Meilhaus Electronic Manual

Intelligent Driver System ME-iDS 3.0E

Intelligent Driver System for

Windows 2000/XP/Vista, Windows 7

Imprint

Manual Meilhaus Intelligent Driver System (ME-iDS)

Revision 3.0

Revised: 2021-09-06

Meilhaus Electronic GmbH
Am Sonnenlicht 2
D-82239 Alling bei München
Germany
www.meilhaus.de

© Copyright 2021 Meilhaus Electronic GmbH

All rights reserved. No part of this publication may be reproduced or distributed in any form whether photocopied, printed, put on microfilm or be stored in any electronic media without the expressed written consent of Meilhaus Electronic GmbH.

Important note:

The information contained in this manual has been reviewed with great care and is believed to be complete and accurate. Meilhaus Electronic assumes no responsibility for its use, any infringements of patents or other rights of third parties which may result from use of this manual or the product. Meilhaus Electronic assumes no responsibility for any problems or damage which may result from errors or omissions. Specifications and instructions are subject to change without notice.

Note the Meilhaus Electronic general terms of business:
www.meilhaus.de/en/infos/my-shop/tob/

All trademarks acknowledged. All trademarks are property of their respective owners.

Content

1 Introduction

Valued customer,

Thank you for purchasing this device from Meilhaus Electronic. You have chosen an innovative high-technology product that left our premises in a fully functional and new condition.

Please take the time to carefully examine the contents of the package for any loss or damage that may have occurred during shipping. If there are any items missing or if an item is damaged, please contact us immediately.

With the Meilhaus Intelligent Driver System (ME-iDS) programming of all supported Meilhaus devices becomes unified and simple. It was developed with the aim of offering a common programming interface to cover all devices and all operating systems. To say it in simplified terms the concept is based on a question and answer game between software and hardware. The software can ask the supported devices for their components resp. their capabilities. In the next step this information can be used to access the appropriate functional groups of the hardware (in the following named as „subdevices“). The ME-iDS knows the following subdevices:

ME-iDS Generation 2.0

With release 2.0 the ME-iDS API was extended by so-called „properties“. Properties are characteristics of a device, subdevice or a channel and so on, which can be determined via the property tree and can also be set if applicable (see also chap. 3.3.2 from page 21). The new property functions enable full access to the functionality of your device. This means a clear expansion and a smart way of access to the feature of the hardware for some models. The configuration of the standard function of your devices - also of the new ME-5000 series - remains unaffected from that and can also be done by the known approach.

Additionally, all properties are accessible via the ME-iDC configuration tool, which was also extended significantly.

Note: The property functions are available with ME-iDS 2.0 and higher and completely implemented for the ME-5000 series under Windows. For all other devices only the general properties are supported at the moment. In future releases of the ME-iDS this will be extended and completed more and more.

In the chapter „Programming“ you find basic information regarding the order of operation for programming. In the chapter „Function Reference“ from page 78 the functions are described in detail.

1.1 Supported Devices

Supported hardware	Windows	Notes
ME-Synapse USB/LAN	✓	
ME-Axon USB/LAN	✓	
ME-94/95/96 cPCI/PCI	✓	Interrupt not supported
ME-630 cPCI/PCI/PCIe	✓	
ME-1000 Serie cPCI/PCI	✓	
ME-1400 Serie cPCI/PCI	✓	OSC output only supported in Linux
ME-1600 Serie cPCI/PCI	✓	
ME-4600 Serie cPCI/PCI/PCIe	✓	
ME-5001	✓	Plug-on board for ME-5000 series
ME-5002	✓	Plug-on board for ME-5810
ME-5004	✓	Plug-on board for ME-5000 series
ME-5100 cPCI/PCIe	✓	
ME-5310 Serie PCIe/PXIe	✓	
ME-5314 Serie PCIe/PXIe	✓	
ME-5351 Serie PCIe/PXIe	✓	
ME-5810(/S) cPCI/PCIe	✓	
ME-5820	✓	
ME-6000 Serie cPCI/PCI	✓	
ME-8100 Serie cPCI/PCI	✓	
ME-8200 Serie cPCI/PCI/PCIe	✓	
MEphisto-Digi (ME-1400 USB)	✓	*
MEphisto-Opto (ME-8200 USB)	✓	*
MEphisto-Scope (UM202, UM203)	✓	limited functionality
MEphisto-Switch (ME-630 USB)	✓	*

Table 1: Supported hardware

*MEphisto-Digi, MEphisto-Opto and MEphisto-Switch are not supported with 64-bit operating systems.

1.1.1 Use in Accordance with the Requirements

The PC boards of the ME-series are designed for acquisition and output of analog and digital signals with a PC. Depending on type install the models of the ME-series into:

a free PCI Express slot (PCIe versions) or

a free CompactPCI slot (3 HE cPCI versions)

For information on how to install a plug-in board or connect a USB device, please read the manual of your PC.

Please note the instructions and specifications as presented in this manual (Appendix A, Specifications):

- Please ensure sufficient heat dissipation for the board within the PC housing.
- All unused inputs should be connected to the ground reference of the appropriate functional section. This avoids cross talk between the input lines.
- The opto-isolated inputs and outputs achieve an electrical isolation of the application relative to PC ground.
- Note that the computer must be powered up prior to connecting signals by the external wiring of the board.
- As a basic principle, all connections to the board should only be made or removed in a powered-down state of all components.
- Ensure that no static discharge occurs while handling the board or while connecting/disconnecting the external cable.
- Ensure that the connection cable is properly connected. It must be seated firmly on the D-Sub connector and must be tightened with both screws, otherwise proper operation of the board cannot be guaranteed.

1.1.2 Improper Application

PC plug-in boards for the PCI-, PCI-Express- or CompactPCI-bus may not be taken into operation outside of the PC. Never connect the devices with voltage-carrying parts, especially not with mains voltage. As power supply for the USB models only an authorized power adaptor may be used.

Make sure that no contact with voltage-carrying parts can happen by the external wiring of the device. As a basic principle, all connections should only be made or removed in a powered-down state.

1.1.3 Unforeseeable Misapplications

The device is not suitable to be used as a children's toy, in the household or under unfavourable environmental conditions (e.g. in the open). Appropriate precautions to avoid any unforeseeable misapplication must be taken by the user.

1.2 System Requirements

The Meilhaus Intelligent Driver System supports Windows 2000/XP/Vista and Windows 7 (32-bit and 64-bit* versions available). The driver system supports multiprocessor systems.

*MEphisto-Digi, MEphisto-Opto and MEphisto-Switch are not supported with 64-bit operating systems.

1.3 Naming Conventions

The API functions of the ME-iDS function library are valid for all supported cPCI/PCI/PCIe boards and USB devices, if the feature is supported by the respective device type. The function name consists of the prefix "me" and several components representing the respective function as descriptive as possible (e.g. "IO" for input/output function).

For the description of the functions, the following standards will be used:

function names	are written italic in the body text e.g. *meIOStream Red().*
`<Parameter>`	Parameters follow the Hungarian notation and are written in brackets in font `Courier`. p – pointer i – integer f – float d – double c – char(string)
[square brakets]	are used to indicate physical units.
`main (…)`	Parts of programs will be in `Courier` type.

1.4 Documentation

The 3 pillars of documentation in the context of ME-iDS:

1. This ME-iDS manual offers a largely general description of concept and programming as well as an extensive function reference (quick access via the ME-iDS system tray or by the Windows start menu).

2. The ME-iDS help (CHM file format) will be installed automatically (quick access via the ME-iDS system tray or by the Windows start menu). It covers the following items:

- Hardware specific aspects of programming.
- Example code.
- Constant definition (see also medefines.h).
- Listing of error codes.
- Listing of properties.
- Frequently asked questions (FAQs).
- Version history.

3. Hardware manuals describe the particular hardware (settings, pinouts, specifications).

.

2 Installation

2.1 Installation under Windows

Please refer to the installation instructions for Windows in the readme files contained in the ME-iDS package.

Please note! If you have already installed a ME-iDS revision 1.2.x or older on your computer you must uninstall this version with the program „meIDSWin-Remove.exe“ before installing ME-iDS revision 1.3.0 or newer. „meIDS-WinRemove.exe“ can be found in the path „C:\Meilhaus\ME-iDS\install“ by default.

Please note: it is necessary to install the ME-iDS driver software before installing the hardware. This is of particular relevance for an initial installation under Windows 7. Otherwise a proper installation and operation cannot be guaranteed.

2.2 Configuration Utility (ME-iDC)

The Meilhaus Intelligent Device Configuration Utility (ME-iDC) makes it easier to keep the overview and offers options for convenient configuration. ME-iDS and ME-iDC communicate through the configuration file me-config.xml.

In Windows the utility is installed automatically and can be run either from the ME system tray icon in the info area of the task bar or via the entry „Meilhaus ME-iDS“ in the Windows start menu.

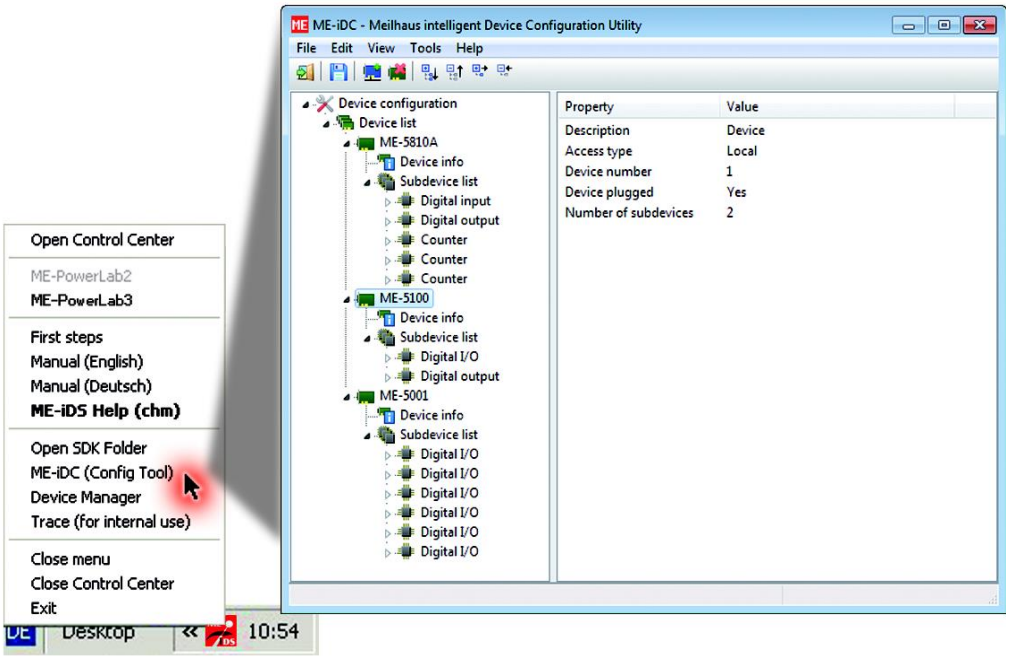

Diagram 1: ME-iDC

The most important functions of the configuration utility:

- Device information can be checked easily. The structure of the installed devices is shown.
- Properties can be determined as set if applicable. The structure of the property tree can be shown.
- The device index (Parameter `<iDevice>`) used to access a device can be changed.
- Accessories can be registered at the driver system. It simplifies programming considerably.
- „Memory function“ for devices which are removed from the system temporarily.
- Deleting devices which have been removed from the system.
- Changing the subdevice configuration.
- Loading a new firmware.
- Registering remote devices like ME-Synapse-LAN.

A detailed desciption can be found in the ME-iDC help file.

2.2.1 Subdevice Configuration

On some models the standard functionality of subdevices can be changed by the user selecting an alternative configuration. The designated configu-

ration will be activated before starting your application via the ME-iDC configuration tool. With the standard configuration (ID 0) the subdevice is ready-to-run at once.

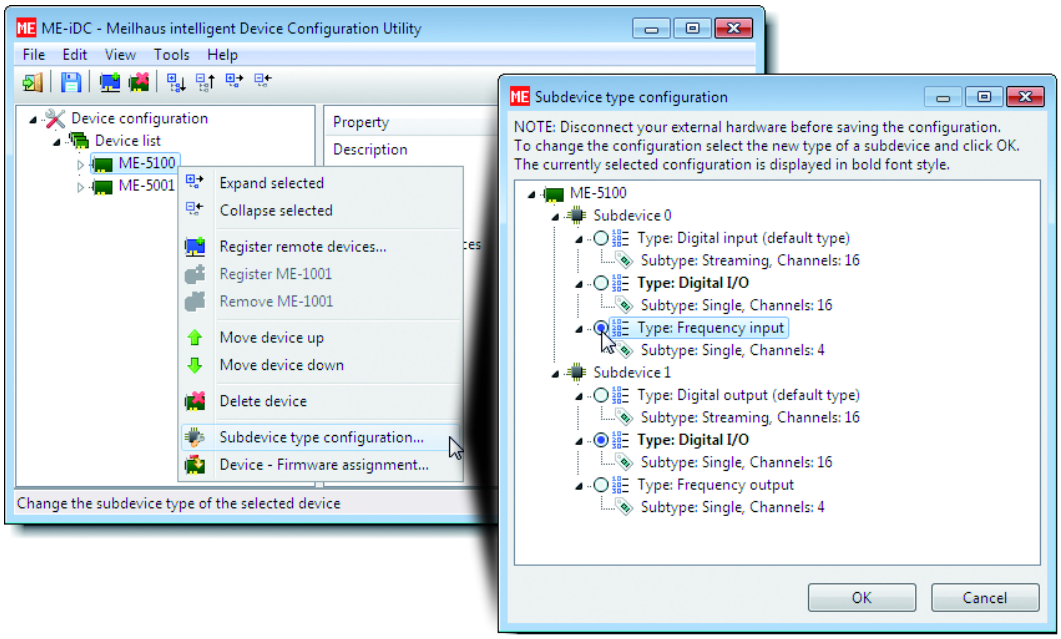

Diagram 2: Subdevice Configuration

2.2.2 Firmware Configuration

On some models the standard functionality of the device can be changed by the user selecting an alternative firmware. See ME-iDC help for this procedure.

2.2.3 Registering a Remote Device

The access to a remote device (e.g. ME-Synapse LAN) under Windows via the ME-iDS API assumes the installation of a so-called ONC RPC client software for Windows, which requires buying a licence key. The abbreviation ONC RPC means „Open Network Computer Remote Procedure Call“.

The used „ONC RPC/XDR for C/C++ Client Runtime“ software from Distinct implements the Sun Microsystems RPC standard on your Windows computer. By the remote procedure calls (RPC) an application will be able to communicate with ONC RPC clients via a TCP/IP based network, as data are transferred using the external data representation (XDR) in a format independent of the processors and operating systems involved.

To install the RPC software choose the option „Network Installation“ in the ME-iDS installation program. A separate installation program will be started. Within this procedure a licence key must be entered.

Please note that you need separate keys for 32-bit and 64-bit version of the RPC software. If you haven't got a licence key with your product package you can buy one at Meilhaus Electronic. Please contact our sales team (sales@meilhaus.com).

Additionally, you have to register the remote device with its IP address at the driver system.

Proceed as follows:

- Install the ME-iDS using the setup type „Network Installation“ and enter the licence key when asked.
- Call the ME-iDC (Config Tool) via the ME-iDS system tray or by the Windows start menu.
- Introduce the IP address of the remote device to the driver system (see also ME-iDC help). Click on the blue icon „Register remote devices“.
- Open the dialog and enter the current IP address.

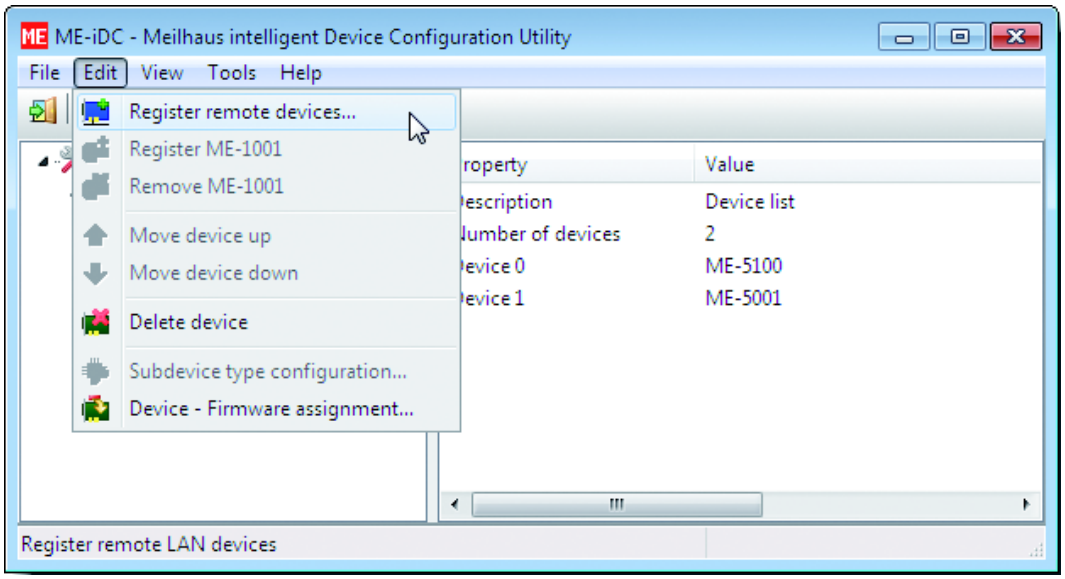

Diagram 3: Registering IP address

2.3 Setting the IP Address

If necessary, ask your network administrator what IP address should be used. To change the IP address of the ME-Synapse LAN and ME-Axon LAN start a web browser and enter the default IP address (or the IP address you have selected last). In case of the factory default address type: http://192.168.20.228 (see also manual of the device). The IP configuration page will be shown.

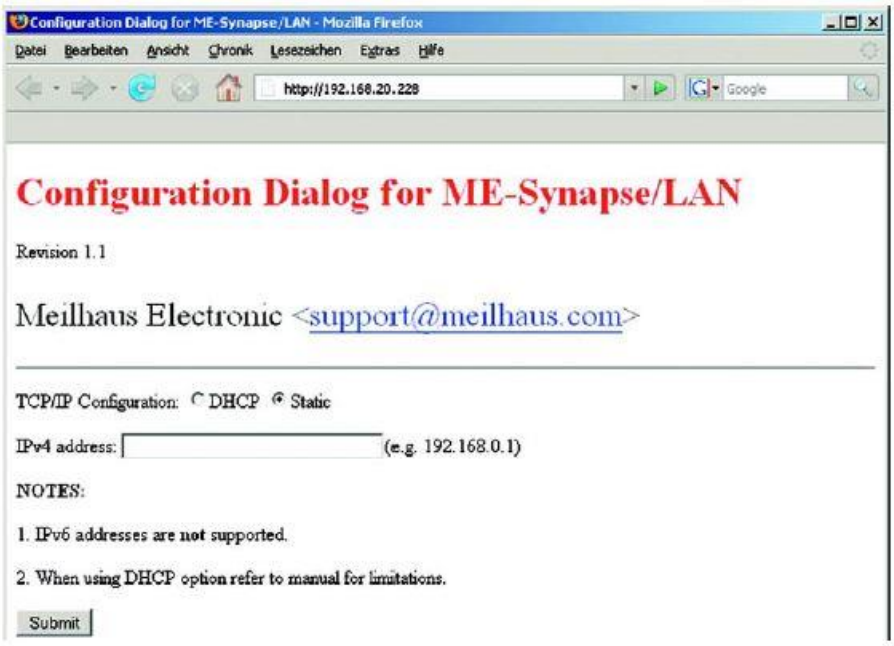

Diagram 4: Setting the IP Address

Static

Select „Static“ and enter the desired address in the field IPv4 address. Write down the address (networking experts may determine the address later for example with the „ipconfig“ command under Windows „Run…“. Introduce the IP address in the ME-iDC to the driver system.

