Characters/Bytes	1450 Characters/Bytes if a 1:1-Ethernet connections is used

The data is always exchanged in the question -> answer - procedure, i.e. the measuring software sends a request to the Irinos system and it sends an answer. Each request / answer consists of a kind of "header", the actual data and an end identifier.

The header can be used to distinguish between different commands by means of an identifier. The following data are optional depending on the question/answer. The end identifier is terminated via "CarriageReturn (CR)" and "LineFeed (LF)" and is represented in the following as `{CRLF}`. CR has the ASCII code 0x0D and LF the ASCII code 0x0A.

Note: The Irinos system recognizes only CR or only LF as end, but always sends both back in the response.

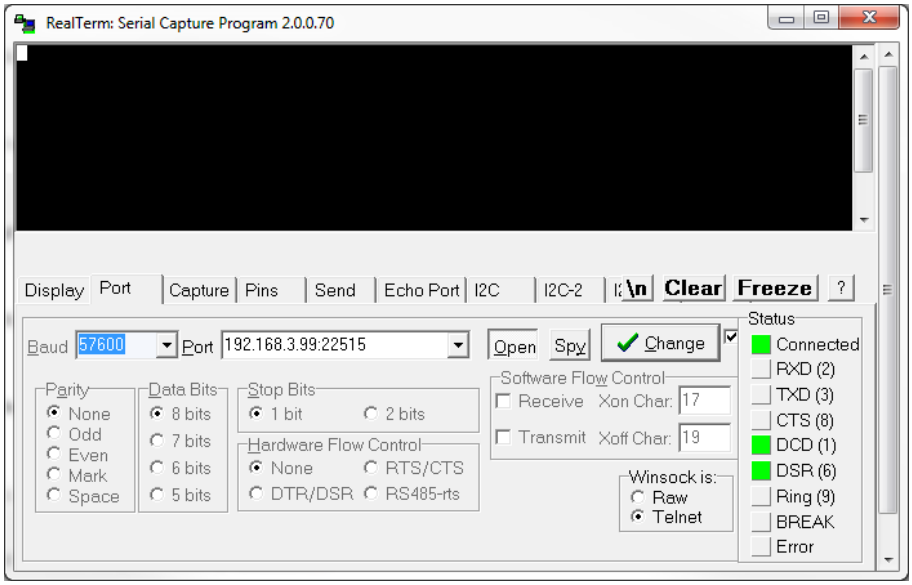
The protocol is explained using the following examples:

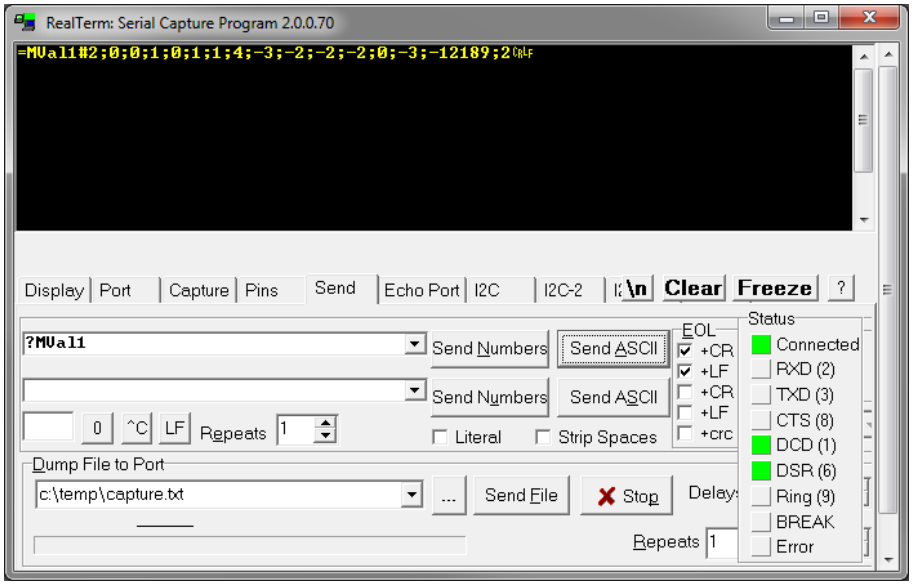
Command	**Request**	**Response**	**Notes**
Dummy command	`?Nop1{CRLF}`	`=Nop1#OK{CRLF}`	Helpful for testing

			communication as well as "Heartbeat".
Read measurement values	?MVal1{CRLF}	=MVal1#0;-3;-12	Example for 4 measurement channels. With more measurement channels, the response is correspondingly longer.
Read digital inputs	?DIn1{CRLF}	=DIn1#0;1;0;1;1	Example for 16 digital inputs. Values can be 0 or 1.
Set digital outputs	?DOutSet1#1;1;0	=DOutSet1#1;1;0	In the response, the actual state of all digital outputs is returned. This can also be more than in the request, if only a part of the outputs was set here. Values can be 0 or 1.
Read measurement values and digital inputs together	?MValDIn1{CRLF}	=MValDIn1#0;-3;	Combination of "MVal1" and "DIn1". First come the measured values, then the digital inputs.

Tip:

For test purposes, data can also be exchanged via Telnet using numerous terminal programs such as "RealTerm".





2.7.3 MscDLL Quick Overview

This is only a short overview!

Detailed information can be found in the reference manual of the MscDLL.

The MscDLL has long been established on the market and is supported by numerous providers of measurement software.

For compatibility with these software packages, the Irinos EC also supports this interface in a limited form: everything is supported except dynamic measurement. **This means that the Irinos EC in combination with this interface is only suitable for static measurements.**

2.8 Troubleshooting & First Aid

The Irinos system has numerous internal monitoring mechanisms that can support setup and servicing.

Depending on the configuration, a detected diagnostic event 63 is stored in the diagnostic memory 72 and/or output via the software interface 56 to the PC software.

The factory configuration of the diagnostic events is designed in such a way that no reconfiguration is required for typical applications.

2.8.1 Diagnostic events

Each Irinos box has a central event handler. As soon as a special event occurs in the firmware, it is reported to the event handler. Depending on its configuration, the event is

- signalled to the use/application,
- stored in the diagnostic memory 72.

No event should occur during normal operation.

In order to distinguish between the events, there are different event types which are distinguished by the event number.

With the appropriate configuration, the occurrence of an event is indicated via the status LED 31 of the Irinos Box. The event can also be read out via the software interface 56.

The Irinos system has a very detailed error handling strategy to ensure reliable operation. Most events triggered by an error are hypothetical in nature. These are therefore not documented. Contact support if such an event occurs.

The following is a list of those events that are relevant to practice:

Event 1: „System“	
Description	Common system event
Type	Information
Cause	○ System has been started ○ Diagnostic memory 72 has been cleared
Forwarding to user/application	No, cannot be activated
Entry in diagnostic memory	Yes, cannot be changed

Event 4: „MscDll communication error“	
Description	A problem was detected in the communication between the Irinos system and the MscDll.
Type	Error
Cause	○ An invalid opcode has been used (Event text in diagnostic memory: „Invalid opcode in RX packet“) -> Use only valid opcodes ○ The send an receive buffer size if too small (Event text in diagnostic memory: „Too much TX data“) -> Use the port number and buffers sizes given in the reference manual in the file Msc.cfg
Forwarding to user/application	Yes, cannot be de-activated

Entry in diagnostic memory	Yes, can be changed

Event 12: „EC-Link module detection error“	
Description	A problem with the recognition or termination of the slave box(s) occurred when starting the Irinos system.
Type	Error
Cause	○ Invalid EC-Link-wiring or broken EC-Link-cable -> Check EC-Link wiring 44 ○ Multiple master boxes in one Irinos system -> Only one master-box 28 is allowed ○ Irinos-Box broken -> Replace the Irinos-Box
Forwarding to user/application	For Irinos systems with multiple slave boxes: Test the system first with 1, then with 2, then with 3, and so on slave boxes to find out where the problem occurs.
Entry in diagnostic memory	Yes, can be de-activated
Eintragung in Diagnose-Speicher	Yes, can be changed

Event 13: „EC-Link communication error“

Description	Communication via the EC-Link interface is faulty.
Type	Error
Cause	○ Invalid EC-Link-wiring or broken EC-Link-cable -> Check EC-Link wiring 44 ○ Improper power supply (e.g. short voltage drops) -> Use an appropriate power supply 29.
Forwarding to user/application	The link communication has an integrated data check as well as a packet repetition in case of an error. If the packet retry fails multiple times, this event is triggered.
Entry in diagnostic memory	Yes, can be de-activated
Eintragung in Diagnose-Speicher	Yes, can be changed

Event 15: „Sine-oscillator“	
Description	The sine oscillator for the inductive probes 35 has been overloaded (short circuit).
Type	Error
Cause	○ Defect of a probe -> Replace the probe ○ Measuring probe incorrectly connected, e.g. when using an

	extension cable -> Check wiring / pin assignment
Notes	It is checked cyclically whether the oscillator short-circuit is still present. As soon as it is no longer present, the event is automatically deleted. ➔To find the cause, remove the probes one after the other. Wait 10s after removing a probe. As soon as the event is no longer active (indicated by the red error LED 35), the defective probe has been removed.
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

Event 24: „Inc. encoder power error“	
Description	The power supply of one or more incremental encoder channels was switched off due to overload / short circuit.
Type	Error
Cause	○ Defect of an incremental encoder or an incremental encoder cable -> Replace incremental encoder ○ Incorrect connection of an incremental encoder. -> Check wiring / pin assignment ○ Incremental encoder power consumption too high

	-> Observe permissible connected loads.
Notes	○ In the event of an overload / short circuit at a single incremental encoder input, only the input itself is deactivated. All other inputs remain functional. As soon as the overload or short-circuit has been removed, the event is automatically deleted. ○ In the event of a total overload, the power supply for the incremental encoder inputs is permanently switched off. The incremental encoder inputs can only be used again after the Irinos system has been restarted.
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

Event 25: „Inc. encoder application error“	
Description	The input signals of an incremental encoder input were / are outside the permissible range.
Type	Error
Cause	○ Incremental encoder connector has been removed during operation. ○ Incremental encoder connector not fitted properly (loose contact). -> Use connector screws for proper fixation.

	○ Input frequency 78 of the incremental encoder too high -> Reduce speed of incremental encoder / avoid mechanical shock ○ Incremental encoder cable too long -> Use short cable ○ Improper wiring of the incremental encoder -> Check pin assignment ○ Encoder signals out of specification -> Check signals with Irinos Tool 51
Notes	○ The incremental encoder signals can be checked with the Irinos Tool 51 (only 1Vpp). ○ See application notes for incremental encoders 77. ○ The incremental input channel can be reset via the software interface 56 (see reference manual for detailed information). Using NmxDLL: NMX_ChannelSetParameter Using MscDLL: Opcode opcSP
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

Event 27: „Firmware update error“	
Description	An error occurred during the execution of the firmware update.

Type	Error
Cause	○ Invalid firmware file -> Use valid firmware file ○ Transmission error -> Repeat the firmware update
Notes	After a failed firmware update, the "old" firmware version is still active.
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

Event 28: „Firmware update successful“	
Description	A firmware update was successfully performed.
Type	Information
Forwarding to user/application	No, cannot be activated
Entry in diagnostic memory	Yes, cannot be changed

Event 36: „Digital I/O error“	
Description	The output driver for the digital outputs was overloaded (thermal overload).
Type	Error

Cause	○ Too high continuous load of the digital outputs. -> Adapt maximum output load to specification.
Notes	As soon as the output driver has cooled down, the outputs are automatically enabled again and the event is deleted.
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

Event 43: "NMX sampling configuration invalid"	
Description	The configuration of the real-time sampling is invalid.
Type	Error
Cause	○ Invalid sampling period
Notes	
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

Event 44: "NMX sampling error"	
Description	Error during execution of real-time sampling.
Type	Error

Cause	○ Internal buffer overflow / measurement values have not been readout in-time. ○ Communication error between multiple boxes (e.g. cable broken).
Notes	
Forwarding to user/application	Yes, can be de-activated
Entry in diagnostic memory	Yes, can be changed

2.8.2 Diagnostic Memory

Each Irinos box has an integrated, non-volatile diagnostic memory in which the events that have occurred are stored (provided that saving is activated for the respective event). It can be read out via the webserver 52 as well as via the Irinos-Tool 51.

The diagnostic memory is therefore an important tool for tracing occurring problems and limiting their causes. This is especially true if an error occurs sporadically.

At least 32 entries per Irinos box can be stored in the diagnostic memory. As soon as it is full, the oldest entries are automatically deleted, creating space for new entries.

In addition to the actual diagnosis event, a diagnosis entry contains some additional information that can help you find the cause. These include the system time (EC-Link time 27) and the absolute time.

The EC-Link time corresponds to the time in µs since the Irinos system was started. It is uniform on all Irinos boxes of a system.

The absolute time contains the date and time of the diagnostic entry. Since the Irinos system does not contain a real-time clock, the absolute time is always 0 when the system is started. It should then be written from the PC. This is done via the software interface 56. Each subsequent diagnostic entry is then provided with the absolute time.

After switching on, the diagnostic entry "System (1)" with the help text "System started" is stored in each Irinos box. This makes it possible to see whether an event occurred before or after the Irinos system was last switched on.

2.8.3 First Aid "Network Connection"

➔ This chapter provides help for typical network connection problems. More information is available in the users manual of the Irinos-Tool 51.

Usually one or more of the following reasons lead to connection problems:

- Network cabling is invalid.
- The network configuration of the PC differs from the network configuration of the Irinos-System.
- The communication settings for the software interface 56 are wrong.

Checking network cabling

a) Check, if the network interface of the Irinos-System is connected to the network interface of the PC.
A working electrical connection is signalled by the "Ethernet LED 31". This LED must either be turned on or flashing.
Proceed with the next step, if this is the case.

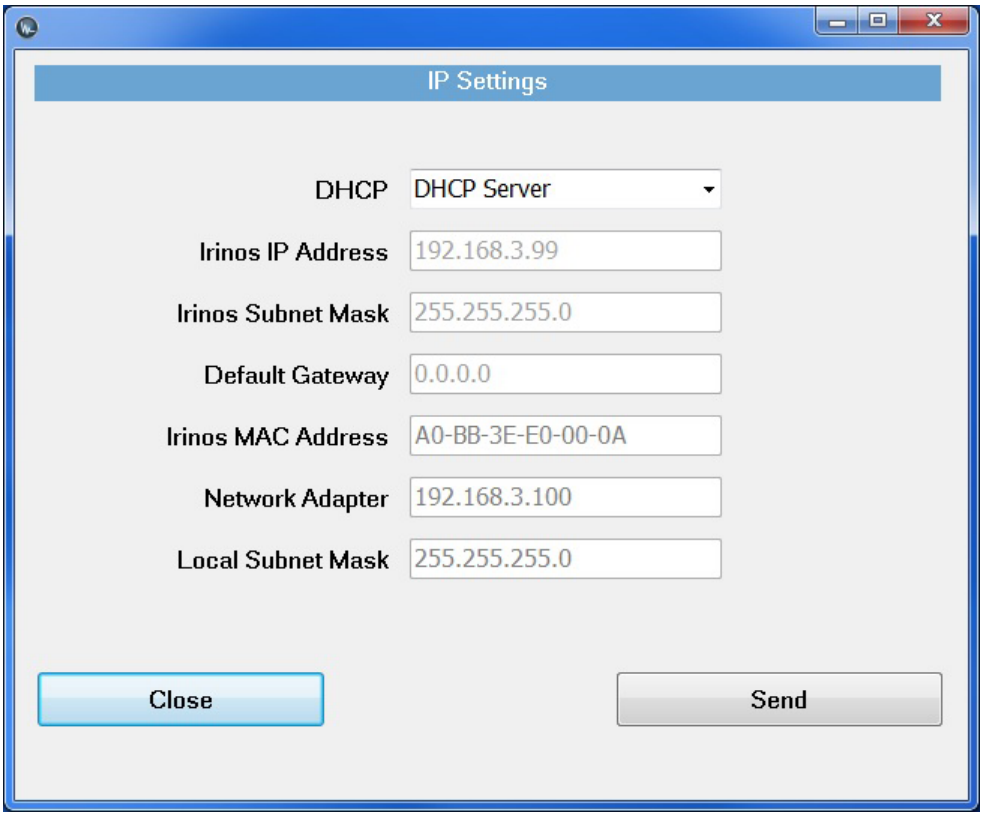
Verifying the network configuration

b) Start the Irinos-Tool. It searches for all available Irinos-Systems in the network and lists them in a table. Your Irinos-System should be listed in this table. You can use the MAC address to verify this (the MAC address is printed on the type-plate of the Master Irinos-Box).
The IP settings will also be displayed in this table.

c) Try to connect to the Irinos-System via the Irinos-Tool.

Proceed with the next step, if the connection cannot be established.
Otherwise proceed with step f).

d) Open the network configuration of the Irinos-System by douple-clicking in the table row. The following window opens:

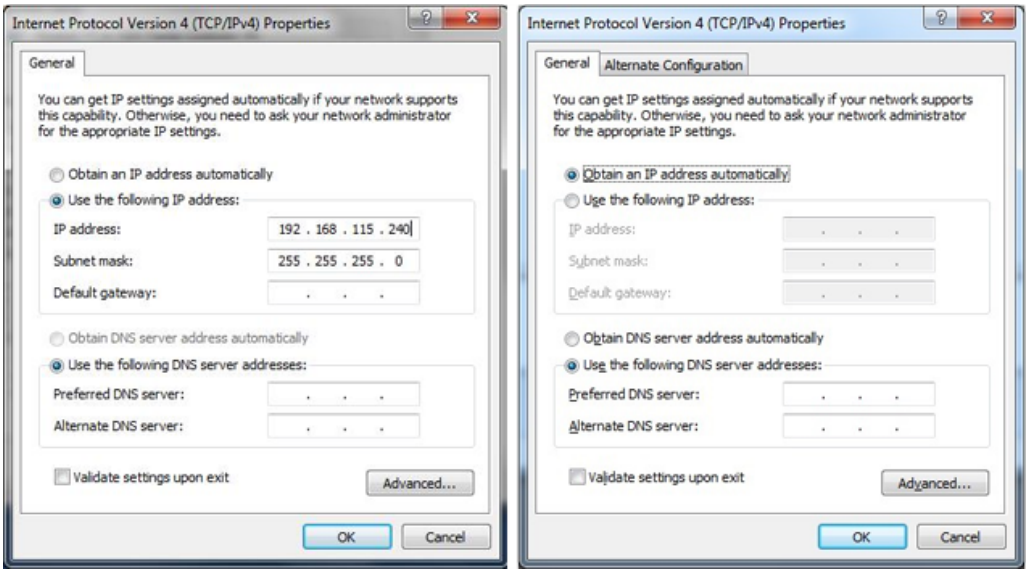


e) Open the Windows configuration settings of the network adapter, which is connected to the Irinos-System. Open the settings for "internet protocol version 4 (TCP/IPv4)".

In case the DHCP-Server of the Irinos-System is enabled, the option "Optain an IP address automatically" must be enabled (see left figure below).

In case the Irinos-System has a fixed IP address, the PC must also have a fixed IP address. Both IP addresses must be within the same subnet. In most cases the subnet mask 255.255.255.0 is used. In this case the first three elements of the IP addresses must be the same. If the Irinos-System for example has the IP address 192.168.178.1, the PC must have an IP address of the range 192.168.178.2 to 192.168.178.254 (see right figure below).

If necessary, change the Windows IP settings.



Windows IP configuration

Adopting the communication settings for the measurement application

f) Check the IP address stored in the measurement software. This must match the IP address of the Irinos system (factory setting 192.168.3.99).

g) Start the application. The connection should now be established.

2.8.4 Maintenance, Cleaning & Disposal

Maintenance

The Irinos system is designed for maintenance-free continuous operation.

It is recommended to check the fastening of the external plugs at regular intervals, e.g. monthly. This will prevent wear-out of the connectors. In addition, the possible occurrence of error sources is prevented as a preventive measure.

Cleaning

	**Caution**
	Unexpected / unintended reaction while cleaning the Irinos-System If the Irinos-System is in operation while cleaning, this may result in unintended actions. This may lead to personal injury or damage at machinery. Always turn off the Irinos-System before cleaning.

For intensive use, carry out the cleaning operations listed in the following table.

If the environment is particularly dirty, more frequent cleaning may be necessary. In return, the cleaning intervals can be extended for occasional use or in a clean environment.

Interval	Cleaning procedure
3 Month	Cleaning of the connector surface from oil and dust. Use a paper towel moistened with detergent water for cleaning. Do not turn on the Irinos system until the connectors are completely dry.
Monthly	Clean the cabinet with a paper towel moistened with detergent water. Use a scratch-free towel.

Disposal

Dispose both the Irinos system and the accessories via the electronic scrap recycling system of your respective country. Do not dispose it with household waste.

2.9 Application Notes

Geben Sie hier den Text ein.

2.9.1 Incremental Encoders

Incremental encoders are reliable and precise measuring devices if potential problems are encountered during the project planning phase of the application. Therefore, please observe the following application notes:

- Referencing 77
- Input frequency 78
- Interpolation of 1Vpp signals 78

2.9.1.1 Referencing for absolute measurement

Incremental encoders are not absolute measuring devices. In order to obtain absolute measured values, referencing is always necessary after switching on and after a signal error. The Irinos-Box EC-INC offers the following possibilities for referencing:

- Referencing via index

  The counter value is set to 0, if the index signal is passed.

- Referencing via software:

  The counter value can be set by software any time. It is possible to set the value 0 as well as to any other value (in the valid value range).

Both actions can be performed in conjunction with the NmxDLL 58 and MscDll 62:

- NmxDLL: NMX_ChannelSetParameter
- MscDLL: Opcode opcSP (0x35)

Please note that the Irinos Box EC-INC can only offer the technical possibility for referencing. The procedure for referencing the measured value depends on the respective measuring procedure. This must therefore already be considered in the planning phase. In particular, consider the procedure after the occurrence of an incremental encoder error.

2.9.1.2 Input frequency

The input frequency of the incremental signals (TTL / RS422) or the signal period (1 Vpp) is limited. Details can be found in the specifications for the respective Irinos-Box.

In most measurement applications the theoretical input frequency is far below the limits. However, in reality it can be exceeded quickly through jerky movements. Examples are:

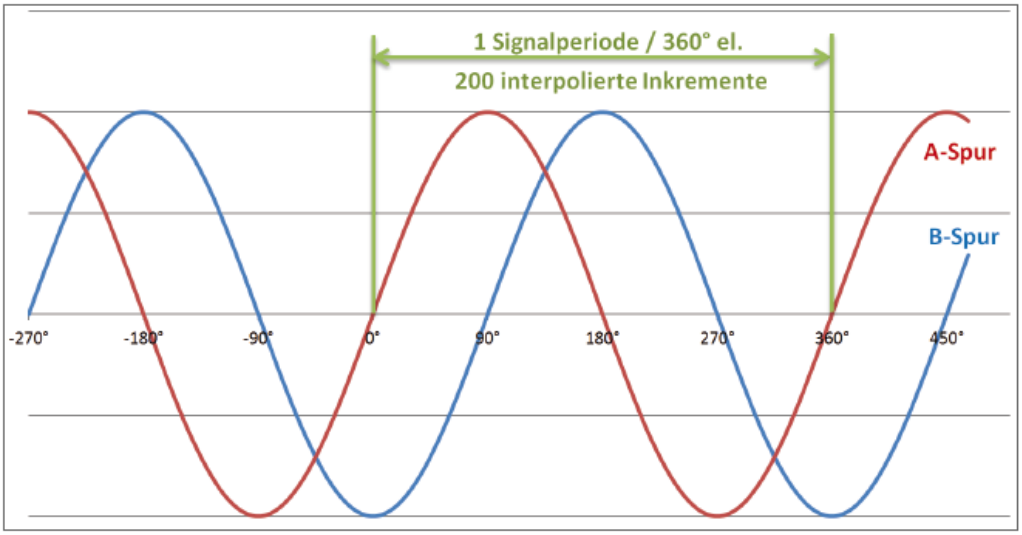
- “Cutting loose” at the beginning of a movement (crossing static friction)
- Mechanical stroke
- Jerky movement due to mechanical tensions

It is recommended to take this into account during the planning phase. If a jerky movement cannot be avoided, this must be considered in the measurement process (e.g. by referencing while moving).

2.9.1.3 Interpolation (only 1Vpp)

An incremental encoder with 1 Vpp - Interface provides two sine waves, each as a differential output signal. The phase shift between these is 90°. One signal period (i.e. 360°) relates to one incremental division of the encoder. The Irinos-Box EC-INC separates a division into 200 incremental steps via analogue interpolation. Thereby the usable resolution of the encoder increases by 200.

Example: An incremental encoder has a resolution of 1.800 divisions/revolution. Via the interpolation, this results in 1.800 divisions/revolution * 200 increments/division = 360.000 increments/revolution.



1Vpp signal period

The accuracy and reliability of the interpolations depends on the quality of the differential sine signals. An ideal signal has the following characteristics:

- The differential voltage is 1 Vpp.
- The signal offset is 0, i.e. at 0° the signal always provides the same value.
- The phase shift between both sine signals is exactly 90°.

In reality, such a signal is rarely available. Because of this, the Irinos-Box EC-INC has an integrated gain- and offset control (patent applies). It corrects these deviations within the allowed value range (see limiting values in the specification section of the datasheet).

Below or above the threshold values, a reliable interpolation is not possible. An integrated signal examination detects such errors. The error status can be read by software in parallel to the measurement. In case of an error, the counter channel should be reset. The referencing procedure should be redone.

Signal quality

The signal quality depends on various factors. Important ones are:

- Speed of the incremental encoder

  The higher the speed of the incremental encoder, the smaller the actual differential voltage. Some incremental encoders have a very good signal quality while standing still or at low speed. As soon as they are moved / turned, the signal quality decreases rapidly.

- Mechanical stability of the incremental encoder and the mechanics

  An incremental encoder or a measurement device, which does not move smoothly, leads to variations in the measurement signal.

- Adjustment of the incremental encoder

  Some incremental encoders (e.g. glass scales) need to be adjusted. An improper adjustment can lead to an insufficient sensor signal (especially for dynamic movements).

- Cable length and cable quality

  The longer the cable, the worse the signal quality.

  The more connectors are used, the worse the signal quality.

  A cable with insufficient shielding or wrong line impedance deteriorates the signal quality.

Often the reason for a signal problem is a combination of these factors.

Suggestions

- Please observe the cut-off frequency of the incremental encoder. It can be found in the encoder datasheet.

  Attention: The cut-off frequency depends on the cable length.

- Check the signal quality during system setup. The signal levels should have enough margin to the limits.
  The Irinos-Tool provides a live-view of the signal levels.

- Make sure that no strong jerk can disturb the signal quality.

- Be prepared to enhance the measurement procedure by the functionalities "reset encoder error" and "restart referencing".

- Use short cables with sufficient shielding (this also applies to the connectors). Avoid extension cables. The Irinos-System allows placing the Irinos-Box next to the incremental encoder.

- Keep distance between the incremental encoder cable and possible sources of noise, like for example frequency changers or motor cables.

2.9.2 Power consumption

The power consumption of an Irinos system depends on the number of Irinos boxes connected and the number of consumers connected. Connected loads are for example measuring probes and sensors.

An overview for estimating the total power consumption can be found in the following table. Please note that all values are guide values. The actual power consumption may differ. Please refer to the respective data sheets for exact details on power consumption.

Irinos-Box	Typical power consumption without probes, sensors, etc.	Recommended calculation value with connected loads
EC-TFV 35	< 2 W	ca. 2 W

It is recommended to check the actual power requirement during system setup by means of a measurement.

2.9.3 Storing data in the non-volatile memory

The non-volatile memory has a limited amount of write operations. By the design of the Irinos-System, this limit is typically not reached. The following table lists the maximum number of write operations:

System function	Maximum number of write operations	Note
Diagnostic memory	4,8 Millions	
Measurement channel configuration	200.000	Executed via ○ NmxDLL: NMX_ChannelSetConfig ○ MscDLL: Opcode opcWCC.
Network configuration	200.000	Changing the IP settings.
Firmware update	100.000	

2.10 Specifications & Dimensions

Detailed technical data can be found in the data sheet of the respective Irinos box.

2.10.1 Common specifications

Measurement value recording	
Static / continuous measurement	Update rate ca. 30 Hz for smooth online display
Realtime sampling	Up to 4.000 samples/s on all channels simultaneously, i.e. 1 channel -> Total sample count: 4.000 measurement values/s 17 channels -> Total sample count 68.000 measurement values/s 32 channels -> Total sample count 128.000 measurement values/s See also chapter synchronization and speed 27.
Synchronization	Simultaneous acquisition of all measurement channels. Synchronous measurement value acquisition, also via cascaded Irinos boxes.

Cascading / EC-Link-Interface	
Maximum number of Irinos-Boxes	8
Maximum number of measurement channels	Depending on the number of channels of the Irinos boxes used. For example with EC-TFV maximum 64 measurement channels.
Maximum cable length EC-Link	10 m (Total length of the EC-Link - wiring) Under certain circumstances more on request.
Termination	Automatically
Box-Adressing	Automatically

Case for EC-TFV 35, EC-INC	
	Design housing aluminium black anodized, back plate stainless steel, front plate black powder-coated
Dimensions	120 x 85 x 49 mm (H x W x D)
Protection	Similar to IP54 Connections for probes and digital inputs/outputs according to the respective connector specification.
Befestigung Standard 41	Via two screws M4
Befestigung Zubehör 41	Adapter for DIN rail mounting

Irinos Tool Users Manual

3 Irinos Tool Users Manual

3.1 Introduction

3.1.1 Imprint

Title	Irinos-Tool users manual
Manufacturer	Messtechnik Sachs GmbH Siechenfeldstraße 30/1 D-73614 Schorndorf Germany Phone +49 7181 99960-0 post@messtechnik-sachs.de
For use with	Measurement modules Irinos IR
Copyright note	© 2019-2020 Messtechnik Sachs GmbH
Trademarks	All product names used in this manual are trademarks of their respective owners.
Material-No.	785-1019
Change not	Subject to change without notice.
Release date	05.06.2020

3.1.2 Revision history

Version	**Datum**	**Changes**
A		

3.1.3 Terms of use for software & documentation

I. Protection rights and scope of use

Messtechnik Sachs provides operating instructions, manuals, documentation, and software programs - all collectively referred to as "LICENSED OBJECT" below - either on portable data storage devices (e.g. diskettes, CD ROMs, DVDs, etc.), in written (printed) form or in electronic form, for a fee and/or free of charge. The LICENSED OBJECT is subject to proprietary safeguarding provisions among other regulations. Messtechnik Sachs or third parties have protection rights for this LICENSED OBJECT. In so far as third parties have whole or partial right of access to this LICENSED OBJECT, Messtechnik Sachs has the appropriate rights of use. Messtechnik Sachs permits the user the use of the LICENSED OBJECT under the following conditions:

1.1) Scope of use for electronic documentation

a) With the acquisition/purchase or relinquishment of a LICENSED OBJECT, you as the user acquire a simple, non-transferable right of use with regard to the respective LICENSED OBJECT. This right of use authorises the user to use the LICENSED OBJECT for the user's own, exclusively company-internal purposes on any number of machines within the user's business premises. This right of use includes exclusively the right to save the LICENSED OBJECT on the central processors (machines) used at the location.
b) Irrespective of the form in which operating instructions and/or documentation are provided, the user may furthermore print out any number of copies on a printer at the user's location, providing this printout is printed with or kept in a safe place together with these complete terms and conditions of use and other user instructions.
c) With the exception of the Messtechnik Sachs logo, the user has the right to use pictures and texts from the operating instructions/documentation for creating the user's own machine and system documentation. The use of the Messtechnik Sachs logo requires written consent from Messtechnik Sachs. The user is responsible for ensuring that the pictures and texts used match the machine/system or the product.
d) Further uses are permitted within the following framework:

Copying exclusively for use within the framework of machine and system documentation from electronic documents of all documented supplier components. Demonstrating to third parties exclusively under guarantee that no data material is stored wholly or partly in other networks or other data storage devices or can be reproduced there.
Passing on printouts to third parties not covered by the regulation in item 3, as well as any processing or other use are not permitted.

1.2) Scope of use for software products

For any type of Messtechnik Sachs software including the associated documentation, the customer shall receive a non-exclusive, non-transferable and time-unlimited right of use on a certain hardware product or on a hardware product to be determined in individual cases. Messtechnik Sachs shall remain the owner of the copyright as well as of any other industrial

property rights. The customer may make copies for back-up purposes only. Any copyright notes may not be removed.

2. Copyright note

Every LICENSED OBJECT contains a copyright note. In any duplication permitted under these provisions, the corresponding copyright note of the original document concerned must be included:

Example: © 2016, Messtechnik Sachs GmbH,
D-73614 Schorndorf

3. Transferring the authorisation of use

The user can transfer the authorisation of use re. the respective LICENSED OBJECT as per these provisions in the scope and with the limitations of the conditions in accordance with items 1 and 2 completely to a third party. The third party must be made explicitly aware of these terms and conditions of use.

II. Exporting the LICENSED OBJECT

When exporting the LICENSED OBJECT or parts thereof, the user must observe the export regulations of the exporting country and those of the acquiring country.

III. Warranty

1. Messtechnik Sachs products are being further developed with regard to hardware and software. If the LICENSED OBJECT, in whatever form, is not supplied with the product, i.e. is not supplied on a data storage device as a delivery unit with the relevant product, Messtechnik Sachs does not guarantee that the electronic documentation corresponds to every hardware and software status of the product. In this case, the printed documentation from Messtechnik Sachs accompanying the product is alone decisive for ensuring that the hardware and software status of the product matches that of the electronic documentation.
2. The information contained in an item of electronic documentation can be amended by Messtechnik Sachs without prior notice and does not commit Messtechnik Sachs in any way.
3. Messtechnik Sachs guarantees that the software program it created agrees with the change description and program specification but not that the functions included in the software run entirely without interruptions and errors or that the functions included in the software can run or meet the requirements in all combinations selected by and in all conditions of use designated by the acquirer.

IV. Liability/limitations on liability

1. Messtechnik Sachs provides LICENSED OBJECTS to allow the user to use - in conformity with the contract - Messtechnik Sachs products which require software for proper operation, or

to assist the user in creating the user's machine and system documentation. In the case of electronic documentation which in the form of data storage devices does not accompany a product, i.e. which is not supplied together with that product, Messtechnik Sachs does not guarantee that the electronic documentation separately available/supplied matches the product actually used by the user.
The latter applies particularly to extracts of the documents for the user's own documentation. The guarantee and liability for separately available/supplied portable data storage devices, i.e. with the exception of electronic documentation provided on the Internet/Intranet, are limited exclusively to proper duplication of the software, whereby Messtechnik Sachs guarantees that in each case the relevant portable data storage device or software contains the latest status of the documentation. In respect of the electronic documentation on the Internet/Intranet, it is not guaranteed that this has the same version status as the last printed edition.
2. Furthermore, Messtechnik Sachs cannot be held liable for the lack of economic success or for damage or claims by third parties resulting from use of the LICENSED OBJECTS by the user, with the exception of claims arising from infringement of protection rights of third parties concerning the use of the LICENSED OBJECTS.
3. The limitations on liability as per paragraphs 1 and 2 do not apply if, in cases of intent or wanton negligence or lack of warranted quality, liability is absolutely necessary. In such a case, the liability of Messtechnik Sachs is limited to the damage recognisable by Messtechnik Sachs when the specific circumstances are made known.

V. Safety guidelines/documentation
Guarantee and liability claims in conformity with the regulations mentioned above (items III and IV) can only be made if the user has observed the safety guidelines of the documentation in conjunction with use of the machine and its safety guidelines or the terms and conditions of use of the software. The user is responsible for ensuring that the electronic documentation, which is not supplied with the product, matches the product actually used by the user.

3.1.4 Preface

3.1.4.1 Purpose

	Warning
	Carefully read this complete users manual and all related documentation before setup and use of the Irinos-System. This applies especially to the safety instructions. Misuse may lead do death, serious injury, injury or damage of man, equipment or machine.

3.1.4.2 Scope of this manual

This manual describes the setup- and diagnostic-software "Irinos-Tool", which is used together with the industrial measurement system "Irinos".

3.1.4.3 Intended use

Irinos is a flexible High-Speed measurement system for the industrial production measurement technology.

The measurement device is not appropriate for use in medical fields or in explosive areas, for aerospace and for home- or office use. Other fields of application, which are not mentioned but similar, are also excluded from use.

In safety critical areas, the safety in operation must be ensured by external equipment (e.g. external emergency stop).

Please note:

	Warning
	Products from Messtechnik Sachs GmbH must only be used for applications, which are mentioned in the dataheet or in the related documentation. If third party products are used, these must be recommended or permitted by Messtechnik Sachs GmbH. Proper and safe operation of the products require appropriate transportation, storage, mounting, usage and maintenance. Environmental conditions stated in the specification must be observed as well as notes in the related documentation.

3.1.4.4 Required knowledge

For using the Irinos-Tool general knowledge in using Windows-based software is required.

For the Irinos-System applies:

For the mechanical integration and mounting, solid knowledge and skills in mechanics and machinery are required.

For the electrical installation and the setup, solid knowledge and skills in electrics and electrical safety are required.

For the setup of the measurement application, profund knowledge in industrial measurement technology is required as well as PC skills.

3.1.4.5 Further documentation

Please note the short booklet, which is delivered with each Irinos module. This applies especially to the safety warnings, which are mentioned in it. The specifications of the Irinos-Boxes can be found in the respective datasheet.

Before using the Irinos-System, please read the users manual carefully.

3.1.4.6 Firmware & Software version

This users manual is related to the Irinos Firmware version V1 and the Irinos-Tool V2.0.

Screenshots may show an older version. These are only updates, if the user interface has changed significantly.

3.1.5 About this help

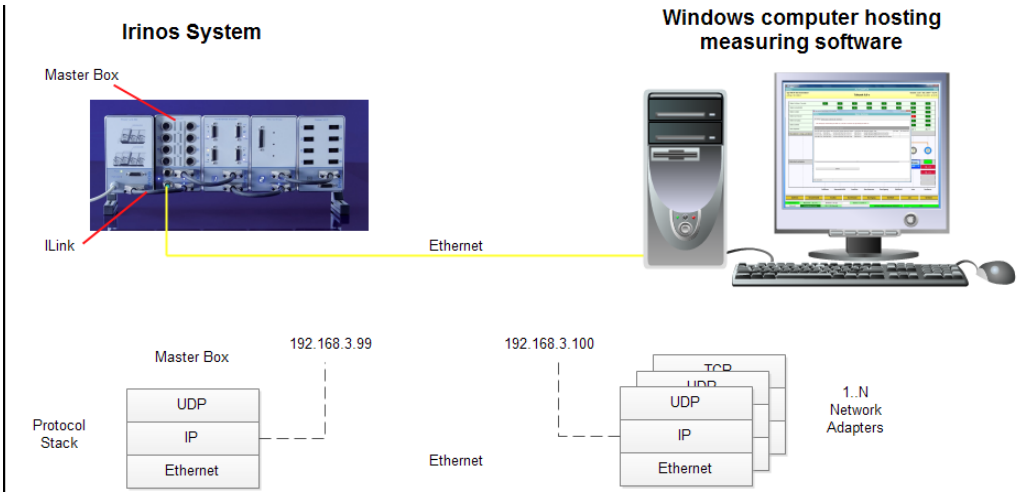
This document describes how the connection between the Irinos system and a host computer is set up. Furthermore, a detailed description of the Irinos-Tool is given. The Irinos-Tool provides a set of functions which support

- connection establishment and verification
- inventory visualization
- configuration setting
- functional tests and
- firmware updates.

Note: It is assumed that the reader is familiar with IP networking principles such as IP address handling, the concept of subnets and finally DHCP.

3.1.6 System overview

As shown in the following figure, the physical connection between the Irinos-System and the PC is made via an Ethernet cable. Typically it has an M12 connector for the Irinos Master-Box and a RJ45 connector for the PC.



System overview

The communication itself is based on the commonly used UDP and IP protocols.

Basically, network devices need to be configured before a communication link can be established. In particular, IP settings are mandatory on both sides of the communication link. Both, the Irinos Master-Box and the host computer need to be equipped at least with an IP address, and a subnet mask. In some cases a default gateway is configured as well.

If set manually, IP settings on the Windows host are done in the Windows Control Panel. For the Irinos system the Irinos Tool is provided, which enables the user to execute basic network settings. Additionally it contains a broad set of utilities, which support putting the Irinos system into operation. A detailed description of the Irinos Tool is given in chapter 3.2.

Manual IP setting, however, is often cumbersome and may not necessarily lead to established communication links. A better choice is DCHP (Dynamic Host Configuration Protocol), a network protocol which handles this task more reliable and user-friendly. DCHP basically knows two roles: A DHCP server, who is responsible for handling a pool of IP addresses, and a DHCP client, who queries the server for an IP address.

The Irinos Master-Box provides such a DHCP server. The DCHP Server is active by factory defaults, so the task of providing proper IP settings can fully be delegated to the DHCP function.

As a prerequisite, the network setting on the Windows computer need to be set accordingly.

3.2 Quick start guide

3.2.1 Requirements

This quick start guide is intended for the standard use case, where the

- Irinos Box is on factory defaults, i.e. the Irinos DHCP Server is on.
- Standard IP settings, as set by the Irinos DHCP Server
  - Irinos Box 192.168.3.99,
  - PC Network adapter 192.168.3.100

are compatible with the customer network policy.