Dynamic

The ME-Synapse-LAN/ME-Axon-LAN can only operate in networks with an initial dynamic assignment of IP addresses, but which then are fixed for every device.

Select „DHCP“. Contact your network administrator to find out the address assigned, for example with a „ping“ command. Introduce the IP address in the ME-iDC to the driver system.

3 Programming

3.1 Architecture of the Driver System

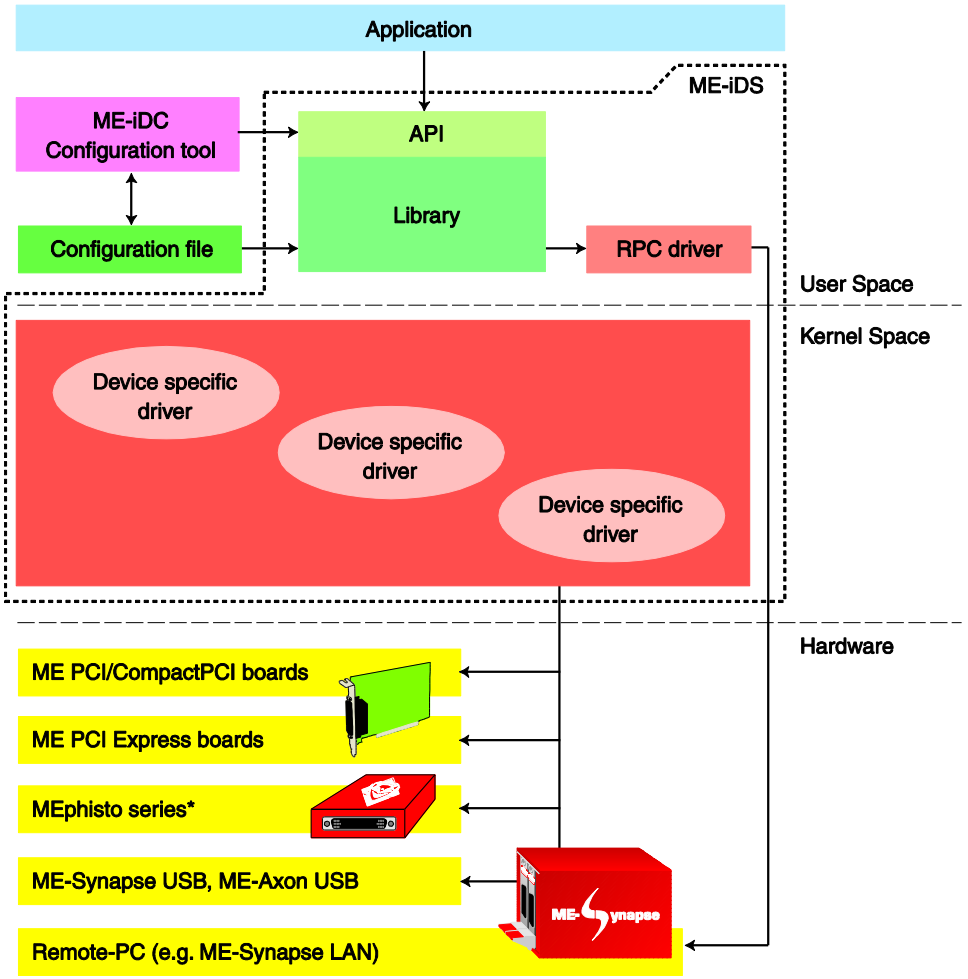

Diagram 5: Architecture of the driver system

*MEphisto-Digi, MEphisto-Opto and MEphisto-Switch are not supported with 64-bit operating systems.

The Meilhaus Intelligent Driver System (ME-iDS) offers a unified programming interface (API) covering different devices and operating systems. It is structured into the level „user space“ with the function library, the level „kernel space“ with the main driver and device-specific driver modules and finally the hardware level. For remote access via a network under Windows an ONC RPC software from Distinct with license is required (see chap. 2.3.3 on page 14), which is not included with the standard ME-iDS package.

3.1.1 Library Files

There are library files in versions without (local) and with RPC support for remote access via network.

Windows

Under Windows the local version is installed by default. Users can choose RPC support as an option (please select it only if you have a valid RPC license available).

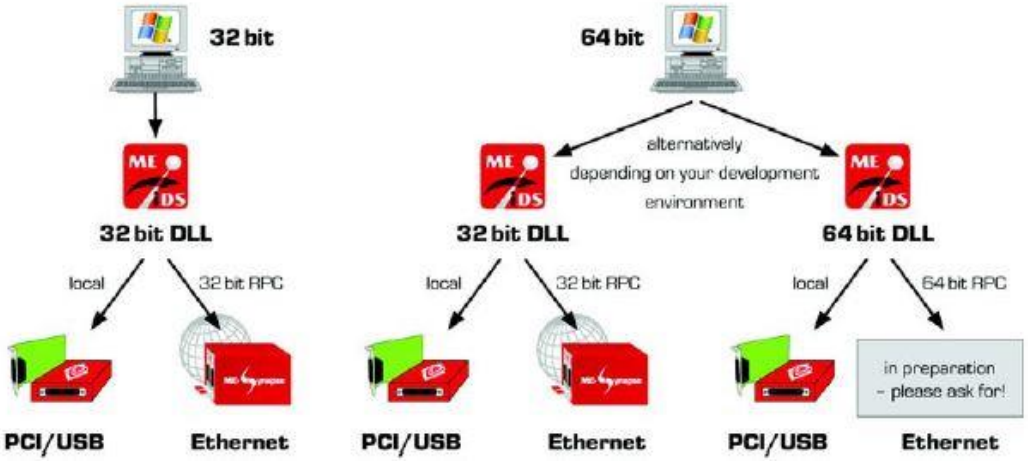

Diagram 6: Library Files

Note: The local version is smaller and faster. The time to establish the connection to a remote device can be very long, especially if a device is not active.

3.2 Language Support

Below you can find an overview of programming languages and development environments, which are supported by default with the ME-iDS.

The ME-iDS can be found on the CD/DVD supplied or under www.meilhaus.de/download.

3.2.1 High-Level Language Support

To make programming as easy as possible we provide simple examples and small projects with source code for each of the languages. The programming examples will be provided as a ZIP file and can be downloaded (see: www.meilhaus.com/download).

Note also the instructions in the appropriate README-files.

	Windows
C/C++	Visual C++, C++ Builder
Visual Basic	Visual Basic 6.0, Visual Basic.NET
Visual C#	✓
Delphi	✓

Table 2: High-level language support

3.2.2 Graphical Programming Tools

For programming with graphical programming environments like Agilent VEE or LabVIEW™ you can use predefined objects („user objects“ resp. “virtual instruments”) and demos which can be included in your project easily.

	Windows
Agilent VEE	✓
LabVIEW	✓

Table 3: Graphical Programming Tools

3.3 Concept of the Library

The concept of the ME-iDS (Meilhaus Intelligent Driver System) can be described as a hierarchical, single rooted tree. All hardware is organized according to the following generic hierarchy.

3.3.1 Hierarchy Levels

- „Driver“: Logical representation of a whole system as a set of devices. Root of hierarchy tree.

- **Device:** A device represents a single hardware unit with one or more so-called subdevices. Each device has its own, unique and unchangeable serial number. To each device a device index is assigned by which it is addressed when calling the functions of the ME-iDS API (parameter `<iDevice>`) **.**
  Important: There is no strict binding between a physical device (hardware) and the device index used to address it. Device indices are assigned dynamically during initialization (on calling *meOpen…()*) and may vary from one application start to the next. For this reason, the *meQuery...()* routines should be used to determine the correct device index for a particular physical device as demonstrated in the programming examples contained in the ME-iDS Software Developer Kit (SDK).
  Note: For information how to assign a fixed device index to a particular hardware unit, please refer to the ME-iDC (Config Tool) help.
  - **Subdevice:** Logical representation of a functional unit (example: analog input). Each subdevice is assigned a subdevice index to be passed when calling the functions of the ME-iDS API (parameter `<iSubdevice>`). Subdevice indices start at 0 and are assigned statically. Two devices of the same type will contain the same subdevices in the same order.
    Important: Each model has its own way of counting the subdevices. Two models of the same family may have different sets of subdevices and so a particular subdevice may have different subdevice indices on two models. For example: the external interrupt (EXT_IRQ) on the ME-1400A has the subdevice index '6' but on the ME-1400B it has the subdevice index '12'. For this reason, the *meQuery...()* routines should be used to determine correct subdevice indices for a particular subdevice as demonstrated in the programming examples contained in the ME-iDS SDK.

– **Channel**: The lowest-level of hardware components represents a single data channel, e.g.: an input or output line (digital, analog, etc.). Each channel can have several parameters, such as: range, reference, polarity (unipolar, bipolar), etc.

3.3.2 Properties

With version 2.0 the ME-iDS-API has been extended by the so-called „Properties“. „Properties“ are characteristics of a device, a subdevice or a channel and so on, which are organized by a tree structure. By the so called property path you have access to all properties and attributes, you can determine their value and you can set them if applicable. The names of the properties are pre-defined.

There are properties of access type „Read only“, which can only be read, e.g. the type of a subdevice (DI, DO, DIO, AI etc.) and those of access type „Read/Write“ e.g. the direction of a bidirectional port, which can be determined (read) as well as set (written).

In the following diagram you see a typical property tree as you can show it via the ME-iDC:

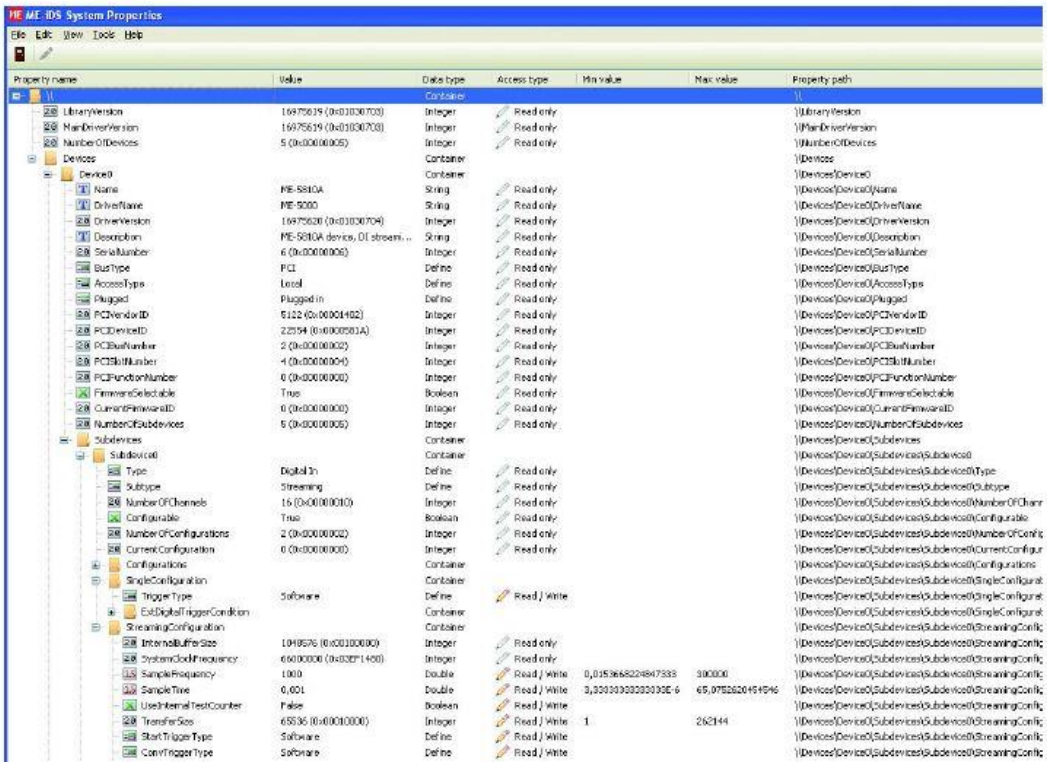

Diagram 7: Property tree

Note: The property functions are available with ME-iDS 2.0 and higher and completely for the ME-5000 series under Windows. For all other devices only the general properties are supported at the moment. In future releases of the ME-iDS this will be extended and completed more and more.

3.3.2.1 Property Pathes

Properties are collected together in containers. A container contains properties and can also contain other containers. By this way, all the properties for the system are organized hierarchically like a tree structure.

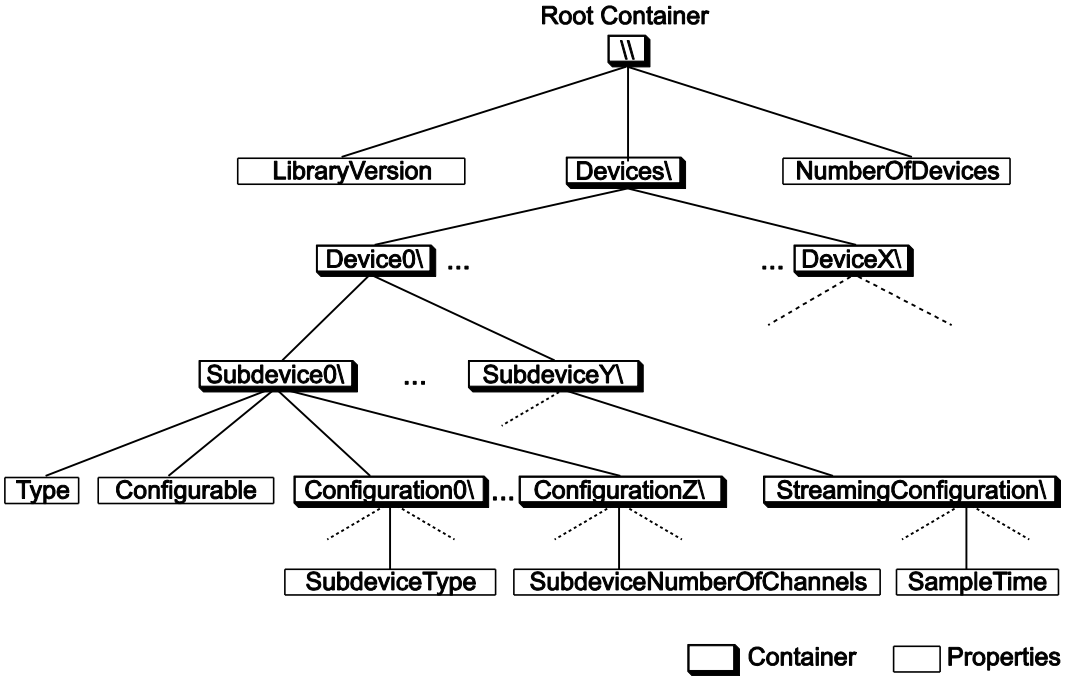

Diagram 8: Tree structure

The root of the property tree is a container represented by two consecutive backslashes "\\". The components of a property path are separated by one backslash "\".

To access a property the entire path to the property from the root "\\" through all the intervening containers, separated by one backslash and ending with the property name must be used.

For example, to access the subdevice type of subdevice 0 on device 0 the correct property path is:

"\\Devices\Device0\Subdevices\Subdevice0\Type“

A backslash must not be appended to the end of a property path.

Note: Depending on the programming language you use, the backslash might be an escape character (e.g. C, C++, C#). In this case in the source code two backslashes must be used to represent one backslash in a property string. For example: \\\\Devices\\Device1.

3.3.2.2 Abbreviations for Property Pathes

For convenience several abbreviations exist to shorten a property path. The following properties resp. containers can be abbreviated:

- **Devices:**
  "\\Devices\Device1" can be abbreviated:
  "\\Device1"
- **Subdevices:**
  "\\Device1\Subdevices\Subdevice0" can be abbreviated:
  "\\Device1\Subdevice0"
- **Channels:**
  "\\Device1\Subdevice0\Channels\Channel7" can be abbreviated:
  "\\Device1\Subdevice0\Channel7"
- **Configurations:**
  "\\Device1\Subdevice0\Configurations\Configuration0" can be abbreviated: \\Device1\Subdevice0\Configuration0.

3.3.2.3 Property Functions

The property functions are implemented as ANSI and Unicode (UTF-16) versions. The ANSI version have suffix 'A' and use NULL terminated ANSI strings (char*). The unicode version with the suffix 'W' use wide character strings (wchar_t*).

Depending on the property type (see chap. 3.3.2.5 on page 26) the appropriate function must be used to read resp. write the value (see also function reference from page 78).

Note: In the following the ANSI routines are used. But all examples are also valid for unicode functions

3.3.2.3.1 Reading Property Values

The following functions are available for reading access to properties:

ANSI:

int mePropertyGetIntA(char pcPropertyPath, int* piValue)*

Reads a property as 32-bit-signed integer value.

int mePropertyGetDoubleA(char pcPropertyPath, double* pdValue)*

Reads a floating point property as 64-bit double value.

int mePropertyGetStringA(char pcPropertyPath, char* pcValue, int iBufferLength)*

Reads a string property as a NULL terminated ANSI character array.

Unicode:

int mePropertyGetIntW(wchar_t pcPropertyPath, int* piValue)*

Reads a property as 32-bit-signed integer value.

int mePropertyGetDoubleW(wchar_t pcPropertyPath, double* pdValue)*

Reads a floating point property as 64-bit double value.

int mePropertyGetStringW(wchar_t pcPropertyPath, wchar_t* pcValue, int iBufferLength)*

Reads a string property as a NULL terminated unicode character array.

3.3.2.4 Attribute

Properties and containers have so-called attributes which can also be accessed to with the property functions. Simply add an additional back-slash „\“ to the property path followed by the attribute name. Attributes provide additional information concerning properties and/or containers, e.g. the data type of the property, whether the access type is read only or writable too, the number of elements a container holds etc.

Examples for attributes:

Name	Defined for …	Type	Possible values
`NumberOfElements`	Container	Integer	-
`PropertyName`	Properties & Container	String	-
`PropertyType`	Properties & Container	Define	ME_PROPERTY_TYPE_CONTAINER, ME_PROPERTY_TYPE_BOOL, ME_PROPERTY_TYPE_INT, ME_PROPERTY_TYPE_DOUBLE, ME_PROPERTY_TYPE_STRING, ME_PROPERTY_TYPE_DEFINE
`PropertyAccess`	Properties	Define	ME_PROPERTY_ACCESS_READ_ONLY, ME_PROPERTY_ACCESS_READ_WRITE

Table 4: Attributes (for example)

All attributes with a short description can be found in the ME-iDS help file (*.chm) under „Related Pages“-„Properties“.

For example the attribute "NumberOfElements" is defined for any container, but not for properties. It is a read only integer value which can be queried using the *mePropertyGetIntA()* function. To find out how many elements (Containers and properties) are in the root container you can call (the number of elements is returned by parameter `<piValue>`):

mePropertyGetIntA("\\NumberOfElements", int piValue);*

Once you know how many elements a container holds you can address the elements even without knowing their names by using an index. So, if you have found out that the root container holds 7 elements, then you can address these elements as "\\0", "\\1", ... "\\6".

Using the attribute "PropertyName" you can query the name of each element of the container. So, to find the name of the first element (with index 0) in the root container you can call:

mePropertyGetStringA("\\0\PropertyName, char pcValue, int iBuffer-Length);*

The name of the property is returned by parameter `<pcValue>`.

Once you know the name of a property, you can use this name instead of the index to address the property. So, if the name of the first element in the root container is "LibraryVersion" then you can find the type of this property by calling either:

mePropertyGetIntA("\\0\PropertyType", int piValue); or...*

mePropertyGetIntA("\\LibraryVersion\PropertyType", int piValue);*

The result is the same.

In this way you can find the names and types of all the elements of the root container. Some of these elements will themselves be containers. You can use the same procedure to find the names and types of the elements in these containers.

For instance, the root container holds a container called "Devices". To find out how many elements are in this container you can call:

mePropertyGetIntA("\\Devices\NumberOfElements", int piValue);*

Next you can find out the name of the first element (with index 0) in this container and so on:

mePropertyGetStringA("\\Devices\0\PropertyName, char pcValue, int iBufferLength);*

In this way you can recursively investigate the entire property tree. The example program `Con_meIDSSystemProperties` im ME-iDS SDK shows how this can be done using C++ source code in a small console program.

3.3.2.5 Property Types

The attribute "PropertyType" is defined for all properties and containers. If "P" is the path to a property or container then the following function call is always valid:

mePropertyGetIntA("P\PropertyType", int piValue)*

The property type is returned in <`piValue`> and is one of the following defines:

- ME_PROPERTY_TYPE_CONTAINER:
  "P" is a container which itself contains further properties and containers.
- ME_PROPERTY_TYPE_BOOL:
  "P" is a two-valued property, for example Off/On, False/True, No/Yes etc. The value can be queried with the function *mePropertyGetIntA()* and in some cases set with the function *mePropertySetIntA().* The value is 0 for False or 1 for True. The value can be also queried using the *mePropertyGetStringA()* functions. These return a readable version of the value, e.g. for display purposes.
- ME_PROPERTY_TYPE_INT:
  "P" is an integer property. The signed 32-bit integer value can be queried with the function *mePropertyGetIntA()* and in some cases set with the function *mePropertySetIntA().* The value can be also queried using the *mePropertyGetStringA()* functions. These return a readable version of the property value, e.g. for display purposes.
- ME_PROPERTY_TYPE_DOUBLE:
  "P" is a double property. The 64-bit floating point double value can be queried with the function *mePropertyGetDoubleA()* and in some cases set with the function *mePropertySetDoubleA().* The value can be also queried using the *mePropertyGetStringA()* functions. These return a readable version of the property value, e.g. for display purposes.
- ME_PROPERTY_TYPE_STRING:
  "P" is a string property, a NULL terminated character sequence. The value can be queried with the function *mePropertyGetStringA()* and in some cases set with the function *mePropertySetStringA()*
- ME_PROPERTY_TYPE_DEFINE:
  "P" is a define property. It can take one of a limited number of predefined constant values. The value can be queried with the function *mePropertyGetIntA()* and in some cases set with the function *mePropertySetIntA().* The value can be also queried using the *mePropertyGetStringA()* functions. These return a readable version of the property value, e.g. for display purposes.

GetStringA() functions. These return a readable version of the define, e.g. for display purposes.

The value of a property can be queried by using the property path without an attribute in the appropriate *mePropertyGet*... function (see above). For example, to query the value of the property "LibraryVersion" mentioned above in the root container you can call either the function

mePropertyGetIntA("\\LibraryVersion", int piValue)*

...which returns the library version asan integer in parameter

<`piValue`> or call the function:

mePropertyGetStringA("\\LibraryVersion", char pcValue, int iBufferLength)*

...which returns the library version in the form of a string in <`pcValue`>.

3.3.2.6 Access Type of Properties

The attribute "PropertyAccess" indicates if a property is read only or writable as well. The property access is defined for all properties, but not for containers.

Example for path of property "ClockSource". The device index is 1, the subdevice should be of type counter with index 2. The property path "P" is defined as follows:

\\Devices\Device1\Subdevices\Subdevice2\ClockSource.

The following function call is valid:

mePropertyGetIntA("P\PropertyAccess", int piValue)*

The property access is returned in parameter <`piValue`> and is one of the following defines:

- ME_PROPERTY_ACCESS_READ_ONLY:
  The value of the property "P" can only be read.
- ME_PROPERTY_ACCESS_READ_WRITE:
  The value of the property "P" can be read and written.

3.3.2.7 System Attributes

These properties are elements of the root container "\\" and can therefore be accessed via the path "\\<Property name>".

Name	Type	Access type
`LibraryVersion`	Integer	Read only
`MainDriverVersion`	Integer	Read only
`NumberOfDevices`	Integer	Read only
`Devices`	Container	-

Table 5: System attributes

3.3.2.8 General Device Properties

The following properties are available for every device in ME-iDS. The path to a device with the index X reads as:

"\\Devices\X\<Property name>" or...

"\\Devices\DeviceX\<Property name>"

or abbreviated...

"\\DeviceX\<Property name>"

Example: Property "Plugged" for device with index 0 "\\Devices\Device0\Plugged" or... "\\Device0\Plugged"

Name	Type	Access type	Possible values
`Name`	String	Read only	-
`DriverName`	String	Read only	-
`DriverVersion`	Integer	Read only	-
`Description`	String	Read only	-
`SerialNumber`	Integer	Read only	-
`BusType`	Define	Read only	ME_BUS_TYPE_PCI, ME_BUS_TYPE_USB, ME_BUS_TYPE_LAN_PCI, ME_BUS_TYPE_LAN_USB
`AccessType`	Define	Read only	ME_ACCESS_TYPE_LOCAL, ME_ACCESS_TYPE_REMOTE

`Plugged`	Boolean	Read only	-
`NumberOfSubde-vices`	Integer	Read only	
`Subdevices`	Container	-	-

Table 6: General device properties

3.3.2.9 Subdevice Properties

The following properties are available for every subdevice in ME-iDS. The path to a subdevice property of subdevice Y on device X reads as:

"\\Devices\DeviceX\Subdevices\Y\<Property name>" or...

"\\Devices\DeviceX\Subdevices\SubdeviceY\<Property name>"

or abbreviated....

"\\DeviceX\SubdeviceY\<Property name>"

Name	Type	Access type
`Typ`	Define	Read only
`Subtype`	Define	Read only
`NumberOfChannels`	Integer	Read only
`Configurable`	Boolean	Read only

Table 7: General subdevice properties

If the boolean property “Configurable”is True then the following three sub-device properties also exist:

Name	Type	Access type
`NumberOfConfigura-tions`	Integer	Read only
`CurrentConfiguration`	Integer	Read only
`Configurations`	Container	

Table 8: Properties for configurable subdevices

3.3.2.10 Properties of Configuration Containers

The Container "Configurations" contains "NumberOfConfigurations" elements, which describe the available subdevice configurations:

The path to a property of configuration Z from subdevice Y on device X reads as:

"\\Devices\DeviceX\Subdevices\SubdeviceY\Configurations\Z\

<Property name>" or...

"\\DeviceX\SubdeviceY\Configurations\ConfigurationZ\<Property name>"

or abbreviated....

"\\DeviceZ\SubdeviceY\ConfigurationZ\<Property name>"

Name	Type	Access type
SubdeviceType	Define	Read only
SubdeviceSubtype	Define	Read only
NumberOfChannels	Integer	Read only

Table 9: Properties of configuration containers

3.3.3 Subdevices

The ME-iDS knows the following subdevices:

- Analog input (ME_TYPE_AI)
- Analog output (ME_TYPE_AO)
- Digital input/output (ME_TYPE_DIO)
- Digital input (ME_TYPE_DI)
- Digital output (ME_TYPE_DO)
- Frequency input/output (ME_TYPE_FIO)
- Frequency input (ME_TYPE_FI)
- Frequency output (ME_TYPE_FO)
- Counter (ME_TYPE_CTR)
- External interrupt (ME_TYPE_EXT_IRQ)
- FPGA (ME_TYPE_FPGA) - planned!

3.3.3.1 Analog Input/Output

Subdevice types for analog ports:

- Input (ME_TYPE_AI)
- Output (ME_TYPE_AO)

meQueryNumberChannels() returns the number of available input or output channels.

3.3.3.2 Digital Input/Output

Subdevice types for digital ports:

- Input (ME_TYPE_DI)
- Output (ME_TYPE_DO)
- Bi-directional (ME_TYPE_DIO)

meQueryNumberChannels() returns the size in bits of a digital subdevice.

3.3.3.3 Frequency Input/Output

Subdevice types for frequency input and output:

- Input (ME_TYPE_FI)
- Output (ME_TYPE_FO)
- Bi-directional (ME_TYPE_FIO)

Note: The subdevices above are only supported in operation mode Single.

3.3.3.4 Counter

Subdevice type for counters of type 8254:

- Counters of type 8254 (ME_TYPE_CTR)

Counter subdevices have always one channel (with index 0). The corresponding parameter <`iChannel`> in the ME-iDS API functions should always be set to '0'. *meQueryNumberChannels()* always returns '1'.

Trigger options are not possible.

3.3.3.5 Interrupt

Subdevice type for external interrupt inputs:

- External interrupt (ME_TYPE_EXT_IRQ)

meQueryNumberChannels() returns the number of interrupt channels. An interrupt subdevice is always of sub-type ME_SUBTYPE_SINGLE. For interrupt handling the following functions are available.

- *meIOIrqStart():* Configure and start interrupt subdevice.
- *meIOIrqStop():* Stop interrupt subdevice.
- *meIOIrqWait():* Event listener waits for interrupt.
- *meIOIrqSetCallback():* By this function you can install an user-defined callback function, waiting for an interrupt in background.

Note: The current state of an interrupt line cannot be read via *meIOSingle().*

3.3.3.6 "FPGA" (planned)

Planned - all information preliminary! Subdevice type ME_TYPE_FPGA (FPGA = „Free Programmable Gate Array") is for subdevices who´s functionality can be designed by the user with the appropriate special know-how. A development software for the firmware design of the appropriate FPGA chip is required.

3.3.4 Structure of the API

The ME-iDS concept of getting properties can be described as „question and answer game". The software can resp. must check the supported devices for their subdevices and capabilities. With this information you can configure your hardware in the next step and finally access to it.

The whole ME-iDS API can be divided into four groups:

- Query functions
- Property functions
- Input/Output functions
- Auxiliary functions

3.3.4.1 Query Functions

Using the query functions, all the properties of the system, a particular device or a particular subdevice can be determined at run-time.

- System queries
- Device queries
- Subdevice queries
- Range queries

3.3.4.2 Property Functions

With the so-called property functions you have the possibility to read and if applicable to write all general and hardware specific properties and attributes. The entity of all properties is like a tree structure. By the so-called property path you can access to a device, a subdevice, a channel or a range and so on. See also chap. 3.3.2 from page 21.

Note: The property functions are available with ME-iDS 2.0 and higher and completely implemented for the ME-5000 series under Windows. For

all other devices only the general properties are supported at the moment. In future releases of the ME-iDS this will be extended and completed more and more.

The property functions are implemented as ANSI and Unicode (UTF-16) versions. The ANSI version have suffix 'A' and use NULL terminated ANSI strings (char*). The unicode version with the suffix 'W' use wide character strings (wchar_t*).

- Reading properties
- Writing properties

3.3.4.3 Input/Output Functions

Prepare and execute data input and output.

- Reset functions
- Interrupt functions
- Single operation
- Streaming operation

3.3.4.4 Auxiliary Functions

- Driver initialization
- Protection functions
- Error handling functions
- Utility functions

3.3.5 Basic Procedure

3.3.5.1 Initialization

Before calling functions accessing to devices of the ME-iDS, the ME-iDS has to be initialized once. The function *meOpen()* does the initialization for the whole driver system:

- Determination of the physically available devices and comparison with the devices in the device configuration which is saved with the ME-iDS.
- Changing of an alternative subdevice configuration.
- Establishes a connection to the driver modules.

To assure clean exit *meClose()* should be used at the end of a program. This function closes the ME-iDS and de-allocates all used resources.

An application can call *meOpen()* and *meClose()* many times to refresh information about connected devices. For reinitialization of the ME-iDS driver system first *meClose()* must be called prior calling *meOpen()*.

3.3.5.2 Protection

To avoid concurrent access from different applications the same resources we recommend using the *meLock...()* functions provided by the ME-iDS.

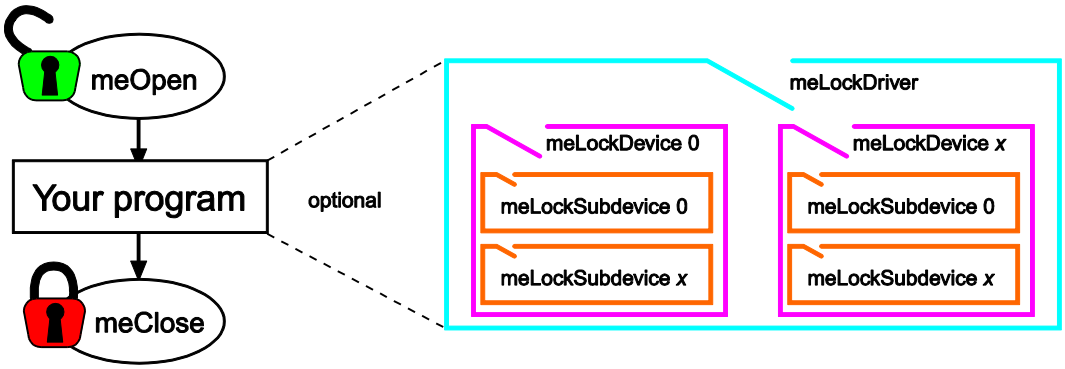

Diagram 9: Lock hierachy

There are three levels of locking resources:

- Driver
  - Device
    - Subdevice

A resource cannot be locked if:

…the resource itself or one of its sub-resources is already locked by another application. For example, you cannot lock a device if one of its subdevices is already locked by another application.

…the resource itself or one of its sub-resources is not in idle state. For example, you cannot lock a device if one of its subdevices is waiting for an external trigger.

If a resource is locked by an application and a second application attempts to call an IO function accessing the resource or one of its sub-resources then the function returns the error code ME_ERRNO_LOCKED. If the access to locked resourses should be allowed again, it must be done by the application which locked the resources prior by calling the function *meLock…().*

The *meQuery…()* and *meUtility…()* functions are unaffected by locks. These routines can be called at any time by any application providing *meOpen()* has been called first.

IMPORTANT: Locks only prevent concurrency among different applications (processes). If you wish to prevent concurrency among different threads running in the same process, then you must use one of the synchronization methods provided by your operating system (e.g. mutex, semaphore etc.).

If you can be sure that at any time only a single application will be accessing a resource (for example, if you have a single application using the ME-iDS) then you will not need to use the *meLock…()* routines at all.

3.3.5.3 Error handling

All of the ME-iDS functions return the error status as an integer. See the ME-iDS help file for a list of all error codes.

On success ME_ERRNO_SUCCESS (equal to '0') is returned. Otherwise an error code different from ME_ERRNO_SUCCESS is returned and the internal variable <`iErrno`> is set accordingly.

For convenience the ME-iDS provides the function *meErrorGetMessage()* which converts error codes into a textual error description.

ME-iDS provides two functions to determine the last error occured:

- *meErrorGetLast()* - returns the last error code.
- *meErrorGetLastMessage()* - returns the error description of the last error.

NOTE: If a function tries to create a situation which already exists, then the routine will return ME_ERRNO_SUCCESS. For example, if one of the *meLock…()* routines is called to unlock a resource which is not locked, then it will return ME_ERRNO_SUCCESS.

Common errors:

- ME_ERRNO_NOT_OPEN: ME-iDS has not been opened correctly.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested device index.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested subdevice index.
- ME_ERRNO_DEVICE_UNPLUGGED: device is physically not available at the moment.
- ME_ERRNO_NOT_SUPPORTED: function is not supported by the subdevice (example: *meIOStreamConfig()* applied to a subdevice which is only for single operation).

For function specific errors see function reference. If necessary the user can enable error logging:

- *meErrorSetDefaultProc()*:
  - **Windows:** A message box with error code and error description is displayed. Pressing „OK“ simply dismisses the messagebox and pressing „Cancel“ dismisses the message box and suppresses its display for future errors.

- *meErrorSetUserProc():* call user-defined logging functions.

3.4 Operation Modes

3.4.1 Single Operation

This operation mode serves both the reading as well as the writing of single values. Before any I/O operation can be carried out a subdevice has to be configured by calling *meIOSingleConfig()*. *meIOSingle()* is then called to carry out the I/O operation itself. Once configured, a subdevice can read resp. write many times. Several single operations can be written to a list and run consecutively by calling the function *meIOSingle()*.

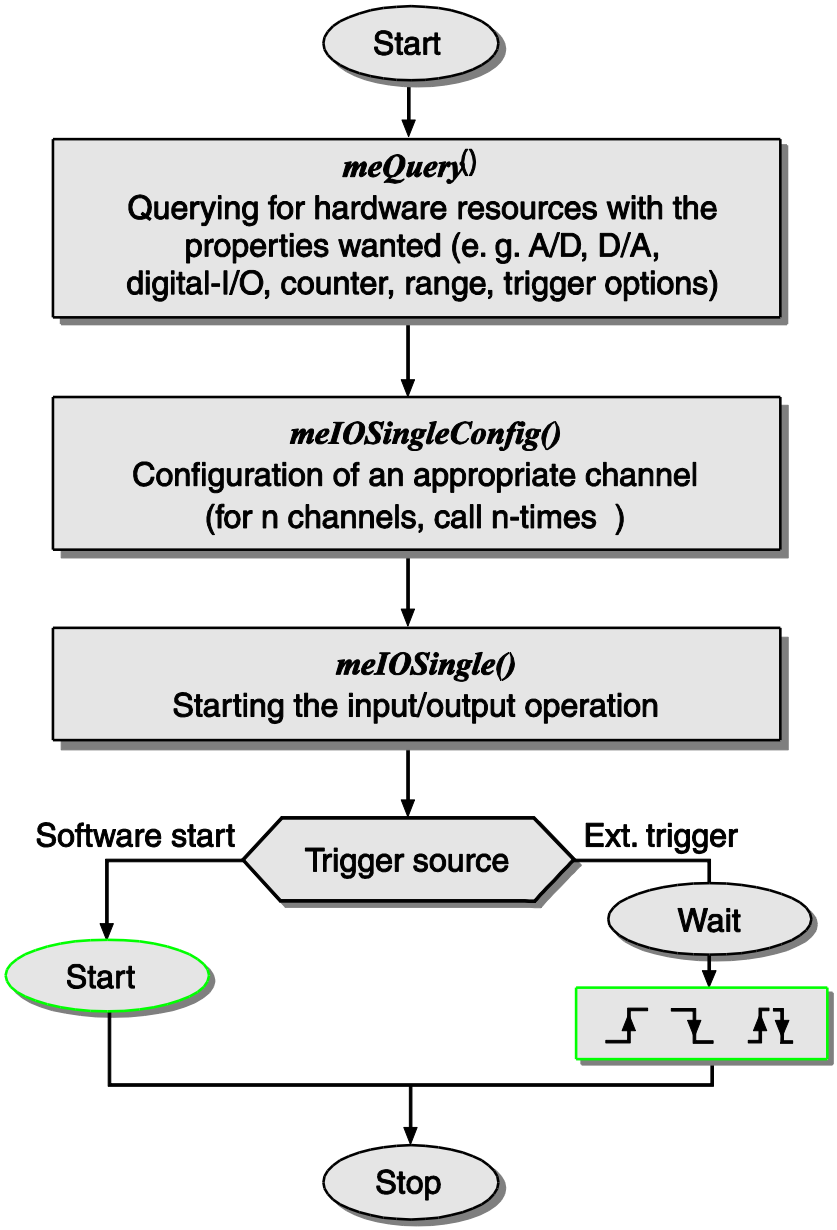

Diagram 10: "Single" operation mode

Important: Operations in a single list of type `meIOSingle_t` are performed in order of entry. If an operation has to wait (e.g. for an external

trigger signal) the next one also must wait. Processing a single list is stopped if one operation fails.