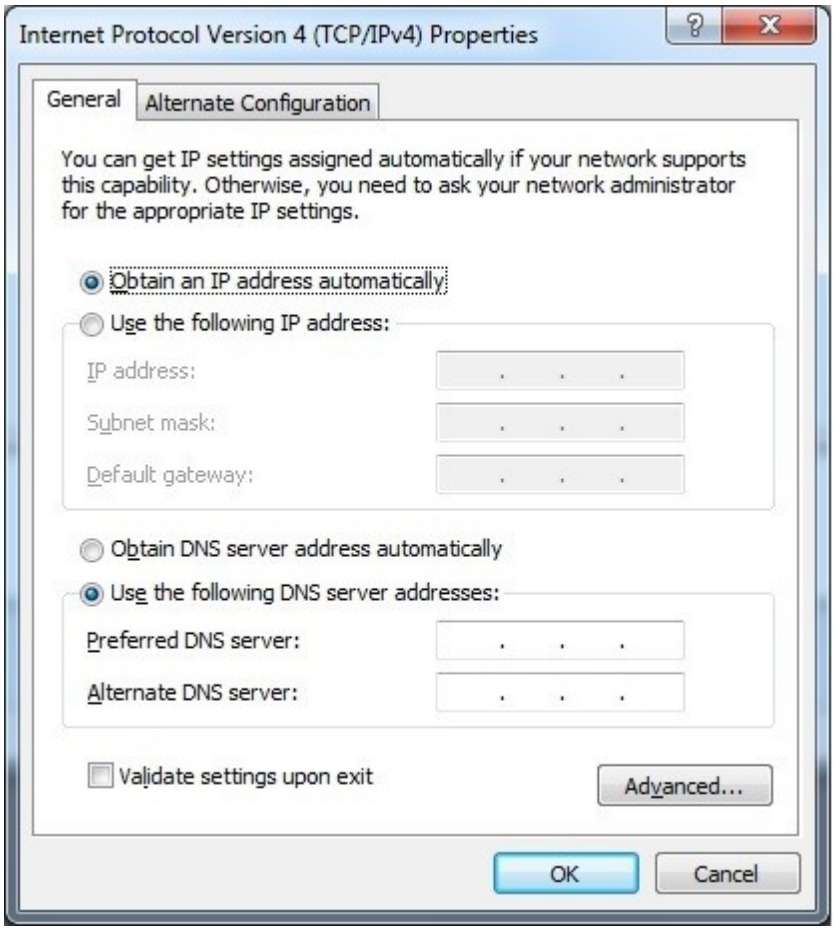
- Irinos box is linked to the host computer.
- User is familiar with the Windows operating system and the IP/DHCP settings at the local host computer.

3.2.2 PC network settings

As a prerequisite on the host computer, the network adapter settings need to be set to DHCP client mode. Therefore it is advisable to check these settings:

1. Open Windows Control Panel and follow Network and Internet -> View Network Status and Tasks –> Change Adapter Settings
2. Select adapter and right-click “Properties”
3. Select “Internet Protocol Version 4 (TCP/IP)” and press the “Properties” Button

The following window opens:



PC IP V4 configuration (DHCP active)

General Settings:

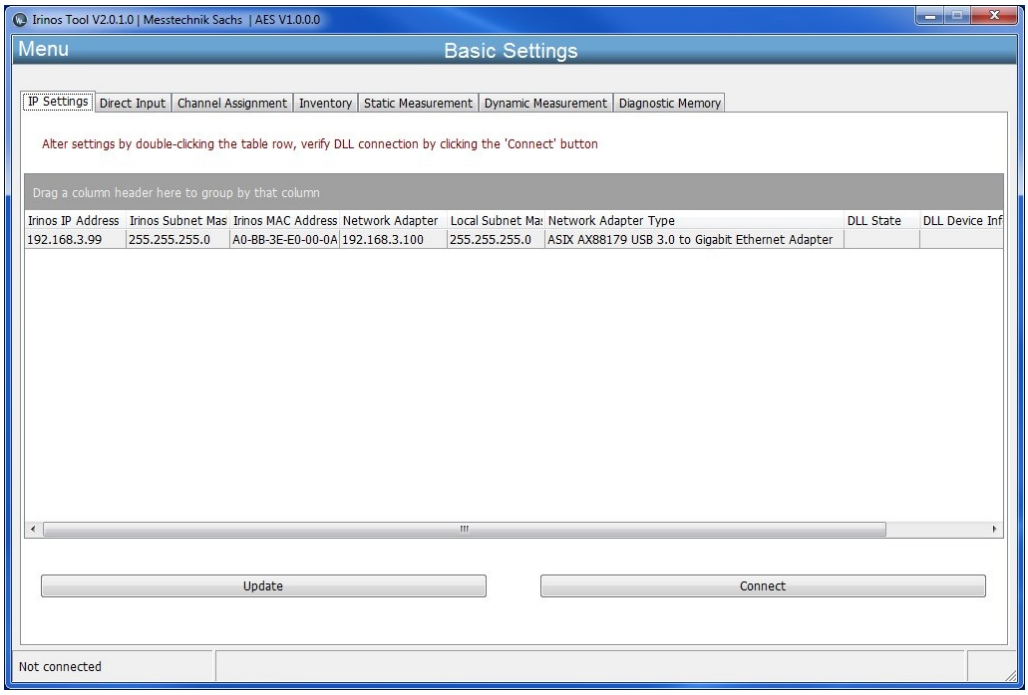
4. Select “Obtain an IP address automatically”

If the network adapter settings are configured as in the screenshot above, no further action is required.

3.2.3 Irinos configuration and connection check

A main benefit of the built-in DCHP server is the simplified connection setup. As the IP settings of the Master-Box are preset, no user-triggered settings are required. The Irinos-System can be connected without applying any settings.

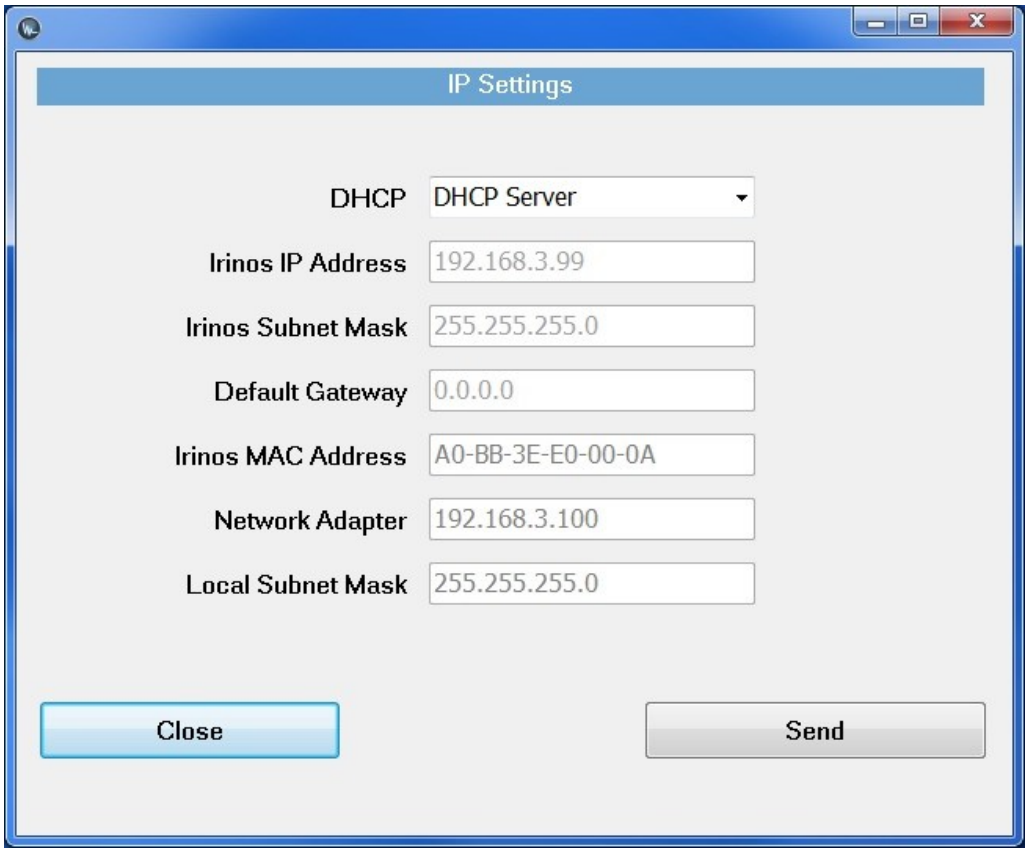
Once the Irinos-System is supplied with power and connected to the host computer, the Irinos Tool should be able to identify the box in the network. Every box found will be presented as one row in the main window of the Irinos Tool:



Start screen of the Irinos-Tool

A double-click on the table row will open the IP Settings window.

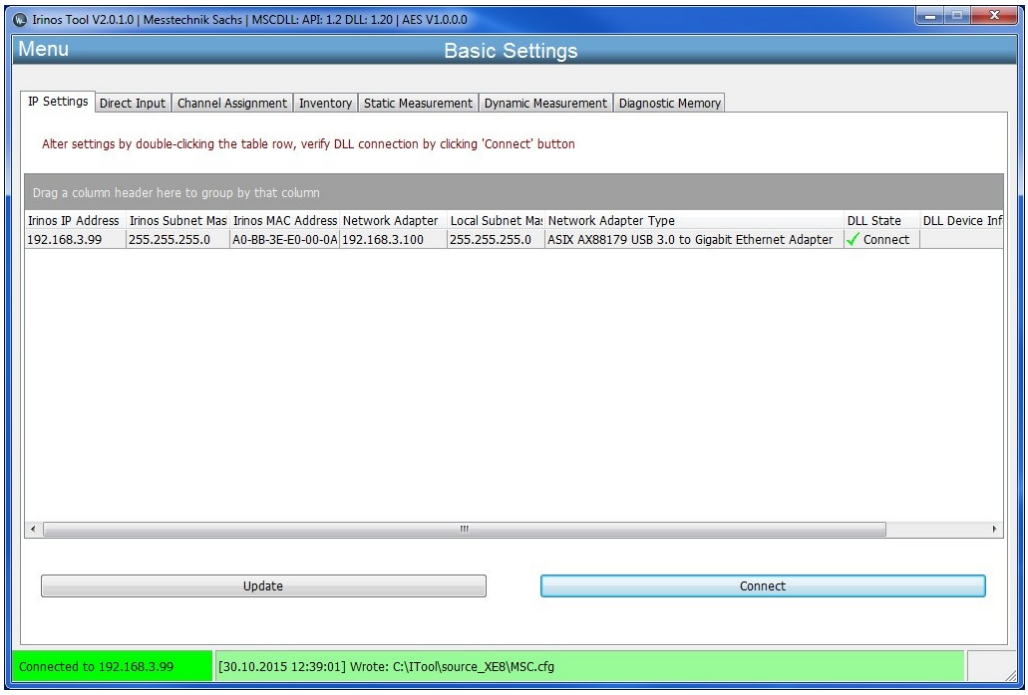
It can be used to review the IP settings of the Master-Box, or to modify the settings. As long as the DHCP server is active (Setting 'DHCP' is on selection 'DHCP Server') no modification is necessary.



IP configuration window

As a final test, a verification of the interface access by means of the provided MscDll should be executed. This is triggered by selecting the appropriate table row and pressing the 'Connect Button'.

A successful connection check is indicated as shown in following figure:



Connection Test via the MscDll

The connection check result is displayed in the column 'DLL State', the same applies for the returned DLL Device Info.

Along with the verification of the DLL connection, the configuration file MSC.cfg is created automatically. This file is necessary to specify the IP address for the MscDll.dll.

The file location of the file MSC.cfg is displayed in the bottom status bar. From this location it can be copied to the measurement software application.

3.3 PC network connection

3.3.1 Ethernet connection

A standard LAN is used to interconnect the Irinos Master-Box and the PC. State of the art network adapters support Auto-MDI(X), a feature to detect the type of cabling used (direct, cross-over) and thereby can work with both types.

However, some very old legacy network adapters, which do not support Auto-MDI(X), might require a cross-over cable to connect the Irinos box.

3.3.2 Network interfaces

Basic network settings need to be configured before a PC is able to communicate with any other network device. Adopting these setting is done by opening the Windows Control Panel. In the Windows Control Panel follow Network and Internet -> View Network Status and Tasks –> Change Adapter Settings.

Network Adapters are displayed as depicted in the following screenshot. Each icon represents a network adapter and displays the connection status: a red cross is shown if no network cable is plugged in. It is removed as soon as a cable connected to another network device is plugged in.

All network adapters not connected

Determining the assignment of the network adapter

Following it is illustrated, how network connections and corresponding LAN sockets can be identified by plugging in or removing the network cable.

1. No network cable plugged in

All network adapters are shown with a red cross:

2. Using the first network adapter

The network cable is plugged into LAN1. Hence one of the connections is shown without the red cross:

3. Using the second network adapter

The network cable is plugged into LAN2. Hence one of the connections is shown without the red cross::

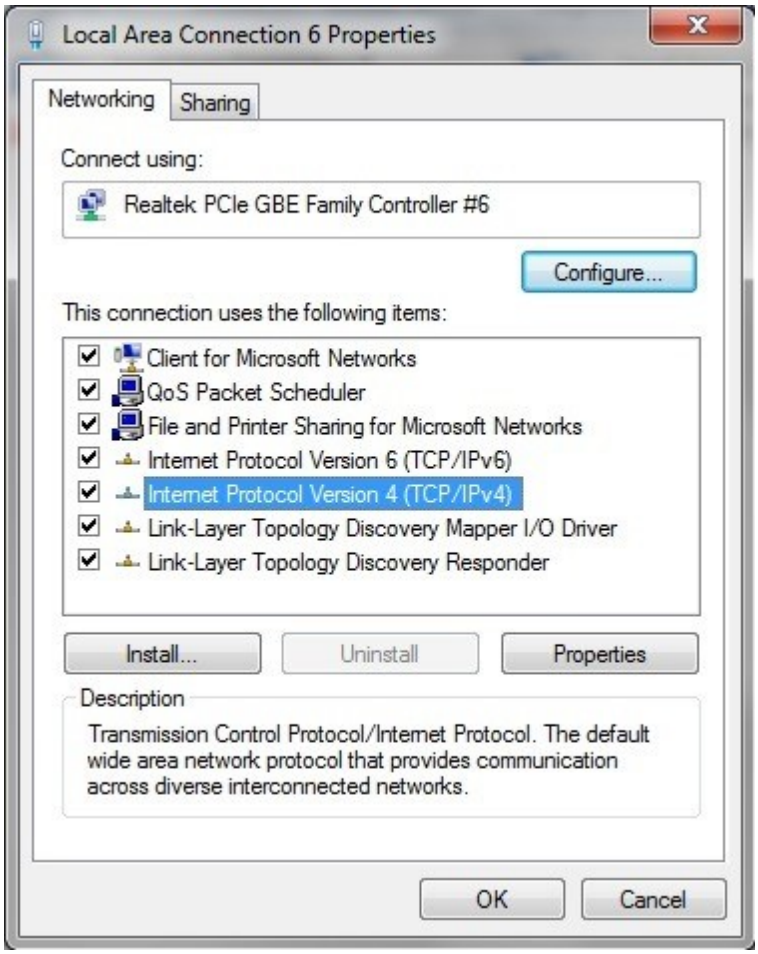
3.3.3 Network settings

3.3.3.1 IP configuration using DHCP

The Irinos System supports easy-to-use connection establishment by a built-in DHCP server. As a factory default, the DHCP server is active and provides basic settings for the network adapter. However, if needed, the DHCP server may be switched off 105 by the Irinos Tool and IP settings can be configured manually 102.

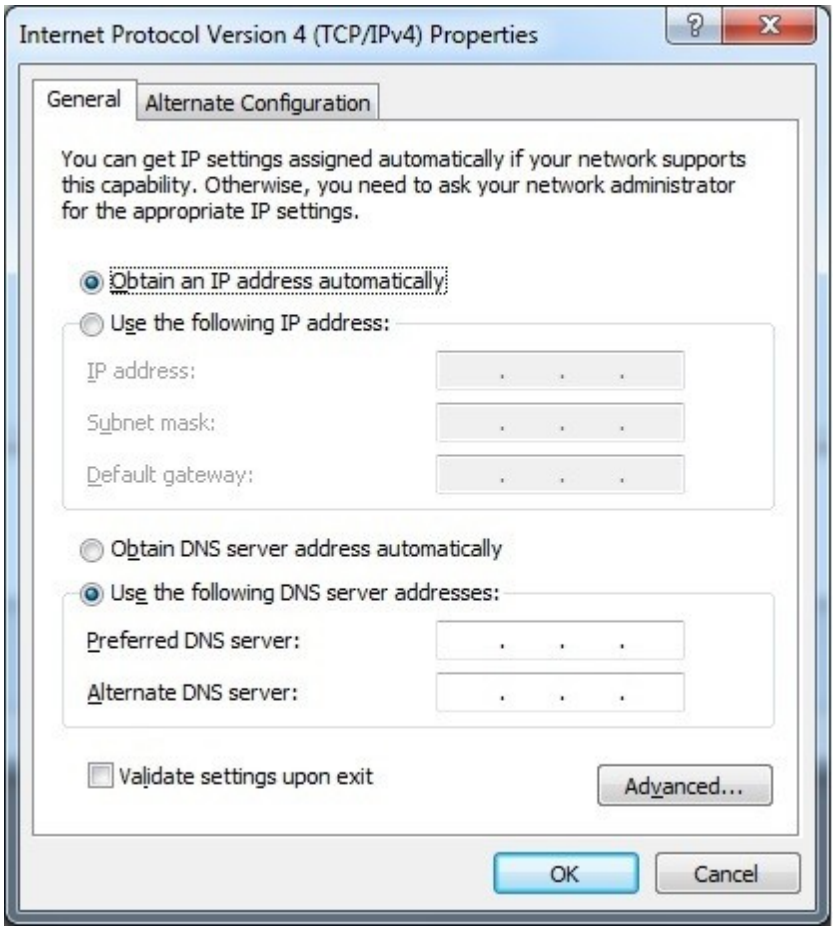
If DCHP is to be used, it is mandatory that the network adapter settings are set to DCHP mode, too. To verify these settings, right-click onto the “Local Area Connection x” item, then selecting the “Properties” item.

An input mask is opening:



-> Select "Internet Protocol Version 4 (TCP/IP)" and press the "Properties" Button.

An input mask is opening:

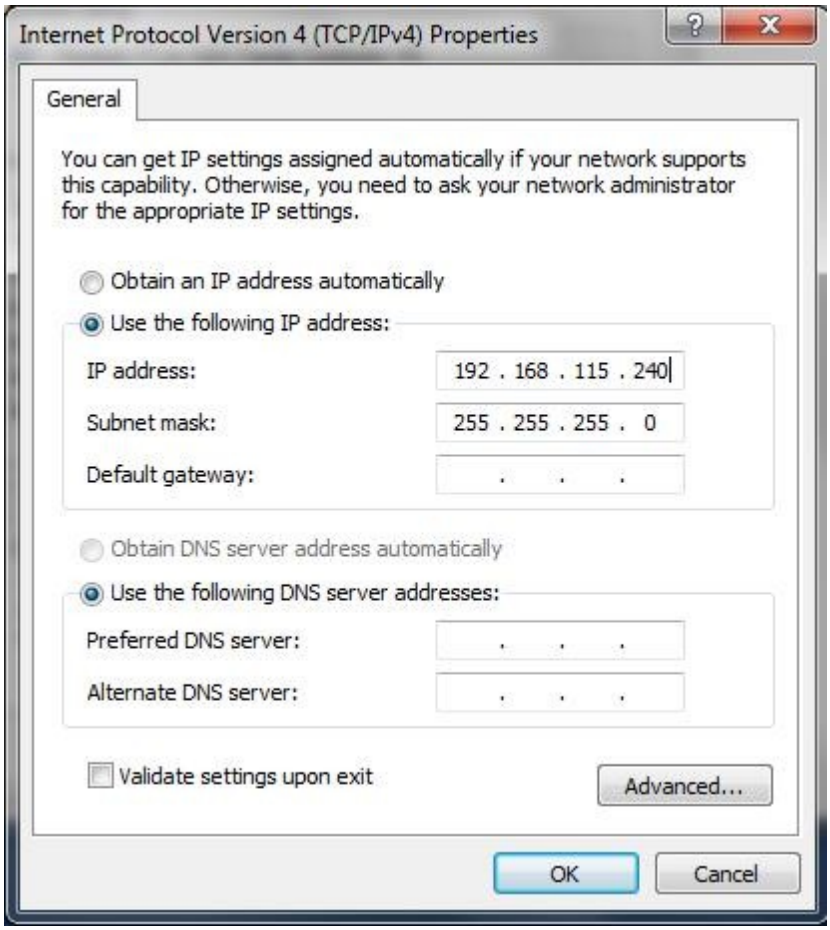


-> Select "Obtain an IP address automatically".

3.3.3.2 IP configuration without DHCP

If the built-in DHCP server of the Irinos-System shall not be used, it can be switched off 105 by means of the Irinos Tool.

If no DHCP server is used, the IP settings of the network adapter need to be configured manually. An example is given below.



--> Select "Use the following IP address".

Then enter IP address and Subnet mask:

Subnet mask 255.255.255.0 is recommended

The IP address needs to be in the same subnet as the device. For example, if the subnet mask is chosen as above, the IP address may be in the range from

192.168.115.1

..

192.168.115.254.

However, it must be different from the one that is used in the Irinos master box.

The settings for the Default Gateway may be left blank. The same applies for the DNS settings.

Finally press "OK" to adopt these settings.

The IP address of the Irinos master box is set in the MSC.cfg file. This file can be generated automatically 109 using the Irinos-Tool.

3.4 Irinos-Tool

3.4.1 General

The Irinos Tool provides a set of utilities which support installation, interconnection and start-up of the Irinos system. It is used to discover and identify 104 Irinos-Systems in an IP network and to configure network settings 105.

It supports connection establishment and verification 109, inventory visualization 112, measuring channel modifications 111, basic functional tests 116 and firmware updates 121.

Furthermore it is gathering information from the master box and generates appropriate configuration 109 files needed for the measurement system.

3.4.2 Installation

The Irinos Tool is delivered as a self-extracting compressed archive. To install, double-click onto the archive-file and select a ***writeable file location*** on your disk (refered to as 'YourFolder' in this document). It is advisable to create a shortcut for the executable file ITool.exe in YourFolder/ITool/source_XE8) and move this shortcut to the windows desktop.

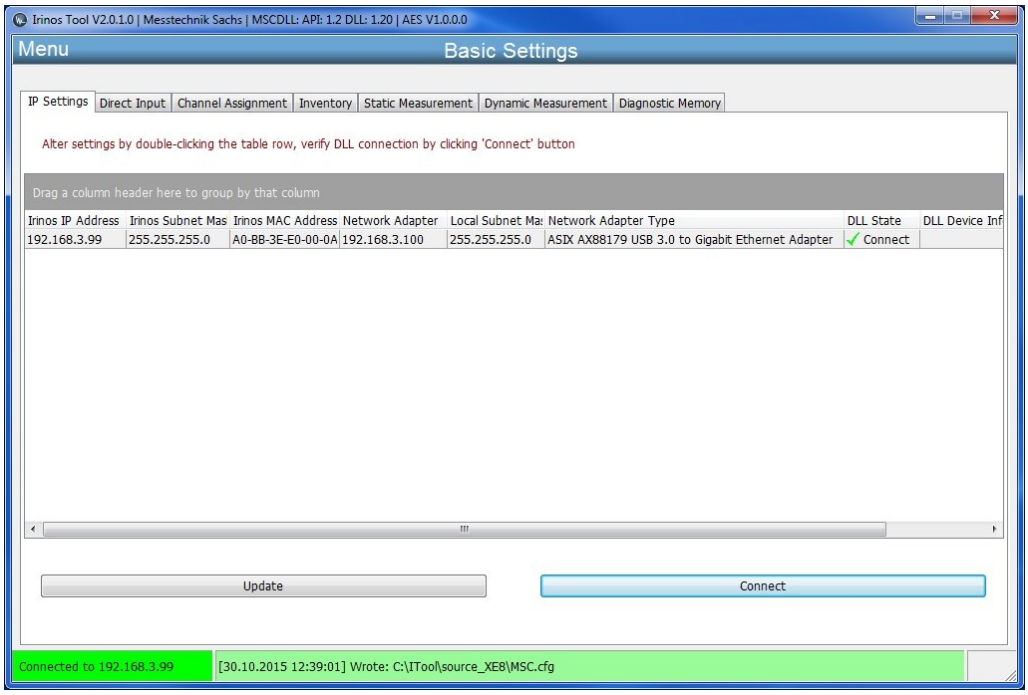
When the Irinos Tool is started the first time, a Windows ***firewall security alert*** will pop-up and ask for an access decision. Please press '***Allow access***'.

3.4.3 Starting the Irinos-Tool

The Irinos V2.0 Irinos Tool is started by a double click on the desktop shortcut created during installation 104.

During startup the Irinos Tool software is querying all activated network adapters and sending broadcast messages to the attached network. Any Irinos box connected to the network is responding with an appropriate response message.

Thereby, the Irinos Tool is able to present a list of all found boxes in the network right after start-up. Typically only 1 Irinos-System is found:



Startfenster des Irinos-Tools

Each table row represents one Irinos master box and contains

- Irinos Box IP Address
- Irinos Box Subnet Mask
- Irinos Box MAC Address
- IP Address of the network adapter the box is connected to
- Subnet Mask of the network adapter
- Network adapter type.

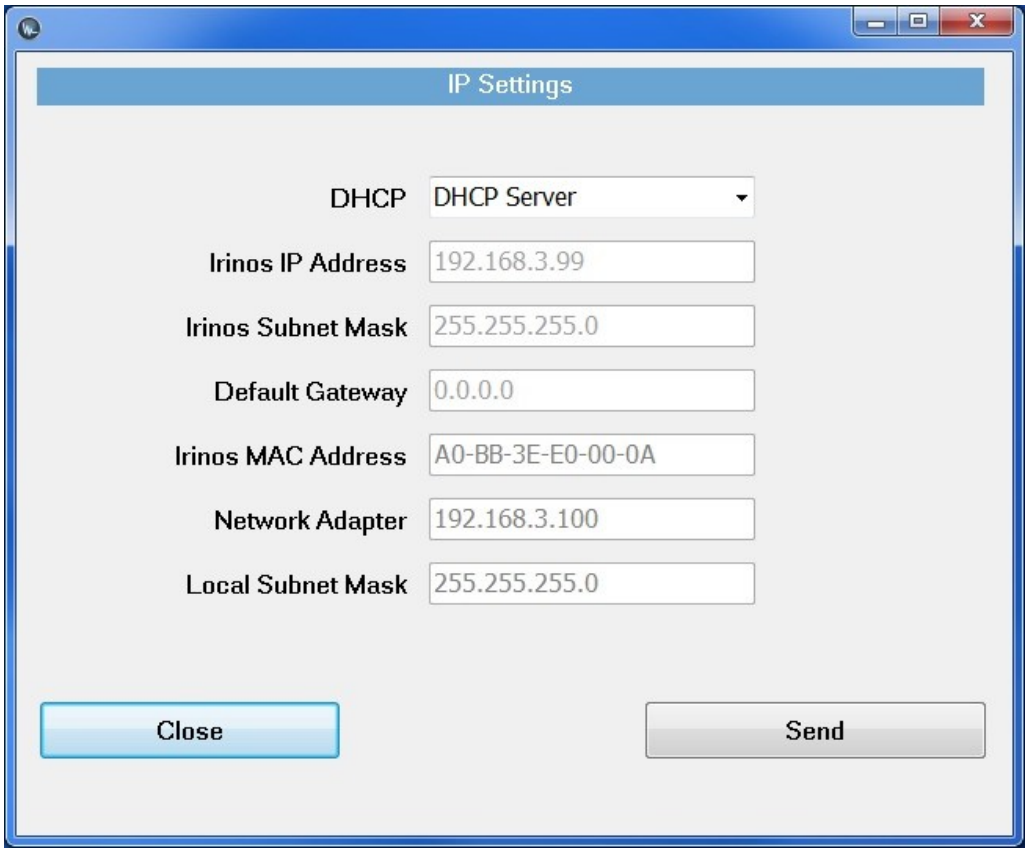
Changes within the network can be obtained by pressing the “Update” button.

3.4.4 IP configuration

As the Irinos boxes are delivered with an activated DHCP server, in most cases there will be no need to alter the IP configuration.

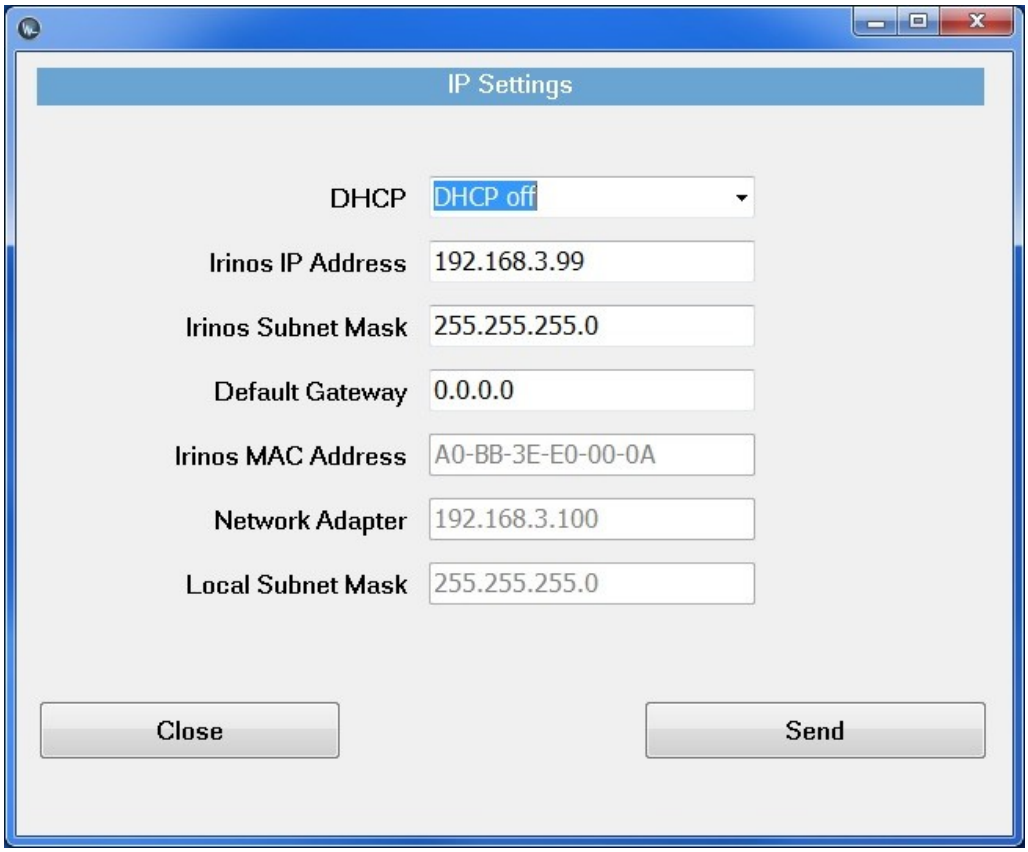
If the IP configuration needs to be changed, a double-click on a particular row in the main window will open the IP Settings window. Initially the window will open up as shown below, with an active DHCP server.

--> Setting DHCP: DHCP Server



If the DHCP setting is altered to “DHCP off”, the window will enable input fields for

- Irinos IP Address
- Irinos Subnet Mask
- Default Gateway



The Irinos MAC address and the local computer's network adapter IP address and subnet mask[102] are shown for guidance reasons only and cannot be modified here.

Now IP Address, Subnet Mask and Default Gateway may be entered according to the requirements of the customer network.

Before the settings are sent towards the Irinos System, several consistency checks are executed:

- Illegal IP addresses such as 127.0.0.1, 127.0.0.0, 255.255.255.255 are refused.
- Does the chosen IP address match the subnet of the local network adapter ?
- Is the chosen IP address the same as the local network adapter IP address ?
- Is a standard IP range used (10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.31.255.255, 192.168.0.0 to 192.168.255.255)

Illegal IP addresses cannot be sent, any other setting can be used after the user has explicitly confirmed his choice.

After the configuration settings were sent to the Irinos-System, it performs a reset. While executing the reset, the appropriate table row in the main window may disappear, if the update button is pressed within this time

period. After approximately 10 seconds the Irinos-System is reachable again. Anew pressing the update button is causing the appropriate table row to show up again. If the Irinos-System setting had been modified, the new settings are now being displayed in the table row.

3.4.5 Direct IP settings

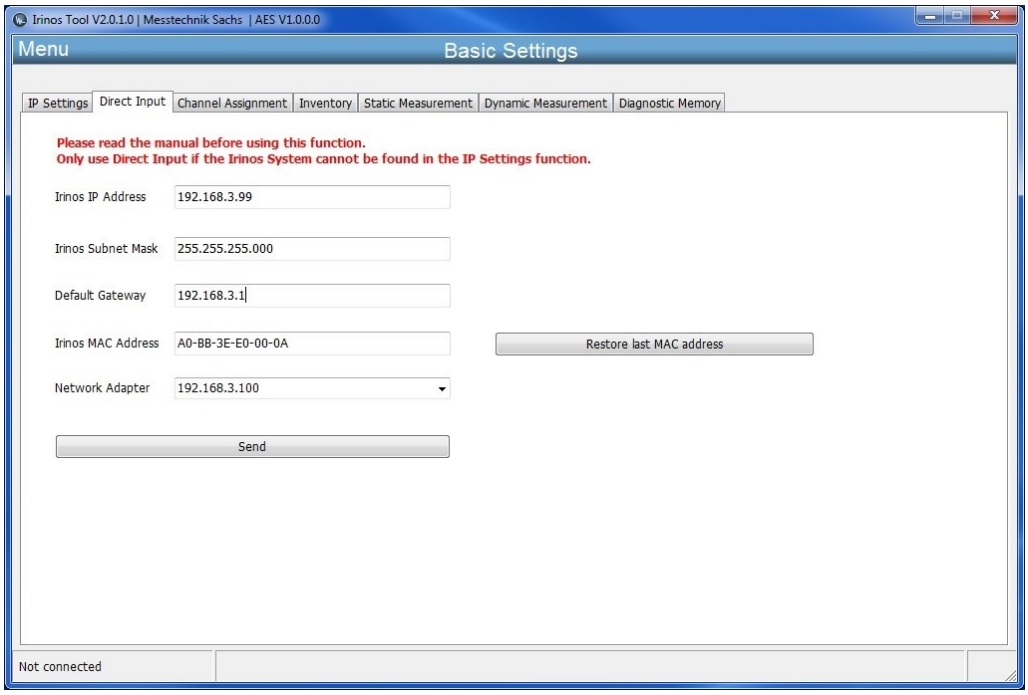
The Direct Input function is ***not needed for normal operation***. It can be used to specify IP setting directly, without picking the Irinos box from the main window list. Moreover, it requires the user to specify IP address and MAC address accurately and, if more than one, select the network adapter the Irinos box is connected to.

This function is intended for cases, where an Irinos box is not showing up in the main window although connected to the computers network adapters. This might be the case when it has been configured with an IP address which is not part of the subnet of the network adapter.

If so, the box can be reconfigured by directly entering the desired IP address and the MAC address of the box. If several network adapters are present, the appropriate network adapter needs to be selected as well.

It is possible to retrieve the last MAC address used for configuring the IP settings of a box. This is done by the “Restore last MAC address” button

By pressing the send button the IP settings are broadcasted to the network attached to the selected network adapter. If there is an Irinos box listening to the given MAC address, it is going to alter its IP address to the one specified, and performs a reset. Once it has recovered from reset, and the new IP address is part of the subnet, it is displayed in the main window again, upon pressing the update button.



3.4.6 Checking the connection via the MscDll

To interface any kind of measurement software with the Irinos-System, a dynamically linked library (DLL) named “MscDll.dll” is provided with the Irinos system. This DLL contains basic discovery and access methods to connect to the Irinos box and to establish measuring channels between measurement system and local host computer.

For that reason it is recommended to verify these communication principles before continuing with the final measurement software.

First, as a pre-requirement for communicating with the Irinos box, the IP address of the Irinos master box has to be set in the DLL’s configuration file “MSC.cfg”. From this file the IP address is read by the DLL during start-up.

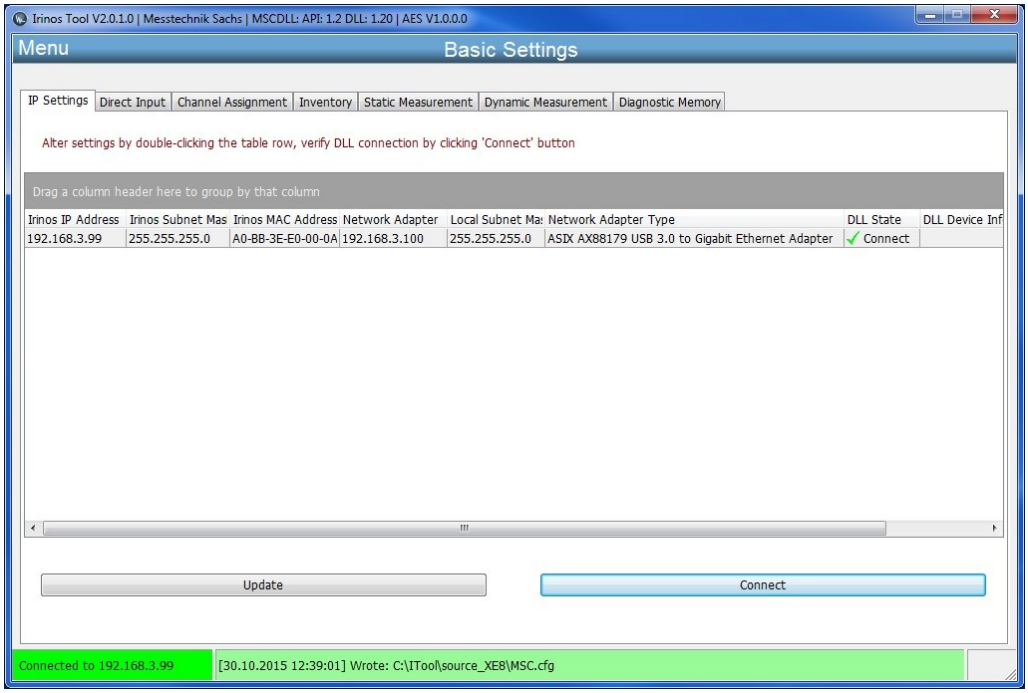
The process of writing the IP address to the configuration file is done automatically by the Irinos Tool, whenever the DLL Connection Verification is started.

Triggering the connection check is done by selecting the appropriate table row and pressing the ‘Connect’ button.

If the connection check turns out ok, the “DLL State” column is showing a green tic followed by a “Connect” indication. Additionally the DLL Device Info, as returned from the DLL, is inserted into the last column of the table.

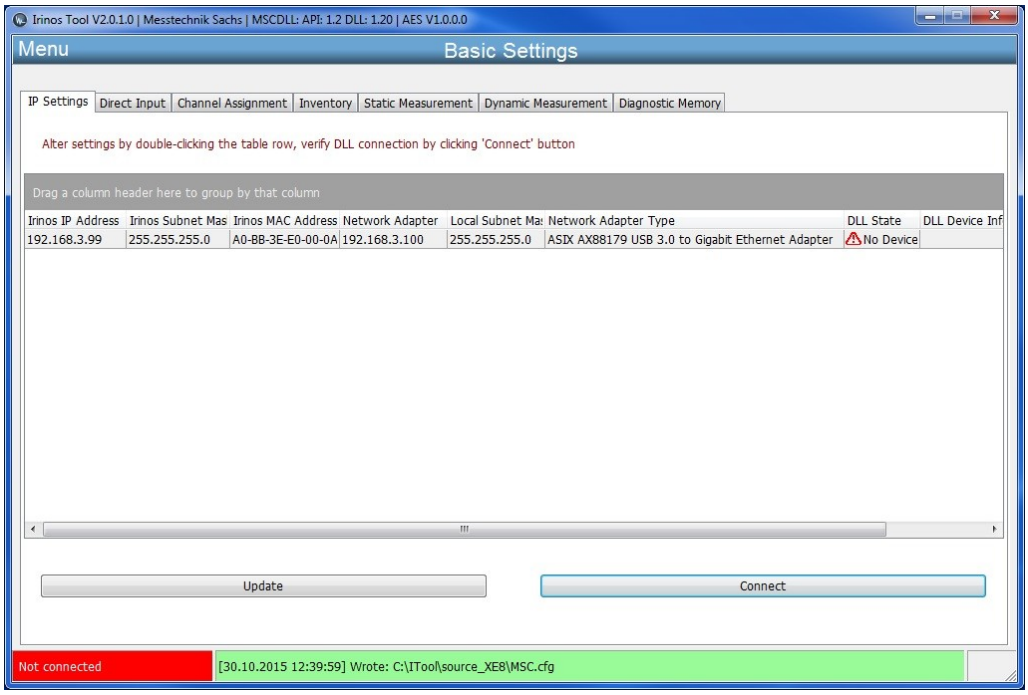
Second, the connection status is shown at the bottom status bar by a green field displaying “Connected to box- ip- address”.

Third, the next field of the bottom status bar indicates the location of the DLL configuration file “MSC.cfg” at the disk and the date/time it was written.



If the connection check fails, the “DLL State” column is showing a red error sign followed by the error reason, as delivered by the DLL while trying to access the box. The connection status at the bottom status bar indicates “Not connected”.

Regardless of the result of the connection check, the location of the DLL configuration file “MSC.cfg”, is depicted in the second field of the bottom status bar.



Reasons for an unsuccessful connection check may be

- No physical connection
- Network adapter not enabled
- No power at the Irinos box

3.4.7 Channel Assignment / Selecting incremental input type

The Channel Assignment function of the Irinos Tool retrieves the measuring channel structure of the Irinos system. The channel assignment is generated by the master box directly after power-on.

It reflects the interconnection of the Irinos boxes and their respective measuring channels. Each single box contributes a certain amount of channels to this structure.

Incremental encoders are characterized by the type of signal they provide at the encoder interface. Two interface specifications are commonly used:

- Encoders providing sinusoidal 1 Volt peak-to-peak signals (referred to as '1Vpp' types) and
- Encoders delivering a square-wave TTL/RS422 signal (referred to as 'TTL' types).

According to the type of the encoder, the corresponding Irinos box needs to be preset with the encoder types to be operational. These pre-settings are configured as factory defaults, either 1Vpp or TTL for all channels of a box.

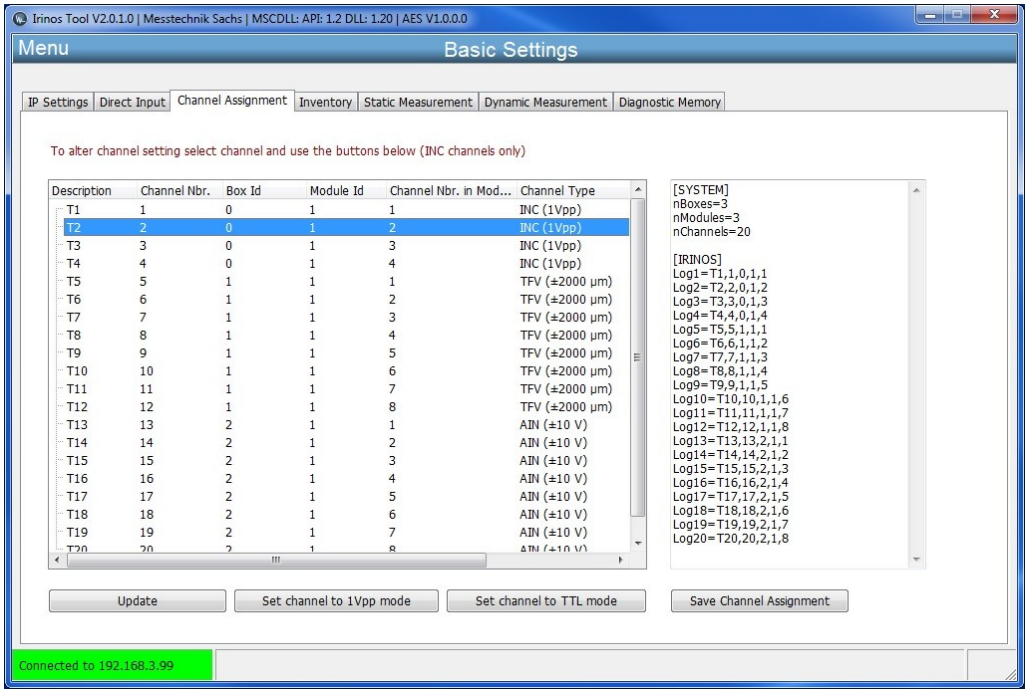
The user might want to alter this setup, for instance to adopt to a configuration which includes both types.

The graphical representation of the channel assignment, as displayed on the left-hand side of the function window, enables the user to alter these channel characteristic.

To modify this channel characteristic, select a channel by clicking the appropriate table row, then click one of the buttons

- Set channel to 1Vpp mode
- Set channel to TLL mode

Modifying a channel characteristic from given factory defaults enables the user to configure his own setup of 1Vpp and TTL channels as needed.



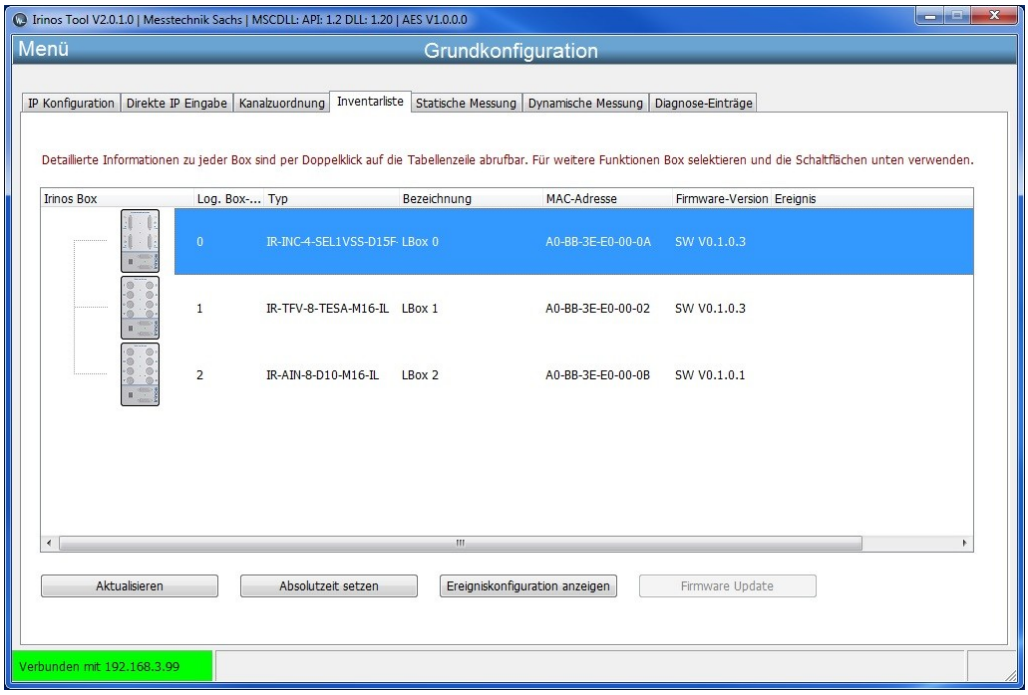
3.4.8 Inventory

The Inventory function of the Irinos Tool retrieves the structure of the Irinos system and displays the composition of boxes.

Additional module dependent data is displayed, such as

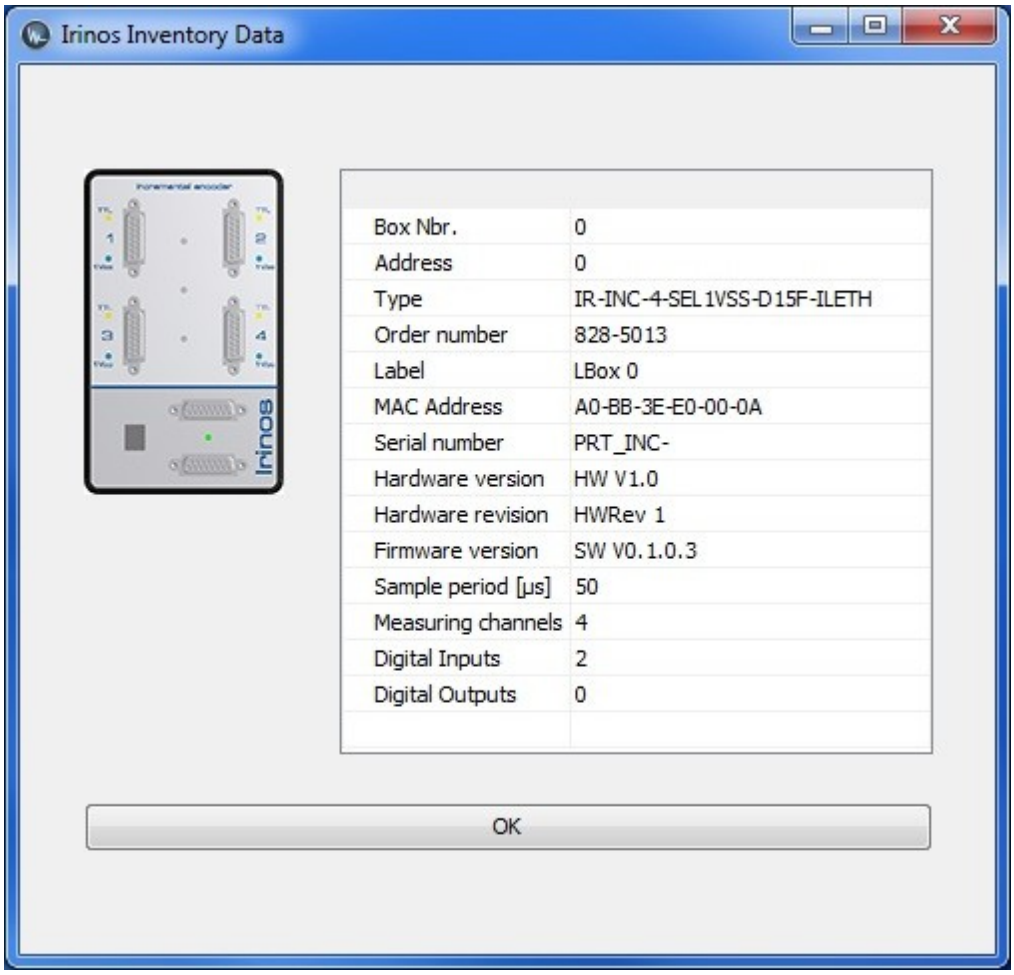
- Logical Box Number
- Box Type

- Box Label
- MAC Address
- Firmware Version
- Event (If any event is currently active in the box)



Changes can be obtained by pressing the “Update” button. However, any modification to the system setup, such as adding or removing boxes, requires a power-off-power-on cycle at the master box before it can be detected by the Irinos Tool.

By double-clicking the table row, detailed box data is displayed in a separate window:



The Inventory display serves as a basis for the functions

- Set Time 114
- Firmware Update 121
- Event configuration 115

All of those functions are accessible by selecting the appropriate table row in the Inventory display.

3.4.8.1 Setting date/time

--> *This function is only available for the Irinos Master-Box.*

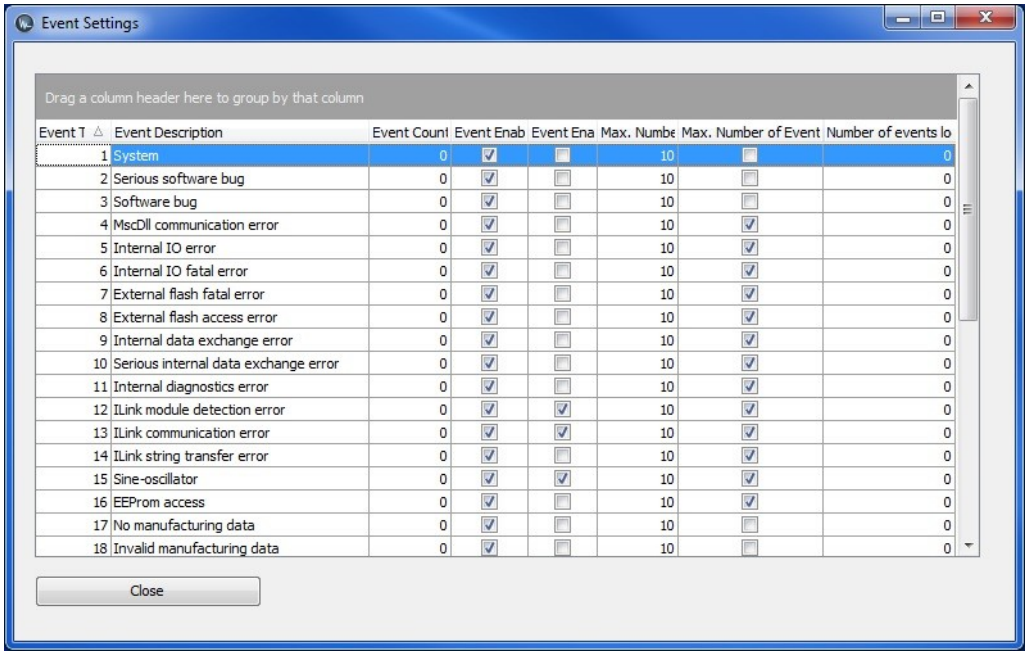
By this function the current clock time and date is sent to the Irinos system. Thus, the Irinos system is able to generate an absolute time reference. The absolute time is held in the Irinos system until it is powered off or reset.

All upcoming events will be equipped with a time stamp on a clock time basis which enables an accurate correlation between event occurrence and time of day.

Within the Irinos Tool, this function serves as a test and analysis resource. When used in measurement software this function should be used once a day to ensure sufficient time accuracy. See the MscDll reference manual for details.

3.4.8.2 Event configuration

The Event Setting window displays the event handling capabilities of the Irinos system per box. The Irinos System supports several configuration options regarding event handling. The behavior of some event types is user-modifiable. For each event type several configuration options are displayed:



Event T	Event Description	Event Count	Event Enab	Event Ena	Max. Numbe	Max. Number of Event	Number of events lo
1	System	0	☑	☐	10	☐	0
2	Serious software bug	0	☑	☐	10	☐	0
3	Software bug	0	☑	☐	10	☐	0
4	MscDll communication error	0	☑	☐	10	☑	0
5	Internal IO error	0	☑	☐	10	☑	0
6	Internal IO fatal error	0	☑	☐	10	☑	0
7	External flash fatal error	0	☑	☐	10	☑	0
8	External flash access error	0	☑	☐	10	☑	0
9	Internal data exchange error	0	☑	☐	10	☑	0
10	Serious internal data exchange error	0	☑	☐	10	☑	0
11	Internal diagnostics error	0	☑	☐	10	☑	0
12	ILink module detection error	0	☑	☑	10	☑	0
13	ILink communication error	0	☑	☑	10	☑	0
14	ILink string transfer error	0	☑	☐	10	☑	0
15	Sine-oscillator	0	☑	☑	10	☑	0
16	EEProm access	0	☑	☐	10	☑	0
17	No manufacturing data	0	☑	☐	10	☐	0
18	Invalid manufacturing data	0	☑	☐	10	☐	0

Event Count	Number of event occurrences
Event Enabled	Event handling is generally enabled.
Event Enabled modifiable	Is the event handling user modifiable? I.e. is the user allowed to turn this event on and off?
Max. Number of Events	How often will this event be logged in the diagnostic memory.
Max. Number of Events modifiable	Is this value (Max. Number of Events) user modifiable? I.e. is the user allowed to alter this value from 10 (default) to any other value?
Number of events	Number of events of this type since system start.

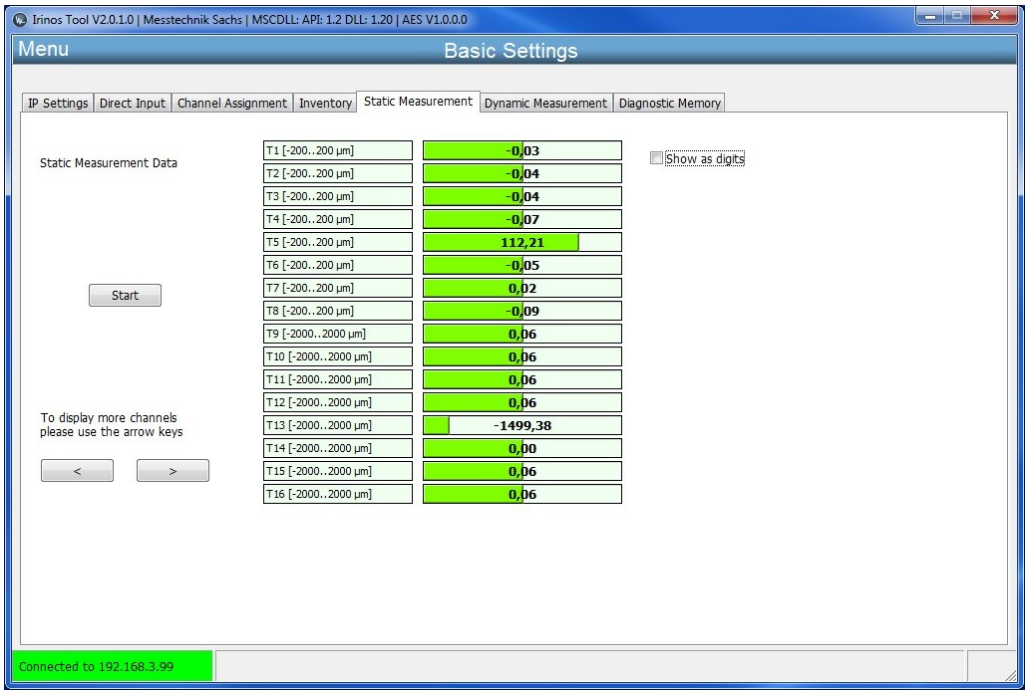
The Irinos Tool is intended ***only to display these settings***. To actually modify these settings, please refer to the reference manual of the MscDll for the Irinos System.

3.4.9 Static measurement

The Irinos Tool provides a measurement display to check and analyze measurement data from each measurement channel.

Once started, it displays “live” data received from the Irinos system. Up to 16 channels are displayed simultaneously. If the system comprises more than 16 channels the user is able to switch to the next set of 16 channels by use of the arrow keys.

Furthermore, it is possible to alter the value display from digits to physical units (such as micrometers, volt, etc.) by means of a tick-box.



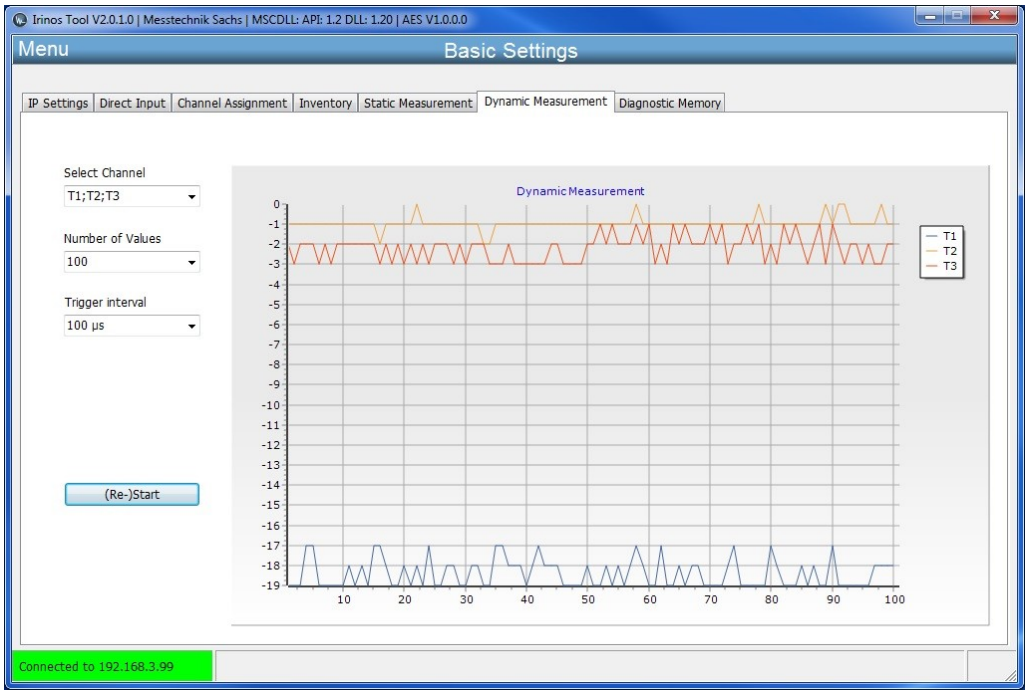
Static measurement is automatically set-up by verifying the connection 109, as it is used as a timeout-monitoring function. Whenever the user directly enters the Static Measurement window, it might be necessary to start the static measurement by means of the “Start” button.

3.4.10 Dynamic measurement

Similar to the static measurement display, the Irinos Tool facilitates setup and verification of dynamic measurement.

To minimize configuration effort only time-triggered measurements are supported. The user only needs to select the desired channels, define number of measurement values and a timer trigger interval before he can start the dynamic measurement.

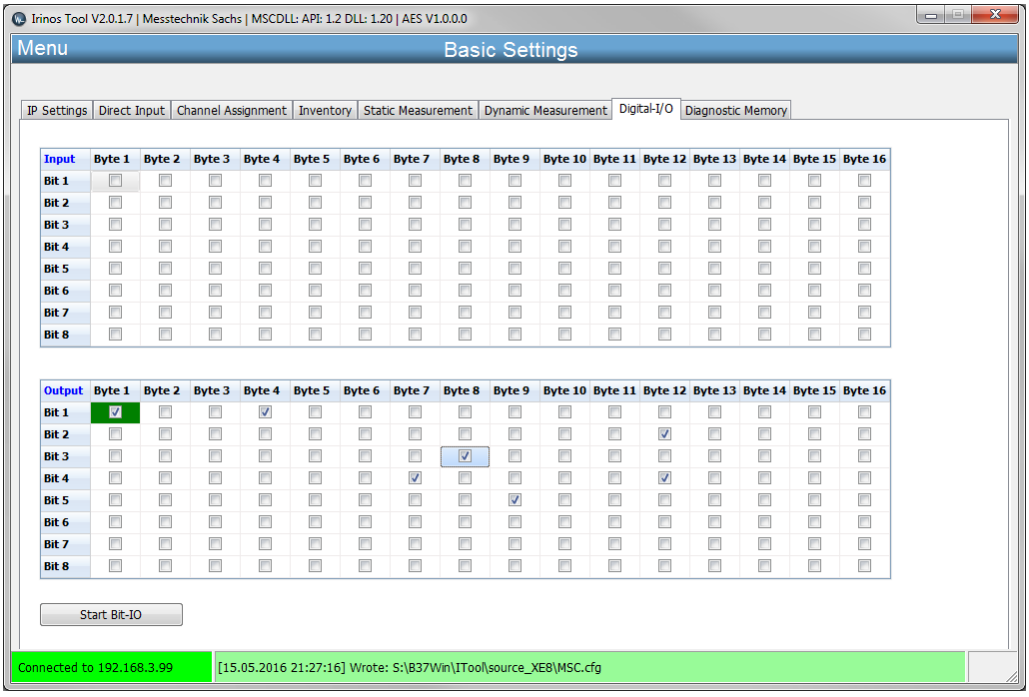
The channel selector offers a predefined list of channels for easy setup. However, it is possible to modify the list i.e. the user can add or remove channels on a textual basis, as long as the semicolon-separated format is retained (e.g. `T1;T2;T3;T9`).



3.4.11 Digital in- & outputs

Requires the IrinosTool version 2.0.1.7 or newer.

The tab Digital-I/O (Bit I/O) provides an online view for up to 128 digital inputs and allows changing the state of up to 128 digital outputs:



Digital in- & outputs

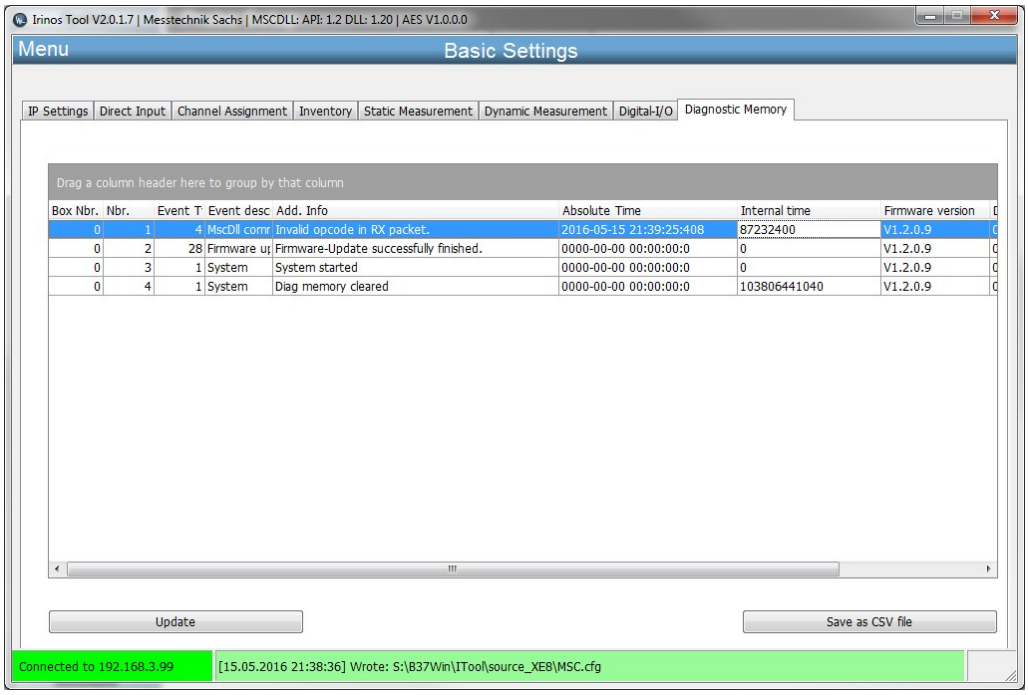
The table "input" shows the status of the digital inputs connected to the Irinos-System. A check-mark in the checkbox signals the high-level of the digital output.

The table "output" allows enabling or disabling digital outputs directly. If a checkbox is checked, the corresponding digital output will be set to high.

3.4.12 Diagnostic memory

For diagnostic reasons, the Irinos system gathers all events reported by the local firmware in a non-volatile event memory. Events can be visualized via the built-in web server, or alternatively upon request within the Irinos Tool.

Diagnostic events are displayed for the entire system, i.e. event data is gathered box by box and finally displayed as a system-wide view. Grouping and sorting of events is supported to facilitate easy cross-box event analysis.



A diagnostic memory entry has the followings attributes:

Box-Nbr	Number of the Irinos-Box: the event occurred at the Irinos-Box with this address.
Nbr	Event number (per Box)
Event Type	Event type (numerical value)
Event description	Event type as text
Add. Info	Additional information for the cause of the event
Absolut Time	Date/Time the event occurred (only available, if the absolute time has been set before).
Internal Time	Internal system time (ILink-Time, [µs])
Firmware version	Firmware version at the time the event occurred.
Debug Info	Further information for manufacturer support.

Newly-created events can be obtained by the update buttons.

If required, the entire event data can be saved into a CSV file by pressing the "Save CSV file" button. After pressing the button a file selection dialogue is displayed. File name and location can be specified by the user.

3.4.13 Firmware update

3.4.13.1 Version numbers

The version number of the firmware consists of 4 parts, which are separated by a dot, e.g. V1.3.4.534. The meaning of the parts is:

Part of the version number	Meaning
1. Part, in the example: `1`	"Major" version number It is incremented, if the firmware is completely redesigned (-> happens seldom).
2. Part, in the example: `3`	"Minor" version number It is incremented, if new functionality has been implemented.
3. Part, in the example: `4`	"Patch" It in incremented, if one ore more bugs have been fixed.
4. Part, in the example: `534`	"Build" Internal number for firmware identification.

In addition, firmware can be marked as "customer specific" or as "Beta version". A warning note is displayed before such a firmware can be downloaded. It must only be used after written notice by Messtechnik Sachs.

3.4.13.2 Executing the update

A basic feature of the Irinos Tool is to support firmware updates for the Irinos System.

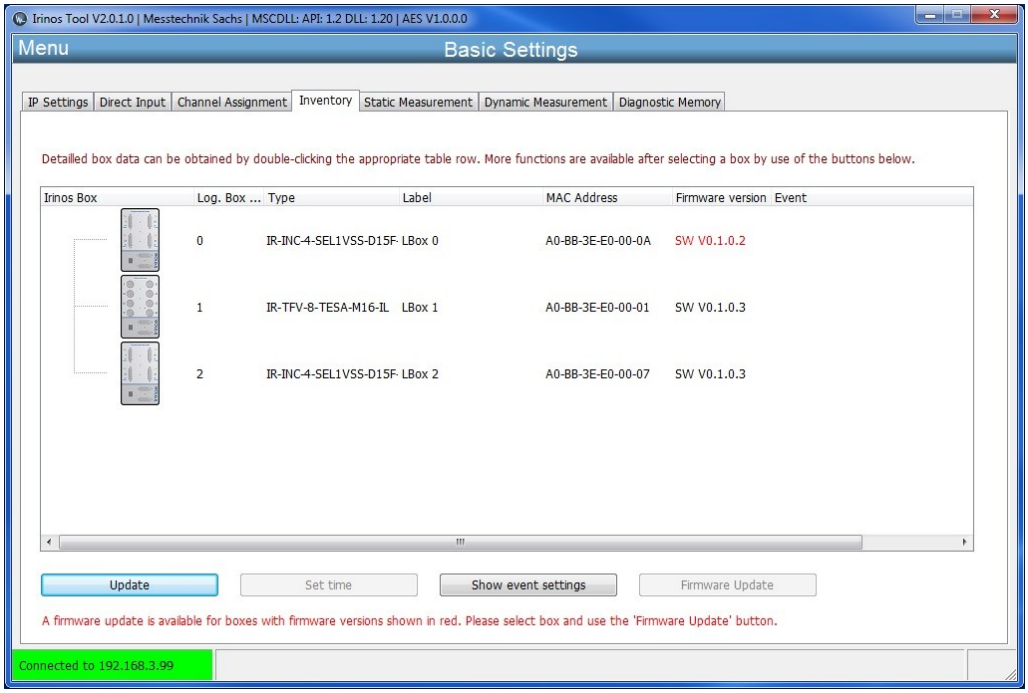
A firmware package is delivered by Messtechnik Sachs GmbH as particular type of file with the extension “SFF”. Firmware Files are issued box type specific. I.e. a firmware package developed for an incremental encoder box can only be used for this type of box.

While setting up the Irinos Tool, a file folder “Firmware” is created in the “source_XEn” directory. Firmware files need to be put here to be found by the Irinos Tool.

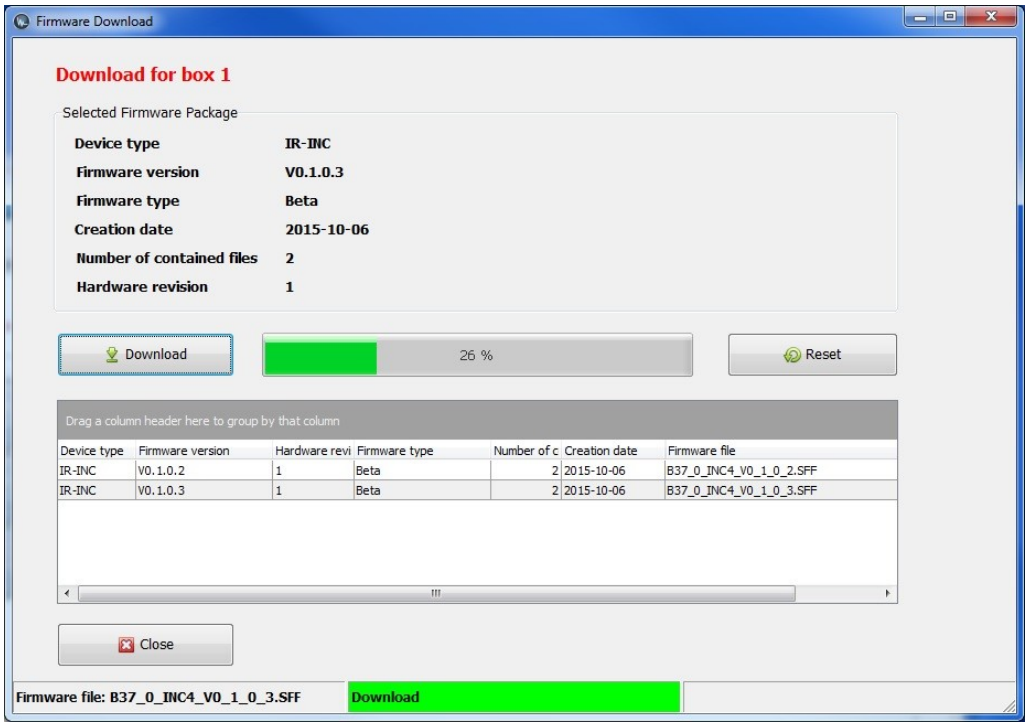
After the firmware files have been put there, the Irinos Tool needs to be restarted. During start-up the firmware files are read and evaluated.

If a new firmware package is found for a particular box type, the firmware version is highlighted in red in the Inventory window. Additionally, a hint is

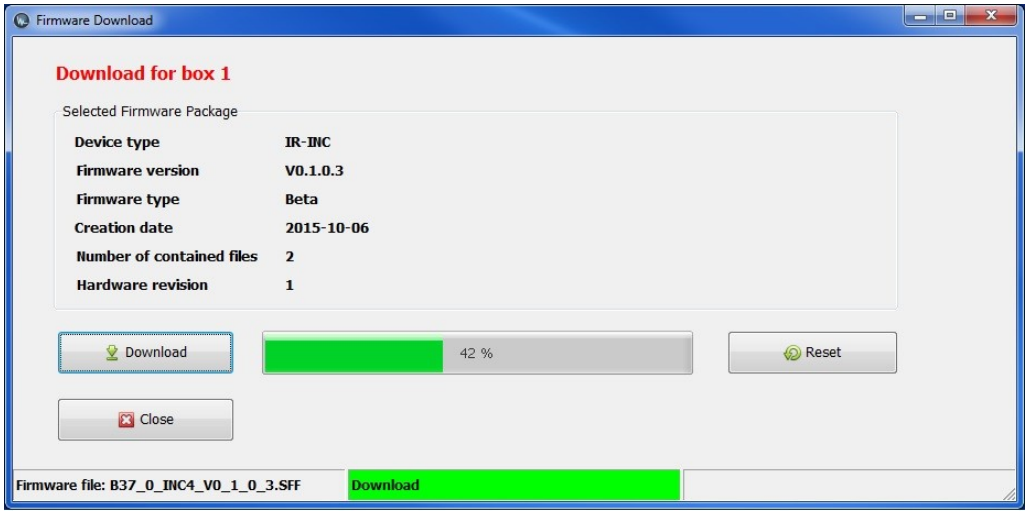
shown, indicating that a firmware update for those boxes is available. After selecting the appropriate box by clicking the table row, pressing the 'Firmware Update' button will open the firmware download window:



If more than one firmware package for a box type has been found, the Firmware Download window offers a selection table for the different firmware packages. Once the user has selected a package by clicking the appropriate entry, the download can be started by pressing the "Download" button:



If only one firmware package for the box type has been detected, the Firmware Download window automatically selects the package. The download can be started by pressing the "Download" button:



After all packages have been downloaded to the Irinos system, a system reset is required to activate the downloaded firmware. Pressing the "Reset" button sends an appropriate command to the Irinos system.

An internal algorithm ensures that slave boxes are reset first, before the master is resetting. This is necessary as the master always requires slaves to be operational, when returning from reset.

3.4.14 Incremental channel diagnostics

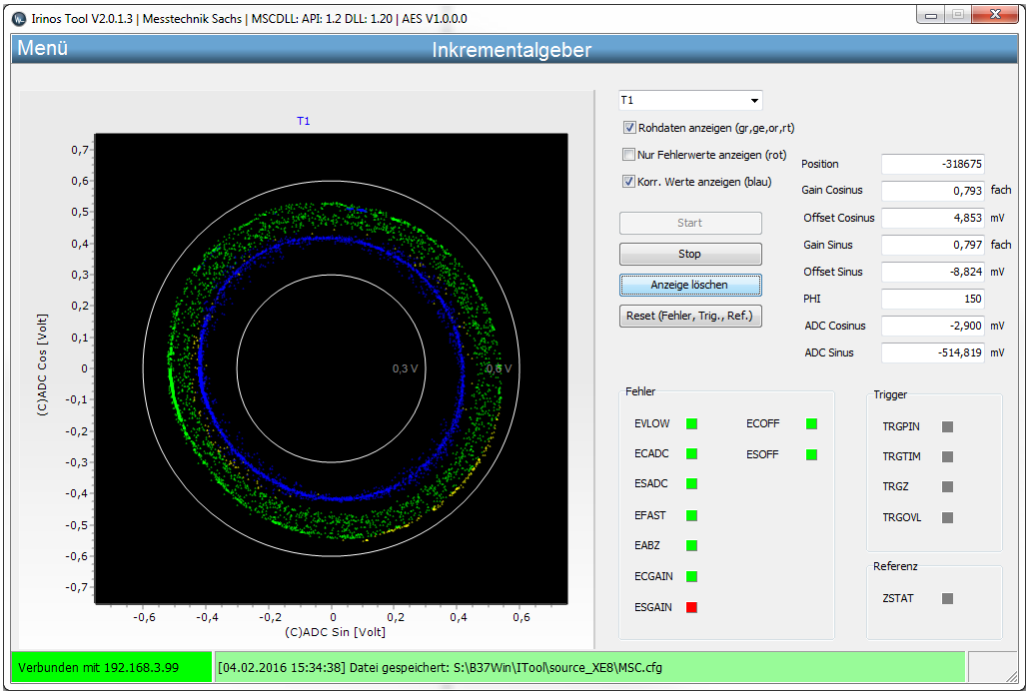
3.4.14.1 Live view (only 1Vpp)

If incremental encoders with 1Vpp interface are used, a high signal quality is required for an accurate and reliable position value. The incremental input channels of the Irinos-System are equipped with a sophisticated signal analysis. Therefore the Irinos-System is able to detect errors, where many other systems would deliver a possibly invalid position value.

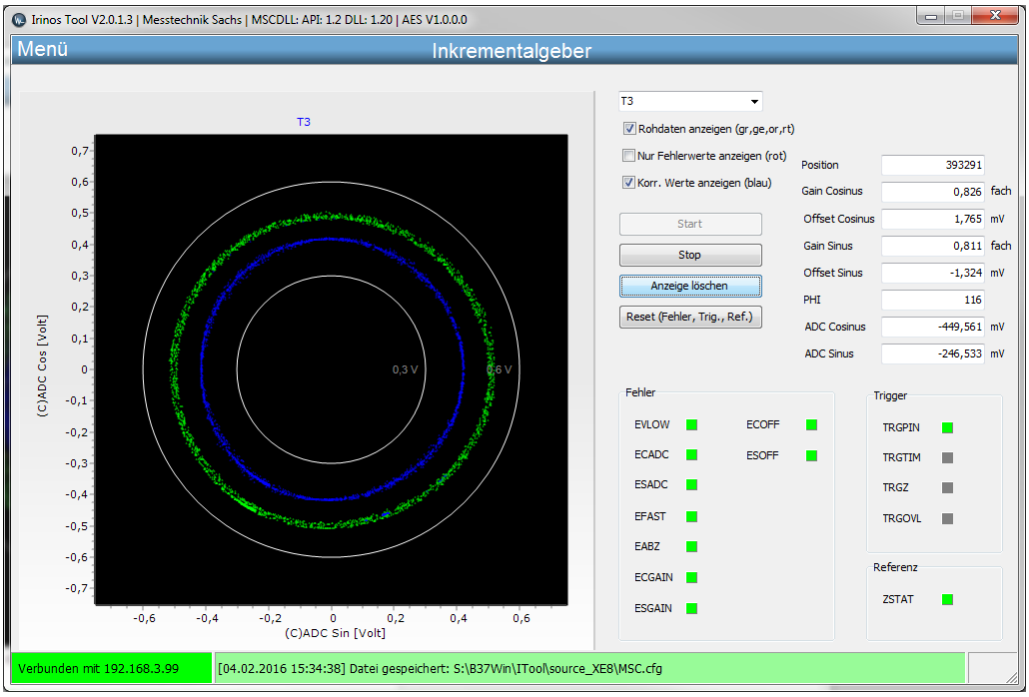
A signal level outside of the specified range may lead to an invalid position. The incremental channel will then provide an error flag in the hardware status. Further information can be found in the application notes in the users manual and in the MscDll reference manual (opcRHS).

The Irinos-Tool provides a live-view of the incremental encoder signals in a Lissajous diagram. In such a diagram, the signal level of the sine-signal is on the x-axis and the corresponding cosine-signal signal level is on the y-axis. If ideal signal levels would be available, a perfect circle would be the result if the encoder would move one electrical rotation. In reality such signals never exist.

The following examples show various signal levels:



Live view with unstable signals



Live view with very good signal levels

For technical reasons, the signals never exactly match the specified 1Vpp input level. Hence the Irinos-Box IR-INC has a tolerance range of 0.6Vpp .. 1.2Vpp.

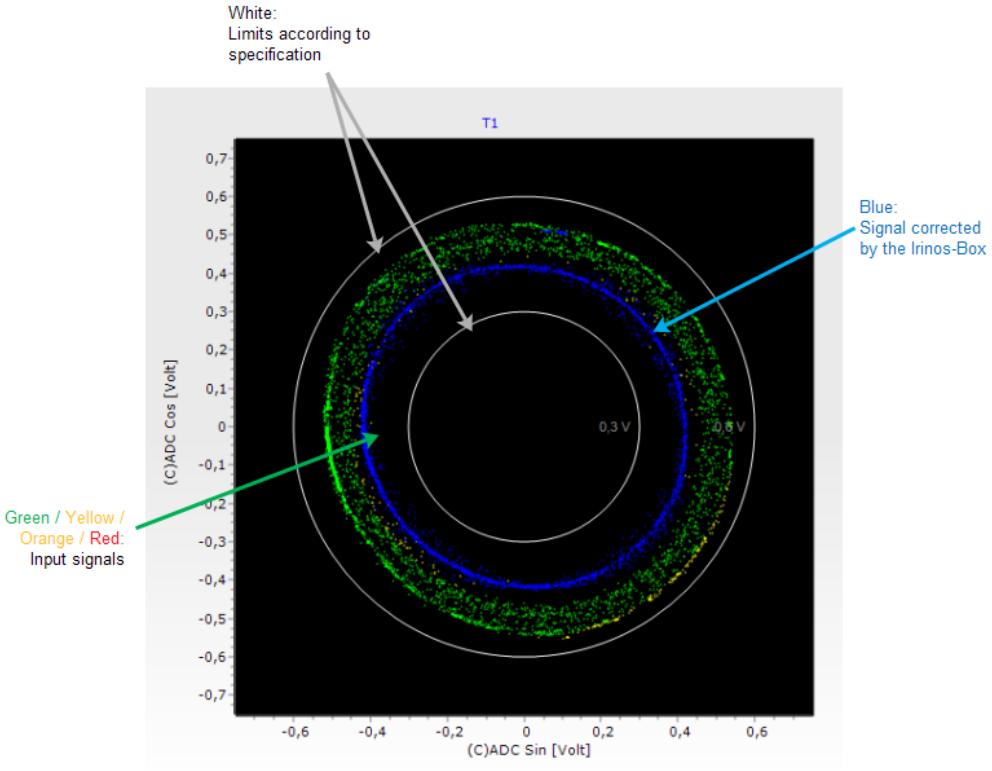
Lissajous diagram

The Lissajous diagram allows estimating the signal quality of the incremental encoder. Therefore the raw values are continuously requested from the incremental input channel of the Irinos-System. The live view can be started as follows:

1. Connect 109 to the Irinos-System.
2. Open the Live view via Menu -> Incremental encoder.
3. Select the desired input channel (e.g. T3).
4. Press "Start".
5. Move / rotate the incremental encoder.

The resulting signal vector if the input signals ("raw values") will no be displayed. If the signal vector is next to the ideal signal level, it will be coloured green. The more it deviates, the more the colour changes to yellow, orange or red.

The inner and outer limits are displayed as a white circle.



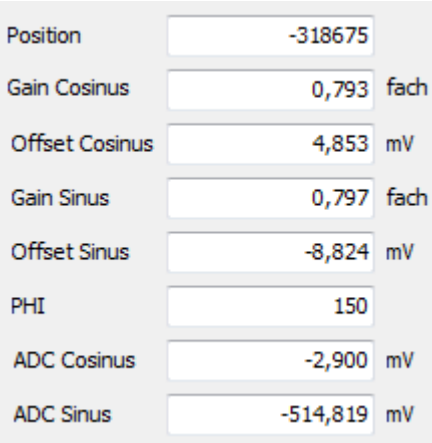
Lissajous diagram

The Irinos-Box continuously corrects the input signal via its offset- and gain-control functionality. The corrected values are displayed in blue. Most of the signal deviations can be corrected. However, if the signal levels become too low or too high, this is technically impossible.