3.4.1.1 Start Operation/Trigger Options

A single operation can be triggered in various ways. The parameter <`iTrigType`> of the function *meIOSingleConfig()* determines the kind of trigger used and the options available here depend on the subdevice in question. Here is a list of the available trigger types:

- No trigger available (ME_TRIG_TYPE_NONE):
  Some subdevices (like counter) have no triggering options. The operation is always performed immediately after calling *meIOSingle()*. No timeout is available.
- Software trigger (ME_TRIG_TYPE_SW):
  Subdevice is read resp. written directly after calling *meIOSingle()* or via the synchronous list (see page 80).
- External digital trigger (ME_TRIG_TYPE_EXT_DIGITAL): Subdevice can be triggered by an external trigger event. The digital trigger can be used to start the synchronous list (see page 80).
- External analog trigger (ME_TRIG_TYPE_EXT_ANALOG): Subdevice can be triggered by an external analog signal analysed by an internal comparator (see the following diagram).

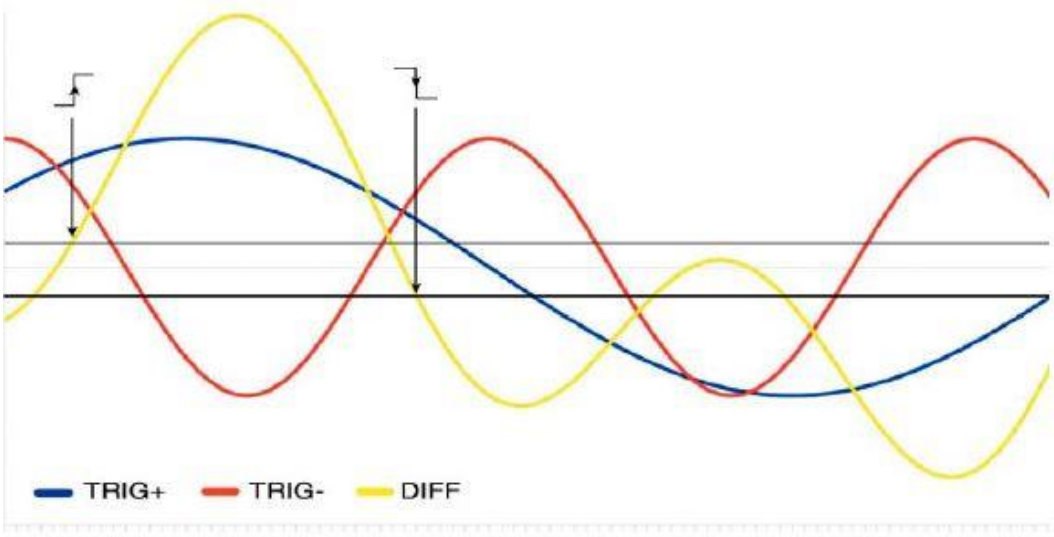

Diagram 11: Analog trigger

NOTE: Analog trigger has certain hysteresis behaviour. For details see the appropriate hardware manual.

External trigger can be configured via the parameter <`iTrigEdge`> to act on rising, falling or any trigger edge.

Diagram 12: Trigger edges

Parameter <iTrigChan> is used for adding entries to a synchronous list. For details see chapter "Synchronous Start" on page 74.

NOTE: Please check hardware manual for possible triggering modes.

IMPORTANT: Parameters passed in *meIOSingleConfig()* and *meIO- Single()*, like <iChannel>, have different meanings depending on subdevice type. Detailed infos ca be found in the ME-iDS help file.

3.4.1.2 Analog Input/Output

Values are read and written as integers. They can be converted into physical units with the function *meUtilityPhysicalToDigital()*. The function *meUtilityDigitalToPhysical()* provides the opposite conversion. The parameters <dMin>, <dMax> and <iMaxData> in the conversion functions must correspond to the parameters <pdMin>, <pdMax> and <piMaxData> (see *meQueryRangeInfo()*) of that range which is used in parameter <iSingleConfig> of the function *meIOSingleConfig()*.

ME-iDS provides several functions that help the user to find the most suitable range for a particular input or output operation.

- *meQueryNumberRanges()*: returns the number of available ranges for a particular subdevice.
- *meQueryRangeInfo()*: returns the parameters which define the chosen range:
  - <piUnit> Physical unit (e.g.: ME_UNIT_VOLTS for Volts).
  - <pdMin> Lower range limit of selected physical unit.
  - <pdMax> Upper range limit of selected physical unit.
  - <piMaxData> Integer value representing the digital resolution (e.g.: 65535 for 16-bit resolution). The lower range limit is always 0.
- *meQueryRangeByMinMax():* returns the index of the most suitable range. This is the smallest range which includes the given minimum and maximum values.

The ground reference for analog measurements has to be configured using the parameter <iRef>. With differential measurements only bipolar input ranges can be used (see also ME-iDS help file and hardware manual).

3.4.1.3 Digital Input/Output

Basically digital ports have to be configured using the function *meIOSingleConfig()* before a read or write operation is carried out. The pins of a bidirectional subdevice are set to input after a reset to avoid undefined signal levels. Because of the configuration can be done in various bit width (bit, byte, word,…) - depending on hardware - the name „blocks“ was introduced. The parameter <iChannel> represents the index of the block to be configured (see also function *meQuerySubdeviceCaps()* on page 104).

The function *meQuerySubdeviceCaps()* returns the width of a block. For example: ME_CAPS_DIO_DIR_BYTE means, that the port has to be configured with the block size of a byte. It is not possible to configure bit 0 as input and the other seven bits as output. Compared to this ME_CAPS_DIO_DIR_BIT means that the block size is one bit and every bit can be configured independently. With the flag ME_IO_ SINGLE_CONFIG_NO_FLAGS all channels of the subdevice will be used.

The parameter <iSingleConfig> defines the direction of a block. There are different hardware implementations possible. If only standard input/output operations are possible use ME_SINGLE_CONFIG_ DIO_INPUT or ME_SINGLE_CONFIG_DIO_OUTPUT. Otherwise please refer to the ME-iDS help file and the hardware manual for availability and default setting.

For example the opto-isolated boards ME-5810 and ME-8100 provide output buffers with additional features:

- Output with sink driver
  (ME_SINGLE_CONFIG_DIO_SINK)
- Output with source driver
  (ME_SINGLE_CONFIG_DIO_SOURCE)
- High impedance state
  (ME_SINGLE_CONFIG_DIO_HIGH_IMPEDANCE)

A read or write operation with function *meIOSingle()* requires the port width in parameter <iFlags> for each entry (ME_IO_SINGLE_ TYPE_DIO_...). Parameter <iChannel> represents the index of the block to be accessed. With the flag ME_IO_SINGLE_CONFIG_NO_ FLAGS all channels of the subdevice will be used.

If you want to output a timer-controlled bit pattern with the ME-4680 to one (or several) digital ports, pass ME_SINGLE_CONFIG_DIO_BIT_PATTERN in the parameter <iSingleConfig> of function *meIOSingleConfig()* and set the parameter <iRef> to ME_REF_DIO_ FIFO_LOW resp.

ME_REF_DIO_FIFO_HIGH. Otherwise use ME_REF_NONE. For details see chap. 3.4.3.2 „Bit-Pattern Output of ME-4680“ on page 73 and appendix A3 from page 182.

3.4.1.4 Frequency Input/Output

A subdevice for frequency measurement (input) resp. pulse output (pulse generator) can contain one or more channels. The assignment of the channels to the subdevices can be found in the ME-iDS help file.

Basically a subdevice of type “ME_TYPE_FI“, “ME_TYPE_FO“ or “ME_TYPE_FIO“ must be configured for operation mode single using the function *meIOSingleConfig()*. The number of channels can be queried with the function *meQueryNumberChannels()*. Pass the index of the channel in parameter <`iChannel`>.

To describe the rectangular signal two variables have been introduced, which apply for input and output likewise. The one value defines the period T and the other value the first phase of the period t_{1P}. When measuring frequency the measurement starts on the first positive edge and ends with the consecutive positive edge. The negative edge between defines the end of the first phase. In pulse generator operation the output starts with a high level and toggles to low after the end of the first phase.

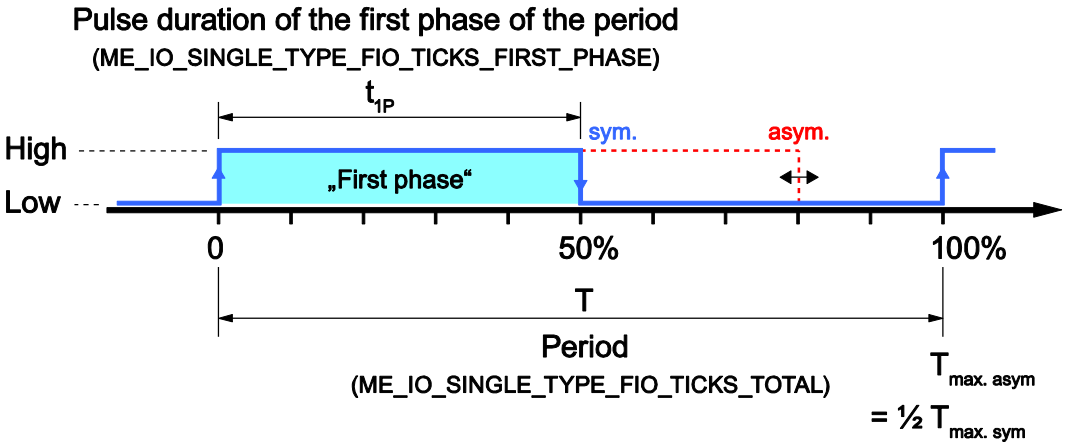

Diagram 13: Signal definition

Note: according to the system clock different values for the maximum period T_{max} are valid depending on the duty-cycle. There is a difference between rectangular signals with asymmetrical duty-cycle $T_{max.asym.}$ and symmetrical duty-cycle $T_{max.sym}$.

As a rule it applies: $T_{max.asym.} = ½ T_{max.sym.}$

3.4.1.4.1 Frequency Measurement

Using the operation mode frequency measurement (FI="Frequency Input") you can determine the period and duty-cycle of rectangular signals. Each frequency input channel will be accessed as subdevice of type ME_TYPE_FI, sub-type ME_SUBTYPE_SINGLE. The functions *meQuery…(), meIOSingleConfig(), meIOSingle() und meIOSingleTicks-ToTime()* are relevant:

- Determine subdevice by the *meQuery…()* functions.
- Configuration of the subdevice by the function *meIOSingleConfig():*
  - Pass the channel index in parameter <`iChannel`>
  - Pass ME_SINGLE_CONFIG_FIO_INPUT in the parameter <`iSingleConfig`>
  - The parameters <iRef>, <iTrigChan>, <iTrigType> and <`iTrigEdge`> are not required here. Please pass ME_VALUE_NOT_USED.
  - Pass ME_IO_ SINGLE_CONFIG_FI_SINGLE_MODE in parameter <`iFlags`>.
- To read period and duration of the first phase of the period you have to call the function *meIOSingle()* twice. Depending on the option in parameter <`iFlags`> either the total period (in ticks) or the duration of the first phase of the period (in ticks) will be returned in parameter <`iValue`>.

For standard operation we recommend the following procedure:

1. Reading the period by bitwise OR-linking the flags: ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL „or" ME_IO_SINGLE_TYPE_ NONBLOCKING.
2. Reading the duration of the first period by bitwise OR-linking the flags: ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_PHASE „or" ME_IO_SINGLE_TYPE_NONBLOCKING.

For repeated acquisition of slow frequencies the following version is possible (see also parameter <`iFlags`> of the function *meIO-SingleConfig()):*

1. Reading the period by bitwise OR-linking the flags: ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL „or" ME_IO_SINGLE_TYPE_FI_LAST_VALUE „or" ME_IO_SINGLE_TYPE_NONBLOCKING.
2. Reading the duration of the first period by bitwise OR-linking the flags: ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_PHASE „or" ME_IO_SINGLE_TYPE_FI_LAST_VALUE „or" ME_IO_SINGLE_TYPE_NONBLOCKING.

• For easy conversion of ticks into seconds you can use the function *meIOSingleTicksToTime()*. You have to call the function separately for period

and duration of the first phase of the period. Note that the option for <iTimer> corresponds with <iFlags> in the function *meIOSingle().*

In parameter <iTicksLow> the ticks from <iValue> are passed (see above). Finally parameter <pdTime> returns a pointer to the calculated value in seconds.

Note: If you need the dimensions frequency and duty-cycle they can be easily calculated by the return values of <pdTime>. It applies:

Frequency [Hz] = 1/period [s]

Duty-cycle [%] = („Duration of the first phase of the period“ [s]

/ period [s]) × 100

The parameter <iTicksHigh> is reserved for future enhancements. Please pass 0 here.

3.4.1.4.2 Pulse Generator

Using the operation mode pulse generator (FO=“Frequency Output“) you can output a rectangular signal with variable duty cycle by a resolution of one tick. Each pulse generator channel (output) is accessed as subdevice of type ME_TYPE_FO, sub-type ME_SUBTYPE_SINGLE. The functions *meQuery…(), meIOSingleConfig(), meIOSingle()* und *meIOSingleTime-ToTicks()* are relevant:

- Determine subdevice by the function *meQuery…().*
- Configuration of the subdevice by the function *meIOSingleConfig():*
  - Pass the channel index in parameter <iChannel>
  - Pass ME_SINGLE_CONFIG_FIO_OUTPUT in the parameter <iSingleConfig>.
  - In parameter <iTrigChan> pass either ME_TRIG_CHAN_DEFAULT or ME_TRIG_CHAN_SYNCHRONOUS for a synchronous start of several pulse generators (see chap. 3.4.3.3 on page 74)
  - The parameters <iRef>, <iTrigType> and <iTrigEdge> are not required here. Please pass ME_VALUE_NOT_USED.
  - Pass ME_IO_SINGLE_CONFIG_NO_FLAGS in parameter <iFlags>.
- For easy conversion of the signal to be output from seconds into ticks the function *meIOSingleTimeToTicks()* is useful. You have to call the function separately for period and duration of the first phase of the period.

  - In parameter <iTimer> first choose ME_TIMER_FIO_ TOTAL for the period and then ME_TIMER_FIO_FIRST_ PHASE for the first phase of the period. Pass the desired appropriate value in seconds to the parameter <pdTime>.
  - Each time <piTicksLow> returns a pointer with the ticks to be passed in parameter <iValue> of the function *meIOSingle()* corresponding to <iFlags>.
- To pass the period and the duration of the first phase of the period you must call the function *meIOSingle()* twice.

For standard operation we recommend the following procedure:

1. Pass the period by bitwise OR-linking the flags:
   ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL „or“ ME_IO_SINGLE_TYPE_FO_UPDATE_ONLY
2. Pass the duration of the first phase of the period by bitwise OR-linking the flags:
   ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_PHASE
   „or“ ME_IO_SINGLE_TYPE_FO_START_SOFT.
   - Pass ME_DIR_OUTPUT in parameter <iDir>.
     Use ME_DIR_INPUT for readback operation.
   - In parameter <iValue> the ticks from <piTicksLow> (see above) are passed. Note that function *meIOSingle()* must be called twice and that the value corresponds with <iFlags>.
   - Starting the output operation can be controlled by appropriate combination of the flags in parameter <iFlags>. It is done by bitwise OR-linking
     ME_IO_SINGLE_TYPE_FIO_TICKS_ TOTAL resp. ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_ PHASE with one or more of the following options:
   - ME_IO_SINGLE_TYPE_FO_UPDATE_ONLY
     The output value should be updated but not output at once. No linking with other flags possible. Default: the new value is output immediately.
   - ME_IO_SINGLE_TYPE_FO_START_SOFT
     The value is output not until the end of the current period (if already running). Can be bitwise OR-linked with ME_IO_SINGLE_TYPE_FO_START_LOW. Default: the new value is output immediately.
   - ME_IO_SINGLE_TYPE_TRIG_SYNCHRONOUS
     All subdevices, which have been added to the synchronous list by parameter <iTrigChan> of the function *meIOSingleConfig()* will be started simultanously (see also chap. 3.4.3.3 on page 74). Default: subdevice starts independently.
   - ME_IO_SINGLE_TYPE_FO_START_LOW

By default the first phase of the rectangular signal is „high“. If the flag is set, the output starts with „low“ level. Can be OR-linked bitwise with ME_IO_SINGLE_TYPE_FO_START_SOFT.

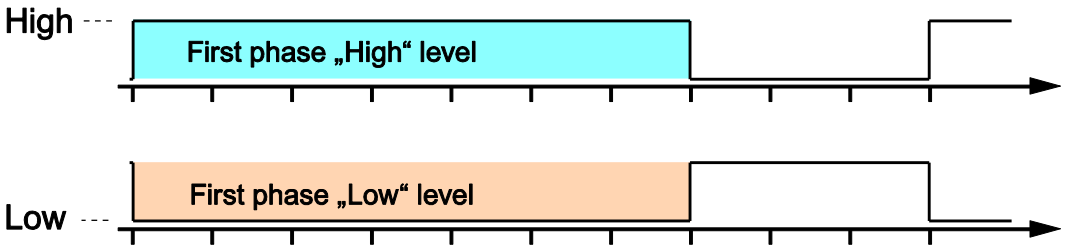

Diagram 14: Negated pulse output

3.4.1.5 Counter Operation

Although the ME-iDS can support various types of counters, currently only the standard counter chip of type 8254 is implemented which provides three 16-bit counters.

Each counter is accessed as a subdevice of type ME_TYPE_CTR, subtype ME_SUBTYPE_CTR_8254. For counter subdevices the parameter <`iChannel`> should be set 0. *meQueryNumberChannels()* always returns 1.

NOTE: There is no „STOP“ function for counters. Use *meIOReset- Subdevice()* instead. Trigger options are not available for counters.

Parameter <`iRef`> defines the clock source for the counters:

- Use external clock generator as clock source (ME_REF_CTR_EXTERNAL)
- Use output of the previous counter as clock source (ME_REF_CTR_PREVIOUS)
- Use the internal 1 MHz clock generator as clock source (ME_REF_CTR_INTERNAL_1 MHZ)
- Use the internal 10 MHz clock generator as clock source (ME_REF_CTR_INTERNAL_10 MHZ)

Parameter <`iSingleConfig`> represents the chosen counting mode. The three counters of the chip can be configured indpen-dently of each other for the following 6 operation modes (see also appendix A1 on page 177):

- Mode 0: Change state at zero (ME_SINGLE_CONFIG_CTR_8254_MODE_0)
- Mode 1: Retriggerable „One Shot“ (ME_SINGLE_CONFIG_CTR_8254_MODE_1)
- Mode 2: Asymmetric divider (ME_SINGLE_CONFIG_CTR_8254_MODE_2)
- Mode 3: Symmetric divider (ME_SINGLE_CONFIG_CTR_8254_MODE_3)
- Mode 4: Counter start by software trigger (ME_SINGLE_CONFIG_CTR_8254_MODE_4)
- Mode 5: Counter start by hardware trigger (ME_SINGLE_CONFIG_CTR_8254_MODE_5)

Note: The real voltage level at the inputs/outputs of the counters depends on the respective hardware. For example on opto-isolated versions of the ME-4600 series a high level at the output corresponds to the „high impedance“ state and a low level to the state „driving“. Please consult the corresponding hardware manual. The logic levels in the following description apply for the counter chip without further circuitry.