This example diagram shows large deviations of the input signal. Most of the values are green, but some are yellow or orange. This diagram has been recorded during moderate speed. If the speed increases, it is likely that the input signals will be out of specification.

Numerical values

Next to the Lissajous diagram, the corresponding numerical values are displayed:



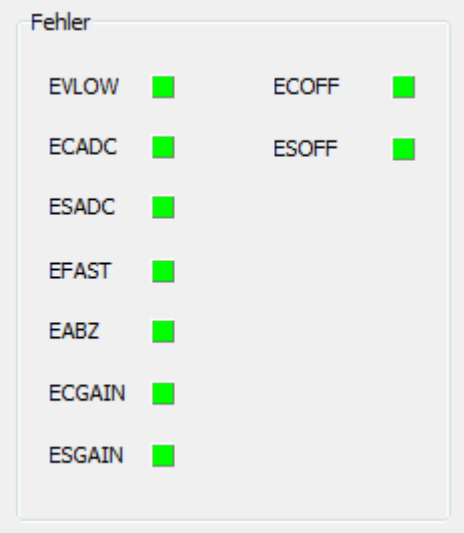
Numerical values

Position	Measurement / position value of the incremental input channel
Gain Cosinus	Gain factor for the cosine signal. It is continuously updated by the internal gain control.
Offset Cosinus	Offset for the cosine signal. This value is continuously updated by the internal offset control.
Gain Sinus	Gain factor for the sine signal. It is continuously updated by the internal gain control.
Offset Sinus	Offset for the sine signal. This value is continuously updated by the internal offset control.
PHI	Phase angle of the input signal. 0 -> 0° 200 -> 360°
ADC Cosinus	Analogue voltage measured at the cosine signal input. For an ideal signal, this value is in the range between -500mV .. +500mV.
ADC Sinus	Analogue voltage measured at the sine signal input. For an ideal signal, this value is in

the range between -500mV .. +500mV.

Error

Different error statuses are displayed via the respective error bits (green = ok, red = error):



Error flags

Error-Flag	Reason
EVLOW	The signal vector generated from the sinusoidal and cosinusoidal signals is smaller than 30% of the nominal amplitude. Usually, the cause is a partly or completely disconnected sensor. Another cause are input signals with a very large offset and a low amplitude at the same time.
ECADC	The A/D converter for the cosine signal is overdriven. The cause is that the signal amplitude is too high. This error may also occur with

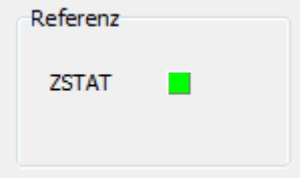
	signals with very large offset at simultaneously high amplitude.
ESACD	The A/D converter for the sinusoidal signal is overdriven. The cause is that the signal amplitude is too high. This error may also occur with signals with very large offset at simultaneously high amplitude.
EFAST	The input frequency is so high that the direction can no longer be detected.
EABZ	Internal error flag, which is disabled for standard applications.
ECGAIN	The gain controller for the cosine signal has reached its limit. The cause is either that the signal amplitude is too low or the sensor is partly or fully disconnected.
ESGAIN	The gain controller for the sine signal has reached its limit. The cause is either that the signal amplitude is too low or the sensor is partly or fully disconnected.
ECOFF	The offset controller for the cosine signal has reached its limit. The cause is an excessive signal offset or a partly or fully disconnected sensor.
ESOFF	The offset controller for the sine signal has reached its limit. The cause is an excessive signal offset or a partly or fully disconnected sensor.

Please note regarding errors:

Because of the limited data bandwidth, the live view only shows a part of the available raw values. Therefore it may happen that an error occurs but no corresponding value is visible in the live-view.

Reference index

The Bit ZSTAT becomes enabled once the reference index has been crossed.



3.4.14.2 History (only 1Vpp)

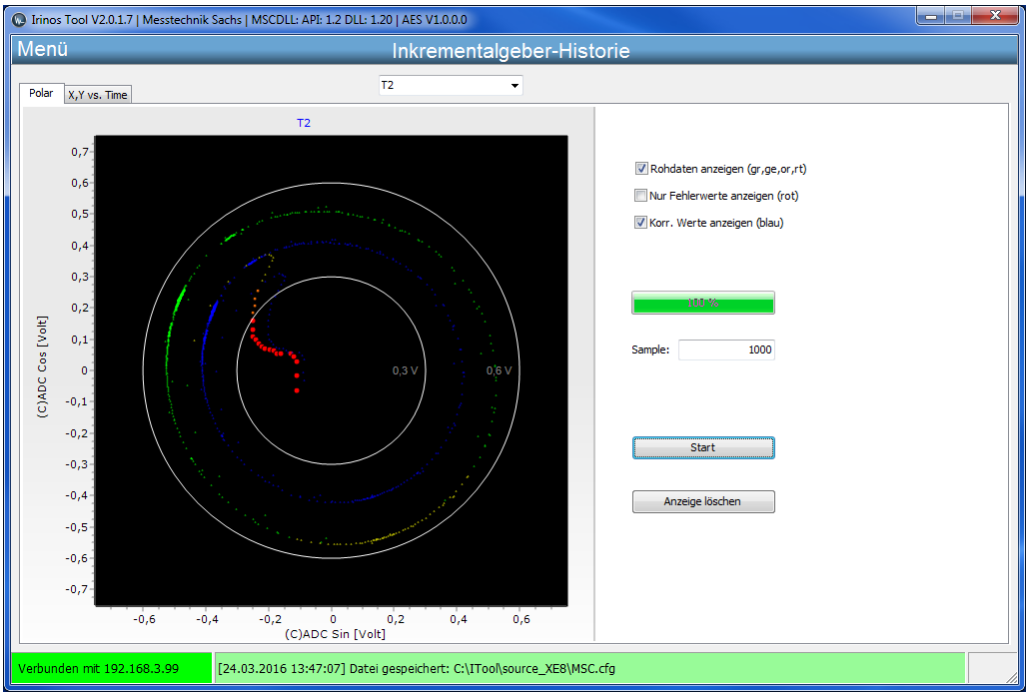
Requires the IrinosTool version 2.0.1.7 or newer.

If incremental encoders with 1Vpp interface are used, the signal quality has a major impact on the reliability of the measurement system. If the signal levels are out of specification, this can lead to measurement errors, which are detected by the Irinos-System.

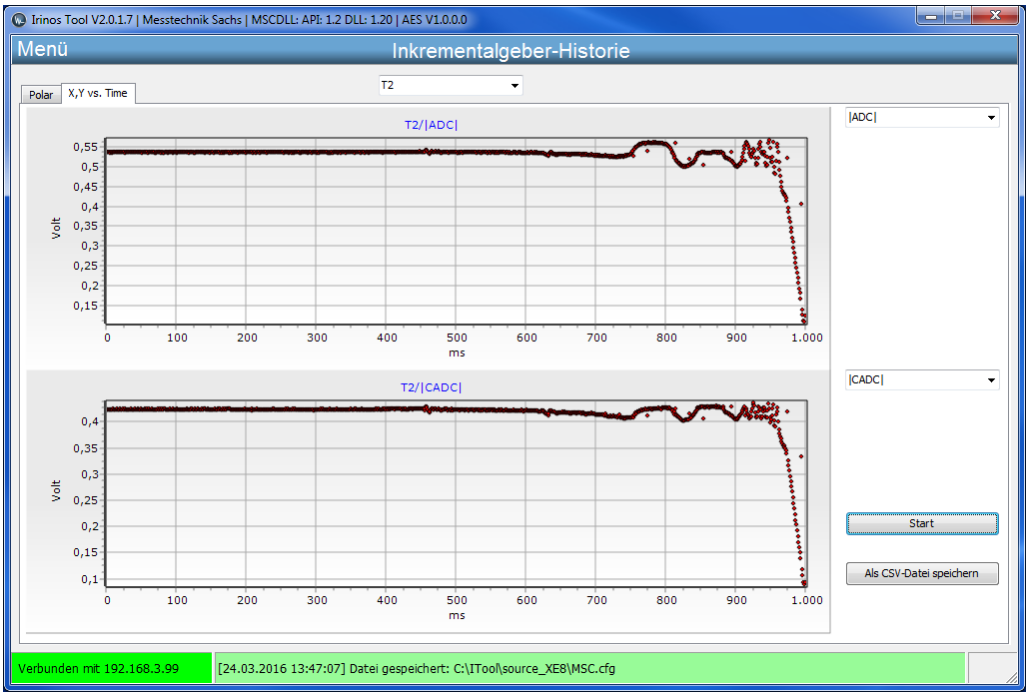
To get a better understanding, why an error occurred, the Irinos-System stores the last 1000 signal values (-> 1 second) before the error. These can be readout using the incremental encoder history of the Irinos-Tool. Like in the live view 124, they can be displayed in a Lissajous diagram. Alternatively they can be displayed over time.

The following screenshot shows an example of a Lissajous diagram (tab "Polar"). Similar to the live view, the measurement values are coloured according to the signal quality. Green ones are very good. If the signal quality becomes worse, they are coloured in yellow, orange or red (worst).

In the example, the final 16 values are displayed in red, showing a very poor signal quality.



Another possibility is displaying the values over time:



Two vertically arranged diagrams allow for a time-based correlation of different values. Via selection menus besides the diagrams, the values to be displayed can be selected:

- Absolute value of the ADC-Signals (corresponds to the signal vector)

- Absolute value of the corrected ADC signals (as shown in the example above)
- Sine value of the ADC signals
- Cosine value of the ADC signals
- Corrected sine value
- Corrected cosine value

In addition to this, the error flags can be displayed. The different error flags are listed in the live-view 124 chapter.

NmxDLL Reference Guide

4 NmxDLL Reference Guide

4.1 Introduction

4.1.1 Imprint

Title	Nmx DLL reference manual
Provided by	Messtechnik Sachs GmbH Siechenfeldstraße 30/1 D-73614 Schorndorf Germany Phone +49 7181 26935-0 post@messtechnik-sachs.de
For use with	Irinos Measurement modules, Type IR and EC, Firmware version 2 and upwards
Copyright note	© 2019-2020 Messtechnik Sachs GmbH Messtechnik Sachs GmbH
Trademarks	All product names used in this manual are trademarks of their respective owners.
Change note	Subject to change without notice.
Release date	05.06.2020

4.1.2 Revision history

Version	Datum	Changes
A	2019-05-05	First revision
B	2019-	Documentation of .Net-DLL 152 added.

	05-24	
C	2020-05-08	New: High-Level sampling, type TFT 232 *Minimum DLL version: 1.1.0.11*
D	2020-06-05	New: Position triggered sampling 229 *Minimum DLL version: 1.2.0.13* In addition, some minor typos have been corrected.

4.1.3 Legal notes

4.1.3.1 Terms of use for documentation & software

I. Protection rights and scope of use

Messtechnik Sachs provides operating instructions, manuals, documentation, and software programs - all collectively referred to as "LICENSED OBJECT" below - either on portable data storage devices (e.g. diskettes, CD ROMs, DVDs, etc.), in written (printed) form or in electronic form, for a fee and/or free of charge. The LICENSED OBJECT is subject to proprietary safeguarding provisions among other regulations. Messtechnik Sachs or third parties have protection rights for this LICENSED OBJECT. In so far as third parties have whole or partial right of access to this LICENSED OBJECT, Messtechnik Sachs has the appropriate rights of use. Messtechnik Sachs permits the user the use of the LICENSED OBJECT under the following conditions:

1.1) Scope of use for electronic documentation

a) With the acquisition/purchase or relinquishment of a LICENSED OBJECT, you as the user acquire a simple, non-transferable right of use with regard to the respective LICENSED OBJECT. This right of use authorises the user to use the LICENSED OBJECT for the user's own, exclusively company-internal purposes on any number of machines within the user's business premises. This right of use includes exclusively the right to save the LICENSED OBJECT on the central processors (machines) used at the location.

b) Irrespective of the form in which operating instructions and/or documentation are provided, the user may furthermore print out any number of copies on a printer at the user's location, providing this printout is printed with or kept in a safe place together with these complete terms and conditions of use and other user instructions.

c) With the exception of the Messtechnik Sachs logo, the user has the right to use pictures and texts from the operating instructions/documentation for creating the user's own machine and system documentation. The use of the Messtechnik Sachs logo requires written consent from Messtechnik Sachs.

The user is responsible for ensuring that the pictures and texts used match the machine/system or the product.

d) Further uses are permitted within the following framework:
Copying exclusively for use within the framework of machine and system documentation from electronic documents of all documented supplier components. Demonstrating to third parties exclusively under guarantee that no data material is stored wholly or partly in other networks or other data storage devices or can be reproduced there.
Passing on printouts to third parties not covered by the regulation in item 3, as well as any processing or other use are not permitted.

1.2) Scope of use for software products
For any type of Messtechnik Sachs software including the associated documentation, the customer shall receive a non-exclusive, non-transferable and time-unlimited right of use on a certain hardware product or on a hardware product to be determined in individual cases. Messtechnik Sachs shall remain the owner of the copyright as well as of any other industrial property rights. The customer may make copies for back-up purposes only. Any copyright notes may not be removed.

2. Copyright note
Every LICENSED OBJECT contains a copyright note. In any duplication permitted under these provisions, the corresponding copyright note of the original document concerned must be included:
Example: © 2019, Messtechnik Sachs GmbH,
D-73614 Schorndorf

3. Transferring the authorisation of use
The user can transfer the authorisation of use re. the respective LICENSED OBJECT as per these provisions in the scope and with the limitations of the conditions in accordance with items 1 and 2 completely to a third party. The third party must be made explicitly aware of these terms and conditions of use.

II. Exporting the LICENSED OBJECT
When exporting the LICENSED OBJECT or parts thereof, the user must observe the export regulations of the exporting country and those of the acquiring country.

III. Warranty
1. Messtechnik Sachs products are being further developed with regard to hardware and software. If the LICENSED OBJECT, in whatever form, is not supplied with the product, i.e. is not supplied on a data storage device as a delivery unit with the relevant product, Messtechnik Sachs does not guarantee that the electronic documentation corresponds to every hardware and software status of the product. In this case, the printed

documentation from Messtechnik Sachs accompanying the product is alone decisive for ensuring that the hardware and software status of the product matches that of the electronic documentation.
2. The information contained in an item of electronic documentation can be amended by Messtechnik Sachs without prior notice and does not commit Messtechnik Sachs in any way.
3. Messtechnik Sachs guarantees that the software program it created agrees with the change description and program specification but not that the functions included in the software run entirely without interruptions and errors or that the functions included in the software can run or meet the requirements in all combinations selected by and in all conditions of use designated by the acquirer.

IV. Liability/limitations on liability

1. Messtechnik Sachs provides LICENSED OBJECTS to allow the user to use - in conformity with the contract - Messtechnik Sachs products which require software for proper operation, or to assist the user in creating the user's machine and system documentation. In the case of electronic documentation which in the form of data storage devices does not accompany a product, i.e. which is not supplied together with that product, Messtechnik Sachs does not guarantee that the electronic documentation separately available/supplied matches the product actually used by the user.
The latter applies particularly to extracts of the documents for the user's own documentation. The guarantee and liability for separately available/supplied portable data storage devices, i.e. with the exception of electronic documentation provided on the Internet/Intranet, are limited exclusively to proper duplication of the software, whereby Messtechnik Sachs guarantees that in each case the relevant portable data storage device or software contains the latest status of the documentation. In respect of the electronic documentation on the Internet/Intranet, it is not guaranteed that this has the same version status as the last printed edition.
2. Furthermore, Messtechnik Sachs cannot be held liable for the lack of economic success or for damage or claims by third parties resulting from use of the LICENSED OBJECTS by the user, with the exception of claims arising from infringement of protection rights of third parties concerning the use of the LICENSED OBJECTS.
3. The limitations on liability as per paragraphs 1 and 2 do not apply if, in cases of intent or wanton negligence or lack of warranted quality, liability is absolutely necessary. In such a case, the liability of Messtechnik Sachs is limited to the damage recognisable by Messtechnik Sachs when the specific circumstances are made known.

V. Safety guidelines/documentation

Guarantee and liability claims in conformity with the regulations mentioned above (items III and IV) can only be made if the user has observed the safety guidelines of the

documentation in conjunction with use of the machine and its safety guidelines or the terms and conditions of use of the software. The user is responsible for ensuring that the electronic documentation, which is not supplied with the product, matches the product actually used by the user.

4.1.3.2 Qualified personnel

The product system described in this documentation must only be handled by qualified personal according to the given scope of work. All documentation relevant for the scope of work must be observed, especially the safety and warning notes. Due to its education and experience, qualified personal is able to identify risks and possible dangers when using this products / systems.

4.1.3.3 Disclaimer

The content of this documentation has been carefully reviewed to comply with the documented hard- and software. We can, however, not exclude discrepancies and do therefore not accept any liability for the exact compliance. This documentation is reviewed regularly. Corrections may be contained in newer versions.

4.1.4 Preface

4.1.4.1 Purpose

This reference manual describes the functions of the NMX DLL for the use with the measurement system. The target audience is software developers, who want to integrate the DLL into their application software (measurement software).

4.1.4.2 Scope of this reference manual

This reference manual is valid for the industrial measurement system Irinos together with the NMX DLL. The NMX software interface is supported from Firmware Version 2 and upwards.

4.1.4.3 Required knowledge

Profound knowledge in PC based software development using Windows is required for integrating and using the NMX DLL.

4.1.4.4 Further documentation

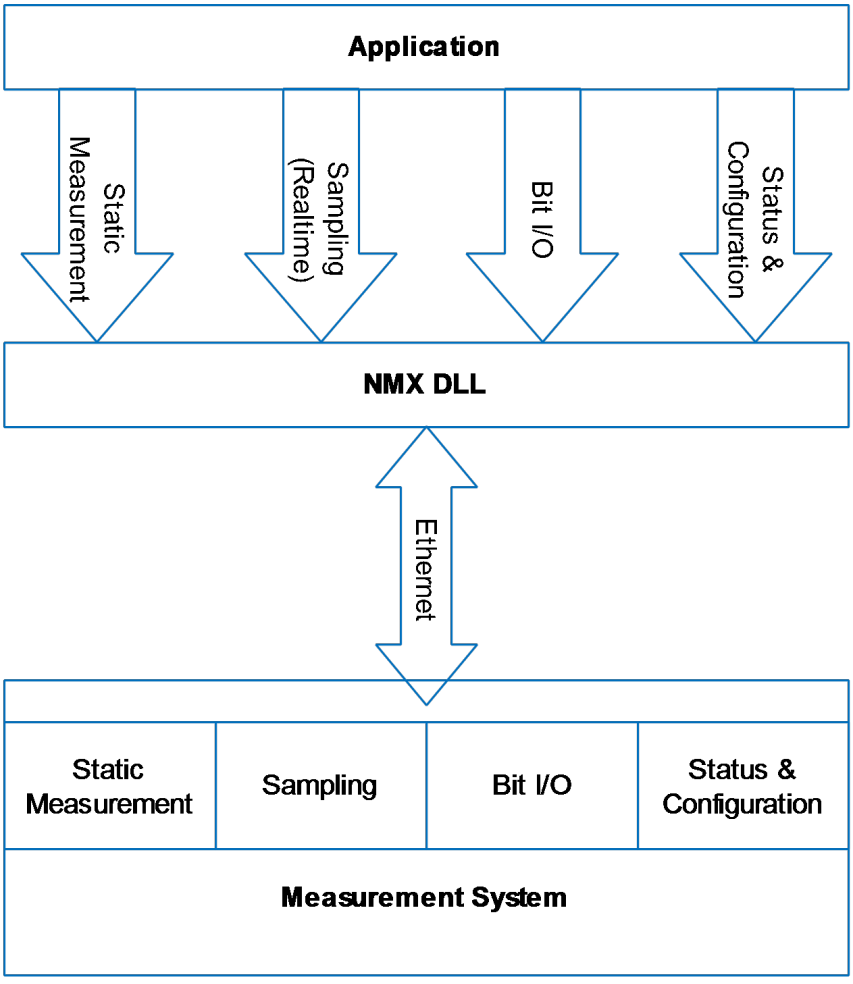
This reference manual is limited to the software interface of your measurement system.

For information about the measurement system, please consult its user manual and technical specification.

4.2 Nmx DLL Overview

The NMX DLL is the link between the application software (measurement software) and the Irinos-System. It supports the application in the following fields:

- Read measurement values from the measurement system. Reading static or sampled 140 data is available.
- Read status information from the measurement system.
- Read / write digital in-/output data.
- Parametrize the measurement system.



The DLL provides a wide range of function calls 153. These allow for a flexible use of the measurement system.
For small systems with limited requirements, just a few of these functions are required.

In most use cases only one measurement system is connected. In this case only one connection is required.
For very special applications, multiple measurement systems can be connected. In this case up to 8 measurement systems can be connected in parallel. These are distinguished by their "Handle".

4.2.1 Static vs. Sampling

Two different types of getting measurement and I/O data are available, "static" and "sampling". Both can be used in parallel.

Static data acquisition is very easy to use and the perfect choice for many applications. It is always active. The typical use case are measurement applications, where the working piece is fixed ("static") during measurement.

The measurement and I/O data is updated continuously with an update rate of approximately 30 Hz. The data is neither synchronized nor transferred in realtime. The update rate is not guaranteed and can be longer due to data packet loss.

It is started automatically after connecting to a device. The NMX DLL then periodically request new data from the device. This data is copied into an internal buffer. It can be readout at any time.

Often the static measurement is also used to achieve an online-view of the current measurement values.

Sampling is used to gather synchronized real-time data from the device. The typical use case are dynamic measurement applications, where high speed and determinism are required, e.g. for getting measurement curves.

It can be limited in time or it can be endless. Once started, the measurement system (device) acquires the measurement and digital I/O data in realtime and stores it into the internal device buffer. From there it is transferred in packets to the NMX DLL without real-time requirements.

Before starting sampling, various parameters must be defined, like the measurements channels used, the digital in-/outputs used and the sample period. Depending on the amount of sampled data and the sample period, the sampling must be time limited or can be endless. Consult the user manual / data sheet of your measurement system to see the possibilities. Typically 1000 Samples/s are possible with a large number of measurement channels.

The NMX DLL supports two general types of sampling:

- **Low-Level sampling** offers the possibility to perform time-triggered measurements. It provides the highest flexibility.

- **High-Level sampling** provides a simplified interface for performing complex real-time measurement tasks, e.g. position-triggered sampling. Internally it always uses Low-Level sampling. Therefore it is always optional and the high-level routines could also be implemented into the application software directly.

4.2.2 Sampling Speed with Irinos

For all Irinos systems, the real-time capability is independent of the number of probes (measurement channels), since each Irinos box has its own measurement value buffer. With the

- **Irinos IR**, most boxes are able to acquire **20000 samples/s**, and with the
- **Irinos EC**, most boxes are able to acquire **4000 samples/s**.

Two quick rules of thumb for 99% of applications are:

- **If you limit the sampling speed to 1000 samples/s, there is no need to think about any details. Or:**
- **If you limit the sampling speed to 10000 samples/s and if you have a maximum of 4 Boxes, there is no need to think about any details.** (With the Irinos EC, the max. speed still is 4000 samples/s.)

The sampling period used with the NMX DLL must be an integer multiple of the minimum sampling period of the system. **The following table lists typical sampling speeds:**

Sampling Speed [samples/s]	**Sampling Period**	**Irinos IR**	**Irinos EC**
20.000	50 µs	OK	not supported
10.000	100 µs	OK	not supported
6.666	150 µs	OK	not supported
5.000	200 µs	OK	not supported
4.000	500 µs	OK	OK

2.000	500 µs	OK	OK
1.000	1000 µs = 1 ms	OK	OK
500	2000 µs = 2 ms	OK	OK
200	5000 µs = 5 ms	OK	OK
100	10000 µs = 10 ms	OK	OK

The memory is dimensioned so that the measured values can be **buffered for at least 10 seconds at the maximum sampling rate**. If the sampling rate is lower, this time increases accordingly.

Only the transmission time to the PC depends on the number of channels and the measuring rate. Since the data transmission to the PC and the PC itself do not have realtime capabilities, no guaranteed transmission time can be guaranteed. However, typically a transmission rate can be achieved, which is almost constant and thus close to realtime.

For time-limited real-time measurement, the transmission rate is relevant for calculating the typical time between "start of sampling" and "all measurement data available on the PC". This time is called "transfer time".

For endless measurement, the transmission rate is relevant for determining the maximum possible sampling speed.

Typically achievable transmission rates are listed in the following table:

Irinos System	**Typical transmission rate R_{TR-32} with 32 Bit measurement channels, e.g. Incremental probes**	**Typical transmission rate R_{TR-16} with 16 Bit measurement channels e.g. Inductive probes**

Irinos IR	approx. 200.000 values/s	approx. 400.000 values/s
Irinos EC	approx. 80.000 values/s	approx. 160.000 values/s

As shown in the table, the transmission rate also depends on the native data type(s) of the measurement channels used. If mixed types of measurement channels are used, then the typical transmission rate will be in between the given values.

Some examples are provided here to give a quick guidance. For a detailed examination, calculation formulas are given below.

Examples for Transfer Time for a time-limited sampling with a duration (Start -> Stop) of 10 seconds:

Configuration	Irinos IR	Irinos EC
32 Inductive Probes (16 Bit)	10000 samples/s: 10s + 0s = 10s -> Similar to realtime	4000 samples/s: 10s + 0s = 10s -> Similar to realtime
32 Incremental Probes (32 Bit)	10000 samples/s: 10s + 6s = 16s -> approximately 6 seconds after stop, the transfer is finished	4000 samples/s: 16s + 6s = 16s -> approximately 6 seconds after stop, the transfer is finished
32 Inductive Probes (16 Bit) **+ 4 Incremental Probes** (32 Bit)	10000 samples/s: 10s + 0s = 10s -> Similar to realtime	4000 samples/s: 10s + 0s = 10s -> Similar to realtime

16 Inductive Probes (16 Bit) **+ 16 Incremental Probes** (32 Bit)	10000 samples/s: 10s + 2s = 12s -> approximately 2 seconds after stop, the transfer is finished	4000 samples/s: 10s + 2s = 12s -> approximately 2 seconds after stop, the transfer is finished

Examples for maximum sampling speed for endless sampling:

Configuration	Irinos IR	Irinos EC
32 Inductive Probes (16 Bit)	Theoretical maximum: 12500 samples/s Next possible value: 10000 samples/s Recommended: ≤ 6666 samples/s	Theoretical maximum: 5000 samples/s Next possible value: 4000 samples/s Recommended: ≤ 2000 samples/s
32 Incremental Probes (32 Bit)	Theoretical maximum: 6250 samples/s Next possible value: 5000 samples/s Recommended: ≤ 4000 samples/s	Theoretical maximum: 2500 samples/s Next possible value: 2000 samples/s Recommended: ≤ 1000 samples/s
32 Inductive Probes (16 Bit) **+ 4 Incremental Probes** (32 Bit)	Theoretical maximum: 10000 samples/s Next possible value: 10000 samples/s Recommended: ≤ 6666 samples/s	Theoretical maximum: 4000 samples/s Next possible value: 4000 samples/s Recommended: ≤ 2000 samples/s

16 Inductive Probes (16 Bit) **+ 16 Incremental Probes** (32 Bit)	Theoretical maximum: 8333 samples/s Next possible value: 6666 samples/s Recommended: ≤ 5000 samples/s	Theoretical maximum: 3333 samples/s Next possible value: 2000 samples/s Recommended: ≤ 2000 samples/s
64 Inductive Probes (16 Bit)	Theoretical maximum: 6250 samples/s Next possible value: 5000 samples/s Recommended: ≤ 4000 samples/s	Theoretical maximum: 2500 samples/s Next possible value: 2000 samples/s Recommended: ≤ 1000 samples/s

Formulas

Following a few formulas are provided to get estimations. These formulas use the following variables:

Variable	Meaning	Unit
$t_{Transfer}$	Transfer Time: Time between "Start of sampling" and "All measurement data available on the PC"	s -> seconds
$t_{Sampling}$	Sampling Time: Time between "Start" and "Stop" of sampling	s -> seconds

vs_{Max}	Maximum sampling speed for endless sampling	values / s / channel --> same as: samples / s
R_{TR}	Typical Transmission rate, see table above.	values / s
N_{MCH}	Number of measurement channels used	channels
N_{MCH-16}	Number of 16 bit measurement channels used	channels
N_{MCH-32}	Number of 32 bit measurement channels used	channels
$N_{Samples}$	Total number of samples to be recorded	values / channel

Each formula is provided in a simplified form and in a detailed form.
The simplified form is sufficient for most applications as a quick check.
The detailed form is especially used, when very high performance is required.

Required value	Simplified	Detailed
Transfer time for time-limited sampling (Time between "start of sampling" and "all measurement data is available at the PC")	**Formula 1:** $t_{Transfer} = \frac{N_{Samples} * N_{MCH}}{R_{TR-32}}$ *Note:* $t_{Transfer}$ *is always ≥* $t_{Sampling}$	**Formula 2:** $t_{Transfer} = \frac{N_{Samples} * (N_{MCH-16} + 2 * N_{MCH-32})}{R_{TR-16}}$ *Note:* $t_{Transfer}$ *is always ≥* $t_{Sampling}$

Max. sampling speed for endless sampling	Formula 3: $vs_{Max} = \frac{R_{TR-32}}{N_{MCH}}$	Formula 4: $vs_{Max} = \frac{R_{TR-16}}{N_{MCH-16} + 2 * N_{MCH-32}}$

Example for Formula 1:

In the measurement application with the Irinos IR, 21 inductive probes, 4 analogue channels and 3 incremental probes/encoders are used. The measurement has a duration of 5 seconds at 10000 samples/s.

N_{MCH} = 21 + 4 + 3 = 28 channels

NSamples = 5s * 10000 values/s/channel = 50000 values/channel.

R_{TR-32} = 200000 values/s

$$t_{Transfer} = \frac{50000\ {}^{values}/_{channel} * 28\ channels}{200000\ {}^{values}/_{s}} = 7s$$

--> 2 seconds after stop of sampling, all data is available.

Example for Formula 2:

In the measurement application with the Irinos IR, 21 inductive probes, 4 analogue channels and 3 incremental probes/encoders are used. The measurement has a duration of 5 seconds at 10000 samples/s.

N_{MCH-16} = 21 + 4 = 25 channels

N_{MCH-32} = 3 channels

NSamples = 5s * 10000 values/s/channel = 50000 values/channel.

R_{TR-16} = 400000 values/s

$$t_{Transfer} = \frac{50000\ {}^{values}/_{channel} * (25\ channels + 2 * 3\ channels)}{400000\ {}^{values}/_{s}} = 3{,}875s$$

--> Since $t_{Transfer} < t_{Sampling}$, all data is available immediately after stop of sampling.

Example for Formula 3:

In the measurement application with the Irinos EC, 11 inductive probes + 1 incremental encoder are used.

N_{MCH} = 11 + 1 = 12 channels

R_{TR-32} = 80000 values/s

$$vs_{Max} = \frac{80000\ {}^{values}/_{s}}{12\ channels} = 6666\ {}^{\frac{values}{s}}/_{channel}$$

--> *The maximum speed of the Irinos EC, which is 4000 samples/s, can be used.*

Example for Formula 4:
In the measurement application with the Irinos IR, 31 inductive probes + 6 incremental encoders are used.

N_{MCH-16} = 31 channels

N_{MCH-32} = 6 channels

$$vs_{Max} = \frac{400000\ {}^{values}/_{s}}{31\ channels + 2 * 6\ channels} = 9302\ {}^{\frac{values}{s}}/_{channel}$$

--> *5000 samples/s can be used as maximum sample rate.*

4.2.3 Data Types

The NMX DLL provides all

- measurement values as signed 32 bit values ("signed long") and all
- digital in-/outputs as unsigned 8 bit values ("unsigned char").

The native format of the measurement values can be different. For inductive probes the native format is for example signed 16 bit ("signed short"). To simplify the DLL interface, this native format is converted / "casted" to the common 32 Bit format within the DLL.

Example: An inductive probe has the value -9152, which is 0xDC40. This 16 Bit value is then converted to the 32 Bit value 0xFFFFDC40, which is also -9152.

If required, the native data type of a measurement channel can be retrieved from the NMX DLL using the function NMX_GetChannelInfo_1 190. However, in most cases the conversion factor to a physical unit is of interest and not the data type.

Digital in-/outputs are always transferred byte-wise, which means that each 8 in-/outputs are represented by 1 byte.

Example: A system has 32 digital inputs. These are represented by 32 / 8 = 4 input bytes, whereas

- Byte 0 contains the inputs 1..8
- Byte 1 contains the inputs 9..16
- etc.

4.2.4 Technical Background

The NMX DLL uses the Windows API functions for IP based communication, thread management and timing.
Inside the NMX DLL separate threads are running, which control the communication and the notification 176. The NMX DLL functions provide data to these threads and vice versa.

Some of the threads use normal thread priority while others use highest thread priority.

Please note: Even though the NMX DLL is multi-threaded, its function calls are not thread safe against each other. This means that all DLL functions must be called from within the same thread!

Communication to the measurement system is based on UDP/IPV4. The DLL automatically retransmits a data packet, if it has been lost. A direct ethernet connection between the measurement system and the PC is advised. Complex network structures, e.g. routing, tunneling, VPN, etc. are not supported due to timing efficiency. If you use these, you do it on your own risk.

The NMX DLL has been designed in C++ (VS2017) and uses the **"stdcall" calling convention**. A C based header file is provided.

Example applications for various programming environments are provided. It is good practice to start with one of these.

4.2.5 Limitations

The NMX DLL has several limitations. These have been selected such that in almost all cases they are of theoretical nature.

Note that your measurement system may have limitations below those of the NMX DLL.

The NMX DLL limitations are:

- Max. number of measurement boxes: 64
- Max. number of measurement channels: 256
- Max. number of sampling elements: 512
- Max. number of handles (-> simultaneous connections): 8

4.2.6 Hardware Requirements

The NMX DLL has no special hardware requirements.

Practical tests have shown that the **CPU load is almost negligible** and all of today's CPUs are sufficient.
To give an example: using the Visual C++ demo application, a 2 weeks sampling at 1kSample/s using 64 measurement channels consumed a total CPU time of 22 seconds (CPU i5-6300U). Usually the CPU performance required for the measurement application is far higher than for the DLL.

The **memory usage** mainly depends on the buffer, which is required for sampling. This means, it is defined by you (see NMX_Sampling_PrepareTime_1 218). **For typical applications, the memory consumption is very low (< 10 MBytes).**

4.2.7 Versions

The version number of the NMX DLL consists of 4 parts, which are separated by a dot, e.g. V1.3.0.12. The meaning of the parts is:

Part of the version number	Meaning
1. Part, in the example: 1	"Major" version number It is incremented, if the NMX DLL is completely redesigned (-> happens seldom).
2. Part, in the example: 3	"Minor" version number It is incremented, if new functionality has been implemented.
3. Part, in the example: 0	"Patch" It in incremented, if one ore more bugs have been fixed.
4. Part, in the example: 12	"Build" Internal number for revision identification.

To ensure a long-term backward compatibility, newer versions may have additional functionality, but existing functionality remains unchanged (except bug-fixes). Therefore each function call has a separate suffix at its end: _1

An example is NMX_StaticGet32_**1**.

If a newer version of an existing function is implemented, a copy of the existing function call is made an the suffix is incremented. Following the example, a newer version would be named NMX_StaticGet32_**2**, whereas the existing function call remains unchanged.

It is strongly recommended to check for a minimum DLL version after startup of your measurement software (especially the Major and Minor part). Use NMX_GetDllVersion_1 164 for this purpose.

The NMX DLL may support functionality, which is not supported by the measurement system ("device"). This can be either since it is limited in functionality or since its firmware version is outdated. The function call will then return with the return code 160 "NST_REQ_VERS_NOSUPPORT".

4.2.8 INI-File

Via an INI-File NmxDLL.ini, the behaviour of the NMX DLL can be modified. Currently the following modifications are possible:

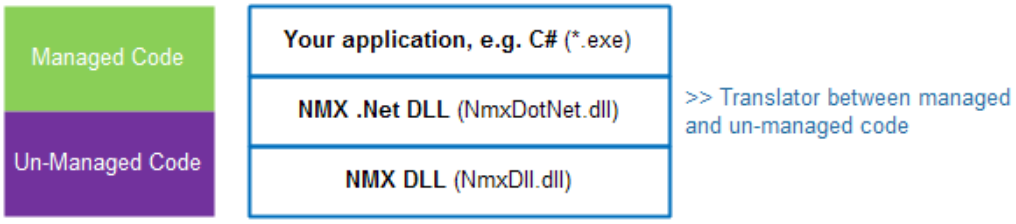
- Enable writing a log file NmxDLL.log.
- Change timing settings.

This INI-File should only be used if advised by the support.

4.2.9 .NET Wrapper DLL

For .Net based environments, like for example C# or VB.Net, a wrapper DLL is available (NmxDotNet.dll).

This DLL provides managed .Net function calls for your application. Internally, it converts between the managed .Net-World and the unmanaged native code of the NMX DLL:



In addition, the .Net DLL is much more convenient to use. There is for example no need to declare any function prototypes, since these are already embedded into the DLL.

Please note:

- The wrapper DLL uses the standard NMX DLL. Therefore both DLLs (NmxDLL.dll and NmxDotNet.dll) must be placed in the same folder.
- Since the wrapper DLL calls native code, it is "unsafe code" from the .Net perspective.
  By default, unsafe code is not allowed in .Net applications. This setting

must be changed in the "Build" section of your project properties, to allow unsafe code.

- The API of the wrapper DLL is slightly different to the native DLLs API. For example arrays are handled differently. However, the basic concept is the same.

4.3 API (programming interface)

4.3.1 Function calls overview

The following table provides a list of all function calls supported by the NMX DLL.

Don't be frightened by the long list of functions. Small measurement applications may need just a few of these function calls. See the HowTo "Small Measurement Application 239" in this manual.

There is quite a simple rule: the more sophisticated your measurement application is, the more function calls you will need.

Category	Function call	Description	Min. DLL version
Miscellaneous	NMX_GetDllVersion_1 164	Get version of the NMX DLL.	V1.0.0.0
	NMX_SystemReset_1 165	Reset the device via Software.	V1.0.0.0
Connecting / Disconnecting	NMX_DeviceIPv4Open_1 173	Establish a connection to the device via IPV4.	V1.0.0.0
	NMX_DeviceClose_1 175	Close a connection to a device.	V1.0.0.0

Category	Function call	Description	Min. DLL version
Notifications	NMX_RegisterMessage_1 178	Register a notification message.	V1.0.0.0
	NMX_RegisterCallback_1 180	Register a notification callback.	V1.0.0.0
Get device information	NMX_GetBoxCount_1 183	Get number of measurement boxes available.	V1.0.0.0
	NMX_GetBoxInfo_1 184	Get information about a measurement box (digital type plate).	V1.0.0.0
	NMX_UpdateChannelInfo_1 188	Re-Read measurement channel information.	V1.0.0.0
	NMX_GetChannelCount_1 189	Get number	V1.0.0.0

Category	Function call	Description	Min. DLL version
		of measurement channels available.	
	NMX_GetChannelInfo_1 190	Get information about measurement channel (digital type plate).	V1.0.0.0
	NMX_GetDigitalInputInfo_1 195	Get information about digital input byte.	V1.0.0.0
	NMX_GetDigitalOutputInfo_1 197	Get information about digital output byte.	V1.0.0.0
Static measurement (Non-Realtime)	NMX_StaticGet32_1 198	Read • static measurement values, • digital input data, • hardware	V1.0.0.0

Category	Function call	Description	Min. DLL version
		status and • current box events.	
	NMX_StaticSetMedianDepth_1 204	Set median filter for static measurement values.	V1.0.0.0
	NMX_SetOutputs_1 205	Set digital outputs.	V1.0.0.0
	NMX_DisableOutputUpdate_1 206	Disable updating digital outputs.	V1.0.0.0
	NMX_DigitalIoConfig_1 207	Configure digital I/O.	V1.0.0.0
	NMX_DigitalOutputsGetState_1 208	Get current state of digital outputs.	V1.0.0.0
Sampling LowLevel (Time-Triggered Realtime	NMX_Sampling_GetMaxSpeed_1 209	Get maximum sampling speed.	V1.0.0.0

Category	Function call	Description	Min. DLL version
Measurement)	NMX_Sampling_Reset_1 210	Reset sampling.	V1.0.0.0
	NMX_Sampling_AddChannelsAll_1 211	Add all measurement channels to the list of sampling elements.	V1.0.0.0
	NMX_Sampling_AddChannel_1 212	Add a single measurement channel to the list of sampling elements.	V1.0.0.0
	NMX_Sampling_AddDigiInAll_1 214	Add all digital input bytes to the list of sampling elements.	V1.0.0.0
	NMX_Sampling_AddDigiInByte_1 215	Add a single digital input byte to the list of sampling	V1.0.0.0

Category	Function call	Description	Min. DLL version
		elements.	
	NMX_Sampling_AddDigiOutAll_1 216	Add all digital output bytes to the list of sampling elements.	V1.0.0.0
	NMX_Sampling_AddDigiOutByte_1 217	Add a single digital output byte to the list of sampling elements.	V1.0.0.0
	NMX_Sampling_PrepareTime_1 218	Prepare sampling.	V1.0.0.0
	NMX_Sampling_Start_1 220	Start sampling.	V1.0.0.0
	NMX_Sampling_Stop_1 221	Stop sampling.	V1.0.0.0
	NMX_Sampling_ReadColumn32_1 221	Read sampled data "column-wise".	V1.0.0.0
	NMX_Sampling_ReadRow32_1 224	Read sampled	V1.0.0.0

Category	Function call	Description	Min. DLL version
		data "row-wise".	
	NMX_Sampling_GetStatus_1 226	Get current sampling status.	V1.0.0.0
Sampling HighLevel (Application specific Realtime Measurement)	NMX_Sampling_PrepareCustomTFT_1 232	Prepare application specific sampling of the type "Trigger + Filter + Tail".	V1.1.0.11
Diagnostics	NMX_DiagClearEvent_1 235	Clear event at Box.	V1.0.0.0
	NMX_DiagGetEventText_1 236	Get text describing the event.	V1.0.0.0
	NMX_SetDateTime_1 238	Set current date & time.	V1.0.0.0

4.3.2 Function Return Codes (NMX_STATUS)

The function calls of the NMX DLL almost all have the same return value of the type NMX_STATUS. Most of the return values will rarely occur. The most common ones are shown in bold in the following table.