The following diagram should explain the program flow briefly:

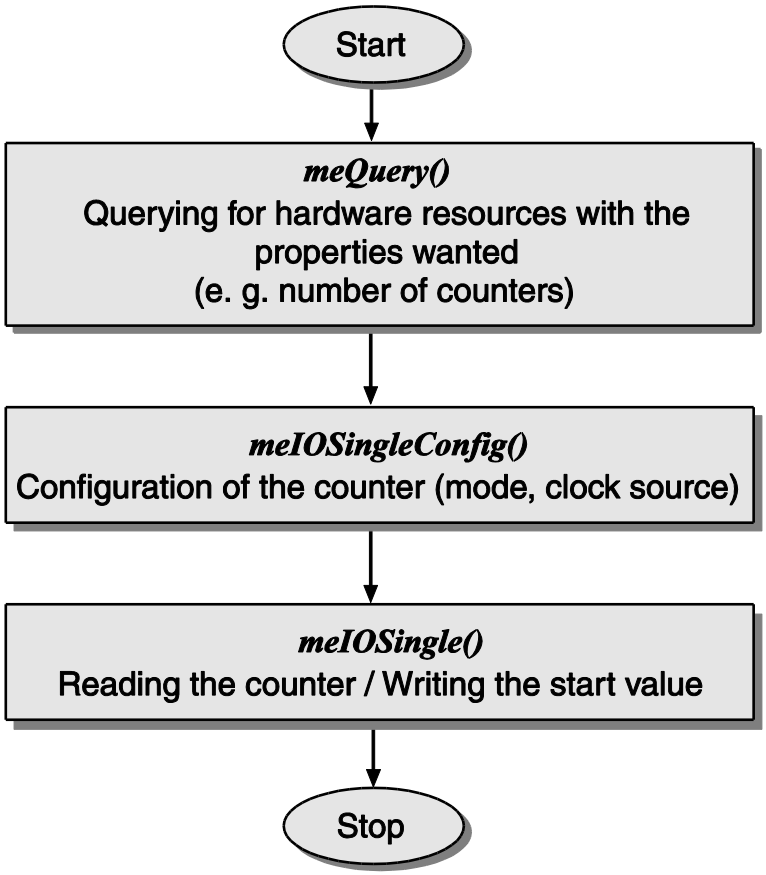

Diagram 15: Programming the counter

3.4.1.5.1 Mode 0: Change State at Zero

This mode of operation can be used e.g. to trigger an interrupt when the counter meets zero. The counter output (OUT_0…2) is set to low when the counter is initialised or when a new start value is loaded. To enable counting, a high level must be applied to the GATE input. As soon as the start value is loaded and the counter is enabled, the counter begins counting downwards and the output remains low.

Upon zero axis crossing, the output is set to high and remains there until the counter is reloaded or initialised again. The counter continues to count down, even after zero is meet. If a counter register is loaded during a count in progress the following occurs:

1. when the first byte is written, the count process is stopped.
2. when the second byte is written, the count process begins again.

3.4.1.5.2 Mode 1: Retriggerable „One-Shot“

The counter output (OUT_0…2) is set to high when the counter is initialised. When a start value is loaded the output becomes low on the next

clock following to the first trigger pulse at the GATE input (positive edge). Upon zero axis crossing, the counter output is set to high again.

On a positive edge at the GATE input, the counter can be reset (retriggered) to the start value. The output remains low until the counter meets zero.

The counter value can be read at any time without effecting the count process.

3.4.1.5.3 Mode 2: Asymmetric Divider

In this mode, the counter is used as a frequency divider. The counter output (OUT_0…2) is set to high after initialisation. When the counter is enabled by applying a high level to the GATE input, the counter is counting downwards and the output remains high. When the count meets the value 0001Hex, the output becomes low for one clock cycle. This process will be repeated periodically as long as the GATE input is enabled (high level), else the output is set to high immediately.

If the counter register is reloaded between two output pulses, the current counter state is not affected. However the new value is used on the next period.

3.4.1.5.4 Mode 3: Symmetric Divider

This mode of operation is similar to mode 2 with the difference that the divided frequency is symmetric (only for even count values). The counter output (OUT_0…2) is set to high after initialisation. When the GATE input is enabled (high level), the counter is counting downwards in steps of 2. The output will toggle its state on a half of the start value number of periods referenced to the input clock (starting with high level). This process will be repeated periodically as long as the GATE input is enabled (high level), else the output is set to high immediately.

If the counter is reloaded between two output pulses, the current counter state is not affected. The new value is used on the next period.

3.4.1.5.5 Mode 4: Counter Start by Software Trigger

The counter output (OUT_0…2) is set to high when the counter is initialised. To enable the counter the GATE input must be enabled (high level). When the counter is loaded (software trigger) and enabled, the counter starts counting downwards, while the output remains high.

Upon zero axis crossing the output becomes low for one clock cycle. Afterwards the output becomes high again and remains there until the counter is initialised and a new start value is loaded.

If the counter is reloaded during a count process, the new start value is used in the next cycle.

3.4.1.5.6 Mode 5: Counter Start by Hardware Trigger

The counter output (OUT_0…2) is set to high when the counter is initialised. After loading a start value to the counter, counting starts on the next clock following to the first trigger pulse at the GATE input (positive edge). Upon zero axis crossing, the output becomes low for one clock cycle. Afterwards the output becomes high again and remains there until the next trigger pulse occurs.

If the counter register is reloaded between two trigger pulses, the new start value is used after the next trigger pulse.

The counter can be reset to the start value (re-triggered) at any time by applying a positive edge to the GATE input. The output remains high until zero axis crossing is meet.

3.4.1.5.7 Mode „Pulse Width Modulation“

A special application for the counters of type 8254 is the output of a rectangular signal with a variable duty cycle („PWM“ mode). With that you can output a rectangular signal of maximum 50 kHz with a variable duty cycle to OUT_2.

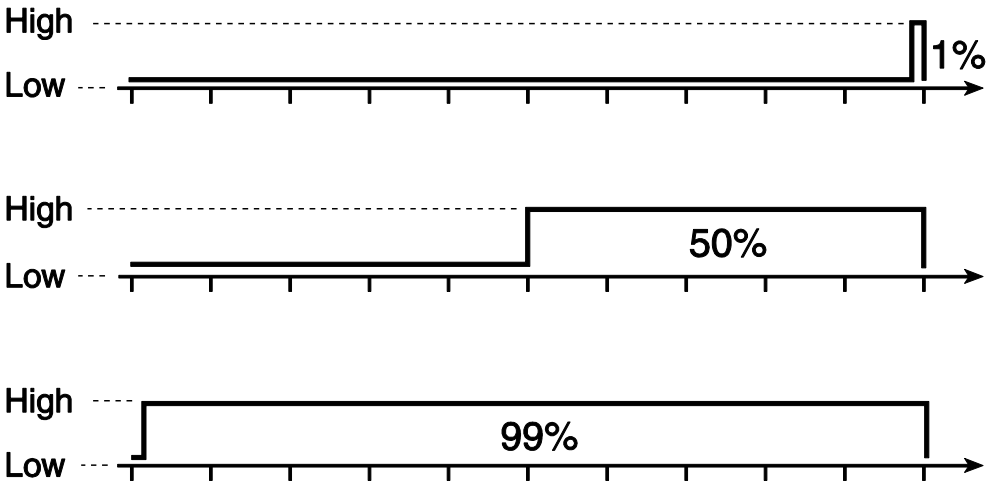

Diagram 16: PWM signal

Condition is a correct switching of inputs and outputs (CLK, GATE, OUT) by the external circuitry. Please read the corresponding chapter regarding the PWM switching - especially of opto-isolated counters - in your hardware manual.

Basically the following switching applies:

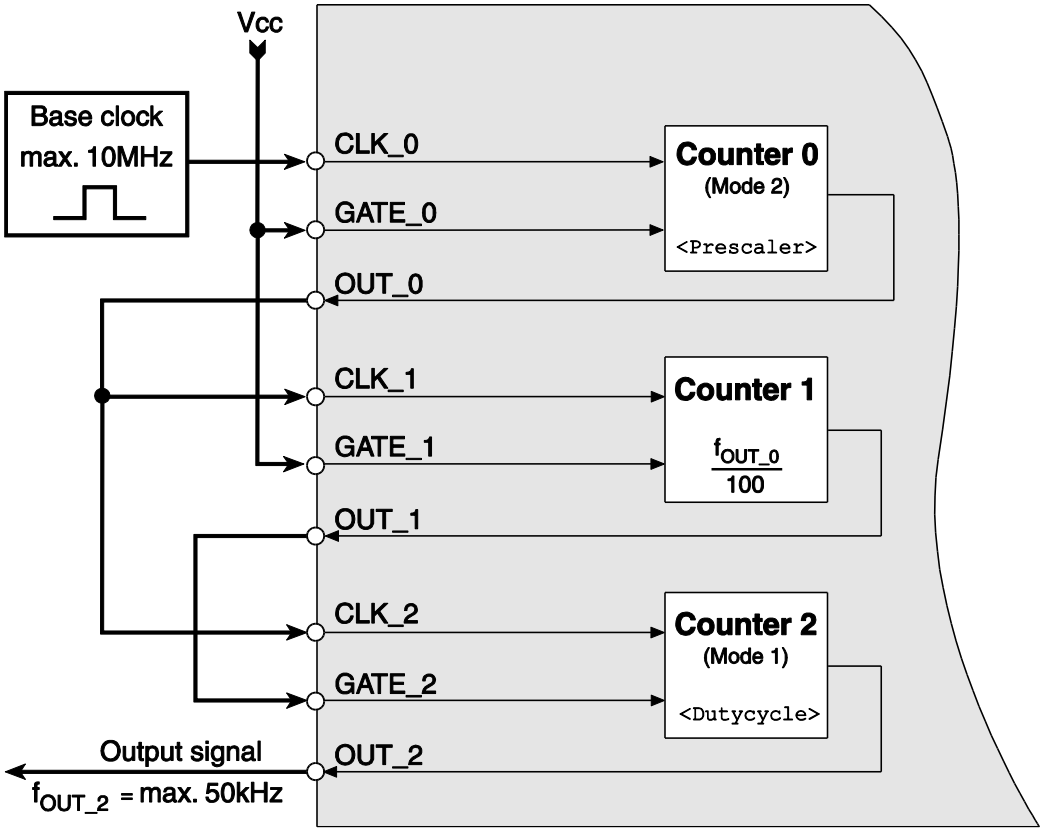

Diagram 17: Switching pulse width modulation

Counter 0 is used as a prescaler for the externally driven base clock. Using the parameter <iPrescaler> you can set the frequency f_{OUT_2} as follows:

$$f_{OUT\_2} = \frac{\text{Base clock}}{\text{<iPrescaler>} \cdot 100} \quad \text{(with <iPrescaler> = 2…(}2^{16}\text{– 1))}$$

By the parameter <iDutyCycle> you can set the duty cycle between 1…99 % in steps of 1 % (see diagram 16 on page 49). The operation is started immediately after calling the function *meUtiltiyPWMStart()* and stopped by the function *meUtiltiyPWMStop().* No further programming of the counters is required.

On opto-isolated devices the output OUT_2 is an open collector output as a rule. I.e. logical „1“ means output is driving and logical „0“ that the output is in a high impedance state (see hardware manual).

3.4.2 Streaming Operation

The term „streaming“ covers all those operations in which data are transfered via FIFO. The timing is controlled by a timer and/or external trigger signals. The subdevice must be of sub-type ME_SUBTYPE_ STREAMING. This can be checked using the function *meQuerySubdeviceType()* (parameter <`piSubtype`>).

If supported by the hardware this applies not only analog input/output but also to digital/output operation. Please refer to appendix A3 on page 182 for a detailed description of the bit-pattern output of the ME-4680.

Proceed as follows:

1. Query the capabilities with the *meQuery…* functions resp. determine the properties with the *meProperty*… functions.
2. Configure the subdevice with the function *meIOStreamConfig()* and *meIOSetChannelOffset()* if you are using a subdevice with an adjustable offset.
3. For an output operation the buffer must be preloaded with the function *meIOStreamWrite().* Pass the option ME_WRITE_ MODE_PRELOAD in parameter <`iWriteMode`>.
4. Start streaming with function *meIOStreamStart().*
5. Read or write data (*meIOStreamRead()* resp. *meIOStreamWrite()*).
6. Stop streaming with the function *meIOStreamStop()* or meIOReset…().

Note: Streaming is not possible for subdevices of type counter (ME_TYPE_CTR), frequency input/output (ME_TYPE_FIO /ME_TYPE_FI/ME_TYPE_FO) and interrupt (ME_TYPE_IRQ).

3.4.2.1 Querying Hardware Properties

For querying the hardware resources you can use either the query functions (*meQuery*…) or the property functions (*meProperty*…). Using the query functions you can query a device for availability of certain subdevices or capabilities (caps) (e.g. „Provides the device a subdevice of type analog input?“). Another way follow the property functions. Similar to a tree structure you can query and set (if applicable) properties and attributes by their property path

3.4.2.2 Configuring Hardware

In order to configure a subdevice for streaming mode two data structures have to be prepared always:

1. The channel-list defines which digital ports resp. analog channels - and further channel specific parameters if applicable - have to be set. See also page 58.
2. The trigger structure defines the conditions how to start a streaming operation and how values should be read or written. See page 58. Choose one of these timing modes:
    - Stream-Timer (see page 56).
    - Stream-Trigger-Sample (see page 59).
    - Stream-Trigger-List (see page 60).

3.4.2.3 Channel List

The channel list (`meIOStreamConfig_t`) defines which digital port resp. analog channels - and further channel specific parameters if applicable - have to be set. Even if no channel specific parameter can be set a channel list with a standard entry must be written. Allocate a buffer of defined size to which the channel list entries are written. The maximum number of channel list entries depends on hardware (e.g. max.1024 entries for the ME-4600 series). This can be queried by the *meQuery*… functions.

Typical settings in the channel list:

- Channel index for analog channels resp. subdevice index for digital inputs/outputs.
- Measurement range for analog channels.
- Ground reference: e.g. single-ended or differential.

3.4.2.4 Trigger Structure

The trigger structure (`meIOStreamTrigger_t`) defines conditions for start and stop of a streaming operation.

For example you can define the trigger type (software or external analog or digital trigger) and trigger edge. Depending on the device different trigger conditions are possible.

There are 3 hierarchical trigger levels and several trigger conditions. The concept should be explained in the following:

Level 1: Concerns the operation as a whole (`iAcq`…)

Level 2: Processing a list e.g. the channel list for analog acquisition (`iScan`…)

Level 3: The conversion of a single value (`iConv`…)

Start of the whole operation (`<iAcqStart…>`)	
	Start of a list processing (scan)

	<iScanStart>	
		Start of a single sample/conversion <iConvStart…>
	Stop of a list processing (scan) <iScanStop…>	
Stop of the whole operation (<iAcqStop…)		

Table 10: Trigger structure

Tip: Initialize the trigger structure with 0. In that way you must only take care of the parameters which are required. At the same time unused parameters are automatically passed correctly.

a. Trigger Type <iAcqStartTrigType>

This parameter defines the trigger type for start of the whole operation. Depending on the used hardware you can choose between the options software start and external analog or digital trigger (see also function description on page 79).

b. Trigger Edge <iAcqStartTrigEdge>

This parameter defines the trigger edge for start of a single conversion. Depending on the trigger type and the used hardware you can choose between different options for the trigger edge (rising, falling, etc.). See also function description on page 79.

c. Trigger Channel <iAcqStartTrigChan>

With this parameter you can choose whether triggering should be done separatly for each channel (standard) or if a channel should be started synchronologically with other channels. E.g. for analog acquisition with sample & hold option or synchronous start of several analog output channels. See also function description on page 79.

d. Offset Time <iAcqStartTicksLow>, <iAcqStartTicksHigh>

Number of ticks between start of the operation and the first conversion. Note that the settling time of the AI-section may not be fallen below. For standard applications we recommend to set the offset time to the minimum chan interval of the appropriate hardware (AcqStartTicks = min. ConvStartTicks). See also function description on page 79.

By combining „AcqStartTicksLow“ and „AcqStartTicksHigh“ values up to 64-bit width are possible, if supported by the hardware. It applies: AcqStartTicks = (AcqStartTicksHigh <<32) ∨ AcqStartTicksLow.

e. AcqStartArgs <iAcqStartArgs>

This parameter is used for extended trigger options of MEphisto-Scope like window trigger or trigger on slope. See also function description on page 79.

f. Trigger Type <iScanStartTrigType>

This parameter defines the trigger type for start of a scan. Depending on the used hardware you can choose between numerous options like external analog or digital trigger, timer-controlled, … See also function description on page 79.

g. Scan Interval

<iScanStartTicksLow>, <iScanStartTicksHigh>

This parameter determines the time interval between the start of two consecutive scans (= channel list processing). Usage is optional. See also function description on page 79.

By combining „ScanStartTicksLow“ and „ScanStartTicksHigh“ values up to 64-bit width are possible, if supported by the hardware. It applies:

ScanStartTicks = (ScanStartTicksHigh <<32) ∨ ScanStartTicksLow.

Note the following relation when calculating the scan interval (see also timing diagrams from page 57):

ScanStartTicks = (number of channel list entries x ConvStartTicks) + „pause“ [Ticks].

h. ScanStartArgs <iScanStartArgs>

This parameter is reserved for future extensions.

i. Trigger Type <iConvStartTrigType>

This parameter defines the trigger type for start of a single conversion. Depending on the used hardware you can choose between the options external analog or digital trigger, timer-controlled,… See also function description on page 78.

j. Chan Interval

<iConvStartTicksLow>, <iConvStartTicksHigh>

This parameter determines the chan interval in number of ticks between two conversions (sample resp. output rate). See also function description on page 163.

By combining „ConvStartTicksLow“ and „ConvStartTicksHigh“ values up to 64-bit width are possible, if supported by the hardware. It applies:

ConvStartTicks = (ConvStartTicksHigh <<32) v ConvStartTicksLow.

k. ConvStartArgs <iConvStartArgs>

This parameter is reserved for future extensions.

l. Trigger Type <iScanStopTrigType>

This parameter defines the trigger type for ending the scan. Depending on the used hardware you can end the scan operation e.g. after the total number of conversions defined in <iScanStopCount>. See also function description on page 78).

m. Number of Conversions <iScanStopCount>

This parameter determines the total number of conversions after which the scan operation should be ended and at the same the operation as a whole. If you want to run the operation for an infinite time please pass 0 here. See also function description on page 78.

n. ScanStopArgs <iScanStopArgs>

This parameter is used for extended trigger options of MEphisto-Scope. See also function description on page 184.

o. Trigger Type <iAcqStopTrigType>

On demand, this parameter defines the trigger type for ending the whole operation. The following options are available (see also function description on page 78):

- The operation will be ended as soon as the number of scans (channel list processing), defined in <iAcqStopCount> has been meet.
- The operation will be ended as soon as the number of conversions, defined in <iScanStopCount> has been meet.

p. Number of Scans <iAcqStopCount>

This parameter defines the number of scans (channel list processing) after which the whole operation will be ended.