Note for the .Net DLL 152: In the .Net DLL, the return values are represented by the enum type NMX_**M**STATUS instead by a binary value. Therefore NST_SUCCESS is for example `NMX_MSTATUS.SUCCESS`.

The return values are defined as follows:

Return value	Hex representation	Description	Possible reasons (excerpt)
NST_SUCCESS	0x00000000	**Everything OK.**	
NST_HANDLE_INVALID	0xF0000000	Invalid handle.	• Connection has not been established yet. • Connection has already been closed.
NST_HANDLE_TOO_MANY	0xF0000001	Too many handles exists.	Too many connections have been opened. The limit is 8.
NST_CONNECT_OPEN_FAILED	0xF0000010	**Failed connecting to a device.**	• **Device is not connected to the PC.** • **Network settings (IP address & Port numbers) are invalid.**
NST_NOT_CONNECTED	0xF0000011	No connection established.	
NST_NOT_AVAILABLE	0xF000001F	The requested data is not available.	No connection established or invalid data received from the device.
NST_SEND_SIZE_TOO_LARGE	0xF0000020	Too much data to send.	
NST_DX_TIMEOUT_1	0xF0000021	**A data exchange**	• **The network connection has been interrupted shortly or permanently.** • **PC timing is too slow, e.g.**
NST_DX_TIMEOUT_2	0xF0000022		

4.3.3 Connection Handle

The NMX DLL allows having multiple connections to different devices, even though in most applications one connection to one device is used.

In order to distinguish between multiple connections, a unique handle is assigned for each of them by the NMX DLL.

Technically spoken, a handle is nothing else than a pointer to an address space managed by the NMX DLL. This is a common programming technique.

The pointer itself resides outside the NMX DLL in the user application. It is assigned if a connection has been established successfully (see NMX_DeviceIPv4Open_1 173). It is then used in almost every DLL function, to identify the relevant connection. It is deleted by closing the connection (see NMX_DeviceClose_1 175).

The Handle has the following type definition (C-Code):

```
typedef void* NMX_PHANDLE;
```

For each connection a handle must be declared. In case only one connection is used, the declaration looks as follows:

```
NMX_PHANDLE pHandle = NULL;
```

Note for the .Net DLL 152: Using the .Net DLL, the handle uses the type `System::IntPtr`. The basic concept is the same. In case only one connection is used, the declaration in C# looks as follows:

```
IntPtr pDevice = IntPtr.Zero;
```

Further information, if you use multiple connections / handles:

Inside the NMX DLL, each connection / handle has its own memory and its own communication tasks. Talking in object orientated programming, a separate object is created for each handle.

If notifications 176 are used, these must be registered for each handle separately.

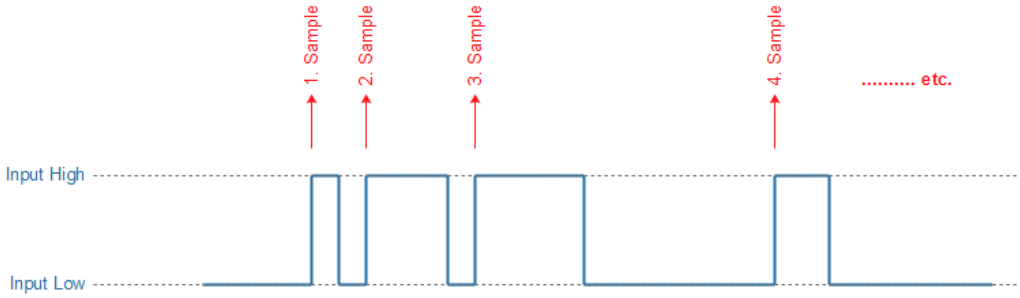
4.3.4 Trigger Modes

Certain sampling functions allow using a digital input for triggering. The following Trigger-Modes are implemented.

Please note that not all trigger modes are supported by every function.

Trigger Mode 1 = "Edge"

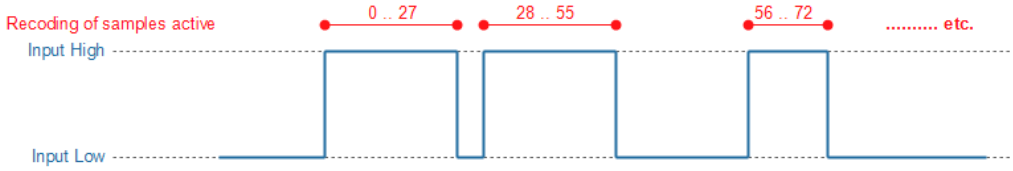
Using the edge trigger, exactly one sample is stored every time an edge of the digital input occurs. The following example shows this for "rising edge":



Please note that the maximum input frequency of the digital input must be ≤ sampling frequency. Otherwise samples could be lost.

Trigger Mode 2 = "Level"

Using the level trigger, measurement values are sampled while the digital input is high (for polarity "high") or low (for polarity "low"). The following example shows this for polarity "high":

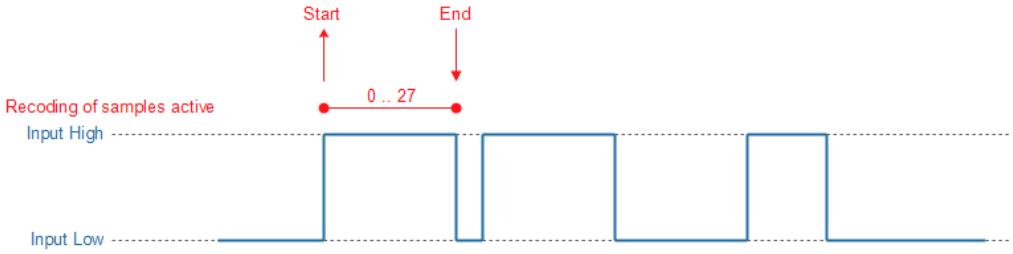


Trigger Mode 3 = "Edge Start"

Using the trigger "Edge Start", recording of sampled data is stored after the first edge of the digital input. Any further edges of the digital input have no effect. The following example shows this for "rising edge":

Trigger Mode 4 = "Level Once"

The trigger mode "Level Once" is similar to the "Level Trigger" (Mode 2), except that sampling is stopped automatically once the trigger condition becomes inactive. The following example shows this for polarity "high":



4.3.5 Miscellaneous

4.3.5.1 NMX_GetDllVersion_1

This function returns the current version of the NMX DLL.

Definition

```
void NMX_GetDllVersion_1(
      unsigned short* pusMajor,
      unsigned short* pusMinor,
      unsigned short* pusPatch,
      unsigned short* pusBuild);
```

Parameter

pusMajor

Major version of the NMX DLL.

pusMinor

Minor version of the NMX DLL.

pusPatch

Patch version of the NMX DLL.

pusBuild

Build number of the NMX DLL.

Typical function call (C example)

```
NMX_GetDllVersion_1(&usMajor, &usMinor, &usPatch, &usBuild);
```

.Net DLL 152 *specific implementation*

```
System::Void GetDllVersion_1(System::UInt16 %pusMajor,
System::UInt16 %pusMinor, System::UInt16 %pusPatch,
System::UInt16 %pusBuild);
```

Comments

See chapter "Versions 150" for more information.

4.3.5.2 NMX_SystemReset_1

This function allows restarting the whole measurement system (device). Use this function only if advised by the support.

Definition

```
NMX_STATUS NMX_SystemReset_1(
      NMX_PHANDLE pHandle,
      unsigned long ulDelayMaster,
      unsigned long ulDelaySlave);
```

Parameter

pHandle

Connection Handle 162.

ulDelayMaster

Time in ms, until the Master-Box (-> first measurement box) will be restarted.

ulDelaySlave

Time in ms, until all Slave-Boxes will be restarted.

Typical function call (C example)

```
NMX_SystemReset_1(pHandle, 2000, 500);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS SystemReset_1(System::IntPtr pHandle,
System::UInt32 ulDelayMaster, System::UInt32 ulDelaySlave);
```

Comments

The delay for the Master must be longer than the delay for the Slaves. Use the following values:

```
ulDelayMaster = 2000;

ulDelaySlave = 500;
```

The reset command is send to all Slave-Boxes via the Link interface between the boxes. If the Link interface does not work properly, no slave can be reset.

4.3.5.3 NMX_ChannelSetParameter_1

This function allows changing a channels parameters during operation. The parameters depend on the measurement channel type. It is used for selected measurement channels, e.g. for incremental encoders.

Definition

```
NMX_STATUS NMX_ChannelSetParameter_1(
      NMX_PHANDLE pHandle,
```

```
      unsigned long ulChannelNo,
      char* pcStringTx, unsigned long ulSizeofTxString,
      char* pcStringRx, unsigned long ulMaxSizeofRxString,
unsigned long* pulSizeofRxString);
```

Parameter

---

pHandle

Connection Handle 162

ulChannelNo

Number of the measurement channel.
The first channel has the number 0.
Having a device with 8 measurement channels, these are numbered 0..7.

pcStringTx

ASCII-encoded string, which shall be send to the device.

ulSizeofTxString

Size of pcStringTx in characters/bytes.

pcStringRx

Buffer for the response-string, which is received from the device.

ulMaxSizeofRxString

Size of the buffer pcStringRx in characters/bytes.

pulSizeofRxString

Length of the response-string, which has been received from the devices.

Typical function call (C example)

```
char acStrTx[] = "#0;REFOFF#";
NMX_ChannelSetParameter_1(pHandleNmx, 2, acStrTx,
strlen(acStrTx), acStrRx, sizeof(acStrRx), &ulRxSize);
```

Strings, which can be sent to incremental en-/decoder channels

---

`#{Position};{Reference index on/off}#`

Position

- New position value for this measurement channel. This allows setting the position of the measurement channel.
- `*` if the position of the measurement channel shall not be changed. This is required, if only the reference index shall be turned on or off.
- `~` in order to reset the gain- and offset-control of the measurement channel. The position value will be reset to 0. This makes only sense for 1Vpp channels.

Reference index on/off

Permissible parameter values:

- `REFON` to enable the reference index.
  In addition, the statusbit "Refmark" (see HardwareStatus 198) will be enabled now, if the reference index is crossed.
- `REFOFF` to disable the reference index.

If the reference index is enabled, the position of the measurement channel will be set to 0, if the index is crossed.

Response string from the measurement system / measurement channel for incremental encoder

---

`#0#` Success

`#-2#` Position parameter in string invalid

`#-3#` Position parameter in string invalid

`#-99#` General syntax error of the request string

Response string from the Irinos-System / measurement channel does not support this opcode

```
#-98#
```

Examples for request strings to incremental measurement channels

```
#-2000;REFOFF#
```

The current position of the selected measurement channel will be set to -2000. The reference index will be disabled.

```
#*;REFON#
```

The reference index of the selected measurement channel will be enabled. The position will not be changed.

```
#~;REFOFF#
```

The gain- and offset control of the selected measurement channel will be reset. The position will be set to 0. The reference index will be disabled.

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS ChannelSetParameter_1(System::IntPtr pHandle,
System::UInt32 ulChannelNo, System::String ^%strExchange);
```

- *Since strExchange is a managed string, no additional string length information must be provided in the function call.*
- *The string strExchange is send to the device.*
  On success, the received string will be returned via the same parameter strExchange.
- *Make sure to use only ASCII-characters. The conversion from unicode to ASCII is made within the wrapper DLL.*

Comments for incremental measurement channels

The error flags and the status bit "Refmark" will be cleared (these can be readout via the hardware status 198). Exception: This does not apply, if the character * is used for the parameter "position".

Notes for 1Vpp measurement channels:

If an error is detected at the 1Vpp-inputs, the signal levels of the incremental encoder are or have been out of specification. After an error has occurred, it is recommended to reset the gain- and offset-control by using the character ~ as the position parameter.

After resetting the gain- and offset-control, it takes a few signal periods until the control has determined the optimal parameters. During this process, the interpolation accuracy is limited. The measurement values can be inaccurate (however, no increments are lost). Further, the signal tolerance is limited during this process.

4.3.5.4 NMX_ChannelSetConfig_1

This function allows changing a channels parameters during operation. The parameters depend on the measurement channel type. It is used for selected measurement channels, e.g. for incremental encoders.

Definition

```
NMX_STATUS NMX_ChannelSetConfig_1(
      NMX_PHANDLE pHandle,
      unsigned long ulChannelNo,
      char* pcStringTx, unsigned long ulSizeofTxString,
      char* pcStringRx, unsigned long ulMaxSizeofRxString,
unsigned long* pulSizeofRxString);
```

Parameter

pHandle

Connection Handle 162.

ulChannelNo

Number of the measurement channel.
The first channel has the number 0.
Having a device with 8 measurement channels, these are numbered 0..7.

pcStringTx

ASCII-encoded string, which shall be send to the device.

ulSizeofTxString

Size of pcStringTx in characters/bytes.

pcStringRx

Buffer for the response-string, which is received from the device.

ulMaxSizeofRxString

Size of the buffer pcStringRx in characters/bytes.

pulSizeofRxString

Length of the response-string, which has been received from the devices.

Typical function call (C example)

```
char acStrTx[] = "#1VSS;1#";
NMX_ChannelSetConfig_1(pHandleNmx, 2, acStrTx, strlen(acStrTx),
acStrRx, sizeof(acStrRx), &ulRxSize);
```

Strings, which can be sent to incremental en-/decoder channels

```
#{Configuration type};{Store}#
```

Configuration type

„1VSS" to change an incremental channel type to 1Vpp.

„TTL" to change an incremental channel type to TTL / RS422 with 1x-decoding (1 increment = 1 digit).

„TTL4X" to change an incremental channel type to TTL / RS422 with 4x-decoding (1 increment = 4 digits).

Store

„0“: The change remains active until the next system restart. Afterwards the old configuration becomes active again.

„1“: The change applies permanently.

Response string from the measurement system / measurement channel for incremental encoder

---

#0# Success

#-2# Position parameter in string invalid

#-3# Position parameter in string invalid

#-99# General syntax error of the request string

Response string from the Irinos-System / measurement channel does not support this opcode

---

#-98#

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS ChannelSetConfig_1(System::IntPtr pHandle,
System::UInt32 ulChannelNo, System::String ^%strExchange);
```

- *Since strExchange is a managed string, no additional string length information must be provided in the function call.*
- *The string strExchange is send to the device.*
  On success, the received string will be returned via the same parameter strExchange.
- *Make sure to use only ASCII-characters. The conversion from unicode to ASCII is made within the wrapper DLL.*

Comments for incremental measurement channels

After configuration, the position of the incremental input channel is reset.

By reconfiguring an incremental input channel after start of the application software, it does not matter how a measurement channel is pre-configured. This allows a quick replacement of a measurement Box without manual reconfiguration of the "new" Box.

4.3.6 Connecting / Disconnecting

A connection to the device must be established before any other function will work properly (exception: NMX_GetDllVersion_1 164).

4.3.6.1 NMX_DeviceIPv4Open_1

This function is used to establish a connection to a device (measurement system) via IPV4.

See also the HowTo 239-Guide.

Definition

```
NMX_STATUS NMX_DeviceIPv4Open_1(
      unsigned char ucIp3,
      unsigned char ucIp2,
      unsigned char ucIp1,
      unsigned char ucIp0,
      unsigned short usPortCmd,
      unsigned short usPortData,
      NMX_PHANDLE *ppHandle);
```

Parameter

ucIp3, ucIp2, ucIp1, ucIp0

IP-Address of the device. The factory default address is 192.168.3.99. For using this address, the parameters are as follows:

```
ucIp3 = 192;
ucIp2 = 168;
ucIp1 = 3;
ucIp0 = 99;
```

usPortCmd

Port number for the DLLs communication channel, which is used for exchanging commands. Use 22517.

usPortData

Port number for the DLLs communication channel, which is used for exchanging data. Use 22516.

ppHandle

If the connection has been established successfully, a connection handle will be returned via this pointer. Otherwise the handle remains unchanged.

Typical function call (C example)

```
NMX_DeviceIPv4Open_1(192, 168, 3, 99, 22517, 22516, &pHandle);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS DeviceIPv4Open_1(
                  System::Byte ucIp3,
                  System::Byte ucIp2,
                  System::Byte ucIp1,
                  System::Byte ucIp0,
                  System::UInt16 usPortCmd,
                  System::UInt16 usPortData,
                  System::IntPtr %ppHandle);
```

Comments

- NMX_PHANDLE is a pointer to a handle (type void*). It is required for all subsequent connection calls.
- It is strongly recommended to read the IP-Address and the Port-Numbers from an INI-File, XML-File, Registry or something similar. Do not hard code these.
- If the connection is not established after having integrated this function into your application, first check the network connection to the device (e.g. by calling its WebServer or via ping).
- Before closing your application, always ensure that this connection is closed before via NMX_DeviceClose_1 175.

- Multiple connections can be established to different devices. In this case a separate handle is provided for each connection. However, this is very rarely required. Please read also the chapter about limitations 149.

4.3.6.2 NMX_DeviceClose_1

This function is used to close a connection to a device (measurement system).

See also the HowTo 241-Guide.

Definition

```
NMX_STATUS NMX_DeviceClose_1(
      NMX_PHANDLE *ppHandle);
```

Parameter

ppHandle

Pointer to Connection Handle 162.

Typical function call (C example)

```
NMX_DeviceClose_1(&pHandle);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS DeviceClose_1(System::IntPtr %ppHandle);
```

Comments

If the connection is not established, nothing happens except that NST_HANDLE_INVALID is returned.

An established connection must be closed before terminating your application. It is good practice calling this function in any case before terminating.

4.3.7 Notifications

Using notifications, the application can be informed about different events. The use of notifications is recommended, but not required.

The following notifications are available:

Category	Notification	Hex Value	Description	M
Connecting / Disconnecting	NMXNOTIFY_DI	0x000	The connection to the device (measurement system) has been closed. Prior to this event the function NMX_DeviceClose_1 175 has been called.	V
	NMXNOTIFY_FA	0x000	Permanent failure during data exchange (e.g. timeout). A typical reason is a broken network connection.	V
	NMXNOTIFY_RE	0x000	The connection has been re-established automatically. Prior to this event, the event NMXNOTIFY_FAILURE_DATA_EXCHANGE occurs.	V
Static measurement (Non-Realtime)	NMXNOTIFY_NI	0x000	The static data inside the DLL has been updated.	V

There are two technical ways for receiving notifications:

- Windows Messages, see NMX_RegisterMessage_1 178, or
- Callback Functions, see NMX_RegisterCallback_1 180.

If technically possible in your application, Messages are recommended. Use Callbacks, if it is difficult or impossible to read the Windows message queue.

Inside the NMX DLL, notifications are send/called from a separate thread. An advantage of this is, that the communication thread is not interrupted, if message or callback handling takes too much time.

Nevertheless it is very important that the notification itself and reading, processing and displaying data is handled in a separate threads. The following approach is suggested:

- In case a notification occurs, set a global flag.
- Check this flag cyclically in the same thread, which handles the DLL function calls. This cyclic check can for example be implemented into a 30ms timer-event of the GUI.

Further comments:

- All notifications are automatically removed, when a connection is closed. You will have to re-register them after establishing a new connection.
- It is possible to combine multiple notifications to the same Windows Message or Callback. Register each of the Notifications, which shall be combined, using the same Windows Message or Callback Function.
- If you have multiple connections, the notifications must be registered for each connection separately.

4.3.7.1 NMX_RegisterMessage_1

This function is used to register a notification message at the DLL.

Definition

```
NMX_STATUS NMX_RegisterMessage_1(
      NMX_PHANDLE pHandle,
      unsigned long ulNotification,
      HWND hWnd,
      unsigned long ulMsgCode,
```

```
    WPARAM tWParam,
    LPARAM tLParam);
```

Parameter

---

pHandle

Connection Handle 162.

ulNotification

Notification type 176, e.g. NMXNOTIFY_NEW_STATIC32.

hWnd

Window Handle of the Windows Window, which handles the message queue.

ulMsgCode

Message number, which must be defined by the application (see also comment below).

tWParam

The wParam of the Windows message.

tLParam

The lParam of the Windows message.

Typical function call (C example)

```
NMX_RegisterMessage_1(pHandle, NMXNOTIFY_NEW_STATIC32,
static_cast<HWND>(Handle.ToPointer()), WM_MESSAGE_NEW_STATIC32,
0, 0);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS RegisterMessage_1(
                    System::IntPtr pHandle,
                    NMX_NOTIFICATION eNotification,
                    System::IntPtr hWnd,
```

```
                    System::UInt32 ulMsgCode,
                    System::UInt32 tWParam,
                    System::UInt32 tLParam);
```

NMX_NOTIFICATION is an enum type, which is defined in the .Net - DLL. The available notifications are identical to the standard notifications[176], except that they don't have a fixed binary representation.

Comments

If hWnd is NULL, a registered message will be cleared.

Message Code (ulMsgCode)

The message-number is defined by your application. Using Visual C++, this can be done for example as follows:

```
#define WM_MESSAGE_DISCONNECTED                 (WM_USER +
NMXNOTIFY_DISCONNECTED)
#define WM_MESSAGE_FAILURE_DATA_EXCHANGE        (WM_USER +
NMXNOTIFY_FAILURE_DATA_EXCHANGE)
#define WM_MESSAGE_RECONNECTED                  (WM_USER +
NMXNOTIFY_RECONNECTED)
#define WM_MESSAGE_NEW_STATIC32                 (WM_USER +
NMXNOTIFY_NEW_STATIC32)
#define WM_MESSAGE_SAMPLING_NEWDATA             (WM_USER +
NMXNOTIFY_SAMPLING_NEW_DATA)
#define WM_MESSAGE_SAMPLING_ALLRECEIVED         (WM_USER +
NMXNOTIFY_SAMPLING_ALL_DATA_RECEIVED)
#define WM_MESSAGE_SAMPLING_FINISHED            (WM_USER +
NMXNOTIFY_SAMPLING_FINISHED)
#define WM_MESSAGE_SAMPLING_ERROR               (WM_USER +
NMXNOTIFY_SAMPLING_ERROR)
#define WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW     (WM_USER +
NMXNOTIFY_SAMPLING_BUFFER_OVERFLOW)
#define WM_MESSAGE_SAMPLING_TIMEOUT             (WM_USER +
NMXNOTIFY_SAMPLING_TIMEOUT)
```

4.3.7.2 NMX_RegisterCallback_1

This function is used to register a notification callback function at the DLL.

Definition

```
NMX_STATUS NMX_RegisterCallback_1(
```

```
    NMX_PHANDLE pHandle,
    unsigned long ulNotification,
    NMX_NOTIFICATION_CALLBACK* pCbFunction,
    void* pvContext);
```

Parameter

pHandle

Connection Handle 162.

ulNotification

Notification type 176, e.g. NMXNOTIFY_NEW_STATIC32.

pCbFunction

Pointer to callback function.

pvContext

A caller provided context pointer which is passed unchanged to the callback function.

Typical function call (C example)

```
RegisterCallback_1(pHandle, NMXNOTIFY_NEW_STATIC32,
&Callback_NewStatic32, (void*)&ulCallbackContext);
```

Comments

If pCbFunction is NULL, a registered callback will be cleared.

Callback function

This function is the prototype for a callback notification.

Definition

```
void NMX_CALLCONV NMX_NOTIFICATION_CALLBACK(IN void*
pvContext);
```

Parameter

---

pvContext

This parameter is the same which was passed to the function NMX_RegisterCallback_1. The application can store a context information in this pointer.

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS RegisterCallback_1(
                  System::IntPtr pHandle,
                  NMX_NOTIFICATION eNotification,
                  OnNotification^ NotificationDelegate);
```

NMX_NOTIFICATION is an enum type, which is defined in the .Net - DLL. The available notifications are identical to the standard notifications 176, except that they don't have a fixed binary representation.

In .Net, delegates are used instead of function pointers. The .Net - DLL calls this delegates in case of a notification.

In C#, an example for a delegate function is:

```
public void OnNewStaticData(IntPtr pHandle)
{
}
```

The handle of the calling DLL-instance is provided as a function parameter. If you have only one instance (= 1 connection to a device), don't care about it. This is the most common case.

In case you have multiple instances, the handle can be used to distinquish between multiple connections.

An example for registering the delegate is:

```
cNmx.RegisterCallback_1(pDevice, NMX_NOTIFICATION.NEW_STATIC32,
OnNewStaticData);
```

Note: Since the delegate is directly forwarded to the unmanaged DLL, the .Net wrapper fixes the delagte pointer in the garbage collector (gc.Alloc).

Comments

This function is called in a different thread context. The application must handle the synchronisation.

4.3.8 Get device information

Several functions are available to get information about the measurement system.

Using this information is optional.

The following is available:

- Information about measurement boxes 184 (digital type plate).
- Information about measurement channels 190 (type information and digital type plate of probe).
- Information about digital inputs 195 & outputs 197.

4.3.8.1 NMX_GetBoxCount_1

This function is used to read the number of measurement boxes, which the device has.

Definition

```
NMX_STATUS NMX_GetBoxCount_1(
      NMX_PHANDLE pHandle,
      unsigned long* pulBoxCount);
```

Parameter

pHandle

Connection Handle 162.

pulBoxCount

Number of measurement boxes, which the system has.

Typical function call (C example)

```
NMX_GetBoxCount_1(pHandle, &ulBoxCount);
```

.Net DLL [152] *specific implementation*

```
NMX_MSTATUS GetBoxCount_1(System::IntPtr pHandle,
System::UInt32 %pulBoxCount);
```

Comments

pulBoxCount only returns a value, if the function return code is NST_SUCCESS. Otherwise the value remains unchanged.

4.3.8.2 NMX_GetBoxInfo_1

This function is used to read information about a measurement box (digital type plate).

Definition

```
NMX_STATUS NMX_GetBoxInfo_1(
      NMX_PHANDLE pHandle,
      unsigned long ulBoxNo,
      unsigned long* pulInfoData, unsigned long
ulInfoDataNElements,
      unsigned long long* pudMacAddress,
      char* pcSerNo, unsigned long ulSizeofSerNo,
      char* pcProdCode, unsigned long ulSizeofProdCode,
      char* pcOrderNo, unsigned long ulSizeofOrderNo,
      char* pcName, unsigned long ulSizeofName);
```

Parameter

pHandle

Connection Handle [162].

ulBoxNo

Number of the measurement box, for which the information shall be provided.
The first measurement box has the number 0.
In a system with 5 measurement boxes, these are numbered 0..4.

pulInfoData

Pointer to an array of unsigned 32 Bit values.
To avoid a large amount of function parameters, several information will be stored in this array.
See below for more information about the array content.
An array size of 32 elements is recommended. Initialize these with 0.

ulInfoDataNElements

Number of elements of the array, to which pulInfoData points.
Example: The array has 32 elements, its size is 128 Bytes. Then:
`ulInfoDataNElements = 32;`

pudMacAddress

MAC address of the box.

pcSerNo

ASCII based string with serial number of the box. The maximum string length is 17 characters (16 + Termination).

ulSizeofSerNo

Maximum size of pcSerNo in Bytes/Characters.

pcProdCode

ASCII based string with production code of the box. The maximum string length is 17 characters (16 + Termination).

ulSizeofProdCode

Maximum size of pcProdCode in Bytes/Characters.

pcOrderNo

ASCII based string with order number of the box. The maximum string length is 33 characters (32 + Termination).

ulSizeofOrderNo

Maximum size of pcOrderNo in Bytes/Characters.

pcName

ASCII based string with name of the box. The maximum string length is 129 characters (128 + Termination).

ulSizeofName

Maximum size of pcName in Bytes/Characters.

Typical function call (C example)

```
NMX_GetBoxInfo_1(pHandle, ulBoxNo, aulInfoData,
sizeof(aulInfoData) / 4, &udMacAddress, acSerNo,
sizeof(acSerNo), acProdCode, sizeof(acProdCode), acOrderNo,
sizeof(acOrderNo), acName, sizeof(acName));
```

Content of the array pulInfoData

Element	Content	Description
0	Box number	Number of the box.
1	Hardware Version Ma	Version of the electronics.
2	Hardware Version Mir	
3	Hardware Revision	Compatibility code between the hardware and the firmware. It ensures that a firmware update is only allowed if the firmware version is compatible to the hardware revision.
4	Firmware Version Maj	Firmware version of the box. The first part of the firmware version is incremented in case of major changes. The second part of the firmware version is incremented in case new functionality has been implemented. The third part of the firmware version is incremented in case one or more bugs were fixes. The fourth part of the firmware version is an internal counter.
5	Firmware Version Min	
6	Firmware Version Patc	
7	Firmware Version Buil	
8	Number of measurem	Total number of measurement channels.
9	Number of 64 Bit mea	Number of 64 Bit measurement channels, which the box has.
10	Number of 32 Bit mea	Number of 32 Bit measurement channels, which the box has.
11	Number of 16 Bit mea	Number of 16 Bit measurement channels, which the box has.
12	Number of 8 Bit meas	Number of 8 Bit measurement channels, which the box has.
13	Number of digital inp	Total number of digital input bits. For data readout: • The number of digital inputs is always rounded up to a multiple of 8. If for example 2 digital inputs are available. These are rounded up to 8, whereas the inputs 3-8 are always low. • Each 8 Bits are combined in 1 Byte

.Net DLL [152] *specific implementation*

```
NMX_MSTATUS GetBoxInfo_1(
                    System::IntPtr pHandle,
                    System::UInt32 ulBoxNo,
                    array<System::UInt32>^aulInfoData,
                    System::UInt64 %pudMacAddress,
                    System::String ^%strSerial,
                    System::String ^%strProdCode,
                    System::String ^%strOrderNo,
                    System::String ^%strName);
```

No Length/Sizeof-Parameters are required for the array aulInfoData and the strings, since this information is not required in a .Net environment.

For aulInfoData a size of 32 elements is recommended. The strings strSerial, strProdCode, strOrderNo and strName are directly returned as Unicode-strings. Hence no additional conversion is required.

Comments

It is good practice reading the box count [183] first. Then read the box information for each box.

The function only returns values, if the function return code is NST_SUCCESS. Otherwise the values remain unchanged.

4.3.8.3 NMX_UpdateChannelInfo_1

This function forces the NMX DLL to re-read measurement channel information..

Definition

```
NMX_STATUS NMX_UpdateChannelInfo_1(
      NMX_PHANDLE pHandle);
```

Parameter

pHandle

Connection Handle [162].

Typical function call (C example)

```
NMX_UpdateChannelInfo_1(pHandle);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS UpdateChannelInfo_1(System::IntPtr pHandle);
```

Comments

The NMX DLL reads the channel information from the device once after connecting. In case the information at the device side changes afterwards, e.g. due to connecting a different probe, the data within the NMX DLL is outdated.

Call this function to update this data.

4.3.8.4 NMX_GetChannelCount_1

This function is used to read the number of measurement channels and the number of digital in-/outputs, which the device has.

Definition

```
NMX_STATUS NMX_GetChannelCount_1(
      NMX_PHANDLE pHandle,
      unsigned long* pulChannelCount,
      unsigned long* pulNDigitalInputBytes,
      unsigned long* pulNDigitalOutputBytes);
```

Parameter

pHandle

Connection Handle 162.

pulChannelCount

Number of measurement channels, which the system has.

pulNDigitalInputBytes

Number of digital input bytes, which the system has.
(= Number of digital inputs / 8)

pulNDigitalOutputBytes

Number of digital output bytes, which the system has.
(= Number of digital outputs / 8)

Typical function call (C example)

```
NMX_GetChannelCount_1(pHandle, &ulChannelCount,
&ulNDigiInBytes, &ulNDigiOutBytes);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS GetChannelCount_1(
                  System::IntPtr pHandle,
                  System::UInt32 %pulChannelCount,
                  System::UInt32 %pulNDigitalInputBytes,
                  System::UInt32 %pulNDigitalOutputBytes);
```

Comments

The function only returns values, if the function return code is NST_SUCCESS. Otherwise the values remain unchanged.

4.3.8.5 NMX_GetChannelInfo_1

This function is used to read information about a measurement channel. If supported by the probe, additional probe information is provided (digital probe type plate).

Definition

```
NMX_STATUS NMX_GetChannelInfo_1(
      NMX_PHANDLE pHandle,
      unsigned long ulChannelNo,
      unsigned long* pulChannelType,
      unsigned long* pulNDigits,
      unsigned long* pulBoxNo,
```

```
unsigned long* pulReserved,
unsigned long* pulBoxChannelNo,
signed long* pslRawDataType,
signed long* pslFactNumerator,
signed long* pslFactDenominator,
float* pflFactDigitsToUnit,
char* pcUnit, unsigned long ulSizeofUnit,
char* pcOrderNo, unsigned long ulSizeofOrderNo,
char* pcSerialNo, unsigned long ulSizeofSerialNo);
```

Parameter

---

pHandle

Connection Handle 162.

ulChannelNo

Number of the measurement channel.
The first channel has the number 0.
Having a device with 8 measurement channels, these are numbered 0..7.

pulChannelType

Measurement channel type. See below for a list of channel types.

pulNDigits

Recommended max. number of decimal places for the measurement value, converted to its unit.
E.g. `pulNDigits = 2` --> 54.**17**µm

pulBoxNo

Number of the Box, where the measurement channel is located.

pulReserved

Reserved for future use.

pulBoxChannelNo

Number of the channel within its measurement box.
Numbering starts at 1 with each Box.

Example: In a system with 2 Boxes à 8 Channels, Channel number 11 is the 4th channel of the second box. Thus the value 4 is returned.

pslRawDataType

Data type, in which the measurement data is acquired.
0 = Unknown (DTRAW_UNKNOWN)
1 = signed 8 Bit (DTRAW_SINT08)
2 = signed 16 Bit (DTRAW_SINT16)
4 = signed 32 Bit (DTRAW_SINT32)
8 = signed 64 Bit (DTRAW_SINT64)
See the data types 148 chapter for more information.

pslFactNumerator

Numerator of the factor, which is used for converting the digit value into its physical unit.
For some channel types, the physical unit is unknown by the device. Then this value is 1.
For some channel types, e.g. inductive probes, this is also the maximum stroke, which the probe has.

pslFactDenominator

Denominator of the factor, which is used for converting the digit value into its physical unit.
For some channel types, the physical unit is unknown by the device. Then this value is 1.

pflFactDigitsToUnit

`= pslFactNumerator / pslFactDenominator.`
Floating point representation of the factor, which is used for converting the digit value into its physical unit.
(ANSI/IEEE Std 754-1985; IEC-60559:1989: single precision 32 Bit)

pcUnit

ASCII based string with the channels physical unit (or "Digits" if the unit is unknown). The maximum string length is 9 characters (8 + Termination).

ulSizeofUnit

Maximum size of pcUnit in Bytes/Characters.

pcOrderNo

ASCII based string with the probe/sensors order number, if available. The maximum string length is 17 characters (16 + Termination).

ulSizeofOrderNo

Maximum size of pcOrderNo in Bytes/Characters.

pcSerialNo

ASCII based string with the probe/sensors serial number, if available. The maximum string length is 17 characters (16 + Termination).

ulSizeofSerialNo

Maximum size of pcSerialNo in Bytes/Characters.

Typical function call (C example)

```
NMX_GetChannelInfo_1(pHandle, ulChannel, &ulChannelType,
&ulNDigits, &ulBoxNo, NULL, &ulBoxChannelNo, &slRawDataType,
&slFactNumerator, &slFactDenominator, &flFactDigitsToUnit,
acUnit, sizeof(acUnit), acOrderNo, sizeof(acOrderNo),
acSerialNo, sizeof(acSerialNo));
```

Channel types

The channel type allows identifying the channel and/or probe type. The list is constantly expanded. Currently the following types are supported:

Type	Hex value	Description
CHANNEL_TYPE_U	0x0000	No channel type available
CHANNEL_TYPE_TI	0x0100	Tesa HalfBridge or compatible, 13 kHz, 3Veff, Default Stroke: ±2mm at 73.75mV/V/mm
CHANNEL_TYPE_TI	0x0101	Tesa HalfBridge or compatible, 13 kHz, 3Veff, Linearized, Max Stroke ±2mm
CHANNEL_TYPE_TI	0x0103	Tesa HalfBridge or compatible, 13 kHz, 3Veff, Linearized, Max. Stroke ±3mm
CHANNEL_TYPE_TI	0x0105	Tesa HalfBridge or compatible, 13 kHz, 3Veff, Linearized, Max. Stroke ±5mm
CHANNEL_TYPE_IE	0x0110	Knäbel IET, 50 kHz, 1.5Veff, Default Stroke: ±200µm
CHANNEL_TYPE_SO	0x0120	Solartron LVDT or compatible, 5 kHz, 3Veff, Default Stroke: ±1mm at 200mV/V/mm
CHANNEL_TYPE_SO	0x0121	Solartron customized type.
CHANNEL_TYPE_FE	0x0130	Feinprüf / Mahr, 20 kHz, 3Veff, Default Stroke: ±1mm
CHANNEL_TYPE_M	0x0140	Marposs LVDT, Default Stroke: ±1mm
CHANNEL_TYPE_IN	0x0200	Incremental Encoder TTL/RS422
CHANNEL_TYPE_IN	0x0210	Incremental Encoder 1Vpp
CHANNEL_TYPE_A	0x0300	Analogue Input ±10V Differential
CHANNEL_TYPE_TI	0x0400	Thermocouple, K-Type
CHANNEL_TYPE_LA	0x0500	Laser-Sensor (Sub-Types available).

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS GetChannelInfo_1(
```

```
System::IntPtr pHandle,
System::UInt32 ulChannelNo,
System::UInt32 %pulChannelType,
System::UInt32 %pulNDigits,
System::UInt32 %pulBoxNo,
System::UInt32 %pulReserved,
System::UInt32 %pulBoxChannelNo,
System::Int32 %pslRawDataType,
System::Int32 %pslFactNumerator,
System::Int32 %pslFactDenominator,
System::Single %pflFactDigitsToUnit,
System::String ^%strUnit,
System::String ^%strOrderNo,
System::String ^%strSerNo);
```

No Length/Sizeof-Parameters are required for the strings, since this information is not required in a .Net environment. They are directly returned as Unicode-strings. Hence no additional conversion is required.

Comments

It is good practice reading the channel count 189 first. Then read the channel information for each channel.

The function only returns values, if the function return code is NST_SUCCESS. Otherwise the values remain unchanged.

4.3.8.6 NMX_GetDigitalInputInfo_1

This function is used to read information about a digital input byte. Each input byte represents 8 digital input bits.

Definition

```
NMX_STATUS NMX_GetDigitalInputInfo_1(
      NMX_PHANDLE pHandle,
      unsigned long ulInputByteNo,
      unsigned long* pulBoxNo,
      unsigned long* pulBoxByteNo);
```

Parameter

---

pHandle

Connection Handle 162.

ulInputByteNo

Number of the digital input byte.
The first byte has the number 0.
Example: Having a device with 4 input bytes, these are numbered 0..3.

pulBoxNo

Number of the Box, where the digital inputs are located.

pulBoxByteNo

Number of the digital input byte within its measurement box.
Numbering starts at 1 with each Box.

Example: In a system with 2 Boxes à 16 digital inputs, input byte 3 is the 1st input byte of the second box. Thus the value 1 is returned.

Typical function call (C example)

```
NMX_GetDigitalInputInfo_1(pHandle, ulInByte, &ulBoxNo,
&ulBoxByteNo);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS GetDigitalInputInfo_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulInputByteNo,
                  System::UInt32 %pulBoxNo,
                  System::UInt32 %pulBoxByteNo);
```

Comments

It is good practice reading the digital input count 189 first. Then read the byte information for each digital input byte.

The function only returns values, if the function return code is NST_SUCCESS. Otherwise the values remain unchanged.

4.3.8.7 NMX_GetDigitalOutputInfo_1

This function is used to read information about a digital output byte. Each output byte represents 8 digital output bits.

Definition

```
NMX_STATUS NMX_CALLCONV NMX_GetDigitalOutputInfo_1(
      NMX_PHANDLE pHandle,
      unsigned long ulOutputByteNo,
      unsigned long* pulBoxNo,
      unsigned long* pulBoxByteNo);
```

Parameter

pHandle

Connection Handle 162.

ulOutputByteNo

Number of the digital output byte.
The first byte has the number 0.
Example: Having a device with 4 output bytes, these are numbered 0..3.

pulBoxNo

Number of the Box, where the digital outputs are located.

pulBoxByteNo

Number of the digital output byte within its measurement box.
Numbering starts at 1 with each Box.

Example: In a system with 2 Boxes à 16 digital outputs, output byte 3 is the 1st output byte of the second box. Thus the value 1 is returned.

Typical function call (C example)

```
NMX_GetDigitalOutputInfo_1(pHandle, ulOutByte, &ulBoxNo,
&ulBoxByteNo);
```

.Net DLL [152] *specific implementation*

```
NMX_MSTATUS GetDigitalOutputInfo_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulOutputByteNo,
                  System::UInt32 %pulBoxNo,
                  System::UInt32 %pulBoxByteNo);
```

Comments

It is good practice reading the digital output count [189] first. Then read the byte information for each digital output byte.

The function only returns values, if the function return code is NST_SUCCESS. Otherwise the values remain unchanged.

4.3.9 Static Measurement (Non-Realtime)

Static measurement is used to get a snapshot of the current measurement values and digital input states. It is very easy to implement

For more information see the chapter "Static vs. Sampling [140]".

4.3.9.1 NMX_StaticGet32_1

This function is used to get a snapshot of:

- Current measurement values
- Current hardware status of the measurement channels
- Current digital input status
- Current status of the measurement boxes ("Box event")

See also the HowTo [242]-Guide.