If you want to run the operation for an infinite time please pass 0 here. See also function description on page 78.

q. AcqStopArgs <iAcqStopArgs>

This parameter is reserved for future extensions.

Notes:

In the following diagrams (page 63 to 67) is only the rising trigger edge active for external trigger signals.

See also chapter „Synchronous Start“ on page 74.

The parameters <iAcqStartTrigType>, <iScanStartTrigType> and <iConvStartTrigType> define all start conditions possible.

For passing the concrete values for the appropriate subdevice please refer to the ME-iDS help file.

Choose one of the operation modes and note the diagrams on the following pages:

- Stream-Timer (see page 56)
- Stream-Trigger-Sample (see page 59)
- Stream-Trigger-List (see page 60)

3.4.2.4.1 Timing Stream-Timer

Timer-controlled streaming operation (with/without scan-timer). Start by software or external trigger pulse after calling the function *meIOStream-Start()* - all further trigger pulses will be ignored.

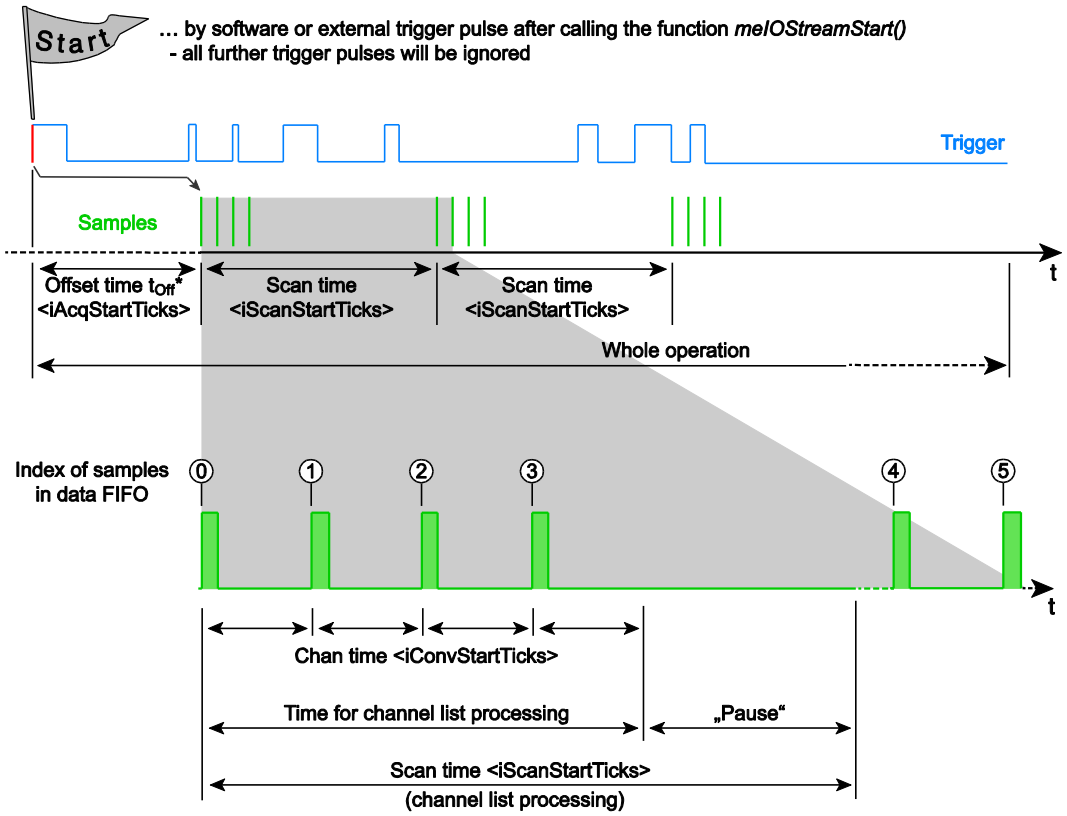

Diagram 18: Timing diagram Stream-Timer

* **Note** that the offset time t_{Off} must consider the settling time of the hardware! For standard applications we recommend to equate the offset time with the minimum chan time of the hardware (<`iAcqStartTicks`> = min. <`iConvStartTicks`>).

** Index of the samples in data FIFO. The values are always sampled corresponding to the order of the channels in the channel list. Independent whether channel specific parameters can be set on the hardware or not, a channel list with a standard entry must be written. See function *meIOStreamConfig()*.

Parameter	Software trigger	External trigger
…without Scan pause		
<iAcqStart-TrigType>	ME_TRIG_TYPE_SW	ME_TRIG_TYPE_EXT_xx
<iAcqStart-TrigEdge>	ME_TRIG_EDGE_NONE	Edge required
<iScanStart-TrigType>	ME_TRIG_TYPE_FOLLOW	ME_TRIG_TYPE_FOL-LOW
<iScanStart-Ticks>	0	0
<iConvStart-TrigType>	ME_TRIG_TYPE_TIMER	ME_TRIG_TYPE_TIMER
…with Scan Pause		
<iAcqStart-TrigType>*	ME_TRIG_TYPE_SW	ME_TRIG_TYPE_EXT_xx
<iAcqStart-TrigEdge>	ME_TRIG_EDGE_NONE	Edge required
<iScanStart-TrigType>	ME_TRIG_TYPE_TIMER	ME_TRIG_TYPE_TIMER
<iScanStart-Ticks>	Time for channel list processing + pause	
<iConvStart-TrigType>	ME_TRIG_TYPE_TIMER	ME_TRIG_TYPE_TIMER

Table 11: Timing Stream-Timer

Details for Stream-Timer: For reliable acquisition of an external trigger pulse, make sure that the high phase of the external trigger pulse t_{PH} is minimum 1 tick (depends on hardware). See also specifications of your device.

Depending on the device further trigger conditions to start this operation are possible. See also parameter <iAcqStartTrigType> of the function *meIOStreamConfig()*.

3.4.2.4.2 Timing Stream-Trigger-Sample

Event-controlled streaming operation. Start by software or external trigger pulse after calling the function *meIOStreamStart()* - on each trigger pulse one value is sampled.

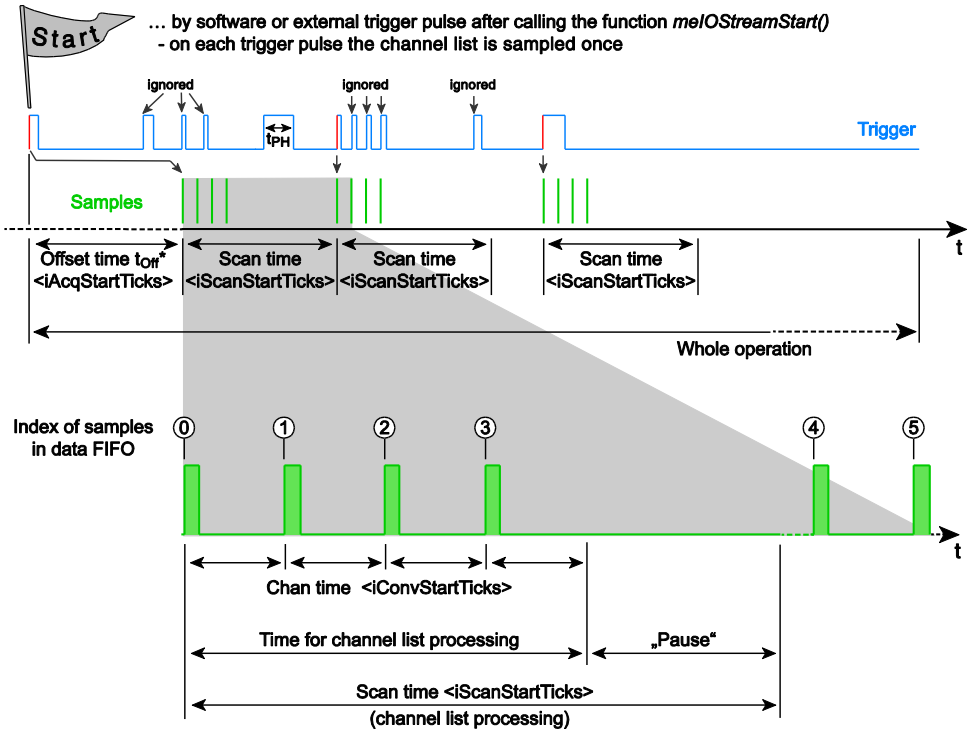

Diagram 19: Timing diagram Stream-Trigger-Sample

* **Note** that the offset time t_{Off} must consider the settling time of the hardware! For standard applications we recommend to equate the offset time with the minimum chan time of the hardware (<`iAcqStartTicks`> = min. <`iConvStartTicks`>).

**Index of the samples in data FIFO. The values are always sampled corresponding to the order of the channels in the channel list. Independent whether channel specific parameters can be set on the hardware or not, a channel list with a standard entry must be written. See function *meIOStreamConfig()*.

Parameter	Software trigger	External trigger
<iAcqStart-TrigType>	ME_TRIG_TYPE_SW	ME_TRIG_TYPE_EXT_xx
<iAcqStart-TrigEdge>	ME_TRIG_EDGE_NONE	Flanke erforderlich
<iScanStart-TrigType>	ME_TRIG_TYPE_SW	<iAcqStartTrigType>
<iScanStart-TrigEdge>	ME_TRIG_EDGE_NONE	<iAcqStartTrigEdge>
<iConvStart-TrigType>	ME_TRIG_TYPE_SW	<iAcqStartTrigType>
<iConvStart-TrigEdge>	ME_TRIG_EDGE_NONE	<iAcqStartTrigEdge>

Table 12: Timing Stream-Trigger-Sample

Details for Stream-Trigger-Sample: For reliable acquisition of an external trigger pulse, make sure that the high phase of the external trigger pulse t_{PH} is minimum 1 tick (depends on hardware). See also specifications of your device.

Depending on the device further trigger conditions to start this operation are possible. See also parameter <iAcqStartTrigType> of the function *meIOStreamConfig().*

In this operation mode an adjustable chan time of some devices is not supported. Please refer to the ME-iDS help file and the hardware manual.

3.4.2.4.3 Timing Stream-Trigger-List

Event-controlled streaming operation. Start by software or external trigger pulse after calling the function *meIOStreamStart()* - on each trigger pulse the channel list is sampled once.

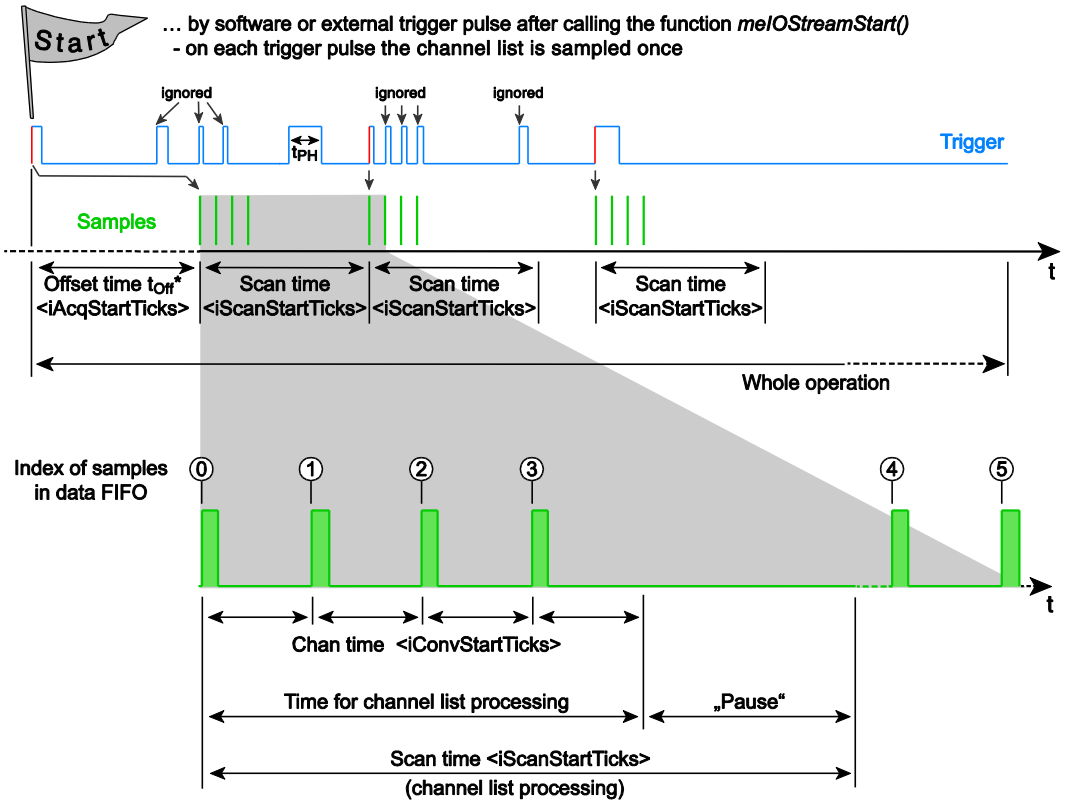

Diagram 20: Timing diagram Stream-Trigger-List

* **Note** that the offset time t_{Off} must consider the settling time of the hardware! For standard applications we recommend to equate the offset time with the minimum chan time of the hardware (<`iAcqStartTicks`> = min.<`iConvStartTicks`>).

** Index of the samples in data FIFO. The values are always sampled corresponding to the order of the channels in the channel list. Independent whether channel specific parameters can be set on the hardware or not, a channel list with a standard entry must be written. See function *meIOStreamConfig()*.

Parameter	Software trigger	External Trigger
<iAcqStart-TrigType>	ME_TRIG_TYPE_SW	ME_TRIG_TYPE_EXT_xx
<iAcqStart-TrigEdge>	ME_TRIG_EDGE_NONE	Flanke erforderlich
<iScanStart-TrigType>	ME_TRIG_TYPE_SW	ME_TRIG_TYPE_EXT_xx
<iScanStart-TrigEdge>	ME_TRIG_EDGE_NONE	<iAcqStartTrigEdge>
<iConvStart-TrigType>	ME_TRIG_TYPE_TIMER	ME_TRIG_TYPE_TIMER
<iConvStart-TrigEdge>	ME_TRIG_EDGE_NONE	ME_TRIG_EDGE_NONE

Table 13: Timing Stream-Trigger-List

Details for Stream-Trigger-List: For reliable acquisition of an external trigger pulse, make sure that the high phase of the external trigger pulse t_{PH} is minimum 1 tick (depends on hardware). See also specifications of your device.

Depending on the device further trigger conditions to start this operation are possible. See also parameter <iAcqStartTrigType> of the function *meIOStreamConfig()*.

In this operation mode the pause between consecutive channel list processing of some devices is not supported. Please refer to the ME-iDS help file and the hardware manual.

3.4.2.5 Reading Data

Reading the data by repeatedly calling the function *meIOStreamRead()* (see page 63). The following diagram applies for analog acquisition. Availability of shown blocks depends on hardware.

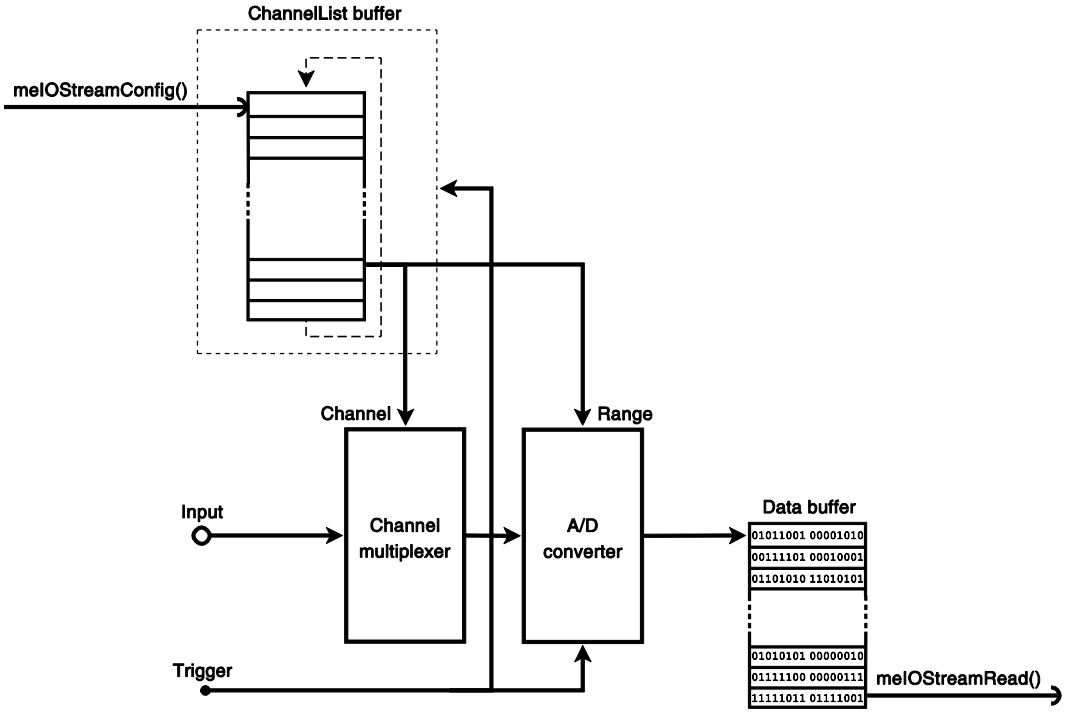

Diagram 21: Reading data

3.4.2.5.1 Procedure Reading Data

Retrieve the data by repeatedly calling the function *meIOStreamRead()* (see page 63).

The diagrams on the next pages explain the program flow under the following conditions:

a. Retrieval of data without callback function (BLOCKING or NONBLOCKING). See diagram 23 on page 71.

b. Retrieval of data with an user-defined callback function in the background. On demand you can install three different callback functions by the function *meIOStreamSetCallbacks().* They can be called at the beginning, during or after the acquisition.See diagram 24 on page 66.

Also note the programming examples within the ME-iDS SDK.