Definition

```
NMX_STATUS NMX_StaticGet32_1(
      NMX_PHANDLE pHandle,
      signed long aslValues[], unsigned long ulValuesNElements,
      unsigned char aucHardwareStatus[], unsigned long
ulSizeofHardwareStatus,
      unsigned char aucDigiIn[], unsigned long ulSizeofDigiIn,
      unsigned char aucBoxStatus[], unsigned long
ulSizeofBoxStatus,
```

```
    unsigned long* pulUpdateCtr);
```

Parameter

---

pHandle

Connection Handle 162.

aslValues

Array, in which the measurement values shall be stored.
The data type of the array is "signed 32 Bit". See chapter "Data Types 148" for more information.
The array must be provided by your application.
Technically, this parameter is the same as a pointer to the array. Its type is: `signed long*`

ulValuesNElements

Number of elements of the array aslValues.
Example: aslValues points to an array with 64 elements, each with 32 Bit (Total size = 256 Bytes). Then: `ulValuesNElements = 64;`

aucHardwareStatus

Array, in which the hardware status for the measurement channels shall be stored.
The data type of the array is "unsigned 8 Bit".
The array must be provided by your application.
Technically, this parameter is the same as a pointer to the array. Its type is: `unsigned char*`

ulSizeofHardwareStatus

Size of the array aucHardwareStatus in Bytes.

aucDigiIn

Array, in which the digital input bytes shall be stored.
The data type of the array is "unsigned 8 Bit".
The array must be provided by your application.
Technically, this parameter is the same as a pointer to the array. Its type is: `unsigned char*`

ulSizeofDigiIn

Size of the array aucDigiIn in Bytes.

aucBoxStatus

Array, in which the status information for each box shall be stored.
The data type of the array is "unsigned 8 Bit".
The array must be provided by your application.
Technically, this parameter is the same as a pointer to the array. Its type is: `unsigned char*`

ulSizeofBoxStatus

Size of the array aucBoxStatus in Bytes.

pulUpdateCtr

A DLL-internal counter is updated each time new static data has been received from the device. This counter can be read-out here.

Typical function call (C example)

```
NMX_StaticGet32_1(pHandle, aslMeasVal, sizeof(aslMeasVal)/4,
aucHardStat, sizeof(aucHardStat), aucDigiIn, sizeof(aucDigiIn),
aucBoxStatus, sizeof(aucBoxStatus), &ulNUpdates);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS StaticGet32_1(
                  System::IntPtr pHandle,
                  array<System::Int32>^aslValues,
                  array<System::Byte>^aucHardwareStatus,
                  array<System::Byte>^aucDigiIn,
                  array<System::Byte>^aucBoxStatus,
                  System::UInt32 %pulUpdateCtr);
```

No Length/Sizeof-Parameters are required for the arrays, since this information automatically available in a .Net environment.

The arrays will not be resized within the function call for performance reasons. This means they should be large enough to store all the data.

Example: in a system with 24 measurement channels, the arrays aslValues and aucHardwareStatus should have a minimum size of 24 array elements.

If the array is shorter, not all data will be available for your application. (The size is checked -> no risk for crash.)

Comments

The notification 176 NMXNOTIFY_NEW_STATIC32 can be used to get informed about new data.

Never use digital in-/outputs for emergency critical applications. If required, use an external emergency circuit. See the users manual of your measurement system for more information.

Status-Byte for measurement channels for 1Vpp incremental encoders (INC)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PwrOvld		Refmark	Vector	GComp	OComp	AmpErr	Fast

PwrOvld

Error: A power supply overload of the incremental encoder(s) has been detected.

Refmark

The reference index has been crossed.

Vector

Error: The signal vector, which has been calculated from the cosine- and sine-signal, is too small. (Can only occur with 1Vpp incremental encoders.)

GComp

Error: The gain-control has reached its limit. (Can only occur with 1Vpp incremental encoders.)

OComp

Error: The offset-control has reached its limit. (Can only occur with 1Vpp incremental encoders.)

AmpErr

Error: One or both AD-converters for the measurement of the sine-/cosine-signal is/are overdriven. (Can only occur with 1Vpp incremental encoders.)

Fast

Error: The input frequency is too high.

Status-Byte for inductive probes (TFV)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LinActive	IdentActive		LinFailure				ShortCirc

ShortCirc

Error: Short circuit of the sine-oscillator

LinFailure

Failed reading / applying the linearisation data.
This bit is only available, if the measurement hardware and the probe support linearisation.

IdentActive

Reading the probe identification and linearisation data is active.
This bit is only available, if the measurement hardware and the probe support linearisation.

LinActive

Probe linearisation is active.
This bit is only available, if the measurement hardware and the probe support linearisation.

Status-Byte for analogue inputs (AIN)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
24VOvld	VRefOvld						

24VOvld

Error: Overload of the 24V output supply.

VRefOvld

Error: Overload of the reference voltage output.

Status-Byte for temperature measurement (TEMP)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TempInvalid	SenseFailure	ADError	ColdJunctionErrorHw	ColdJunctionErrorSw	SenseVoltHigh	SenseVoltLow	OutOfRange

OutOfRange

Sensor input voltage is out of range

SenseVoltLow

Sensor reading is below normal range.

SenseVoltHigh

Sensor reading is above normal range.

ColdJunctionErrorSw

Cold junction sensor result is beyond normal range.

ColdJunctionErrorHw

Cold junction sensor has a hardware fault error.

ADError

Bad ADC reading. Could be large external noise event.

SenseFailure

Bad sensor reading.

TempInvalid

Temperature value may be invalid.

4.3.9.2 NMX_StaticSetMedianDepth_1

This function is used to modify, enable or disable the median filter, which is applied on static measurement values.

Definition

```
NMX_STATUS NMX_StaticSetMedianDepth_1(
      NMX_PHANDLE pHandle,
      unsigned long ulMedianDepth);
```

Parameter

pHandle

Connection Handle 162.

ulMedianDepth

Number of samples, which are used to calculate their median value.
Using an odd value (3, 5, 7, ...) is recommended.
A value of 1 disables the median calculation.

Typical function call (C example)

```
NMX_StaticSetMedianDepth_1(pHandle, 3);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS StaticSetMedianDepth_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulMedianDepth);
```

Comments

The default depth is 5.

The value should not be too high. Otherwise you get a long delay.

4.3.9.3 NMX_SetOutputs_1

This function is used to change the state of the digital outputs.

Definition

```
NMX_STATUS NMX_SetOutputs_1(
      NMX_PHANDLE pHandle,
      unsigned char* pucDigiOut, unsigned long ulSizeofDigiOut,
      unsigned char ucForceSendImmediately);
```

Parameter

pHandle

Connection Handle 162.

pucDigiOut

Pointer to the array, in which the digital output bytes are stored.

ulSizeofDigiOut

Size of the array "behind" pucDigiOut in Bytes.

ucForceSendImmediately

0 -> Default: The output data will be send with the next communication cycle. The default cycle time is approximately 30ms.
1 -> Send the output data immediately. Use this only, if an urgent update is required. Sending urgent updates repeatedly at a high frequency could slow down transfer rate for sampled data.

Typical function call (C example)

```
NMX_SetOutputs_1(pHandle, aucDigiOut, sizeof(aucDigiOut), 0);
```

.Net DLL [152] *specific implementation*

```
NMX_MSTATUS SetOutputs_1(
                  System::IntPtr pHandle,
                  array<System::Byte>^aucDigiOut,
                  System::Byte ucForceSendImmediately);
```

No Length/Sizeof-Parameter is required for the array aucDigiOut, since this information automatically available in a .Net environment.

Comments

With ucForceSendImmediately = 0, the output data is only updated inside the NMX DLL with this function call. Each time cyclic data is exchanged between the DLL and the device, the output data is send to the device. No matter if it has been changed or not.

Never use digital in-/outputs for emergency critical applications. If required, use an external emergency circuit. See the users manual of your measurement system for more information.

4.3.9.4 NMX_DisableOutputUpdate_1

Digital output data is normally transferred cyclically from the NMX DLL to the device. This function is used to stop this cyclic update.
It is typically not used within standard measurement applications. It could be helpful, if a manual manipulation of the outputs shall be done, e.g. using the configuration tool of the measurement system.

Definition

```
NMX_STATUS NMX_DisableOutputUpdate_1(
      NMX_PHANDLE pHandle);
```

Parameter

pHandle

Connection Handle 162.

Typical function call (C example)

```
NMX_DisableOutputUpdate_1(pHandle);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS DisableOutputUpdate_1(System::IntPtr pHandle);
```

4.3.9.5 NMX_DigitalIoConfig_1

This function is used to disable or enable the automatic reset of digital outputs after a communication timeout.
By default this is enabled, which means that all outputs are set to low, if there is a breakdown of the communication between the NMX DLL and the digital output.

Definition

```
NMX_STATUS NMX_DigitalIoConfig_1(
      NMX_PHANDLE pHandle,
      unsigned char ucOutputResetEnabled);
```

Parameter

pHandle

Connection Handle 162.

ucOutputResetEnabled

0 -> Resetting the digital outputs to low state after communication breakdown is disabled.
1 -> Default: Resetting the digital outputs to low state after communication breakdown is enabled.

Typical function call (C example)

```
NMX_DigitalIoConfig_1(pHandle, 0);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS DigitalIoConfig_1(
                    System::IntPtr pHandle,
                    System::Byte ucOutputResetEnabled);
```

Comments

By default, the state of all digital outputs is automatically set to low after communication breakdown is enabled. This means: if the communication breaks down: all outputs are set to low.

4.3.9.6 NMX_DigitalOutputsGetState_1

This function allows reading the current state of all digital outputs from the device.

Definition

```
NMX_STATUS NMX_DigitalOutputsGetState_1(
      NMX_PHANDLE pHandle,
      unsigned char* pucOutputState,
      unsigned long ulSizeofOutputState);
```

Parameter

pHandle

Connection Handle 162.

pucOutputState

Pointer to the array, in which the digital output bytes shall be stored.
The data type of the array is "unsigned 8 Bit".
The array must be provided by your application.

ulSizeofOutputState

Size of pucOutputState in Bytes.

Typical function call (C example)

```
NMX_DigitalOutputsGetState_1(pHandle, aucOutputs,
sizeof(aucOutputs));
```

.Net DLL [152] *specific implementation*

```
NMX_MSTATUS DigitalOutputsGetState_1(
                  System::IntPtr pHandle,
                  array<System::Byte>^aucOutputState);
```

No Length/Sizeof-Parameter is required for the array aucOutputState, since this information automatically available in a .Net environment.

The array will not be resized within the function call for performance reasons. This means it should be large enough to store all the data.

Example: in a system with 32 digital outputs (= 4 outbut bytes), the array aucOutputState should have a minimum size of 4 array elements.

If the array is shorter, not all data will be available for your application. (The size is checked -> no risk for crash.)

Comments

The purpose of this function is getting the state of all digital outputs once after establishing a connection. It should not be called cyclically.

The state reflects the internal data inside the measurement system. It does not reflect the physical state of an output. Usually both are the same, but under fault conditions they may differ.

4.3.10 Sampling LowLevel (Time-Triggered Realtime Measurement)

Sampling is used to get real-time data from the measurement system. This covers measurement values as well as digital in-/output states.

For more information see the chapter "Static vs. Sampling [140]".

4.3.10.1 NMX_Sampling_GetMaxSpeed_1

This function is used to get the maximum possible sampling speed.

Definition

```
NMX_STATUS NMX_Sampling_GetMaxSpeed_1(
```

```
        NMX_PHANDLE pHandle,
        unsigned long *pulInSamplePeriodUs);
```

Parameter

---

pHandle

Connection Handle 162.

pulInSamplePeriodUs

Minimum sample period in µs. The maximum sampling speed is the reciprocal value.

Typical function call (C example)

```
NMX_Sampling_GetMaxSpeed_1(pHandle, &ulMinSamplePeriod);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_GetMaxSpeed_1(
                  System::IntPtr pHandle,
                  System::UInt32 %pulInSamplePeriodUs);
```

Comments

The maximum possible speed for endless sampling may be slower. Consult the users manual of the measurement system for more information about sampling speed.

4.3.10.2 NMX_Sampling_Reset_1

Reset the whole sampling configuration. If sampling is active, stop it.

This function must be called before a new list of sampling elements is created via NMX_Sampling_Add...

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_Reset_1(
```

```
NMX_PHANDLE pHandle);
```

Parameter

---

pHandle

Connection Handle 162.

Typical function call (C example)

```
NMX_Sampling_Reset_1(pHandle, &ulMinSamplePeriod);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_Reset_1(System::IntPtr pHandle);
```

Comments

If an active sampling is stopped via NMX_Sampling_Stop_1 and then shall be restarted with the same sampling elements, there is no need for calling NMX_Sampling_Reset_1.

However, if a new list of sampling elements shall be created, an existing list must be cleared by calling NMX_Sampling_Reset_1.

4.3.10.3 NMX_Sampling_AddChannelsAll_1

Use this function to add all measurement channels to the list of sampling elements.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_AddChannelsAll_1(
      NMX_PHANDLE pHandle,
      unsigned long* pulNElementsTotal);
```

Parameter

pHandle

Connection Handle 162.

pulNElementsTotal

Total number of sampling elements, which have already been added since the last call of NMX_Sampling_Reset_1 210.

Typical function call (C example)

```
NMX_Sampling_AddChannelsAll_1(pHandle, &ulNElements);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS Sampling_AddChannelsAll_1(
                  System::IntPtr pHandle,
                  System::UInt32 %pulNElementsTotal);
```

Comments

It is also possible to add only a selection of measurement channels to the list of sampling elements (see NMX_Sampling_AddChannel_1 212).

This may

- allow for higher sampling speed with endless sampling or
- shorten the transfer time of the samples with time-limited sampling.

Please note: the same measurement channel cannot be added twice. Thus you can either use NMX_Sampling_AddChannelsAll_1 or NMX_Sampling_AddChannel_1, but not both of them.

4.3.10.4 NMX_Sampling_AddChannel_1

Use this function to add a single measurement channel to the list of sampling elements.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_AddChannel_1(
      NMX_PHANDLE pHandle,
      unsigned long ulChannelNumber,
      unsigned long* pulNElementsTotal);
```

Parameter

---

pHandle

Connection Handle 162.

ulChannelNumber

Number of the measurement channel, which shall be added.
The first channel has the number 0.
Having a device with 24 measurement channels, these are numbered 0..23.

pulNElementsTotal

Total number of sampling elements, which have already been added since the last call of NMX_Sampling_Reset_1 210.

Typical function call (C example)

```
NMX_Sampling_AddChannel_1(pHandle, 5, &ulNElements);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_AddChannel_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulChannelNumber,
                  System::UInt32 %pulNElementsTotal);
```

Comments

It is also possible to add all measurement channels to the list of sampling elements (see NMX_Sampling_AddChannelsAll_1 211).

Please note: the same measurement channel cannot be added twice. Thus you can either use NMX_Sampling_AddChannelsAll_1 or NMX_Sampling_AddChannel_1, but not both of them.

4.3.10.5 NMX_Sampling_AddDigiInAll_1

Use this function to add all digital input bytes to the list of sampling elements.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_AddDigiInAll_1(
      NMX_PHANDLE pHandle,
      unsigned long* pulNElementsTotal);
```

Parameter

pHandle

Connection Handle 162.

pulNElementsTotal

Total number of sampling elements, which have already been added since the last call of NMX_Sampling_Reset_1 210.

Typical function call (C example)

```
NMX_Sampling_AddDigiInAll_1(pHandle, &ulNElements);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS Sampling_AddDigiInAll_1(
                  System::IntPtr pHandle,
                  System::UInt32 %pulNElementsTotal);
```

Comments

It is also possible to add only a selection of digital input bytes to the list of sampling elements (see NMX_Sampling_AddDigiInByte_1 215).

This may

- allow for higher sampling speed with endless sampling or

- shorten the transfer time of the samples with time-limited sampling.

Please note: the same digital input byte cannot be added twice. Thus you can either use NMX_Sampling_AddDigiInAll_1 or NMX_Sampling_AddDigiInByte_1, but not both of them.

4.3.10.6 NMX_Sampling_AddDigiInByte_1

Use this function to add a single digital input byte to the list of sampling elements.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_AddDigiInByte_1(
      NMX_PHANDLE pHandle,
      unsigned long ulDigiInByteNo,
      unsigned long* pulNElementsTotal);
```

Parameter

pHandle

Connection Handle 162.

ulDigiInByteNo

Number of the digital input byte.
The first byte has the number 0.
Example: Having a device with 6 input bytes, these are numbered 0..5.

pulNElementsTotal

Total number of sampling elements, which have already been added since the last call of NMX_Sampling_Reset_1 210.

Typical function call (C example)

```
NMX_Sampling_AddDigiInByte_1(pHandle, 0, &ulNElements);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS Sampling_AddDigiInByte_1(
                    System::IntPtr pHandle,
                    System::UInt32 ulDigiInByteNo,
                    System::UInt32 %pulNElementsTotal);
```

Comments

It is also possible to add all digital input bytes to the list of sampling elements (see NMX_Sampling_AddDigiInAll_1 214).

Please note: the same digital input byte cannot be added twice. Thus you can either use NMX_Sampling_AddDigiInAll_1 or NMX_Sampling_AddDigiInByte_1, but not both of them.

4.3.10.7 NMX_Sampling_AddDigiOutAll_1

Use this function to add all digital output bytes to the list of sampling elements.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_AddDigiOutAll_1(
      NMX_PHANDLE pHandle,
      unsigned long* pulNElementsTotal);
```

Parameter

pHandle

Connection Handle 162.

pulNElementsTotal

Total number of sampling elements, which have already been added since the last call of NMX_Sampling_Reset_1 210.

Typical function call (C example)

```
NMX_Sampling_AddDigiOutAll_1(pHandle, &ulNElements);
```

.Net DLL[152] *specific implementation*

```
NMX_MSTATUS Sampling_AddDigiOutAll_1(
                  System::IntPtr pHandle,
                  System::UInt32 %pulNElementsTotal);
```

Comments

It is also possible to add only a selection of digital output bytes to the list of sampling elements (see NMX_Sampling_AddDigiOutByte_1[217]).

This may

- allow for higher sampling speed with endless sampling or
- shorten the transfer time of the samples with time-limited sampling.

Please note: the same digital output byte cannot be added twice. Thus you can either use NMX_Sampling_AddDigiOutAll_1 or NMX_Sampling_AddDigiOutByte_1, but not both of them.

4.3.10.8 NMX_Sampling_AddDigiOutByte_1

Use this function to add a single digital output byte to the list of sampling elements.

See also the HowTo[250]-Guide.

Definition

```
NMX_STATUS NMX_Sampling_AddDigiOutByte_1(
      NMX_PHANDLE pHandle,
      unsigned long ulDigiOutByteNo,
      unsigned long* pulNElementsTotal);
```

Parameter

pHandle

Connection Handle[162].

ulDigiOutByteNo

Number of the digital output byte.
The first byte has the number 0.
Example: Having a device with 6 output bytes, these are numbered 0..5.

pulNElementsTotal

Total number of sampling elements, which have already been added since the last call of NMX_Sampling_Reset_1 210.

Typical function call (C example)

```
NMX_Sampling_AddDigiOutByte_1(pHandle, 1, &ulNElements);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_AddDigiOutByte_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulDigiOutByteNo,
                  System::UInt32 %pulNElementsTotal);
```

Comments

It is also possible to add all digital output bytes to the list of sampling elements (see NMX_Sampling_AddDigiOutAll_1 216).

Please note: the same digital output byte cannot be added twice. Thus you can either use NMX_Sampling_AddDigiOutAll_1 or NMX_Sampling_AddDigiOutByte_1, but not both of them.

4.3.10.9 NMX_Sampling_PrepareTime_1

This function is used to prepare a time-based sampling. Before starting the sampling, it must be prepared.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_PrepareTime_1(
      NMX_PHANDLE pHandle,
      unsigned long ulSamplePeriod,
      unsigned long ulDLLArrayLength,
      unsigned long long udMaxSamples);
```

Parameter

---

pHandle

Connection Handle 162.

ulSamplePeriod

Time period between two samples in µs, e.g. 1000 for 1ms.
See the chapter "Sampling Speed with Irinos" for more information. 141

ulDLLArrayLength

Data received from the device is internally buffered in the DLL. This parameter specifies the internal buffer length in samples.
Example: The sampling speed is 1ms -> 1000Samples/s and the buffer shall contain a maximum of 10 seconds. Then `ulDLLArrayLength = 10000;`

udMaxSamples

Maximum number of samples, which shall be recorded.
Use 0 for endless sampling.
Example: The sampling speed is 1ms -> 1000Samples/s. Data shall be recorded for a maximum of 15 seconds. Then udMaxSamples = 15000;

Typical function call (C example)

Time-limited: `NMX_Sampling_PrepareTime_1(pHandle, 1000, 10000, 10000);`
Endless: `NMX_Sampling_PrepareTime_1(pHandle, 1000, 10000, 0);`

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_PrepareTime_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulSamplePeriod,
                  System::UInt32 ulDLLArrayLength,
                  System::UInt64 udMaxSamples);
```

Comments

- For short measurements of a few seconds, it is common practice using the same value for ulDLLArrayLength and for udMaxSamples.
- The internal buffer is allocated in the heap memory of your application.
- The larger the DLL-internal buffer is, the larger the memory consumption. For typical time-limited sampling, this is not a problem and the memory size will be between a few kBytes up to a few MBytes. This applies even to larger systems.
  However, if you would use for example 256 measurement channels of 32 Bit size, and ulDLLArrayLength would be 100000, then 100 MBytes would be required. This may be too large for your heap memory.

4.3.10.10 NMX_Sampling_Start_1

This function is used to start sampling. Before starting the sampling, it must be prepared 218.

See also the HowTo 250-Guide.

Definition

```
NMX_STATUS NMX_Sampling_Start_1(
      NMX_PHANDLE pHandle);
```

Parameter

pHandle

Connection Handle 162.

Typical function call (C example)

```
NMX_Sampling_Start_1(pHandle);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS Sampling_Start_1(System::IntPtr pHandle);
```

Comments

4.3.10.11 NMX_Sampling_Stop_1

This function is used to stop an active sampling.

Definition

```
NMX_STATUS NMX_Sampling_Stop_1(
      NMX_PHANDLE pHandle);
```

Parameter

pHandle

Connection Handle 162.

Typical function call (C example)

```
NMX_Sampling_Stop_1(pHandle);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_Stop_1(System::IntPtr pHandle);
```

Comments

If sampling is inactive, nothing happens.

4.3.10.12 NMX_Sampling_ReadColumn32_1

Read sampled values column-wise. Column-wise means, that multiple samples of the same sampling element are read-out. This could for example be samples 0..999 from measurement channel 1.

See also the HowTo 268-Guide.

Definition

```
NMX_STATUS NMX_Sampling_ReadColumn32_1(
      NMX_PHANDLE pHandle,
      signed long aslSamples[],
```

```
    unsigned long ulMaxSamples,
    unsigned long ulElementNo,
    unsigned long ulDoNotDelete,
    unsigned long* pulSamplesCopied,
    unsigned long long* pudNoFirstSample);
```

Parameter

---

pHandle

Connection Handle 162.

aslSamples

The measurement samples will be written into this array. For simplicity, all sampled values are provided with data type signed long. See chapter "data types 148" for more information.

ulMaxSamples

Maximum number of samples to read-out. This value must be ≤ the total number of elements of aslSamples.
Example: aslSamples is defined as "signed long aslSamples[1000]". Then ulMaxSamples must be ≤ 1000.

ulElementNo

Number of the sampling element, starting with 0 for the first sampling element.

ulDoNotDelete

0: samples will be deleted from the DLL-internal buffer after readout.
1: samples will NOT be deleted from the DLL-internal buffer after readout.

pulSamplesCopied

Number of samples copied into aslSamples. This value will be ≤ ulMaxSamples.

pudNoFirstSample

Number of the first sample in aslSamples, counted from the beginning of sampling, starting at 0.
Example:
The 1. time after the start of sampling, this function is called and 162 samples are read-out. `pudNoFirstSample = 0` will be returned.
The 2. time this function is called, 97 samples are read-out. `pudNoFirstSample = 162` will be returned.
The 3. time this function is called, 212 samples are read-out. `pudNoFirstSample = 259` will be returned (162 + 97 = 259).

Typical function call (C example)

Assuming aslSamples is defined as: `signed long aslSamples[10000];`

Example 1: `NMX_Sampling_ReadColumn32_1(pHandle, aslSamples, 10000, 4, 0, &ulSamplesCopied, &udNoFirstSample);`
Example 2: `NMX_Sampling_ReadColumn32_1(pHandle, &aslSamples[162], 10000-162, 4, 0, &ulSamplesCopied, &udNoFirstSample);`

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_ReadColumn32_1(
                  System::IntPtr pHandle,
                  array<System::Int32>^aslSamples,
                  System::UInt32 ulArrayIndex,
                  System::UInt32 ulMaxSamples,
                  System::UInt32 ulElementNo,
                  System::UInt32 ulDoNotDelete,
                  System::UInt32 %pulSamplesCopied,
                  System::UInt64 %pudNoFirstSample);
```

The array aslSamples will not be resized within the function call for performance reasons. This means it should be large enough to store all the data.

Via the parameters ulArrayIndex and ulMaxSamples, it can be defined in which area of aslSamples the sampled data can be written. This is especially helpful, if the sampled data is read in multiple blocks of data (-> no additional copying required).

If the sampled data is read at once, typically ulArrayIndex = 0 and ulMaxSamples = aslSamples.Length.

Comments

- The standard way is deleting sampled data inside the NMX DLL, after it has been read-out. This is done by setting: `ulDoNotDelete = 0;`
  If you want to be able to read it out more than once, it is possible to set `ulDoNotDelete = 1;`. Naturally this makes no sense for endless sampling!
- It is possible to get informed about new data by the notification[176] NMXNOTIFY_SAMPLING_NEW_DATA.
- It is good practice reading sampled data in small portions, for example each time new data has arrived at the DLL.

4.3.10.13 NMX_Sampling_ReadRow32_1

Read sampled values row-wise. Row-wise means, that all sampling elements of a single sampling point are read-out. Each time this function is called, the oldest sample data is read-out and deleted afterwards.

See also the HowTo[273]-Guide.

Definition

```
NMX_STATUS NMX_Sampling_ReadRow32_1(
      NMX_PHANDLE pHandle,
      signed long aslSamples[],
      unsigned long ulMaxSamples,
      unsigned long* pulSamplesCopied,
      unsigned long long* pudSampleNo);
```

Parameter

pHandle

Connection Handle[162].

aslSamples

The measurement samples will be written into this array. For simplicity, all sampled values are provided with data type signed long. See chapter "data types[148]" for more information.

ulMaxSamples

Maximum number of samples to read-out. This value must be ≤ the total number of elements of aslSamples.
Example: aslSamples is defined as "signed long aslSamples[64]". Then ulMaxSamples must be ≤ 64.

pulSamplesCopied

Number of samples copied into aslSamples. This value will be ≤ ulMaxSamples.

pudSampleNo

Number of the this sample, counted from the beginning of sampling, starting at 0.
Example:
The 1. time after the start of sampling, this function is called, `pudSampleNo = 0` will be returned.
The 2. time this function is called, `pudSampleNo = 1` will be returned.
The 3. time this function is called, `pudSampleNo = 2` will be returned.

Typical function call (C example)

Assuming aslSamples is defined as: `signed long aslSamples[64];`

```
NMX_Sampling_ReadRow32_1(pHandle, aslSamples, 64,
&ulSamplesCopied, &udNoSample);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_ReadRow32_1(
                  System::IntPtr pHandle,
                  array<System::Int32>^aslSamples,
                  System::UInt32 %pulSamplesCopied,
                  System::UInt64 %pudSampleNo);
```

The array aslSamples will not be resized within the function call for performance reasons. This means it should be large enough to store all the data.

If for example 23 sampling elements have been assigned with NMX_Sampling_Add..., the the minimum array size should be 23.
If it is smaller, it won't crash but not all data will be available to your application.

Comments

- It is possible to get informed about new data by the notification 176 NMXNOTIFY_SAMPLING_NEW_DATA.
- Using NMX_Sampling_ReadRow32_1 it is not possible, reading the same data twice.

4.3.10.14 NMX_Sampling_GetStatus_1

This function is used to read the current sampling status.

See also the HowTo 277-Guide.

Definition

```
NMX_STATUS NMX_Sampling_GetStatus_1(
      NMX_PHANDLE pHandle,
      unsigned char* pucStatus,
      unsigned long* pulNSamplingElements,
      unsigned long long* pudSamplesReceived,
      unsigned long long* pudMaxSamples);
```

Parameter

pHandle

Connection Handle 162.

pucStatus

Current sampling status. See definition below.

pulNSamplingElements

Number of sampling elements used.

pudSamplesReceived

Provide the number of samples, which have been received by the DLL from the device.

pudMaxSamples

Provide the maximum number of samples, that will be recorded.
If you started endless sampling, this value will be 0xFFFFFFFFFFFFFFFF = 18446744073709551615.

Typical function call (C example)

Assuming aslSamples is defined as: `signed long aslSamples[64];`

```
NMX_Sampling_ReadRow32_1(pHandle, aslSamples, 64,
&ulSamplesCopied, &udNoSample);
```

Sampling Status

The sampling status is defined as follows:

Status	Hex value	Description
EMS_INACTIVE	0x00	No sampling active.
EMS_PREPARED	0x01	Sampling has been prepared, but not yet started.
EMS_ACTIVE	0x02	Sampling is active.
EMS_DATA_TRANS	0x03	Sampling has ended or has been stopped, but data transfer is still active.
EMS_FINISHED	0x04	Sampling is finished.
EMS_ERROR_CONF	0xF0	Error during configuration / preparation of sampling.
EMS_ERROR_STAR	0xF1	Error during start of sampling.
EMS_ERROR_NOTP	0xF2	Error: start not possible, sampling has not been prepared.
EMS_ERROR_RUN	0xF8	Error occurred while sampling was active.

By definition, all values > 0xF0 are error states.

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS Sampling_GetStatus_1(
                    System::IntPtr pHandle,
                    System::Byte %pucStatus,
                    System::UInt32 %pulNSamplingElements,
                    System::UInt64 %pudSamplesReceived,
                    System::UInt64 %pudMaxSamples);
```

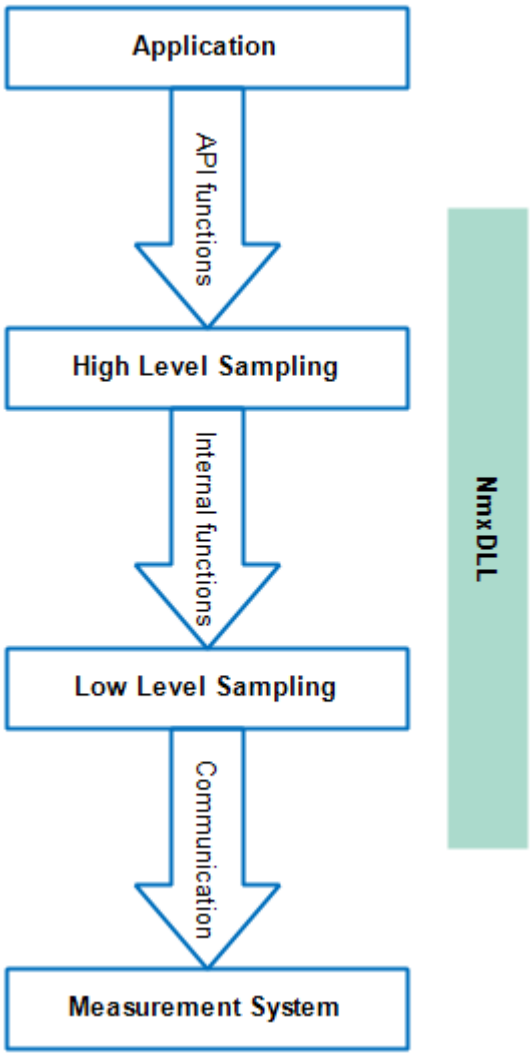
Comments

It is possible to get informed about various sampling events via notfications 176.

4.3.11 Sampling HighLevel (Application-specific Realtime Measurement)

Using the high-level functionality is optional! *If only time-triggered realtime measurement is required, using the low-level sampling functionality is sufficient!*

The High-Level Sampling functionality provides an easy way to implement intelligent real-time measurement functions. All high-level functions internally use the the low-level time-triggered realtime sampling 209.



The following high-level sampling types are available:

- TFT[232] stands for "**T**rigger + **F**ilter + **T**ail values" and was developed to meet the specific needs of one customer.

4.3.11.1 NMX_Sampling_PreparePosition_1

Minimum DLL-Version: V1.2.0.13

This function is used to prepare a position-triggered sampling. With position-triggered sampling, one measurement channel is used as a trigger source. In defined position distances, it triggers the other channels. This results in sets of measurement values (samples), which are recorded at equidistant position distances. Technically, position-triggered sampling uses an under

Before starting the sampling, it must be prepared.

See also the HowTo[284]-Guide.

Definition

```
NMX_STATUS NMX_Sampling_PreparePosition_1(
      NMX_PHANDLE pHandle,
      unsigned long ulSamplePeriod,
      unsigned long ulArrayLength,
      unsigned long long udMaxSamples,
      unsigned long ulTriggerChannelNumber,
      double fdScale,
      double fdStart,
      double fdDistance);
```

Parameter

pHandle

Connection Handle 162.

ulSamplePeriod

Sampling-period for the underlying time-based sampling.
Time period between two samples of the measurement hardware in µs, e.g. 1000 for 1ms.
This is the sampling period (-> measurement frequency) which will be used by the measurement system for acquiring the time-triggered measurement samples.
See the chapter "Sampling Speed with Irinos" for more information. 141
If you are unsure which value to use, 1000 (= 1ms) is a good value for starting your development.

ulArrayLength

Triggered data is internally buffered in the DLL. This parameter specifies the internal buffer length in samples.
Since position-triggered sampling is usually not endless, this parameter has typically the same value as udMaxSamples.

udMaxSamples

Maximum number of samples, which shall be recorded.
(0 can be used for endless sampling, even though this typically makes no sense with position-triggered sampling.)

Example: 360° shall be sampled with a position distance of 0.1°. This udMaxSamples = 360° / 0.1° = 3600.

To stop position triggering before udMaxSamples has been reached, use NMX_Sampling_Stop_1[221].

ulTriggerChannelNumber

Number of the measurement channel, which shall be used as source for the position trigger.
The first channel has the number 0.
Having a device with 24 measurement channels, these are numbered 0..23.
This measurement channel can be part of the sampled channels (this means it has been added to sampling via NMX_Sampling_AddChannel_1[212] or NMX_Sampling_AddChannelsAll_1[211]). If this is not the case, it will be added automatically.

fdScale

Format: 64 Bit floating point number according to IEEE 754 (-> double).

Using the scale value, the position value of the Trigger Channel can be converted to a physical unit, e.g. from increments to degrees. It thereby sets the unit of the parameters fdStart and fdDistance.
If 1.0 is used, then the raw value from the trigger channel is used.

Example: An encoder with 720000 increments is used. The preferred resolution is 1°. Then fdScale = 720000 / 360 = 2000.
If a trigger distance (-> see fdDistance) of 0.1° is required, then fdDistance will now be 0.1 instead of 200.

fdStart

Format: 64 Bit floating point number according to IEEE 754 (-> double).

Start position for triggering: Triggering will not start, before this position has been crossed.
In case fdDistance ≥ 0, the start position must be crossed from a value below fdStart to a value equal/above fdStart.
In case fdDistance ≤ 0, the start position must be crossed from a value above fdStart to a value equal/below fdStart.

The unit of this parameter depends on fdScale.

fdDistance

Format: 64 Bit floating point number according to IEEE 754 (-> double).

Position distance between two trigger positions.

The unit of this parameter depends on fdScale.

Typical function call (C example)

```
NMX_Sampling_PreparePosition_1(pHandle, 1000, 10000, 10000, 8,
2000.0f, 0.0f, 0.1f);
```

Comments

- For short measurements of a few seconds, it is common practice using the same value for ulArrayLength and for udMaxSamples.
- The internal buffer is allocated in the heap memory of your application.
- The larger the DLL-internal buffer is, the larger the memory consumption. For typical time-limited sampling, this is not a problem and the memory size will be between a few kBytes up to a few MBytes. This applies even to larger systems.
  However, if you would use for example 256 measurement channels of 32 Bit size, and ulDLLArrayLength would be 100000, then 100 MBytes would be required. This may be too large for your heap memory.

4.3.11.2 NMX_Sampling_PrepareCustomTFT_1

Minimum DLL-Version: V1.1.0.11

This function is used to prepare a time-based sampling with optional triggering, arithmetic average filtering and "recording tail values after stop". Before starting the sampling, it must be prepared.

See also the HowTo 288-Guide.

Definition

```
NMX_STATUS NMX_Sampling_PrepareCustomTFT_1(
      NMX_PHANDLE pHandle,
      unsigned long ulSamplePeriod,
      unsigned long ulFilteredPeriod,
      unsigned long ulArrayLength,
      unsigned long long udMaxSamples,
```

```
    unsigned long ulNTailSamples,
    unsigned long ulTriggerDigiInByteNo,
    unsigned long ulTriggerDigiInBitNo,
    unsigned long ulTriggerMode,
    unsigned long ulTriggerPolarity);
```

Parameter

---

pHandle

Connection Handle 162.

ulSamplePeriod

Time period between two samples of the measurement hardware in µs, e.g. 1000 for 1ms.
This is the sampling period (-> measurement frequency) which will be used by the measurement system for acquiring the time-triggered measurement samples.
See the chapter "Sampling Speed with Irinos" for more information. 141

ulFilteredPeriod

Time period between two filtered samples provided to the application / measurement software in µs, e.g. 5000 for 5ms.
This value must be an integer multiple of ulSamplePeriod.
With this parameter, an average filter can be applied to the measurement samples. To disable the Filter, use the same values for ulSamplePeriod and ulFilteredPeriod.

Example: ulSamplePeriod is 1ms (=1000), then ulFilteredPeriod can be 1, 2, 3, 4, 5, ..., 10, ... 100ms.
If the SamplePeriod is 1ms and the FilteredPeriod is 5ms, then an average filter of the past 5 samples is used.

ulArrayLength

Data received from the device is internally buffered in the DLL. This parameter specifies the internal buffer length in samples.
Example: The filtered period is 1ms -> 1000Samples/s and the buffer shall contain a maximum of 10 seconds. Then `ulDLLArrayLength = 10000;`

udMaxSamples

Maximum number of samples, which shall be recorded.
Use 0 for endless sampling.
Example: The sampling speed is 1ms -> 1000Samples/s. Data shall be recorded for a maximum of 15 seconds. Then udMaxSamples = 15000;

ulNTailSamples

If this value is 0, then sampling will be stopped immediately, when the stop condition occurs.
If this value is >0, then ulNTailSamples will be sampled after the stop condition occurs.
The tail samples could for example be used for additional filtering.

ulTriggerDigiInByteNo

Number of the digital input byte, which contains the digital input information. Counting starts with 0.
This parameter is only relevant, if Triggering is used (ulTriggerMode > 0).

ulTriggerDigiInBitNo

Number of the bit in ulTriggerDigiInByteNo, which contains the digital input information. Counting starts with 0, maximum is 7.
This parameter is only relevant, if Triggering is used (ulTriggerMode > 0).

Example: A measurement system has 16 digital inputs, counted from 1 to 16. For input no 11, the parameters are:
ulTriggerDigiInByteNo = 1;
ulTriggerDigiInBitNo = 2;

ulTriggerMode

See general information about trigger mode 163.
The following modes are supported:
0 = Trigger disabled
1 = Edge
2 = Level
3 = Edge Start
4 = Level Once

ulTriggerPolarity

1 = falling edge / low active
2 = rising edge / high active
This parameter is only relevant, if Triggering is used (ulTriggerMode > 0).

Typical function call (C example)

Time-limited: `NMX_Sampling_PrepareCustomTFT_1(pHandle, 1000, 5000, 10000, 10000, 4, 0, 0, 0, 0);`
Endless: `NMX_Sampling_PrepareCustomTFT_1(pHandle, 1000, 5000, 10000, 0, 0, 1, 4, 2);`

Comments

- For short measurements of a few seconds, it is common practice using the same value for ulArrayLength and for udMaxSamples.
- The internal buffer is allocated in the heap memory of your application.
- The larger the DLL-internal buffer is, the larger the memory consumption. For typical time-limited sampling, this is not a problem and the memory size will be between a few kBytes up to a few MBytes. This applies even to larger systems.
  However, if you would use for example 256 measurement channels of 32 Bit size, and ulDLLArrayLength would be 100000, then 100 MBytes would be required. This may be too large for your heap memory.

4.3.12 Diagnostics

Your measurement system is equipped with an internal diagnostic service. It supports identifying special events and errors.

4.3.12.1 NMX_DiagClearEvent_1

This function allows clearing an event, which has occurred.

Definition

```
NMX_STATUS NMX_DiagClearEvent_1(
      NMX_PHANDLE pHandle,
      unsigned long ulBoxNo,
      unsigned char ucEventNo);
```

Parameter

pHandle

Connection Handle 162.

ulBoxNo

Number of the Box, where the event shall be cleared.
The first measurement box has the number 0.
In a system with 5 measurement boxes, these are numbered 0..4.

ucEventNo

Number of the Event, which shall be cleared.
Current events are readout using NMX_StaticGet32_1 198 -> Box-Status.

Typical function call (C example)

```
NMX_DiagClearEvent_1(pHandle, 0, 13);
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS DiagClearEvent_1(
                  System::IntPtr pHandle,
                  System::UInt32 ulBoxNo,
                  System::Byte ucEventNo);
```

Comments

Events are always handled Box-wise.

4.3.12.2 NMX_DiagGetEventText_1

This function provides a textual description of an event. It could for example be used to display the event description instead or together with an event number.

Definition

```
NMX_STATUS NMX_DiagGetEventText_1(
      NMX_PHANDLE pHandle,
```

```
        unsigned char ucEventNo,
        char* pcText, unsigned long ulSizeofText);
```

Parameter

---

pHandle

Connection Handle 162.

ucEventNo

Number of the event.

pcText

ASCII based string with event description. The maximum string length is 129 characters (128 + Termination).

ulSizeofText

Maximum size of pcText in Bytes/Characters.

Typical function call (C example)

```
NMX_DiagGetEventText_1(pHandle, 15, acText, sizeof(acText));
```

.Net DLL 152 specific implementation

```
NMX_MSTATUS DiagGetEventText_1(
                  System::IntPtr pHandle,
                  System::Byte ucEventNo,
                  System::String ^%strText);
```

No Length/Sizeof-Parameter is required for the string strText, since this information is not required in a .Net environment. It is directly returned as Unicode-strings. Hence no additional conversion is required.

Comments

4.3.12.3 NMX_SetDateTime_1

This function is used to update the current date & time at the device by using the PC time.

Definition

```
NMX_STATUS NMX_SetDateTime_1(
      NMX_PHANDLE pHandle);
```

Parameter

pHandle

Connection Handle 162

Typical function call (C example)

```
NMX_SetDateTime_1(pHandle);
```

.Net DLL 152 *specific implementation*

```
NMX_MSTATUS SetDateTime_1(System::IntPtr pHandle);
```

Comments

The measurement system does not have an internal Realtime-Clock. In order to provide advanced information in the diagnostic memory, the current date & time is transferred to the measurement system while connecting.

The internal clock does not take into account leap years of leap seconds. Further it has no high accuracy. Hence it is recommended to rewrite the date/time information once a day. It can be rewritten any time.

The date & time information has no effect on measurement or sampling. It is just for the diagnostic memory.

4.4 HowTo

This chapter is intended as an implementation guideline for the NMX DLL in order to get started quickly.

Another good source for the first steps, is the Demo application, which is available for various programming environments.

4.4.1 Small Measurement Application

For simple applications with low requirements regarding measurement speed, only a small portion of the NMX DLLs functionality is required.

In case reading measurement values from time to time, only 3 simple steps need to be implemented into your application:

1. Establishing a connection 239 to the measurement system. This is typically done after the application has been started.
2. Closing the connection 241 to the measurement system. This is typically done before the application is closed.
3. Reading the measurement values cyclically 243, e.g. within a timer routine. It is good practice to check here for a connection failure. This is done by evaluating the return value of the function NMX_StaticGet32_1 198.

Please note: As an alternative to the NMX DLL, a very simple ASCII based interface is available. It can be accessed either via Telnet (similar to RS232) or via UDP. Please consult the respective documentation.

4.4.2 Establishing a connection

A connection is established using NMX_DeviceIPv4Open_1 173. All required connection parameters are assigned via its function parameters. No additional configuration file or similar is required.

Before establishing a connection, a connection handle 162 must be declared.

Establishing a connection can last a few hundred milliseconds, since the Windows IP stack may need to determine device-specific network data (e.g. MAC address).

Please note: if a firewall is active, the connection must be allowed by this firewall. With common firewalls, e.g. Windows Firewall, the user is asked one-time, if the connection shall be allowed. This has to be answered with "Yes".

In the following examples, the connection is established to the IP address 192.168.3.99. The port numbers 22517 and 22516 are fixed.
It is strongly recommended to store all these parameters in an INI-File, XML-File, in the Registry or something similar.

C / C++

```
// Global definition
NMX_PHANDLE pHandle = NULL;

// Establish the connection.
NMX_STATUS tStatus = NMX_DeviceIPv4Open_1(192, 168, 3, 99,
22517, 22516, &pHandle);
if (tStatus == NST_SUCCESS) {
      // Connection has been established successfully.
      // Do some further initialization here, e.g. register
notifications.
}
else {
      // Failed establishing the connection. Show an error
message.
}
```

Delphi

```
// Global definition
pNmxHandle : NMX_PHANDLE;

// Example: Establish connection within a button event handler.
procedure TMainForm.btnConnectClick(Sender: TObject);
var
   tNmxStat : NMX_STATUS;
begin
   tNmxStat := NMX_DeviceIPv4Open_1(192, 168, 3, 99, 22517, 22516, pNmxHa
   if (tNmxStat = NST_SUCCESS) then begin
      // Connection has been established successfully.
      // Do some further initialization here, e.g. register notifications
   end
   else begin
      // Failed establishing the connection. Show an error message.
   end;
end;
```

C# / .Net

```
// Global definition
IntPtr pDevice = IntPtr.Zero;

// Establish the connection.
if (cNmx.DeviceIPv4Open_1(192, 168, 3, 99, 22517, 22516, ref
pDevice) == NMX_MSTATUS.SUCCESS) {
      // Connection has been established successfully.
      // Do some further initialization here, e.g. register
notifications.
}
else {
      // Failed establishing the connection. Show an error
message.
}
```

4.4.3 Closing a connection

A connection is closed using NMX_DeviceClose_1 175.

Before closing an application, which used the NMX DLL, it must be ensured that all connections are closed. It is common practice to call NMX_DeviceClose_1 175 always before closing the measurement application. If no connection is established, nothing happens.

C / C++

```
NMX_DeviceClose_1(&pHandleNmx);
```

Delphi

```
NMX_DeviceClose_1(pNmxHandle);
```

C# / .Net

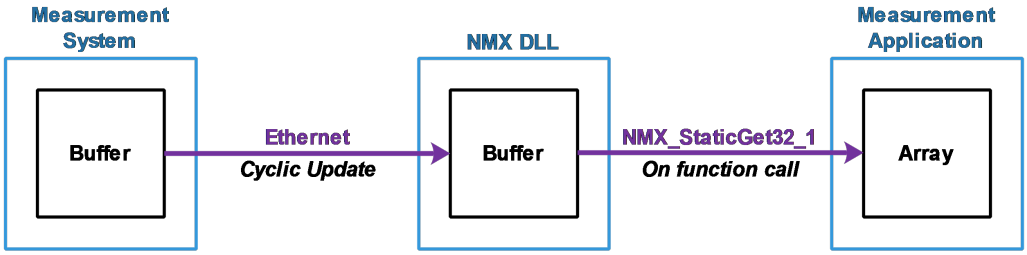
```
cNmx.DeviceClose_1(ref pDevice);
```

4.4.4 Reading static data

Reading static data is in most cases an essential part of the measurement application. It is very simple to integrate.

The static data is transferred cyclically from the measurement system to the NMX DLL. Here it is stored in the internal buffer.
The typical update rate is approximately 30 Hz, but it is not guaranteed.
This cyclic update starts automatically after establishing the connection.

The measurement application can readout the static data from the DLL-internal buffer any time using NMX_StaticGet32_1 198. It will then get the newest data.



Typically all data is read-out via a single function call. There is neither a need nor a possibility to read each measurement channel value individually. The same applied to the other static data.

However, it is important to know, that calling the function NMX_StaticGet32_1 198 does not cause any additional data transfer. Thus there is no significant drawback calling it from different places of your source code (but only within the same thread). It is for example no problem reading static measurement values from one part of your software and reading digital input data from another part. Just set the unused pointers to NULL and the respective array size to 0.

There are two common ways of reading static data:

- Event based 245 after receiving a notification or
- Calling NMX_StaticGet32_1 cyclically 243.

The technically better solution is event based. It is strongly recommended, if sampling 250 is used. For very simple applications the cyclic call could be an alternative.

4.4.4.1 Cyclically (Polling)

Cyclically reading data, also known as polling, is the most simple way. It is done by calling NMX_StaticGet32_1 198 regularly.

Typically it is called in a time-interval of approximately 30-50 ms.

C / C++

```
// Global definition
#define N_STATIC_CH_DISPLAY         (64)  // Max number of
static channels used. Adapt it to your needs.
#define N_DIGIIO_DISPLAY            (128) // Max number of
digital inputs used. Adapt it to your needs.
#define N_MAX_BOXES                 (32)  // Max number of
boxes supported.


// ###-> Implement timer and timer-event-handler, Interval e.g.
30ms
private: System::Void tmrStatic_Tick(System::Object^  sender,
System::EventArgs^  e) {
      // Disable this timer
      tmrStatic->Enabled = false;

      // Update static data, if new data available.
      signed long aslMeasVal[N_STATIC_CH_DISPLAY];
      unsigned char aucHardStat[N_STATIC_CH_DISPLAY];
      unsigned char aucDigiIn[N_DIGIIO_DISPLAY / 8];
      unsigned char aucBoxStatus[N_MAX_BOXES];
      unsigned long ulNUpdates = 0;
      memset(aslMeasVal, 0, sizeof(aslMeasVal));
      memset(aucHardStat, 0, sizeof(aucHardStat));
      memset(aucDigiIn, 0, sizeof(aucDigiIn));
      memset(aucBoxStatus, 0, sizeof(aucBoxStatus));
      if (NMX_StaticGet32_1(pHandleNmx, aslMeasVal,
N_STATIC_CH_DISPLAY, aucHardStat, sizeof(aucHardStat),
aucDigiIn, sizeof(aucDigiIn), aucBoxStatus,
sizeof(aucBoxStatus), &ulNUpdates) == NST_SUCCESS) {
            // New data successfully received. Display,
Calculate, Store, ...
      }
      else {
            // Error reading data. Typical error:
NST_DX_TIMEOUT_COMMON due to communication breakdown.
      }
```

```
      // Re-enable this timer
      tmrStatic->Enabled = true;
}
```

Delphi

```
const cMaxTotalChannels = 256;      // Max number of static channels used. Ada
      cMaxDigiInBytes = 16;         // Max number of digital input bytes used
      cMaxBoxes = 32;               // Max number of boxes supported.

// ###-> Implement timer and timer-event-handler, Interval e.g. 30ms
procedure TMainForm.tmrUpdateStaticTimer(Sender: TObject);
var
   aslValues : array [0..(cMaxTotalChannels-1)] of integer;
   aucHardwareStatus : array [0..(cMaxTotalChannels-1)] of Byte;
   aucDigiInBytes : array [0..(cMaxDigiInBytes-1)] of Byte;
   aucBoxStatus : array [0..(cMaxBoxes-1)] of Byte;
   ulUpdateCtr : Cardinal;
   tStatus : NMX_STATUS;
begin
  tmrUpdateStatic := false;

  tStatus := NMX_StaticGet32_1(pNmxHandle, @aslValues[0], cMaxTotalChannels,
                               @aucHardwareStatus[0], cMaxTotalChannels,
                               @aucDigiInBytes[0], cMaxDigiInBytes,
                               @aucBoxStatus[0], cMaxBoxes,
                               ulUpdateCtr);
  if (tStatus = NST_SUCCESS) then begin
     // New data successfully received. Display, Calculate, Store, ...
  end
  else begin
     // Error reading data. Typical error: NST_DX_TIMEOUT_COMMON due to co
  end;

  tmrUpdateStatic := true;
end;
```

C# / .Net

```
const int N_STATIC_CH_DISPLAY = 64;  // Max number of static
channels used. Adapt it to your needs.
const int N_DIGIIO_DISPLAY = 128;    // Max number of digital
input bytes used. Adapt it to your needs.
const int N_BOXES_MAX = 32;          // Max number of boxes
supported.


// ###-> Implement timer and timer-event-handler, Interval e.g.
30ms
```

```
private void tmrStatic_Tick(object sender, EventArgs e)
{
    /* Disable this timer */
    tmrStatic.Enabled = false;

    Int32[] aslMeasVal = new Int32[N_STATIC_CH_DISPLAY];
    Byte[] aucHardStat = new Byte[N_STATIC_CH_DISPLAY];
    Byte[] aucDigiIn = new Byte[N_DIGIIO_DISPLAY / 8];
    Byte[] aucBoxStatus = new Byte[N_BOXES_MAX];
    UInt32 ulNUpdates = 0;
    if (cNmx.StaticGet32_1(pDevice, aslMeasVal, aucHardStat,
aucDigiIn, aucBoxStatus, ref ulNUpdates) ==
NMX_MSTATUS.SUCCESS)
    {
        // New data successfully received. Display, Calculate,
Store, ...
    }
    else
    {
        // Error reading data. Typical error:
NST_DX_TIMEOUT_COMMON due to communication breakdown.
    }

    /* Re-enable this timer */
    tmrStatic.Enabled = true;
}
```

4.4.4.2 Event based

Reading event based first requires registering a notification[176]. In the following example, message based notifications are used.

A good practice is:

Every-time the notification NMXNOTIFY_NEW_STATIC32 occurs (= a message is received), a flag is set. In a separate timer routine, this flag is checked. If set, static data is readout via NMX_StaticGet32_1[198].

If event based data read-out is used, then a connection breakdown is typically handled using the notification NMXNOTIFY_FAILURE_DATA_EXCHANGE. Therefore no special error handling is done in the following sample code.

C / C++

```
// Define Message. In VisualStudio, WM_USER is defined in
WinUser.h as:
```

```
// #define WM_USER 0x0400
#define     WM_MESSAGE_NEW_STATIC32             (WM_USER +
NMXNOTIFY_NEW_STATIC32)

// Global declaration
BOOL bNewStatic = false;

// Global definition
#define N_STATIC_CH_DISPLAY         (64)  // Max number of
static channels used. Adapt it to your needs.
#define N_DIGIIO_DISPLAY            (128) // Max number of
digital inputs used. Adapt it to your needs.
#define N_MAX_BOXES                 (32)  // Max number of
boxes supported.


// ###-> Register notification, e.g. directly after connecting.
if (NMX_RegisterMessage_1(pHandle, NMXNOTIFY_NEW_STATIC32,
static_cast<HWND>(Handle.ToPointer()), WM_MESSAGE_NEW_STATIC32,
0, 0) != NST_SUCCESS) {
      // Handle error
}


// ###-> Implement message handler
protected:  virtual void WndProc(Message% m) override
{
      /* Listen for operating system messages. */
      switch (m.Msg)
      {
      case WM_MESSAGE_NEW_STATIC32:
            bNewStatic = true;
            break;
      default:
            /* Pass all standard messages to the GUI form. */
            Form::WndProc(m);
            break;
      }
}

// ###-> Implement timer and timer-event-handler, Interval e.g.
30ms
private: System::Void tmrStatic_Tick(System::Object^  sender,
System::EventArgs^  e) {
      // Disable this timer
      tmrStatic->Enabled = false;

      // Update static data, if new data available.
```

```
	if (bNewStatic != false)
	{
		bNewStatic = false;
		signed long aslMeasVal[N_STATIC_CH_DISPLAY];
		unsigned char aucHardStat[N_STATIC_CH_DISPLAY];
		unsigned char aucDigiIn[N_DIGIIO_DISPLAY / 8];
		unsigned char aucBoxStatus[N_MAX_BOXES];
		unsigned long ulNUpdates = 0;
		memset(aslMeasVal, 0, sizeof(aslMeasVal));
		memset(aucHardStat, 0, sizeof(aucHardStat));
		memset(aucDigiIn, 0, sizeof(aucDigiIn));
		memset(aucBoxStatus, 0, sizeof(aucBoxStatus));
		if (NMX_StaticGet32_1(pHandleNmx, aslMeasVal,
N_STATIC_CH_DISPLAY, aucHardStat, sizeof(aucHardStat),
aucDigiIn, sizeof(aucDigiIn), aucBoxStatus,
sizeof(aucBoxStatus), &ulNUpdates) == NST_SUCCESS)
		{
			// New data successfully received. Display,
Calculate, Store, ...
		}
	}

	// Re-enable this timer
	tmrStatic->Enabled = true;
}
```

Delphi

```
// Define Message for "new static data available"
const WM_MESSAGE_NMX_NEWSTATIC = WM_USER + $10;
       cMaxTotalChannels = 256;     // Max number of static channels used. Ada
       cMaxDigiInBytes = 16;        // Max number of digital input bytes used
       cMaxBoxes = 32;              // Max number of boxes supported.

// Declaration of class variables.
bNewStaticData : Boolean;

// Declaration of message handler. Integrate it into the class declaration
procedure OnMessageNewStatic32(var Msg: TMessage); message WM_MESSAGE_NMX

// ###-> Register notification, e.g. directly after connecting.
NMX_RegisterMessage_1(pNmxHandle, NMXNOTIFY_NEW_STATIC32,
MainForm.Handle, WM_MESSAGE_NMX_NEWSTATIC, 0, 0);

// ###-> Implement message handler
procedure TMainForm.OnMessageNewStatic32(var Msg: TMessage);
begin
   bNewStaticData := true;
```

```
end;

// ###-> Implement timer and timer-event-handler, Interval e.g. 30ms
procedure TMainForm.tmrUpdateStaticTimer(Sender: TObject);
var
   aslValues : array [0..(cMaxTotalChannels-1)] of integer;
   aucHardwareStatus : array [0..(cMaxTotalChannels-1)] of Byte;
   aucDigiInBytes : array [0..(cMaxDigiInBytes-1)] of Byte;
   aucBoxStatus : array [0..(cMaxBoxes-1)] of Byte;
   ulUpdateCtr : Cardinal;
   tStatus : NMX_STATUS;
begin
  tmrUpdateStatic := false;

  if bNewData then begin
   bNewData := false;

   tStatus := NMX_StaticGet32_1(pNmxHandle, @aslValues[0], cMaxTotalChannels,
                                @aucHardwareStatus[0], cMaxTotalChannels,
                                @aucDigiInBytes[0], cMaxDigiInBytes,
                                @aucBoxStatus[0], cMaxBoxes,
                                ulUpdateCtr);
   if (tStatus = NST_SUCCESS) then begin
      // New data successfully received. Display, Calculate, Store, ...
   end;
  end;

  tmrUpdateStatic := true;
end;
```

C# / .Net

```
const int WM_MESSAGE_NEW_STATIC32 = WM_USER + 0x10;  // Define
Message for "new static data available"
const int N_STATIC_CH_DISPLAY = 64;  // Max number of static
channels used. Adapt it to your needs.
const int N_DIGIIO_DISPLAY = 128;    // Max number of digital
input bytes used. Adapt it to your needs.
const int N_BOXES_MAX = 32;          // Max number of boxes
supported.

// Global declaration
Boolean bNewStatic = false;

// ###-> Register notification, e.g. directly after connecting.
if (cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.NEW_STATIC32, Handle, WM_MESSAGE_NEW_STATIC32,
0, 0) != NMX_MSTATUS.SUCCESS)
{
    // Handle error
}

// ###-> Implement message handler
```

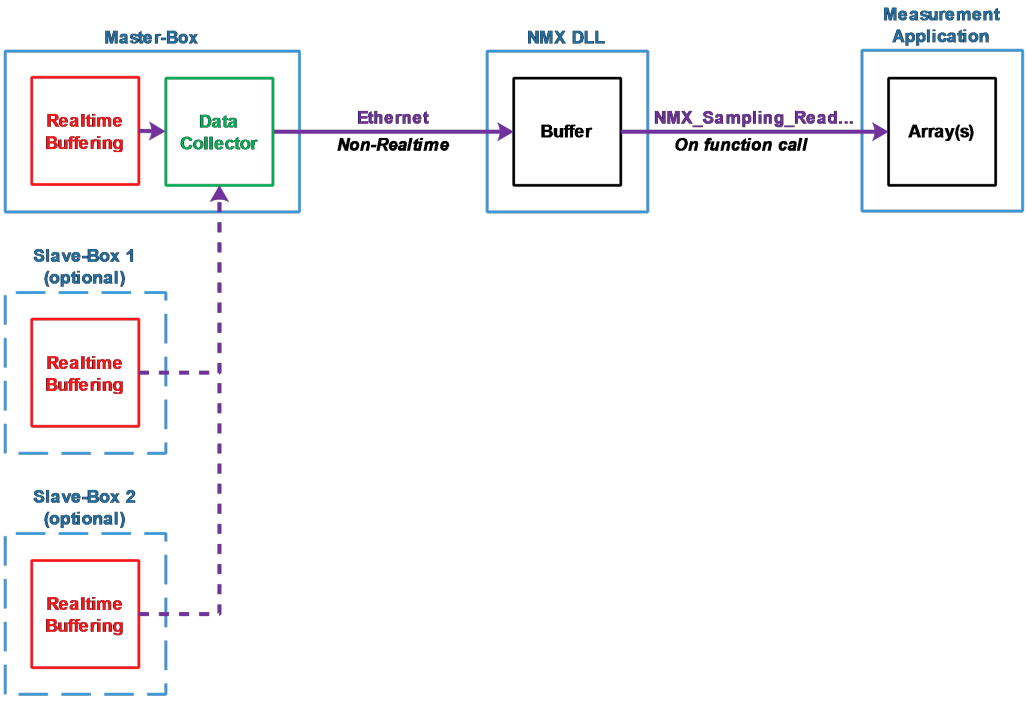
```
protected override void WndProc(ref Message m)
{
    // Listen for operating system messages.
    switch (m.Msg)
    {
        case WM_MESSAGE_NEW_STATIC32:
            bNewStatic = true;
            break;

        default:
            /* Pass all standard messages to the GUI form. */
            base.WndProc(ref m);
            break;
    }
}

// ###-> Implement timer and timer-event-handler, Interval e.g.
30ms
private void tmrStatic_Tick(object sender, EventArgs e)
{
    /* Disable this timer */
    tmrStatic.Enabled = false;

    if (bNewStatic != false)
    {
        bNewStatic = false;

        Int32[] aslMeasVal = new Int32[N_STATIC_CH_DISPLAY];
        Byte[] aucHardStat = new Byte[N_STATIC_CH_DISPLAY];
        Byte[] aucDigiIn = new Byte[N_DIGIIO_DISPLAY / 8];
        Byte[] aucBoxStatus = new Byte[N_BOXES_MAX];
        UInt32 ulNUpdates = 0;
        if (cNmx.StaticGet32_1(pDevice, aslMeasVal,
aucHardStat, aucDigiIn, aucBoxStatus, ref ulNUpdates) ==
NMX_MSTATUS.SUCCESS)
        {
            // New data successfully received. Display,
Calculate, Store, ...
        }
        else
        {
            // Error reading data. Typical error:
NST_DX_TIMEOUT_COMMON due to communication breakdown.
        }
    }

    /* Re-enable this timer */
    tmrStatic.Enabled = true;
}
```

4.4.5 Sampling

Using sampling, data is acquired in realtime (see also chapter "Static vs. Sampling 140"). There are two types of sampling:

- Time-limited sampling, and
- Endless sampling.

Both of them use the same buffering technique. The sampled data is first stored in the internal buffer(s) of the measurement system in realtime. For the data transfer to the PC, there are no realtime requirements.



Time-limited sampling provides more opportunities in regard of the measurement speed and the amount of sampled data. Full speed is supported using all measurement channels.

As its name says, it does not run all the time. Instead, it is started before each measurement cycle. It then runs until the measurement cycle is finished. The typical duration is a few seconds.

Endless sampling provides more flexibility. After starting, the sampled data is available endlessly, similar to an endless data stream.

The maximum data rate depends on the amount of sampled data (= amount of measurement channels used). However, for many applications, it is fast enough. Consult the users manual of the measurement system for more information about the maximum sampling rate. Typically "1000 Samples/s

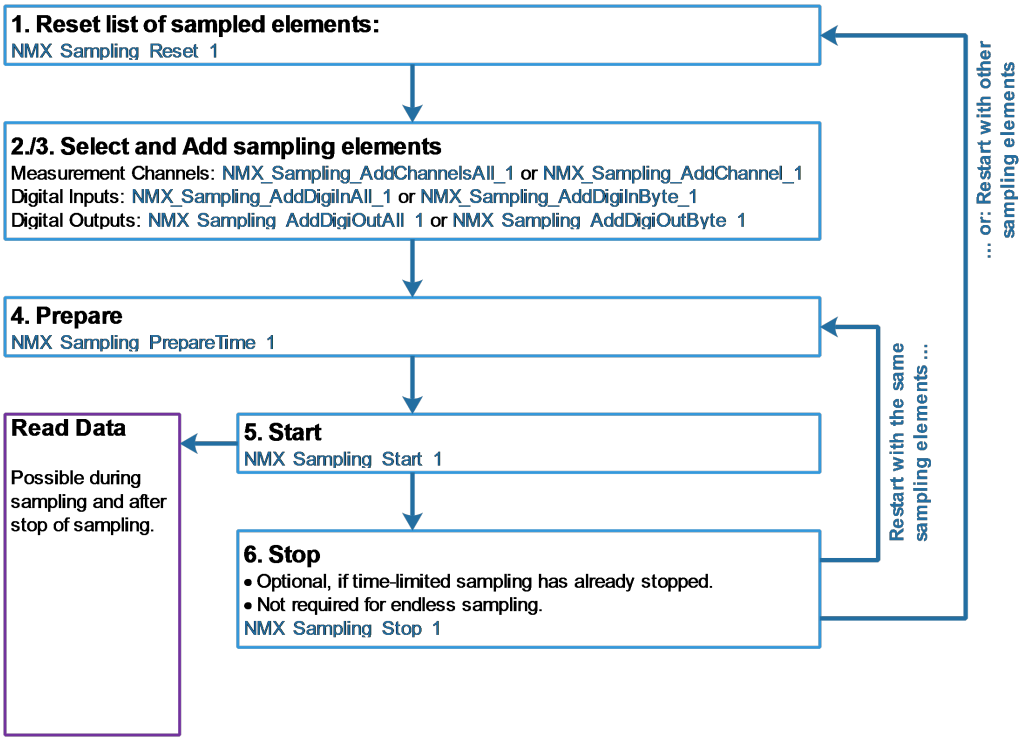
per measurement channel" or more are possible with many measurement channels.

Regarding the integration into the measurement software, both sampling types are similar. The same function calls are used.

Fundamentals are:

- Sampling does not care about measurement boxes. It does not matter whether a single or multiple boxes are used. All measurement channels and digital in-/outputs across the whole measurement system can be used. The boxes are synchronized. The Master-Box collects all data from the Slave-Boxes, sorts it and sends it to the NMX DLL. There it is stored in a table. See the chapter "Reading sampled data 267" for more details.
- At each sampling point, the values of all sampling elements are buffered/stored together with a sample counter.
- As the first step, the sampling elements have to be defined. A sampling element can be a measurement channel, a digital input byte or a digital output byte.
  It is possible to add all of them or only a selection.
- The next step is preparing the sampling.
  Here the sample rate (-> sample period) is defined. Further it is defined, whether it is a time-limited or an endless sampling.
- Then the sampling can be started.
- The data transfer to the PC begins immediately.

These steps are also shown in the following diagram:



As visible in the diagram, the sampling elements only need to be defined once, if it is started repeatedly.

4.4.5.1 Start endless time-based sampling

For the following sample code, endless sampling with a system having 32 measurement channels and 64 digital in-/outputs is shown.

64 digital in-/outputs means there are 8 bytes each.

The sample rate shall be 1000 Samples/s.

The DLL-internal buffer shall be large enough to buffer for 5 seconds, which equals 5000 samples.

C / C++: Start with <u>all</u> measurement channels and digital I/Os

```
unsigned long ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (NMX_Sampling_Reset_1(pHandle) == NST_SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}
```

```
// ###-> 2/3. Add sampling elements
if ( (NMX_Sampling_AddChannelsAll_1(pHandle, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddDigiInAll_1(pHandle, &ulNElements) ==
NST_SUCCESS)   &&
     (NMX_Sampling_AddDigiOutAll_1(pHandle, &ulNElements) ==
NST_SUCCESS)    ) {
      // Successfully added all sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (NMX_Sampling_PrepareTime_1(pHandle, 1000/*µs*/,
5000/*Buffer-Size*/, 0/*0=NoLimit*/) == NST_SUCCESS) {
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (NMX_Sampling_Start_1(pHandle) == NST_SUCCESS) {
      // Start successful.
      // Reading sampled data should start now/soon to avoid a
buffer overflow.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

C / C++: Start with a selection of measurement channels and digital inputs

```
unsigned long ulNElements = 0;
```

```
// ###-> 1. Reset list of sampling elements
if (NMX_Sampling_Reset_1(pHandle) == NST_SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}


// ###-> 2/3. Add measurement channels 0, 5 & 17
//            Add digital input byte 0
if ( (NMX_Sampling_AddChannel_1(pHandle, 0, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddChannel_1(pHandle, 5, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddChannel_1(pHandle, 17, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddDigiInByte_1(pHandle, 0, &ulNElements) ==
NST_SUCCESS) ) {
      // Successfully added selected sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (NMX_Sampling_PrepareTime_1(pHandle, 1000/*µs*/,
5000/*Buffer-Size*/, 0/*0=NoLimit*/) == NST_SUCCESS) {
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (NMX_Sampling_Start_1(pHandle) == NST_SUCCESS) {
      // Start successful.
      // Reading sampled data should start now/soon to avoid a
buffer overflow.
}
else {
```

```
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

Delphi: Start with <u>all</u> measurement channels and digital I/Os

```
// ###-> 1. Reset list of sampling elements
if cNmx.Sampling_Reset_1(pHandleNmx) = NST_SUCCESS then begin
      // List of sampling elements has been reset successfully.
end
else begin
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      exit;
end;


// ###-> 2/3. Add sampling elements
if (cNmx.Sampling_AddChannelsAll_1(pHandleNmx, ulNElements) = NST_SUCCESS) And
      (cNmx.Sampling_AddDigiInAll_1(pHandleNmx, ulNElements) = NST_SUCCESS) And
      (cNmx.Sampling_AddDigiOutAll_1(pHandleNmx, ulNElements) = NST_SUCCESS) then b
      // Successfully added all sampling elements
end
else begin
      // Failed adding sampling elements
      // Do some error handling.
      exit;
end;


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs sample period)
if cNmx.Sampling_PrepareTime_1(pHandleNmx, 1000{µs}, 5000{BufferSize}, 0{0=NoLimit}
      // Sampling successfully prepared
end
else begin
      // Failed preparing sampling
      // Do some error handling.
      exit;
end;


// ###-> 5. Start sampling
if cNmx.Sampling_Start_1(pHandleNmx) = NST_SUCCESS then begin
      // Start successful.
      // Reading sampled data should start now/soon to avoid a buffer overflow.
end
else begin
      // Failed starting sampling
      // Do some error handling.
      exit;
end;
```

Delphi: Start with a selection of measurement channels and digital inputs

```
// ###-> 1. Reset list of sampling elements
if cNmx.Sampling_Reset_1(pHandleNmx) = NST_SUCCESS then begin
      // List of sampling elements has been reset successfully.
end
else begin
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      exit;
end;


// ###-> 2/3. Add sampling elements
if (cNmx.Sampling_AddChannel_1(pHandleNmx, 0, ulNTotal) = NST_SUCCESS) And
      (cNmx.Sampling_AddChannel_1(pHandleNmx, 5, ulNTotal) = NST_SUCCESS) And
      (cNmx.Sampling_AddChannel_1(pHandleNmx, 17, ulNTotal) = NST_SUCCESS) And
      (cNmx.Sampling_AddDigiInByte_1(pHandleNmx, 0, ulNTotal) = NST_SUCCESS) then b
      // Successfully added selected sampling elements
end
else begin
      // Failed adding sampling elements
      // Do some error handling.
      exit;
end;


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs sample period)
if cNmx.Sampling_PrepareTime_1(pHandleNmx, 1000{µs}, 5000{BufferSize}, 0{0=NoLimit}
      // Sampling successfully prepared
end
else begin
      // Failed preparing sampling
      // Do some error handling.
      exit;
end;


// ###-> 5. Start sampling
if cNmx.Sampling_Start_1(pHandleNmx) = NST_SUCCESS then begin
      // Start successful.
      // Reading sampled data should start now/soon to avoid a buffer overflow.
end
else begin
      // Failed starting sampling
      // Do some error handling.
      exit;
end;
```

C# / .Net: Start with <u>all</u> measurement channels and digital I/Os

```
UInt32 ulNElements = 0;
```

```
// ###-> 1. Reset list of sampling elements
if (cNmx.Sampling_Reset_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}

// ###-> 2/3. Add sampling elements
if ( (cNmx.Sampling_AddChannelsAll_1(pDevice, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddDigiInAll_1(pDevice, ref ulNElements) ==
NMX_MSTATUS.SUCCESS)   &&
     (cNmx.Sampling_AddDigiOutAll_1(pDevice, ref ulNElements)
== NMX_MSTATUS.SUCCESS)    ) {
      // Successfully added all sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (cNmx.Sampling_PrepareTime_1(pDevice, 1000/*µs*/,
5000/*Buffer-Size*/, 0/*0=NoLimit*/) == NMX_MSTATUS.SUCCESS)
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (cNmx.Sampling_Start_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // Start successful.
      // Reading sampled data should start now/soon to avoid a
buffer overflow.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
```

```
}
```

C# / .Net: Start with a selection of measurement channels and digital inputs

```
UInt32 ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (cNmx.Sampling_Reset_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}

// ###-> 2/3. Add sampling elements
if ( (cNmx.Sampling_AddChannel_1(pDevice, 0, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddChannel_1(pDevice, 5, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddChannel_1(pDevice, 17, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddDigiInByte_1(pDevice, 0, ref
ulNElements) != NMX_MSTATUS.SUCCESS) ) {
      // Successfully added all sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (cNmx.Sampling_PrepareTime_1(pDevice, 1000/*µs*/,
5000/*Buffer-Size*/, 0/*0=NoLimit*/) == NMX_MSTATUS.SUCCESS)
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}
```

```
// ###-> 5. Start sampling
if (cNmx.Sampling_Start_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // Start successful.
      // Reading sampled data should start now/soon to avoid a
buffer overflow.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

4.4.5.2 Start time-limited sampling

For the following sample code, time-limited sampling with a system having 32 measurement channels and 64 digital in-/outputs is shown.

64 digital in-/outputs means there are 8 bytes each.

The sample rate shall be 1000 Samples/s.

The maximum duration shall be 10 seconds, which equals 10000 Samples. The DLL-internal buffer shall be large enough to store all sampled data.

C / C++: Start with <u>all</u> measurement channels and digital I/Os

```
unsigned long ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (NMX_Sampling_Reset_1(pHandle) == NST_SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}


// ###-> 2/3. Add sampling elements
if ( (NMX_Sampling_AddChannelsAll_1(pHandle, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddDigiInAll_1(pHandle, &ulNElements) ==
NST_SUCCESS)   &&
     (NMX_Sampling_AddDigiOutAll_1(pHandle, &ulNElements) ==
NST_SUCCESS)    ) {
      // Successfully added all sampling elements
}
```

```
else {
	// Failed adding sampling elements
	// Do some error handling.
	return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (NMX_Sampling_PrepareTime_1(pHandle, 1000/*µs*/, 10000,
10000/*Samples*/) == NST_SUCCESS) {
	// Sampling successfully prepared
}
else {
	// Failed preparing sampling
	// Do some error handling.
	return;
}

// ###-> 5. Start sampling
if (NMX_Sampling_Start_1(pHandleNmx) == NST_SUCCESS) {
	// Start successful.
}
else {
	// Failed starting sampling
	// Do some error handling.
	return;
}
```

C / C++: Start with a selection of measurement channels and digital inputs

```
unsigned long ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (NMX_Sampling_Reset_1(pHandle) == NST_SUCCESS) {
	// List of sampling elements has been reset successfully.
}
else {
	// Failed resetting the list of sampling elements.
	// Do some error handling.
	return;
}


// ###-> 2/3. Add measurement channels 0, 5 & 17
```

```
//            Add digital input byte 0
if ( (NMX_Sampling_AddChannel_1(pHandle, 0, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddChannel_1(pHandle, 5, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddChannel_1(pHandle, 17, &ulNElements) ==
NST_SUCCESS) &&
     (NMX_Sampling_AddDigiInByte_1(pHandle, 0, &ulNElements) ==
NST_SUCCESS) ) {
      // Successfully added selected sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (NMX_Sampling_PrepareTime_1(pHandle, 1000/*µs*/, 10000,
10000/*Samples*/) == NST_SUCCESS) {
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (NMX_Sampling_Start_1(pHandleNmx) == NST_SUCCESS) {
      // Start successful.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

Delphi: Start with <u>all</u> measurement channels and digital I/Os

```
// ###-> 1. Reset list of sampling elements
if cNmx.Sampling_Reset_1(pHandleNmx) = NST_SUCCESS then begin
      // List of sampling elements has been reset successfully.
end
else begin
      // Failed resetting the list of sampling elements.
      // Do some error handling.
```

```
      exit;
end;


// ###-> 2/3. Add sampling elements
if (cNmx.Sampling_AddChannelsAll_1(pHandleNmx, ulNElements) = NST_SUCCESS) And
      (cNmx.Sampling_AddDigiInAll_1(pHandleNmx, ulNElements) = NST_SUCCESS) And
      (cNmx.Sampling_AddDigiOutAll_1(pHandleNmx, ulNElements) = NST_SUCCESS) then b
      // Successfully added all sampling elements
end
else begin
      // Failed adding sampling elements
      // Do some error handling.
      exit;
end;


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs sample period)
if cNmx.Sampling_PrepareTime_1(pHandleNmx, 1000{µs}, 10000, 10000{Samples}) = NST_S
      // Sampling successfully prepared
end
else begin
      // Failed preparing sampling
      // Do some error handling.
      exit;
end;


// ###-> 5. Start sampling
if cNmx.Sampling_Start_1(pHandleNmx) = NST_SUCCESS then begin
      // Start successful.
      // Reading sampled data should start now/soon to avoid a buffer overflow.
end
else begin
      // Failed starting sampling
      // Do some error handling.
      exit;
end;
```

Delphi: Start with a selection of measurement channels and digital inputs

```
// ###-> 1. Reset list of sampling elements
if cNmx.Sampling_Reset_1(pHandleNmx) = NST_SUCCESS then begin
      // List of sampling elements has been reset successfully.
end
else begin
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      exit;
end;


// ###-> 2/3. Add sampling elements
if (cNmx.Sampling_AddChannel_1(pHandleNmx, 0, ulNTotal) = NST_SUCCESS) And
```

```
      (cNmx.Sampling_AddChannel_1(pHandleNmx, 5, ulNTotal) = NST_SUCCESS) And
      (cNmx.Sampling_AddChannel_1(pHandleNmx, 17, ulNTotal) = NST_SUCCESS) And
      (cNmx.Sampling_AddDigiInByte_1(pHandleNmx, 0, ulNTotal) = NST_SUCCESS) then b
      // Successfully added selected sampling elements
end
else begin
      // Failed adding sampling elements
      // Do some error handling.
      exit;
end;


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs sample period)
if cNmx.Sampling_PrepareTime_1(pHandleNmx, 1000{µs}, 10000, 10000{Samples}) = NST_S
      // Sampling successfully prepared
end
else begin
      // Failed preparing sampling
      // Do some error handling.
      exit;
end;


// ###-> 5. Start sampling
if cNmx.Sampling_Start_1(pHandleNmx) = NST_SUCCESS then begin
      // Start successful.
      // Reading sampled data should start now/soon to avoid a buffer overflow.
end
else begin
      // Failed starting sampling
      // Do some error handling.
      exit;
end;
```

C# / .Net: Start with <u>all</u> measurement channels and digital I/Os

```
UInt32 ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (cNmx.Sampling_Reset_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}

// ###-> 2/3. Add sampling elements
if ( (cNmx.Sampling_AddChannelsAll_1(pDevice, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
```

```
     (cNmx.Sampling_AddDigiInAll_1(pDevice, ref ulNElements) ==
NMX_MSTATUS.SUCCESS)   &&
     (cNmx.Sampling_AddDigiOutAll_1(pDevice, ref ulNElements)
== NMX_MSTATUS.SUCCESS)    ) {
      // Successfully added all sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (cNmx.Sampling_PrepareTime_1(pDevice, 1000/*µs*/, 10000,
10000/*Samples*/) == NMX_MSTATUS.SUCCESS)
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (cNmx.Sampling_Start_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // Start successful.
      // Reading sampled data should start now/soon to avoid a
buffer overflow.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

C# / .Net: Start with a selection of measurement channels and digital inputs

---

```
UInt32 ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (cNmx.Sampling_Reset_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // List of sampling elements has been reset successfully.
}
else {
```

```
      // Failed resetting the list of sampling elements.
      // Do some error handling.
      return;
}

// ###-> 2/3. Add sampling elements
if ( (cNmx.Sampling_AddChannel_1(pDevice, 0, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddChannel_1(pDevice, 5, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddChannel_1(pDevice, 17, ref ulNElements)
== NMX_MSTATUS.SUCCESS) &&
     (cNmx.Sampling_AddDigiInByte_1(pDevice, 0, ref
ulNElements) != NMX_MSTATUS.SUCCESS) ) {
      // Successfully added all sampling elements
}
else {
      // Failed adding sampling elements
      // Do some error handling.
      return;
}


// ###-> 4. Prepare sampling with 1000 Samples/s (= 1000µs
sample period)
if (cNmx.Sampling_PrepareTime_1(pDevice, 1000/*µs*/, 10000,
10000/*Samples*/) == NMX_MSTATUS.SUCCESS)
      // Sampling successfully prepared
}
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (cNmx.Sampling_Start_1(pDevice) == NMX_MSTATUS.SUCCESS) {
      // Start successful.
      // Reading sampled data should start now/soon to avoid a
buffer overflow.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

4.4.5.3 Stop sampling

Stopping sampling is optional. It can be used to:

- Stopp an endless sampling.
- Stopp a time-limited sampling earlier than defined in the preparation step. Example: Sampling has been prepared with a length of 10 seconds. After 6.5 seconds the measurement is finished. Sampling shall now be stopped.

Inside the NMX DLL / measurement system, the maximum number of samples is stored during the preparation phase (see NMX_Sampling_PrepareTime_1) 218. This value can also be read-out via NMX_Sampling_GetStatus_1 226.

By stopping sampling, this value is adjusted to the new end of sampling.

Example: If sampling with 1000 samples/s has been prepared and started for a maximum duration of 10 seconds, the maximum number of samples is 10000. By calling the stop function after 6.5 seconds, this value is adjusted to 6500.

C / C++

```
// ###-> Stop sampling
if (NMX_Sampling_Stop_1(pHandle) == NST_SUCCESS) {
      // Successfully stopped
}
else {
      // Failed stopping sampling
      // Do some error handling.
}
```

Delphi

```
if cNmx.Sampling_Stop_1(pHandleNmx) = NST_SUCCESS then begin
      // Successfully stopped
end
else begin
      // Failed stopping sampling
      // Do some error handling.
end;
```

C# / .Net

```
if (cNmx.Sampling_Stop_1(pDevice) == NMX_MSTATUS.SUCCESS)
```

```
{
	// Successfully stopped
}
else
{
	// Failed stopping sampling
	// Do some error handling.
}
```

4.4.5.4 Reading sampled data

As soon as sampling has been started, data can be read-out.

For time-limited sampling, it is possible to wait until the measurement is finished and then read the data as a single block.

For endless-sampling, data must be read-out cyclically to ensure that the internal buffer of the NMX DLL does not overflow.

Sampled data is provided by the NMX DLL in the form of a table, or technically speaking in the form of a 2-dimensional array. Thereby:

- the rows are the samples
- and the columns are the sampling elements.

The following table provides an example:

Sampling Element	0	1	2	3	4	5
Source	Channel 0	Channel 1	Channel 2	Channel 3	Input Byte 0	Output Byte 0
Sample 0	21722	695	-11256	147236	0x00000003	0x00000000
Sample 1	21725	695	-11260	147924	0x00000003	0x00000000
Sample 2	21729	694	-11266	148626	0x00000003	0x00000002
Sample 3	21733	695	-11279	149301	0x00000003	0x00000002
Sample 4	21734	695	-11288	149998	0x00000003	0x00000002
Sample 5	21736	695	-11298	150654	0x00000004	0x00000002
Sample 6	21743	695	-11305	151131	0x00000004	0x00000004
Sample 7	21749	694	-11326	151840	0x00000004	0x00000004
...	...	...	...	...	...	...
Sample n	26225	695	15318	222516	0x00000008	0x00000000

The rows and columns are indexed as follows:

Sampling Element	0	1	2	3	4	5
Source	Channel 0	Channel 1	Channel 2	Channel 3	Input Byte 0	Output Byte 0
Sample 0	Col 0 | Row 0	Col 1 | Row 0	Col 2 | Row 0	Col 3 | Row 0	Col 4 | Row 0	Col 5 | Row 0
Sample 1	Col 0 | Row 1	Col 1 | Row 1	Col 2 | Row 1	Col 3 | Row 1	Col 4 | Row 1	Col 5 | Row 1
Sample 2	Col 0 | Row 2	Col 1 | Row 2	Col 2 | Row 2	Col 3 | Row 2	Col 4 | Row 2	Col 5 | Row 2
Sample 3	Col 0 | Row 3	Col 1 | Row 3	Col 2 | Row 3	Col 3 | Row 3	Col 4 | Row 3	Col 5 | Row 3
Sample 4	Col 0 | Row 4	Col 1 | Row 4	Col 2 | Row 4	Col 3 | Row 4	Col 4 | Row 4	Col 5 | Row 4
Sample 5	Col 0 | Row 5	Col 1 | Row 5	Col 2 | Row 5	Col 3 | Row 5	Col 4 | Row 5	Col 5 | Row 5
Sample 6	Col 0 | Row 6	Col 1 | Row 6	Col 2 | Row 6	Col 3 | Row 6	Col 4 | Row 6	Col 5 | Row 6
Sample 7	Col 0 | Row 7	Col 1 | Row 7	Col 2 | Row 7	Col 3 | Row 7	Col 4 | Row 7	Col 5 | Row 7
...	...	...	...	...	...	...
Sample n	Col 0 | Row n	Col 1 | Row n	Col 2 | Row n	Col 3 | Row n	Col 4 | Row n	Col 5 | Row n

There are two possibilities for reading this data: column-wise 268 and row-wise 273. None of them is generally better:

- Column-wise typically has a better performance, since less function calls are required. However, in most systems/applications, this is not even measurable.
  It could be the preferred option for time-limited sampling, if all samples shall be read-out as a whole after the sampling is finished.
- Row-wise could be a little bit easier to use, since there is no need to synchronize the readout of multiple columns. This applies especially to endless sampling.

There is no possibility to read the whole table as a 2-dimensional array, since this is known to be a bad programming style.

Please note that all data is provided as 32 Bit values. For more information please read the chapter "Data Types 148".

4.4.5.4.1 Read Column-Wise

Single block

For time-limited sampling, the easiest way is reading all data together after the measurement is finished. In this case, the function NMX_Sampling_ReadColumn32_1 221 must be called one-time per sampling element. The following example shows this visually:

Sampling Element	0	1	2	3	4	5
Source	Channel 0	Channel 1	Channel 2	Channel 3	Input Byte 0	Output Byte 0
Sample 0	21722	695	-11256	147236	0x00000003	0x00000000
Sample 1	21725	695	-11260	147924	0x00000003	0x00000000
Sample 2	21729	694	-11266	148626	0x00000003	0x00000002
Sample 3	21733	695	-11279	149301	0x00000003	0x00000002
Sample 4	21734	695	-11288	149998	0x00000003	0x00000002
Sample 5	21736	695	-11298	150654	0x00000004	0x00000002
Sample 6	21743	695	-11305	151131	0x00000004	0x00000004
Sample 7	21749	694	-11326	151840	0x00000004	0x00000004
Sample 8	21755	695	-11342	152529	0x00000008	0x00000000
Sample 9	21760	695	-11376	153111	0x00000008	0x00000000
Sample 10	1.	2.	3.	4.	0x 5. 08	0x 6. 01
Sample 11					0x 08	0x 01
Sample 12	21824	695	-11467	153722	0x00000008	0x00000001
Sample 13	21823	695	-11468	153710	0x00000008	0x00000001
Sample 14	21824	695	-11468	153717	0x00000008	0x00000001
Sample 15	21824	694	-11467	153720	0x00000008	0x00000001
Sample 16	21824	695	-11467	153709	0x00000008	0x00000001
Sample 17	21823	695	-11467	153716	0x00000008	0x00000001
Sample 18	21824	694	-11467	153722	0x00000008	0x00000001
Sample 19	21824	695	-11467	153719	0x00000008	0x00000001
Sample 20	21824	695	-11466	153711	0x00000008	0x00000001
Sample 21	21824	695	-11466	153716	0x00000008	0x00000001

According to this example, NMX_Sampling_ReadColumn32_1 must be called 6 times. The function/return parameters are as follows:

Function call	ulMaxSamples	ulElementNo	pulSamplesCopied	pudNoFirstSample
1	≥ 22	0	22	0
2	≥ 22	1	22	0
3	≥ 22	2	22	0
4	≥ 22	3	22	0
5	≥ 22	4	22	0
6	≥ 22	5	22	0

The parameter ulDoNotDelete normally is 0. If you need to read all samples again, set it to 1.

Multiple blocks

Another possibility is reading data in small blocks.
For endless sampling, this is essential.
For time-limited sampling this can be useful, since it allows analysing/displaying data, while sampling is active.

The following example shows visually, how the data from the example above is read in 3 blocks:

Sampling Element	0	1	2	3	4	5
Source	Channel 0	Channel 1	Channel 2	Channel 3	Input Byte 0	Output Byte 0
Sample 0	21722	695	-11256	147236	0x00000003	0x00000000
Sample 1	21725	695	-11260	147924	0x00000003	0x00000000
Sample 2	1	2	3	4	0x 5 03	0x 6 02
Sample 3	21733	695	-11279	149301	0x00000003	0x00000002
Sample 4	21734	695	-11288	149998	0x00000003	0x00000002
Sample 5	21736	695	-11298	150654	0x00000004	0x00000002
Sample 6	21743	695	-11305	151131	0x00000004	0x00000004
Sample 7	21749	694	-11326	151840	0x00000004	0x00000004
Sample 8	7	8	9	10	0x 11 08	0x 12 00
Sample 9					0x 08	0x 00
Sample 10	21767	694	-11441	153724	0x00000008	0x00000001
Sample 11	21773	695	-11468	153736	0x00000008	0x00000001
Sample 12	21824	695	-11467	153722	0x00000008	0x00000001
Sample 13	21823	695	-11468	153710	0x00000008	0x00000001
Sample 14	21824	695	-11468	153717	0x00000008	0x00000001
Sample 15	21824	694	-11467	153720	0x00000008	0x00000001
Sample 16	21824	695	-11467	153709	0x00000008	0x00000001
Sample 17	13	14	15	16	0x 17 08	0x 18 01
Sample 18					0x 08	0x 01
Sample 19	21824	695	-11467	153719	0x00000008	0x00000001
Sample 20	21824	695	-11466	153711	0x00000008	0x00000001
Sample 21	21824	695	-11466	153716	0x00000008	0x00000001

According to this example, NMX_Sampling_ReadColumn32_1 must be called 18 times. The function/return parameters are as follows:

Function call	ulMaxSamples	ulElementNo	pulSamplesCopied	pudNoFirstSample
1	5	0	5	0
2	5	1	5	0
3	5	2	5	0
4	5	3	5	0
5	5	4	5	0
6	5	5	5	0
7	9	0	9	5
8	9	1	9	5
9	9	2	9	5
10	9	3	9	5
11	9	4	9	5
12	9	5	9	5
13	≥ 8	0	8	14
14	≥ 8	1	8	14
15	≥ 8	2	8	14
16	≥ 8	3	8	14

17	≥ 8	4	8	14
18	≥ 8	5	8	14

The parameter **ulDoNotDelete must be 0**.

C / C++

```
// ###-> Get Number of Rows and Columns.
unsigned long long udSamplesReceived = 0;  // --> Rows
unsigned long ulNSamplingElements = 0;     // --> Columns
if (NMX_Sampling_GetStatus_1(pHandleNmx, NULL,
&ulNSamplingElements, &udSamplesReceived, NULL) != NST_SUCCESS)
{
      // Error reading sampling status. Do some error handling.
      return;
}

// ###-> Create table/array for sampled data
signed long **aaslSamples = new signed
long*[ulNSamplingElements];
unsigned long ulSamplesCopied = 0;

for (unsigned long ulColumn = 0; ulColumn <
ulNSamplingElements; ulColumn++)
{
      // Create array for this column
      aaslSamples[ulColumn] = new signed long[(unsigned long)
udSamplesReceived];
      for (unsigned long ulRow = 0; ulRow < udSamplesReceived;
ulRow++) {
            aaslSamples[ulColumn][ulRow] = -2147483647;
      }

      // ###-> Read sampled data column-wise
      if (NMX_Sampling_ReadColumn32_1(pHandle,
aaslSamples[ulColumn], (unsigned long)udSamplesReceived,
ulColumn, 0, &ulSamplesCopied, NULL) != NST_SUCCESS) {
            // Error reading sampled data. Do some error
handling.
            return;
      }
}

// ###-> Now the data can be processed.
//       First array dimension is columns / sampling elements
//       Second array dimension is rows / samples
```

```
// ###-> Don't forget to delete the array.
delete aaslSamples;
```

Delphi

```
var
  ucStatus: byte;
  udSamplesReceived: uint64;
  udSamplesMax: uint64;
  ulNElements: cardinal;
  aaslSamples: packed array of array of integer;
  ulColumn: cardinal;
  ulSamplesCopied: cardinal;
  udNoFirstSample: uint64;
begin
  ucStatus := 0;
  udSamplesReceived := 0;
  udSamplesMax := 0;
  ulNElements := 0;

  // ###-> Get Number of Rows and Columns.
  if cNmx.Sampling_GetStatus_1(pHandleNmx, ucStatus, ulNElements, udSamplesReceived
      // Error reading sampling status. Do some error handling.
      exit;
  end;

  // ###-> Create table/array for sampled data
  SetLength(aaslSamples, ulNElements);
  for ulColumn := 0 to ulNElements-1 do begin
    SetLength(aaslSamples[ulColumn], udSamplesReceived);

    if cNmx.Sampling_ReadColumn32_1(pHandleNmx,
                                    @aaslSamples[ulColumn][0],
                                    Length(aaslSamples[0]),
                                    ulColumn,
                                    1,
                                    ulSamplesCopied,
                                    udNoFirstSample) <> NST_SUCCESS then begin
      // Error reading sampled data. Do some error handling.
    end;
  end;

  // ###-> Now the data can be processed.
  //       First array dimension is columns / sampling elements
  //       Second array dimension is rows / samples
end;
```

C# / .Net

```
public struct TSampleColumn
```

```
{
    public Int32[] aslRows;
}

Byte ucStatus = 0;
UInt32 ulNElements = 0;
UInt64 udSamplesReceived = 0;
UInt64 udMaxSamples = 0;
UInt32 ulSamplesCopied = 0;

/* Get number of samples, which are available. */
cNmx.Sampling_GetStatus_1(pDevice, ref ucStatus, ref
ulNElements, ref udSamplesReceived, ref udMaxSamples);

/* Create arrays for sampling data and read it from DLL */
TSampleColumn[] aaslSData = new TSampleColumn[ulNElements];
for (UInt32 ulColumn = 0; ulColumn < ulNElements; ulColumn++)
{
    aaslSData[ulColumn].aslRows = new Int32[udSamplesReceived];
    for (Int32 I = 0; I < aaslSData[ulColumn].aslRows.Length;
I++) {
        aaslSData[ulColumn].aslRows[I] = Int32.MaxValue;
    }


    /* Feel free to do some more error handling here, e.g.
check ulSamplesCopied and checking the function return code. */
    cNmx.Sampling_ReadColumn32_1(pDevice,
                                 aaslSData[ulColumn].aslRows,
                                 0,
                                 (UInt32)
aaslSData[ulColumn].aslRows.Length,
                                 ulColumn,
                                 1,
                                 ref ulSamplesCopied,
                                 ref udMaxSamples);
}
```

4.4.5.4.2 Read Row-Wise

Reading row-wise is quite simple. The function NMX_Sampling_ReadRow32_1 224 is called for each sample and provides the data for all sampling elements. The following example shows this visually.

Sampling Element	0	1	2	3	4	5	
Source	Channel 0	Channel 1	Channel 2	Channel 3	Input Byte 0	Output Byte 0	
Sample 0	21722	695	-11256	147236	0x00000003	0x00000000	1.
Sample 1	21725	695	-11260	147924	0x00000003	0x00000000	2.
Sample 2	21729	694	-11266	148626	0x00000003	0x00000002	3.
Sample 3	21733	695	-11279	149301	0x00000003	0x00000002	4.
Sample 4	21734	695	-11288	149998	0x00000003	0x00000002	5.
Sample 5	21736	695	-11298	150654	0x00000004	0x00000002	6.
Sample 6	21743	695	-11305	151131	0x00000004	0x00000004	7.
Sample 7	21749	694	-11326	151840	0x00000004	0x00000004	8.
Sample 8	21755	695	-11342	152529	0x00000008	0x00000000	9.

According to this example, NMX_Sampling_ReadRow32_1 224 must be called 9 times. The function/return parameters are as follows:

Function call	ulMaxSamples	pulSamplesCopied	pudSampleNo
1	≥ 6	6	0
2	≥ 6	6	1
3	≥ 6	6	2
4	≥ 6	6	3
5	≥ 6	6	4
6	≥ 6	6	5
7	≥ 6	6	6
8	≥ 6	6	7
9	≥ 6	6	8

C / C++

```
// ###-> Get Number of Sampling elements. This could be done
one-time after starting sampling, since this value does not
change.
unsigned long ulNSamplingElements = 0;     // --> Columns
if (NMX_Sampling_GetStatus_1(pHandleNmx, NULL,
&ulNSamplingElements, NULL, NULL) != NST_SUCCESS) {
      // Error reading sampling status. Do some error handling.
      return;
}

// ###-> Read newest data. This code could for example be done
after the notification NMXNOTIFY_SAMPLING_NEW_DATA.
signed long *aslSamples = new signed long[ulNSamplingElements];
unsigned long ulSamplesCopied = 0;
unsigned long long udSampleNo = 0;
NMX_STATUS tResult = NST_SUCCESS;
```

```
do {
	tResult = NMX_Sampling_ReadRow32_1(pHandleNmx,
aslSamples, ulNSamplingElements, &ulSamplesCopied,
&udSampleNo);
	if (tResult == NST_SUCCESS) {
		// New row has been read-out. Use the data, e.g.
copy it to your own array or ring-buffer.
	}
	else if (tResult == NST_SAMPLING_NO_DATA_AVAILABLE) {
		// No more data available. Wait until new data has
arrived.
		break;
	}
	else {
		// An error occurred. Do some error handling.
		break;
	}
} while (1);
```

Delphi

```
var
  ucStatus: byte;
  udSamplesReceived: uint64;
  udSamplesMax: uint64;
  ulNElements: cardinal;
  aslSamples: packed array of integer;
  ulSamplesCopied: cardinal;
  udSampleNo: uint64;
  tResult: NMX_STATUS;
begin
  ucStatus := 0;
  udSamplesReceived := 0;
  udSamplesMax := 0;
  ulNElements := 0;

  // ###-> Get Number of Sampling elements. This could be done one-time after start
  if cNmx.Sampling_GetStatus_1(pHandleNmx, ucStatus, ulNElements, udSamplesReceived
      // Error reading sampling number of sampling elements. Do some error handling
      exit;
  end;

  // ###-> Read newest data. This code could for example be done after the notifica
  SetLength(aslSamples, ulNElements);
  while (true) do begin
    tResult := cNmx.Sampling_ReadRow32_1(pHandleNmx, aslSamples, Length(aslSamples)
    if tResult = NST_SUCCESS then begin
      // New row has been read-out. Use the data, e.g. copy it to your own array or
    end
    else if tResult = NST_SAMPLING_NO_DATA_AVAILABLE then begin
      // No more data available. Wait until new data has arrived.
      break;
```

```
    end
    else begin
      // Error reading sampled data. Do some error handling.
      break;
    end;
  end;
end;
```

C# / .Net

```
Byte ucStatus = 0;
UInt32 ulNElements = 0;
UInt64 udSamplesReceived = 0;
UInt64 udMaxSamples = 0;
UInt32 ulSamplesCopied = 0;

/* Get number of samples, which are available. */
cNmx.Sampling_GetStatus_1(pDevice, ref ucStatus, ref
ulNElements, ref udSamplesReceived, ref udMaxSamples);

/* Create arrays for sampling data and read it from DLL */
Int32[] aslValues = new Int32[ulNElements];

while (true)
{
    for (Int32 I = 0; I < aslValues.Length; I++) aslValues[I] =
Int32.MaxValue;

    /* Read samples from DLL into local array */
    UInt32 ulSamplesCopied = 0;
    NMX_MSTATUS tStatus = cNmx.Sampling_ReadRow32_1(pDevice,
aslValues, ref ulSamplesCopied, ref udNoFirstSample);
    if (tStatus == NMX_MSTATUS.SUCCESS)
    {
        // New row has been read-out. Use the data, e.g. copy
it to your own array or ring-buffer.
    }
    else if (tStatus == NMX_MSTATUS.SAMPLING_NO_DATA_AVAILABLE)
    {
        // No more data available. Wait until new data has
arrived.
        break;
    }
    else
    {
        // An error occurred. Do some error handling.
        break;
    }
}
```

4.4.5.5 Get sampling status

After sampling has been started, it is common practice to check its current status. There are two possibilities doing this, whereas it may make sense using both of this in parallel:

- By polling a function call 277, the current status and the current sample counter can be read-out from the NMX DLL.
- Sampling-Notifications 278 can be used to get informed about certain events.

4.4.5.5.1 Poll sampling status

Polling the sampling status is done by calling the function NMX_Sampling_GetStatus_1 226. The following information is provided:

- Current sampling status (e.g. running, finished, error).
- Number of sampling elements. This is the number of measurement channels + digitale I/O bytes, which have been selected for sampling. After sampling has been prepared, this value remains static.
- The number of samples, which have already been received by the NMX DLL.
- The maximum number of samples, which will be recorded. For endless measurement, this is the maximum value for the respective data type.

C / C++

```
unsigned char ucStatus = 0;
unsigned long ulNElements = 0;
unsigned long long udSamplesReceived = 0;
unsigned long long udSamplesMax = 0;
if (clNmx->Sampling_GetStatus_1(pHandleNmx, &ucStatus,
&ulNElements, &udSamplesReceived, &udSamplesMax) ==
NST_SUCCESS) {
      /* Use status data, e.g. display it on the GUI. */
}
```

Delphi

```
procedure TForm1.tmrSamplingTimer(Sender: TObject);
var
  ucStatus: byte;
  udSamplesReceived: uint64;
  udSamplesMax: uint64;
```

```
  ulNElements: cardinal;
begin
  ucStatus := 0;
  udSamplesReceived := 0;
  udSamplesMax := 0;
  ulNElements := 0;

  // Disable this timer
  tmrSampling.Enabled := false;

  if cNmx.Sampling_GetStatus_1(pHandleNmx, ucStatus, ulNElements, udSample
       // Use status data, e.g. display it on the GUI.
  end;

  // Re-enable this timer
  tmrSampling.Enabled := true;
end;
```

C# / .Net

```
private void tmrSampling_Tick(object sender, EventArgs e)
{
    /* Disable this timer */
    tmrSampling.Enabled = false;

    Byte ucStatus = 0;
    UInt32 ulNElements = 0;
    UInt64 udSamplesReceived = 0;
    UInt64 udSamplesMax = 0;
    if (cNmx.Sampling_GetStatus_1(pDevice, ref ucStatus, ref
ulNElements, ref udSamplesReceived, ref udSamplesMax) ==
NMX_MSTATUS.SUCCESS)
    {
        /* Use status data, e.g. display it on the GUI. */
    }

    /* Re-enable this timer */
    tmrSampling.Enabled = true;
}
```

4.4.5.5.2 Sampling notifications

Notifications are very useful to get informed about any changes of the sampling. For a complete list of all available notifications, please consult the chapter "Notifications 176".

To receive the notifications, these must be registered. Depending on the selected notification type (messages or callbacks), a message handler or a callback function must be implemented.

In the following sample code, message based notifications are used.

C / C++

```
// Define Message. In VisualStudio, WM_USER is defined in
WinUser.h as:
// #define WM_USER 0x0400
#define WM_MESSAGE_SAMPLING_NEWDATA             (WM_USER +
NMXNOTIFY_SAMPLING_NEW_DATA)              // Message for "new
sampling data" received
#define WM_MESSAGE_SAMPLING_ALLRECEIVED         (WM_USER +
NMXNOTIFY_SAMPLING_ALL_DATA_RECEIVED)     // Message for "all
sampling data has been received"
#define WM_MESSAGE_SAMPLING_FINISHED            (WM_USER +
NMXNOTIFY_SAMPLING_FINISHED)              // Message for
"sampling is finished"
#define WM_MESSAGE_SAMPLING_ERROR               (WM_USER +
NMXNOTIFY_SAMPLING_ERROR)                 // Message for
"sampling error"
#define WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW     (WM_USER +
NMXNOTIFY_SAMPLING_BUFFER_OVERFLOW)       // Message for
"sampling buffer overflow in DLL"
#define WM_MESSAGE_SAMPLING_TIMEOUT             (WM_USER +
NMXNOTIFY_SAMPLING_TIMEOUT)               // Message for
"sampling: timeout receiving data"


// ###-> Register notifications, e.g. directly after
connecting.
if (NMX_RegisterMessage_1(pHandleNmx,
NMXNOTIFY_SAMPLING_NEW_DATA,
static_cast<HWND>(Handle.ToPointer()),
WM_MESSAGE_SAMPLING_NEWDATA, 0, 0) != NST_SUCCESS) {
      // Handle error
}
if (NMX_RegisterMessage_1(pHandleNmx,
NMXNOTIFY_SAMPLING_ALL_DATA_RECEIVED,
static_cast<HWND>(Handle.ToPointer()),
WM_MESSAGE_SAMPLING_ALLRECEIVED, 0, 0) != NST_SUCCESS) {
      // Handle error
}
if (NMX_RegisterMessage_1(pHandleNmx,
NMXNOTIFY_SAMPLING_FINISHED,
```

```
static_cast<HWND>(Handle.ToPointer()),
WM_MESSAGE_SAMPLING_FINISHED, 0, 0) != NST_SUCCESS) {
	// Handle error
}
if (NMX_RegisterMessage_1(pHandleNmx, NMXNOTIFY_SAMPLING_ERROR,
static_cast<HWND>(Handle.ToPointer()),
WM_MESSAGE_SAMPLING_ERROR, 0, 0) != NST_SUCCESS) {
	// Handle error
}
if (NMX_RegisterMessage_1(pHandleNmx,
NMXNOTIFY_SAMPLING_BUFFER_OVERFLOW,
static_cast<HWND>(Handle.ToPointer()),
WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW, 0, 0) != NST_SUCCESS) {
	// Handle error
}
if (NMX_RegisterMessage_1(pHandleNmx,
NMXNOTIFY_SAMPLING_TIMEOUT,
static_cast<HWND>(Handle.ToPointer()),
WM_MESSAGE_SAMPLING_TIMEOUT, 0, 0) != NST_SUCCESS) {
	// Handle error
}

// ###-> Implement message handler
// Please note: Keep the code in the message handler as short
as possible.
// Do not update the GUI from within the message handlers.
protected:  virtual void WndProc(Message% m) override
{
	/* Listen for operating system messages. */
	switch (m.Msg)
	{
		case WM_MESSAGE_SAMPLING_NEWDATA:
			break;

		case WM_MESSAGE_SAMPLING_ALLRECEIVED:
			break;

		case WM_MESSAGE_SAMPLING_FINISHED:
			break;

		case WM_MESSAGE_SAMPLING_ERROR:
			break;

		case WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW:
			break;

		case WM_MESSAGE_SAMPLING_TIMEOUT:
			break;
```

```
        default:
                /* Pass all standard messages to the GUI form. */
                Form::WndProc(m);
                break;
        }
}
```

Delphi

```
// Define Message for "new static data available"
const WM_MESSAGE_NMX_SAMPLING_NEW_DATA      = WM_USER + NMXNOTIFY_SAMPLING
  WM_MESSAGE_NMX_SAMPLING_FINISHED          = WM_USER + NMXNOTIFY_SAMPLING
  WM_MESSAGE_NMX_SAMPLING_ALL_DATA_RECEIVED = WM_USER + NMXNOTIFY_SAMPLING
  WM_MESSAGE_NMX_SAMPLING_ERROR             = WM_USER + NMXNOTIFY_SAMPLING
  WM_MESSAGE_NMX_SAMPLING_BUFFER_OVERFLOW   = WM_USER + NMXNOTIFY_SAMPLING
  WM_MESSAGE_NMX_SAMPLING_TIMEOUT           = WM_USER + NMXNOTIFY_SAMPLING

// Declaration of message handler. Integrate it into the class declaration
procedure OnMessageSamplingNewData(var Msg: TMessage); message WM_MESSAGE_NMX_SAMPL
procedure OnMessageSamplingFinished(var Msg: TMessage); message WM_MESSAGE_NMX_SAMP
procedure OnMessageSamplingAllDataReceived(var Msg: TMessage); message WM_MESSAGE_N
procedure OnMessageSamplingError(var Msg: TMessage); message WM_MESSAGE_NMX_SAMPLIN
procedure OnMessageSamplingBufferOverflow(var Msg: TMessage); message WM_MESSAGE_NM
procedure OnMessageSamplingTimeout(var Msg: TMessage); message WM_MESSAGE_NMX_SAMPL

// ###-> Register notification, e.g. directly after connecting.
if NMX_RegisterMessage_1(pNmxHandle,
NMXNOTIFY_SAMPLING_NEW_DATA, MainForm.Handle,
WM_MESSAGE_NMX_SAMPLING_NEW_DATA, 0, 0) <> NST_SUCCESS then
begin
  // Registering notification failed. Do some error handling
end;
if NMX_RegisterMessage_1(pNmxHandle,
NMXNOTIFY_SAMPLING_FINISHED, MainForm.Handle,
WM_MESSAGE_NMX_SAMPLING_FINISHED, 0, 0) <> NST_SUCCESS then
begin
  // Registering notification failed. Do some error handling
end;
if NMX_RegisterMessage_1(pNmxHandle,
NMXNOTIFY_SAMPLING_ALL_DATA_RECEIVED, MainForm.Handle,
WM_MESSAGE_NMX_SAMPLING_ALL_DATA_RECEIVED, 0, 0) <>
NST_SUCCESS then begin
  // Registering notification failed. Do some error handling
end;
if NMX_RegisterMessage_1(pNmxHandle,
NMXNOTIFY_SAMPLING_ERROR, MainForm.Handle,
WM_MESSAGE_NMX_SAMPLING_ERROR, 0, 0) <> NST_SUCCESS then
begin
  // Registering notification failed. Do some error handling
end;
if NMX_RegisterMessage_1(pNmxHandle,
NMXNOTIFY_SAMPLING_BUFFER_OVERFLOW, MainForm.Handle,
```

```
WM_MESSAGE_NMX_SAMPLING_BUFFER_OVERFLOW, 0, 0) <>
NST_SUCCESS then begin
  // Registering notification failed. Do some error handling
end;
if NMX_RegisterMessage_1(pNmxHandle,
NMXNOTIFY_SAMPLING_TIMEOUT, MainForm.Handle,
WM_MESSAGE_NMX_SAMPLING_TIMEOUT, 0, 0) <> NST_SUCCESS then
begin
  // Registering notification failed. Do some error handling
end;

// ###-> Implement message handlers
// Please note: Keep the code in the message handlers as short as possible
// Do not update the GUI from within the message handlers.
```

C# / .Net

```
// Define Messages. In VisualStudio, WM_USER is 0x0400
const int WM_USER = 0x400;
const int WM_MESSAGE_SAMPLING_NEW_DATA = WM_USER + 0x20;
const int WM_MESSAGE_SAMPLING_FINISHED = WM_USER + 0x21;
public const int WM_MESSAGE_SAMPLING_ALL_DATA_RECEIVED =
WM_USER + 0x22;
public const int WM_MESSAGE_SAMPLING_ERROR = WM_USER + 0x28;
public const int WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW = WM_USER
+ 0x29;
public const int WM_MESSAGE_SAMPLING_TIMEOUT = WM_USER + 0x2A;


// ###-> Register notifications, e.g. directly after
connecting.
NMX_MSTATUS tStatus = NMX_MSTATUS.UNKNOWN;
tStatus = cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.SAMPLING_NEW_DATA, Handle,
WM_MESSAGE_SAMPLING_NEW_DATA, 0, 0);
if (tStatus != NMX_MSTATUS.SUCCESS) { /* Do some error handling
*/ }
tStatus = cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.SAMPLING_ALL_DATA_RECEIVED, Handle,
WM_MESSAGE_SAMPLING_ALL_DATA_RECEIVED, 0, 0);
if (tStatus != NMX_MSTATUS.SUCCESS) { /* Do some error handling
*/ }
tStatus = cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.SAMPLING_FINISHED, Handle,
WM_MESSAGE_SAMPLING_FINISHED, 0, 0);
if (tStatus != NMX_MSTATUS.SUCCESS) { /* Do some error handling
*/ }
tStatus = cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.SAMPLING_ERROR, Handle,
WM_MESSAGE_SAMPLING_ERROR, 0, 0);
```

```
if (tStatus != NMX_MSTATUS.SUCCESS) { /* Do some error handling
*/ }
tStatus = cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.SAMPLING_BUFFER_OVERFLOW, Handle,
WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW, 0, 0);
if (tStatus != NMX_MSTATUS.SUCCESS) { /* Do some error handling
*/ }
tStatus = cNmx.RegisterMessage_1(pDevice,
NMX_NOTIFICATION.SAMPLING_TIMEOUT, Handle,
WM_MESSAGE_SAMPLING_TIMEOUT, 0, 0);
if (tStatus != NMX_MSTATUS.SUCCESS) { /* Do some error handling
*/ }


// ###-> Implement message handler
// Please note: Keep the code in the message handler as short
as possible.
// Do not update the GUI from within the message handlers.
protected override void WndProc(ref Message m)
{
    // Listen for operating system messages.
    switch (m.Msg)
    {
        case WM_MESSAGE_SAMPLING_NEW_DATA:
            break;

        case WM_MESSAGE_SAMPLING_FINISHED:
            break;

        case WM_MESSAGE_SAMPLING_ALL_DATA_RECEIVED:
            break;

        case WM_MESSAGE_SAMPLING_ERROR:
            break;

        case WM_MESSAGE_SAMPLING_BUFFER_OVERFLOW:
            break;

        case WM_MESSAGE_SAMPLING_TIMEOUT:
            break;

        default:
            /* Pass all standard messages to the GUI form. */
            base.WndProc(ref m);
            break;
      }
}
```

4.4.5.6 Start position triggered sampling

Instead of recording the measured values at constant time intervals, the position-triggered measurement enables the acquisition at constant position intervals, e.g. in 0.1° or 10µm distances.

If position-triggered sampling is started, the NMX DLL internally starts an endless time-triggered sampling via the low-level sampling functions. With the Parameter ulSamplingSpeed, the speed of this sampling is defined. Position triggered sampling then reads all the sampled data and processes it. The processed data is then read by the application / measurement software. The accuracy of the position distance depends on the time interval used for sampling.

This is important to know, since the underlying endless time-triggered sampling will only work endless, as long as the sampled data can be transferred from the measurement system to the PC in realtime. For many applications, 1000 samples/s = 1000 µs should be enough and these can almost always be transferred in realtime. In case a higher speed is required, please consult the chapter "Sampling Speed with Irinos 141" for a more detailed information.

Setting up the trigger

For position triggered sampling, it is required to provide the NMX DLL information about the trigger. That information is:

1. Which measurement channel does provide the position information? -> Which encoder or probe does provide the position information, which is required to identify the trigger points?
   In the function call of NMX_Sampling_PreparePosition_1 229, this is the parameter ulTriggerChannelNumber.
2. Where / at which position shall triggering be started?
   In the function call of NMX_Sampling_PreparePosition_1 229, this is the parameter fdStart.
3. What is the position distance between two trigger points?
   In the function call of NMX_Sampling_PreparePosition_1 229, this is the parameter fdDistance.

Optionally, the unit for the parameters fdStart and fdDistance can be set via a scale factor. In the function call of NMX_Sampling_PreparePosition_1 229, this is the parameter fdScale. If fdScale = 1, then no scaling is used.

With these parameters, sometimes several possibilities lead to the same goal. Thus not every possibility for setting up the trigger can be discussed here. However, following two examples are provided:

Example 1:

A measurement system consists of 8 inductive + 4 incremental measurement channels. A rotational incremental encoder is connected to the 2nd incremental channel. The encoder has a resolution of 400000 increments per revolution. Measurement shall be started at 0° and end after one rotation with a position distance of 0.5°. Then the following parameters could be used:

```
ulTriggerChannelNumber = 9;        // 10-1 = 9, since channel numbering is 0-based
fdScale = 400000.0 / 360.0;        // = 1111.1111
fdStart = 0.0;
fdDistance = 0.5;
udMaxSamples = 360.0 / 0.5;        // = 720.0
```

Example 2:

A measurement system consists of 4 incremental measurement channels + 32 inductive measurement channels. A linear encoder is connected to the 1st incremental channel. Measurement shall be started at the position 7500 increments towards the negative direction with a position distance of 100 increments. Measurement shall be stopped at -423600 Then the following parameters could be used:

```
ulTriggerChannelNumber = 0;        // 1-1 = 0, since channel numbering is 0-based
fdScale = 1.0;                     // No scale factor is used
fdStart = 7500.0;
fdDistance = -100.0;               // Minus due to negative direction
udMaxSamples = 4312;               // = ((7500 - (-423600) / 100) + 1
```

Coding position-triggered sampling

Basically, position-triggered sampling is used in the same way as standard low-level sampling. Therefore most of the low-level functions[209] can be used:

- NMX_Sampling_GetMaxSpeed_1[209]
- NMX_Sampling_Reset_1[210]
- NMX_Sampling_AddChannelsAll_1[211]
- NMX_Sampling_AddChannel_1[212]
- NMX_Sampling_AddDigiInAll_1[214]
- NMX_Sampling_AddDigiInByte_1[215]
- NMX_Sampling_AddDigiOutAll_1[216]
- NMX_Sampling_AddDigiOutByte_1[217]
- NMX_Sampling_Start_1[220]
- NMX_Sampling_Stop_1[221]
- NMX_Sampling_ReadColumn32_1[221]
- NMX_Sampling_ReadRow32_1[224]
- NMX_Sampling_GetStatus_1[226]

The major difference is that the function NMX_Sampling_PreparePosition_1[229] instead of the function NMX_Sampling_PrepareTime_1[218] is used.

The use of notifications[176] is the same as with low-level sampling.

As a result of this, the following other HowTo's can be used as well:

- Start endless time-based sampling[252], except that NMX_Sampling_PrepareCustomTFT_1[232] must be used.
- Start time-limited sampling[259], except that NMX_Sampling_PrepareCustomTFT_1[232] must be used.
- Stop sampling[266]
- Reading sampled data[267]
- Get sampling status[277]

Following one example is provided for starting position-triggered sampling:

C / C++: Start with <u>all</u> measurement channels, 250µs sample period, first encoder as trigger source, scale factor 200.0, start position 0°, distance 0.5° and stop after one rotation

```
unsigned long ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (NMX_Sampling_Reset_1(pHandle) == NST_SUCCESS) {
	// List of sampling elements has been reset successfully.
}
else {
	// Failed resetting the list of sampling elements.
	// Do some error handling.
	return;
}


// ###-> 2/3. Add sampling elements
if (NMX_Sampling_AddChannelsAll_1(pHandle, &ulNElements) ==
NST_SUCCESS) {
	// Successfully added all sampling elements
}
else {
	// Failed adding sampling elements
	// Do some error handling.
	return;
}

NMX_STATUS NMX_Sampling_PreparePosition_1(
	NMX_PHANDLE pHandle,
	unsigned long ulSamplePeriod,
	unsigned long ulArrayLength,
	unsigned long long udMaxSamples,
	unsigned long ulTriggerChannelNumber,
	double fdScale,
	double fdStart,
	double fdDistance);

// ###-> 4. Prepare sampling
if (NMX_Sampling_PreparePosition_1(pHandle, 250/*µs*/, 720,
720/*MaxSamples*/, 0 /*first channel*/, 200.0, 0.0, 0.5) ==
NST_SUCCESS) {
	// Sampling successfully prepared
}
else {
	// Failed preparing sampling
```

```
        // Do some error handling.
        return;
}

// ###-> 5. Start sampling
if (NMX_Sampling_Start_1(pHandleNmx) == NST_SUCCESS) {
        // Start successful.
}
else {
        // Failed starting sampling
        // Do some error handling.
        return;
}
```

4.4.5.7 Start TFT high-level sampling

TFT sampling is similar to time-limited sampling, but provides 3 features. These are

- triggering by a digital input,
- applying an arithmetic average filter on the measurement values and
- providing additional samples, called "tail samples", after the sampling has been stopped.

Each of them can be disabled individually.

If TFT sampling is started, the NMX DLL internally starts an endless time-triggered sampling via the low-level sampling functions. With the Parameter ulSamplingSpeed, the speed of this sampling is defined. TFT sampling then reads all the sampled data and processes it. The processed data is then read by the application / measurement software.

This is important to know, since the underlying endless time-triggered sampling will only work endless, as long as the sampled data can be transferred from the measurement system to the PC in realtime. For many applications, 1000 samples/s = 1000 µs should be enough and these can almost always be transferred in realtime. In case a higher speed is required, please consult the chapter "Sampling Speed with Irinos 141" for a more detailed information.

Triggering

TFT sampling provides several possibilities to start / stop the recording of measurement values via a digital input. The trigger modes "Edge", "Level",

"Edge Start" and "Level Once" are available. For more information about these, consult the chapter "Trigger Modes 163".

Here are further notes:

- Triggering can be disabled completely by using the dummy trigger mode "Off".
- Sampling can always be stopped manually by calling NMX_Sampling_Stop_1 221. The digital input then has no effect.
- Besides using a digital input for triggering, all digital input data can still be included into the list of sampling elements (see NMX_Sampling_AddDigiInAll_1 214 and NMX_Sampling_AddDigiInByte_1 215).

Filtering

Via an integrated arithmetic average filter, the measurement values can be smoothened. The results are provided to the user application / measurement software.

Setting the filter is done via the parameters ulSamplePeriod and ulFilterPeriod:

- If ulSamplePeriod = ulFilterPeriod, then filtering is disabled.
- ulFilterPeriod must be an integer multiple of ulSamplePeriod.
  Example for ulSamplePeriod = 1000 (=1 ms):
  Valid values for ulFilterPeriod are 1000, 2000, 3000, 4000, 5000, ..., 10000.
  But for example 2500 would be invalid.
- ulFilterPeriod must be ≥ ulSamplePeriod

Example:

ulSamplePeriod = 1000, which is 1ms or 1000 samples/s.

ulFilterPeriod = 5000, which is 5ms or 200 samples/s.

-> The arithmetic average of 5 incoming samples is calculated and provided to the application / measurement software. This means the application receives 200 samples/s.

Note: Digital inputs or outputs are never filtered.

Tail values

Tail values are recorded after stop of sampling, no matter how the stop condition occurred. Typically they are used to apply an additional filter in the application / measurement software.

The parameter ulNTailSamples defines the number of samples recorded after stop. If this value is 0, then recording tail values is disabled.

Tail values can also be used with endless sampling. After endless sampling is stopped manually via NMX_Sampling_Stop_1 221, the tail values will be recorded.

Coding TFT high-level sampling

Basically, TFT high-level sampling is used in the same way as standard low-level sampling. Therefore most of the low-level functions 209 can be used:

- NMX_Sampling_GetMaxSpeed_1 209
- NMX_Sampling_Reset_1 210
- NMX_Sampling_AddChannelsAll_1 211
- NMX_Sampling_AddChannel_1 212
- NMX_Sampling_AddDigiInAll_1 214
- NMX_Sampling_AddDigiInByte_1 215
- NMX_Sampling_AddDigiOutAll_1 216
- NMX_Sampling_AddDigiOutByte_1 217
- NMX_Sampling_Start_1 220
- NMX_Sampling_Stop_1 221
- NMX_Sampling_ReadColumn32_1 221
- NMX_Sampling_ReadRow32_1 224
- NMX_Sampling_GetStatus_1 226

The major difference is that the function NMX_Sampling_PrepareCustomTFT_1 232 instead of the function NMX_Sampling_PrepareTime_1 218 is used.

The use of notifications 176 is the same as with low-level sampling.

As a result of this, the following other HowTo's can be used as well:

- Start endless time-based sampling 252, except that NMX_Sampling_PrepareCustomTFT_1 232 must be used.
- Start time-limited sampling 259, except that NMX_Sampling_PrepareCustomTFT_1 232 must be used.
- Stop sampling 266
- Reading sampled data 267
- Get sampling status 277

Following one example is provided for starting TFT high-level sampling:

C / C++: Start with all measurement channels, 1ms sample period, 5ms filter period, trigger "Level Once" and 4 Tail values

```
unsigned long ulNElements = 0;

// ###-> 1. Reset list of sampling elements
if (NMX_Sampling_Reset_1(pHandle) == NST_SUCCESS) {
	// List of sampling elements has been reset successfully.
}
else {
	// Failed resetting the list of sampling elements.
	// Do some error handling.
	return;
}


// ###-> 2/3. Add sampling elements
if (NMX_Sampling_AddChannelsAll_1(pHandle, &ulNElements) ==
NST_SUCCESS) {
	// Successfully added all sampling elements
}
else {
	// Failed adding sampling elements
	// Do some error handling.
	return;
}

// ###-> 4. Prepare sampling
if (NMX_Sampling_PrepareCustomTFT_1(pHandle, 1000/*µs*/,
5000/*µs*/, 10000, 10000/*MaxSamples*/, 4, 0, 3, 4, 2) ==
NST_SUCCESS) {
	// Sampling successfully prepared
}
```

```
else {
      // Failed preparing sampling
      // Do some error handling.
      return;
}

// ###-> 5. Start sampling
if (NMX_Sampling_Start_1(pHandleNmx) == NST_SUCCESS) {
      // Start successful.
}
else {
      // Failed starting sampling
      // Do some error handling.
      return;
}
```

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- I -

- M -

- S -

- V -