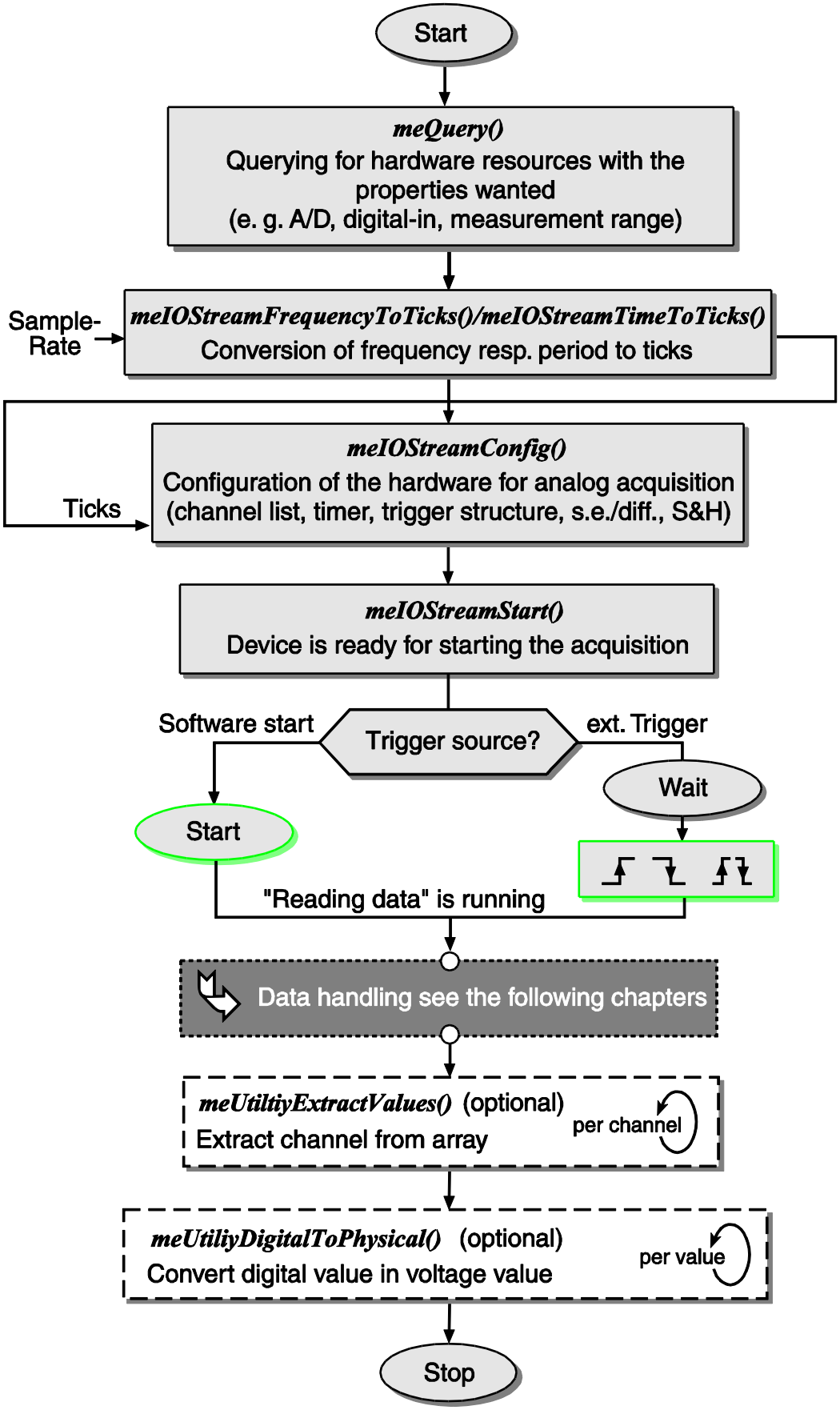

Diagram 22: Procedure programming reading data

3.4.2.5.2 Reading without Callback Function

Reading data by repeatedly calling the function *meIOStreamRead()* (<`iReadMode`>: BLOCKING or NON_BLOCKING):

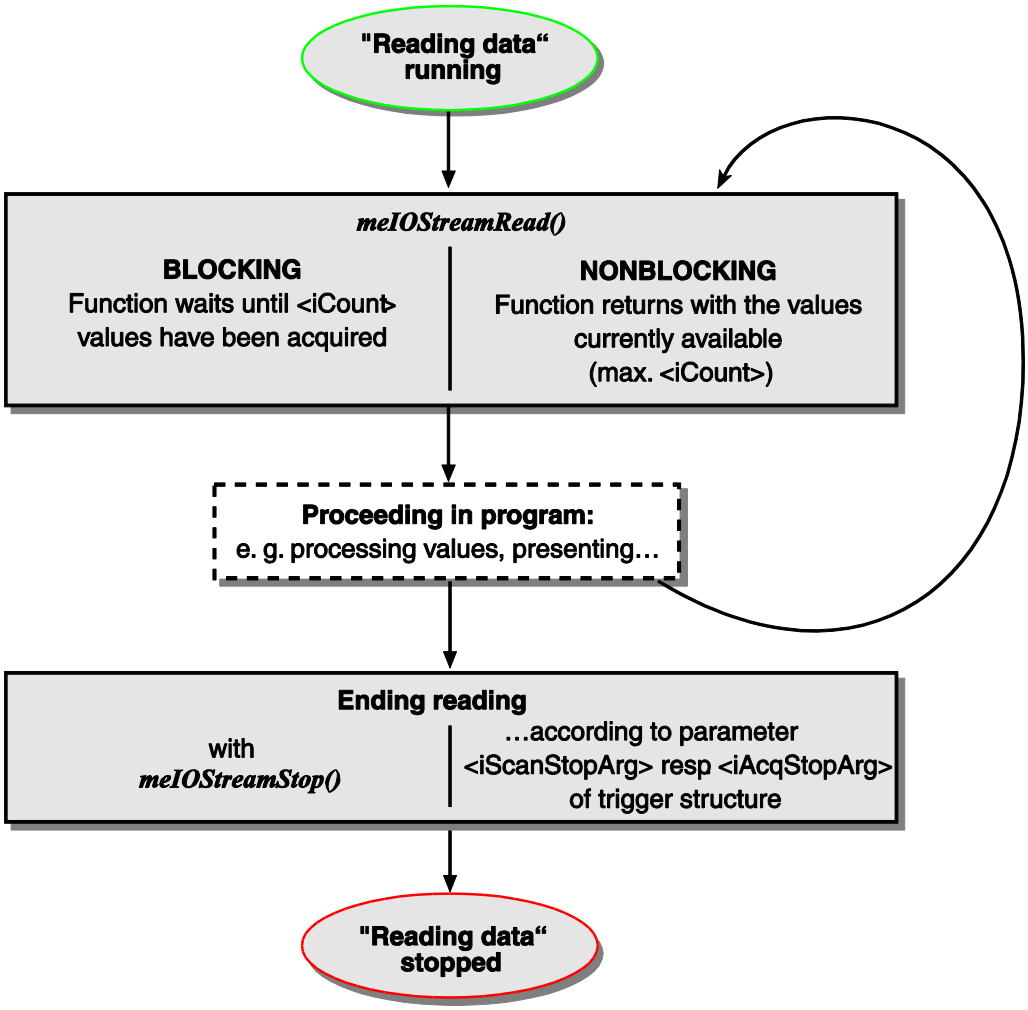

Diagram 23: Reading data without callback function

3.4.2.5.3 Reading with Callback Function

Reading data with a user-defined callback function.

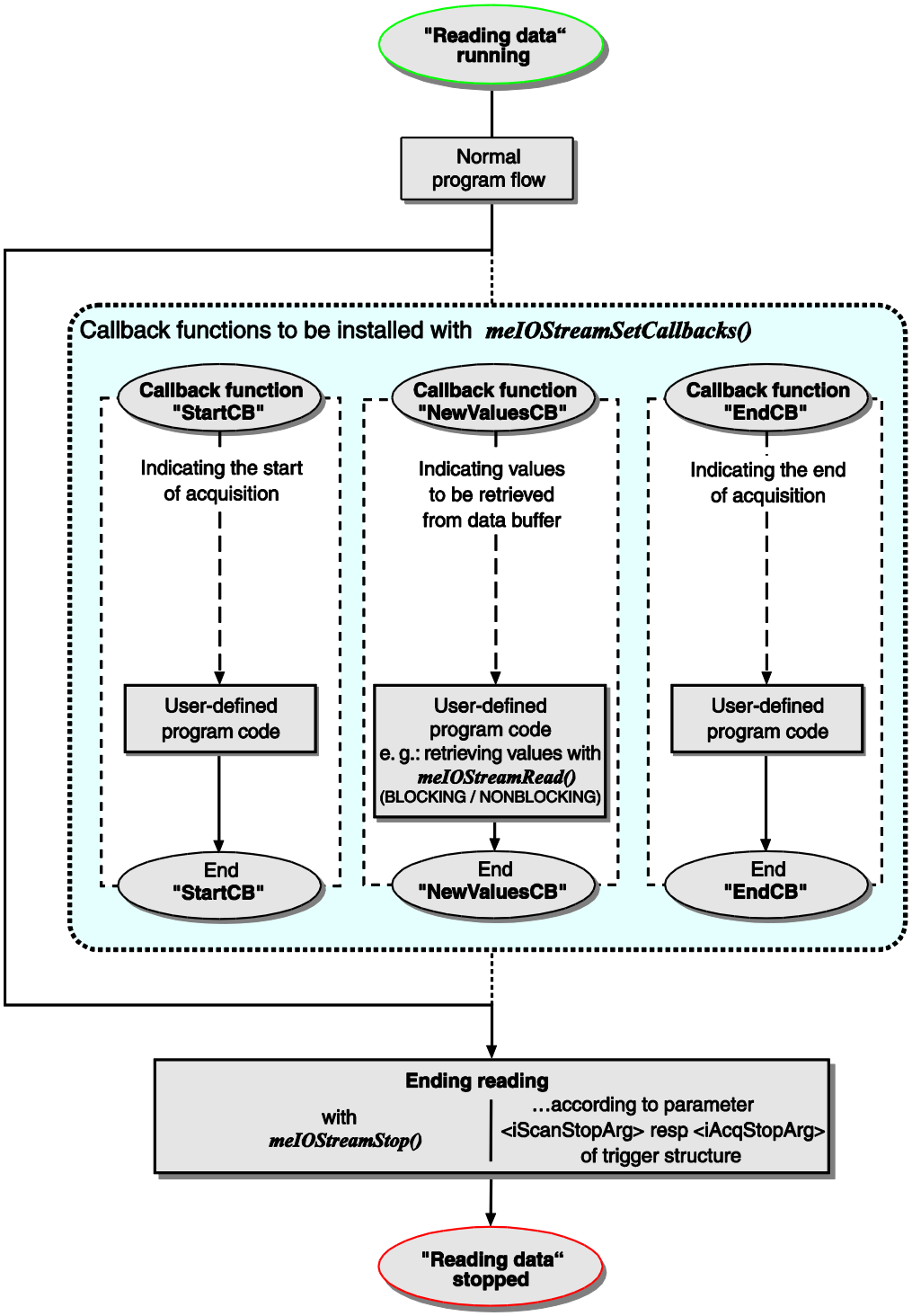

Diagram 24: Reading data with callback function

Returning a value different from 0 by the callback function will stop the streaming operation. You can return the error code of the callback function from parameter <`iErrorCode`>.

3.4.2.6 Writing Data

Writing data by repeatedly calling the function *meIOStreamWrite()* (see page 180). The following diagram applies for analog output. Availability of shown blocks depends on hardware.

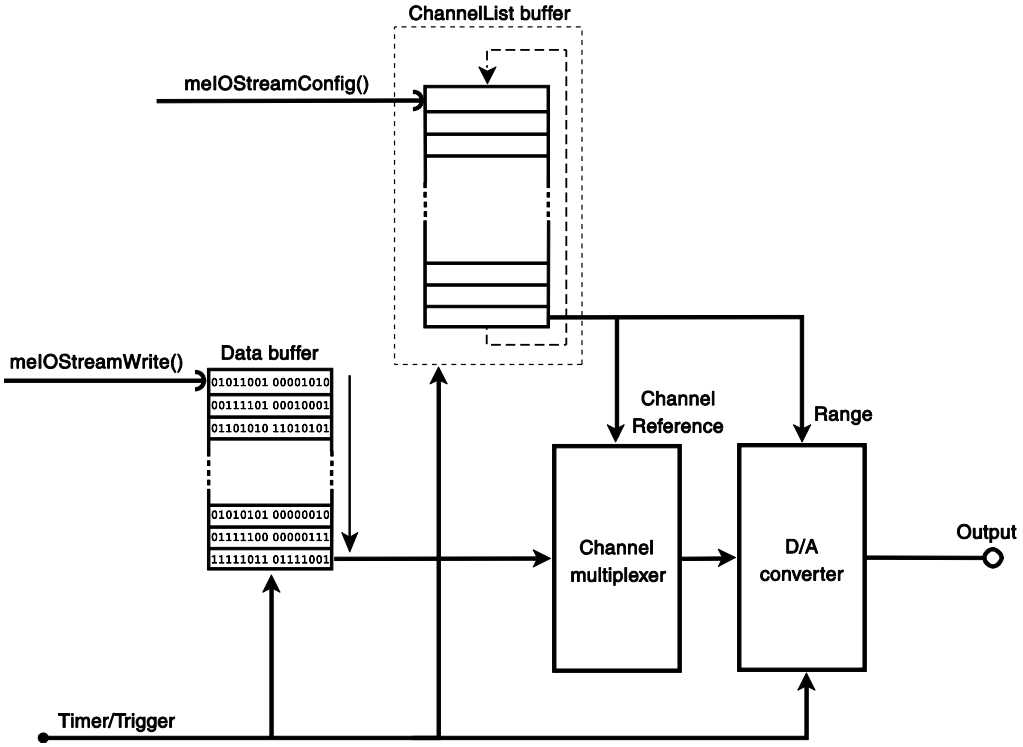

Diagram 25: Writing data

Basically, data are continuously written to a subdevices of subtype ME_SUBTYPE_STREAMING. The application is responsible for reloading the buffer with new values by the function *meIOStreamWrite()*. This offers you the possibility, to change the output values permanently during the operation.

A data buffer of a defined size must be allocated for the values to be output. Before starting the output operation, the first data package must be written to the data buffer by the function *meIOStreamWrite()* using the option ME_IO_WRITE_MODE_PRELOAD in the parameter<`iWriteMode`>.

The chan-timer defines a fixed time grid for output of the values. It must be configured with the function *meIOStreamConfig()* before running the output.

For the wraparound option see chap. 3.4.2.6.4 on page 72).

Note: Regarding the specifics of the timer-controlled bit-pattern output of the ME-4680 please note the detailed description in appendix A3 on page 182.

3.4.2.6.1 Procedure Writing Data

The diagrams on the following pages show the program flow for the following conditions:

a. The output is done without callback function in mode „NONBLOCKING“ only the number of values are reloaded which have enough space in the data buffer. In „BLOCKING“ mode the function waits until all values specified in parameter <`piCount`> could be reloaded. See diagram 27 on page 70.

b. The output is done as a background operation using a user-defined callback function. On demand you can install three different callback functions by the function *meIOStreamSetCallbacks()*. They can be called when starting a read or write operation, when data are available resp. required and when terminating the operation. See diagram 28 on page 71.

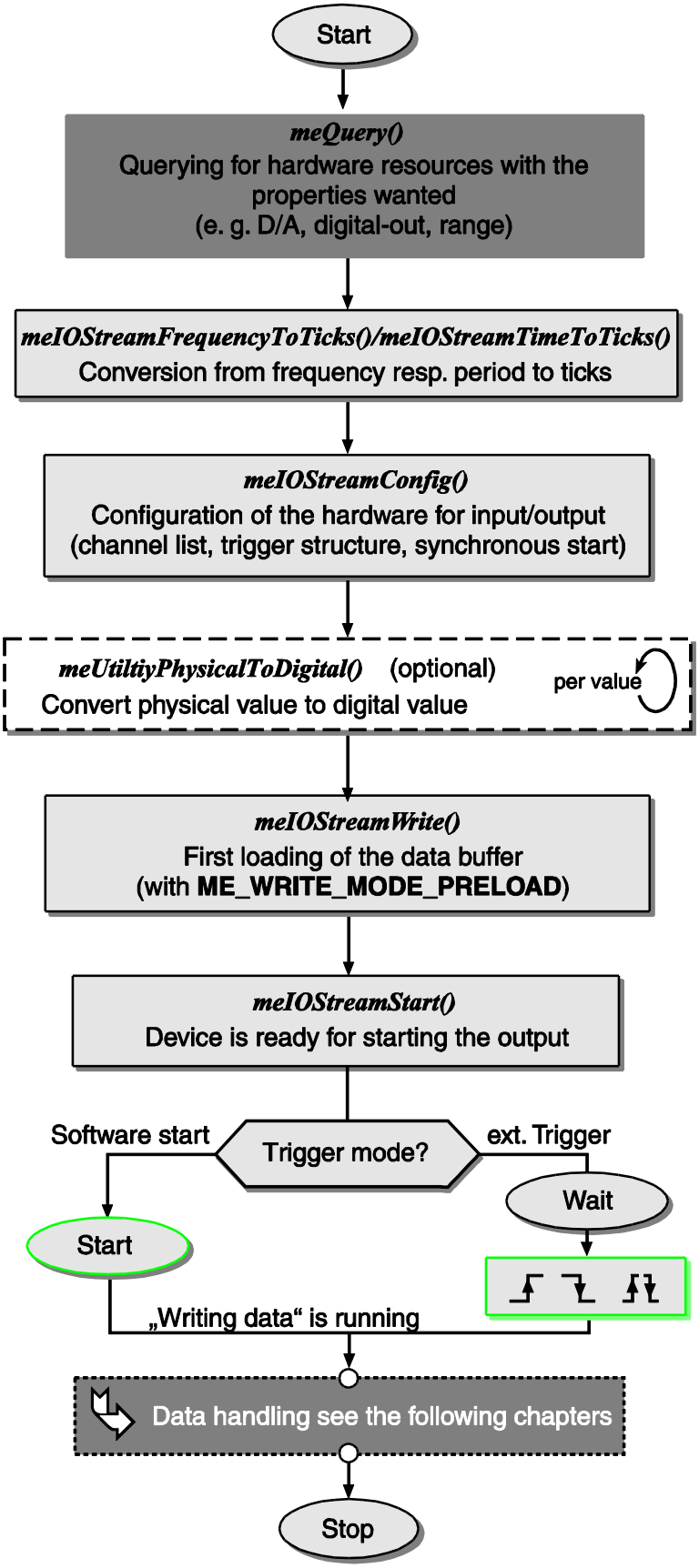

Diagram 26: Procedure programming writing data

3.4.2.6.2 Writing without Callback Function

Writing data by repeatedly calling the function *meIOStreamWrite()* (<`iWriteMode`>: BLOCKING or NON_BLOCKING):

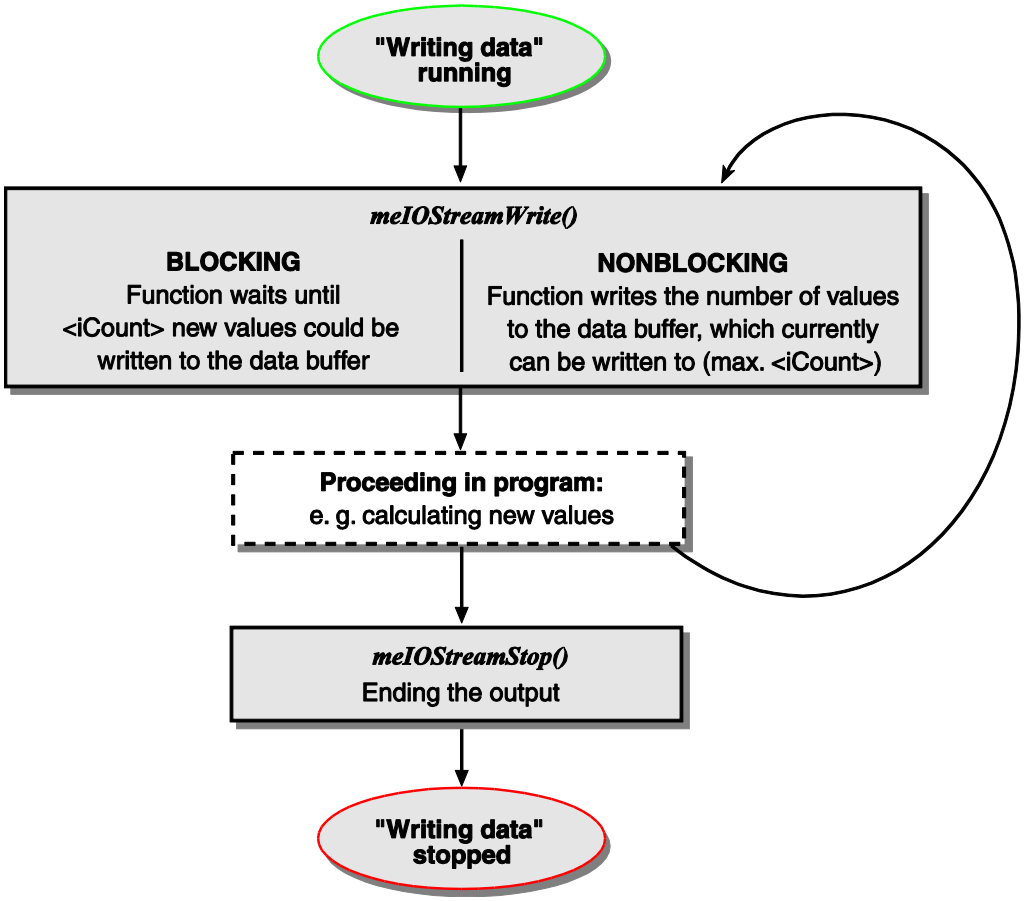

Diagram 27: Writing data without callback function

3.4.2.6.3 Writing with Callback Function

Writing data with a user-defined callback function:

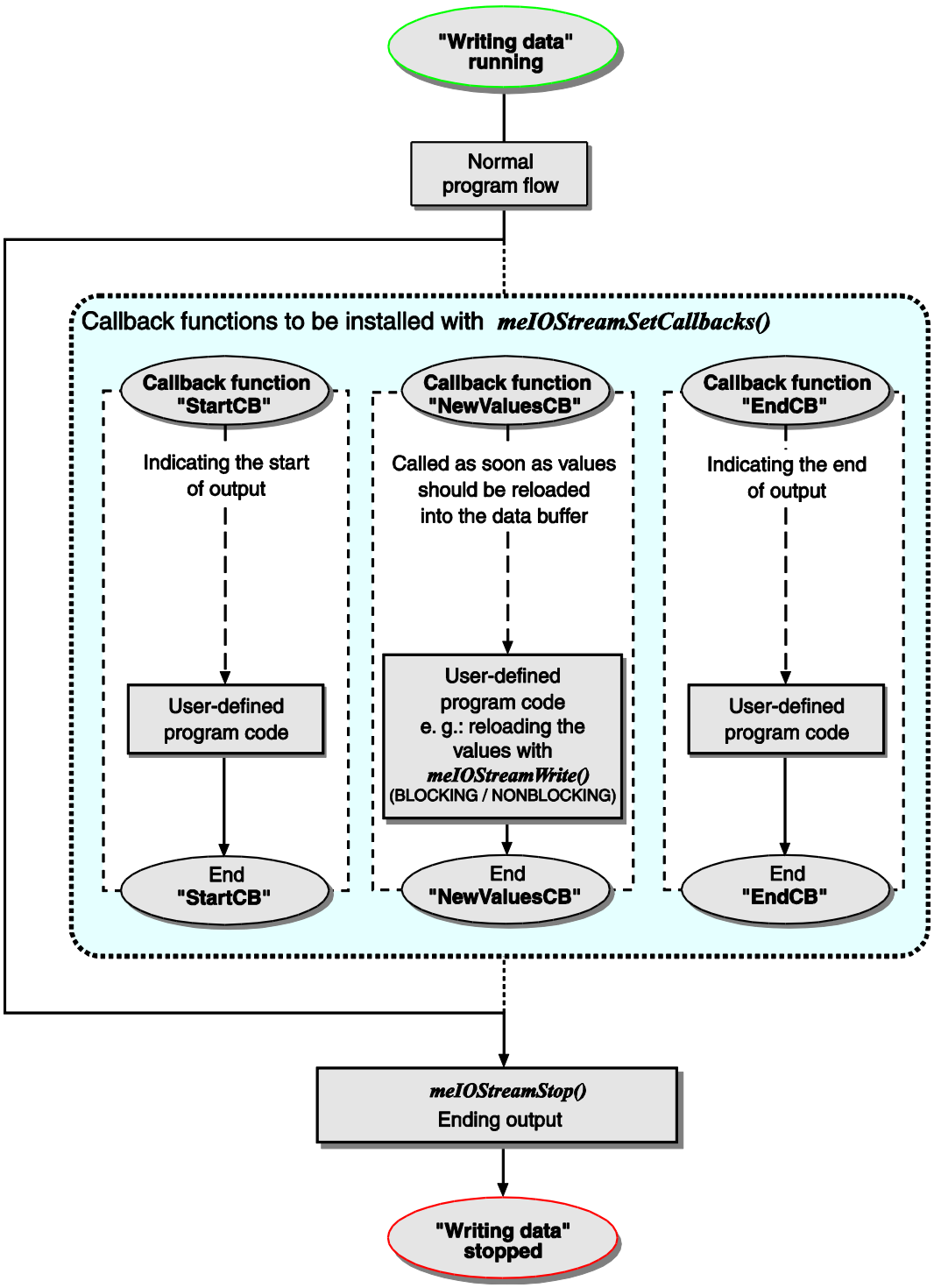

Diagram 28: Writing data with callback function

Returning a value different from 0 by the callback function will stop the streaming operation. You can return the error code of the callback function from parameter <`iErrorCode`>.

3.4.2.6.4 Wraparound Option

By the „wraparound“ option the values are written into the data buffer once and output periodically. There is no possibility to update the buffer after the operation has been started by *meIOStreamStart().*

Depending on various hardware specific parameters (e.g. FIFO size, sampling rate) the operation is running on firmware level without extra load for the host computer.

Before starting the output, the first data package must be written to the data buffer. Use the option ME_IO_WRITE_MODE_PRELOAD in the parameter <`iWriteMode`> of function *meIOStreamWrite().*

3.4.2.7 Stop Streaming Operation

Operation ends if one of the following conditions is true:

1. Cancelled by user with *meIOStreamStop()* or *meIOReset*...()

2. Finished: number of values have been transfered

3. Function *meIOStreamConfig():* The trigger structure *meIOStreamTrigger_t* contains several parameters to stop a streaming operation:

a. Manual stop (in case of infinite operation)

- <`iScanStopTrigType`> = ME_TRIG_TYPE_NONE
- <`iScanStopCount`> = 0
- <`iAcqStopTrigType`> = ME_TRIG_TYPE_NONE
- <`iAcqStopCount`> = 0

b. Stop after a defined number of conversions

- <`iScanStopTrigType`> = ME_TRIG_TYPE_ COUNT
- <`iAcqStopTrigType`> = ME_TRIG_TYPE_ FOLLOW
- <`iAcqStopCount`> = 0

c. Stop after defined number of channel-list processing

- <`iScanStopTrigType`> = ME_TRIG_TYPE_NONE
- <`iScanStopCount`> = 0
- <`iAcqStopTrigType`> = ME_TRIG_TYPE_COUNT

4. Error occured: not enough space in buffer or no values in buffer.

3.4.3 Extra Features

3.4.3.1 Sample and Hold

The “Sample & Hold”-option is used when multiple channels need to be measured at the same time with multiplexed subdevices. The „sample & hold“-channels are „frozen“ simultaneously by the hardware, using a common triggering signal (software or external trigger). Next the values can be read sequentially.

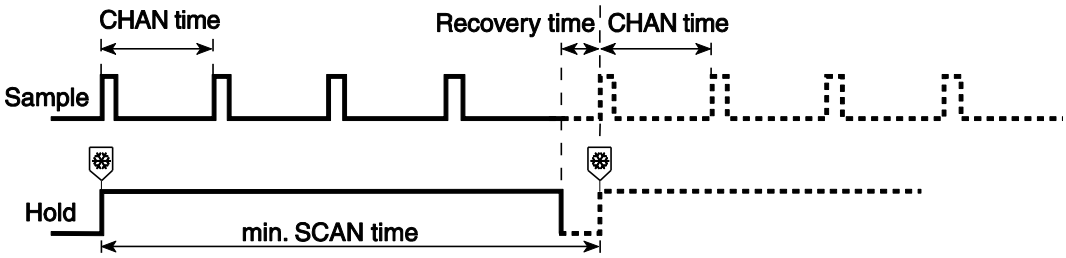

Diagram 29: Sample & Hold

NOTE: “S&H”-operation needs a certain recovery time, therefore executing the channel-list directly after the previous one is not possible. It is only available for analog input channels.

If the „S&H“-option is supported by the hardware the function *meQuerySubdeviceCaps()* returns ME_CAPS_AI_SAMPLE_HOLD.

For more details and limitations check the ME-iDS help file and the appropriate hardware manual.

3.4.3.2 Bit-Pattern Output of ME-4680

A special function is the redirection of a D/A-FIFO from a D/A converter (DAC) to digital ports, as required for the “timer-controlled bit-pattern output” of the ME-4680.

Some of the data FIFOs for analog output can be redirected to other subdevices (with digital output ports) and output a digital data stream.

The programming is done in two steps:

• Disconnecting the DAC: This is programmed in an analog output subdevice of sub-type ME_SUBTYPE_STREAMING. The DAC will be disconnected if the constant ME_IO_STREAM_CONFIG_

BIT_PATTERN will be used in parameter <`iFlags`> of the function *meIOStreamConfig().*

NOTE: After disconnecting the DAC from FIFO the analog output value is preserved.

- Redirecting the FIFO output: This is programmed in a digital port subdevice of type ME_TYPE_DIO or ME_TYPE_DO. Assumed the analog

output FIFO is 16 bits wide the data word can be treated as separate low byte and high byte. The values can be assigned to the digital ports via the parameter <`iRef`> (ME_REF_FIFO_LOW for bit 7…0 and ME_REF_FIFO_HIGH for bit 15…8 in the function *meIOSingleConfig()*. In parameter <`iSingleConfig`> ME_SINGLE_CONFIG_DIO_BIT_ PATTERN must be passed.

See also diagram 33 in appendix A3 on page 182.

NOTES:

- Ports used for „bit-pattern output“ have to be configured as output.
- There is the possibility to assign the same data source to several digital ports.
- After programming a redirection normal access to these digital port (read/write via *meIOSingle())* is not possible any more.
- Programming a redirection has instant effect.
- If only redirection is programmed (without disconnecting the DAC) the analog output port works normally with corresponding value to those on the digital outputs.

3.4.3.3 Synchronous Start

Some of the Meilhaus boards (example: ME-6000) offer the possibility to synchronize the operation of different subdevices. Therefore a so-called „sync-list“ must be generated.

Notes:

- There can be more than one sync-list for a single device.
- One sync-list can contain subdevices of different types.

See ME-iDS help file and hardware manual for details.

By default every subdevice is running completely independent. In that case pass the constant ME_TRIG_CHAN_DEFAULT. In order to add a subdevice to the sync-list this parameters have to be set to ME_TRIG_CHAN_SYNCHRONOUS.

… for **single operation:**

in parameter <`iTrigChan`> of function *meIOSingleConfig().*

Example: Starting the pulse output of several subdevices synchronously:

- Configure each subdevice in the function *meIOSingleConfig()* with the flag ME_TRIG_CHAN_SYNCHRONOUS.
- For each subdevice write the value for FIO_TICKS_TOTAL und FIO_TICKS_FIRST_ PHASE without „sync“ flag set with exeption of the last one which must be passed

with the flag ME_IO_SINGLE_TYPE_TRIG_ SYNCHRONOUS. The last write command with the flag set starts the output on all subdevices, which have been configured with ME_TRIG_CHAN_ SYNCHRONOUS immediately.

… for **streaming operation:**

in parameter <`iAcqStartTrigChan`> in the trigger structure of function *meIOStreamConfig().*

Choose one of the following trigger sources for the sync-list:

- **External trigger:** Active external trigger on any of the subdevices included in the sync-list.
  Software trigger: *meIOSingle()* or *meIOStreamStart()* with the flag _TRIG_SYNCHRONOUS set is called software trigger. Sub-devices that generate a software trigger have not to be included in the sync-list.
    - Function *meIOSingle(),* parameter <`iFlags`>: ME_IO_SINGLE_TYPE_TRIG_SYNCHRONOUS
    - Function *meIOStreamStart()* start entry pStartList[x].iFlags: ME_IO_STREAM_START_TYPE_TRIG_SYNCHRONOUS

IMPORTANT: An external trigger can start a sync-list only if the sub-device is included. Software trigger starts the list even if the subdevice in not part of the sync-list.

3.4.3.4 Offset Setting

If an input channel (e.g. on MEphisto-Scope) provides the feature of adjusting the measurement range by an offset, you have to call the function *meIOSetChannelOffset()* prior of starting the operation. Please note that this feature is only possible with streaming operation. After a reset with the function *meIOResetSubdevice()* the offset for both channels and for all ranges is set to 0.

See also the description of the function *meIOSetChannelOffset()* in the chapter function reference from page 78.

3.4.4 Interrupt Operation

For subdevices of type „external interrupt“ (ME_TYPE_EXT_IRQ) no single operations are possible. On demand you can enable the interrupt operation with the function *meIOIrqStart().* Depending on the hardware you can choose between different interrupt sources via parameter <`iIrqSource`>:

- ME_IRQ_SOURCE_DIO_LINE
  Interrupt source is a dedicated external interrupt input.
- ME_IRQ_SOURCE_DIO_PATTERN
  Operation mode „Bit-Pattern Match“ (e.g.: ME-5810/8100/8200): On bit-pattern match an interrupt is triggered.
- ME_IRQ_SOURCE_DIO_MASK
  Operation mode „Bit-Pattern Change“ (e.g.: ME-5100/5810,
- ME-8100/8200): On change of at least one bit, masked as „sensitive“ an interrupt is triggered.
- ME_IRQ_SOURCE_DIO_OVER_TEMP
  On overheating of a driver chip an interrupt is triggered (e.g.: ME-5810/8200).
- ME_IRQ_SOURCE_DIO_NORMAL_TEMP
  An interrupt is triggered as soon as an overheated driver chip is cooling down to normal temperature.
- ME_IRQ_SOURCE_DIO_CHANGE_TEMP
  An interrupt is triggered on overheating as well as on cooling down to normal temperature.

Use the function *meIOIrqWait()* to analyze the different interrupt sources (depending on hardware) individually. Disable the interrupt operation with the function *meIOIrqStop()*.

See also the description of the *meIOIrq*… functions in the chapter function reference from page 78.

On demand, a user-defined callback function can be called. Returning a value different from 0 by the callback function the interrupt operation will be stopped.

See also the diagram 30:

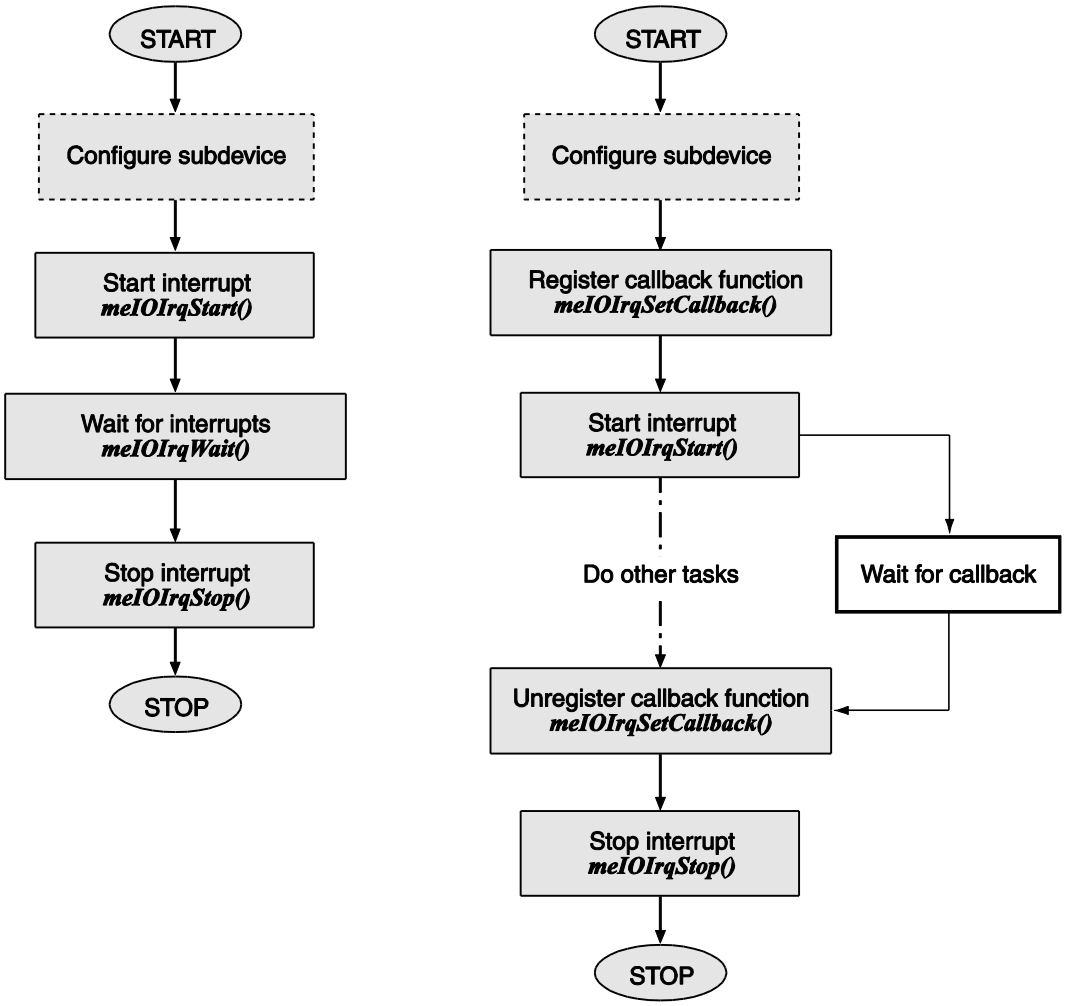

Diagram 30: Interrupt operation without (left) and with callback function (right)

4 Function Reference

4.1 General Notes

- **Function prototypes:**
  In the following description of the functions the generic function prototypes for ANSI C are used. Please refer to the definition resp. header files for the other supported programming languages. The different syntax for pointers like „`int* piName`“ and „`int *piName`“ is equivalent.
- **Execution mode BLOCKING:**
  Note, when using long sample rates or an external trigger which appears later or not it can result in a longer lasting blocking of the task.
- **Callback functions:**
  With Agilent VEE, LabVIEW, older Visual Basic dialects and Python no callback functions can be used.
  Returning a value different from 0 by a callback function a running operation can be stopped.
- **External trigger with time-out:**
  For functions with external trigger you can define a time-out period within the **first** trigger pulse must occur. Else the operation will be cancelled (parameter <`iTimeOut`>). It is not checked if further trigger signals fail to appear e.g. during an acquisition in the operation mode streaming. Note this when programming.
- **Usage of flags:**
  If useful, the constants in the parameters <`iFlags`> can be logically OR-linked.
- **Indexes:**
  Indexes of the parameters <`iDevice`>, <`iSubdevice`>, <`iChannel`> and <`iRange`> start always with 0.

4.2 Description of the API Functions

The functions sorted by togetherness:

Function	Short Description	Page
Query Functions		
meQueryInfoDevice	Querying information like device-ID, serial number, bus type…	82
meQueryNameDeviceDriver	Querying the driver name	84
meQueryNameDevice	Querying the type of the device	82
meQueryDescriptionDevice	Querying a string describing the device	86
meQueryVersionLibrary	Querying the version of the function library	86
meQueryVersionMainDriver	Querying the version of the main driver	87
meQueryVersionDeviceDriver	Querying the version of the device driver	87
meQueryNumberDevices	Querying the number of devices recognized by the ME-iDS	88
meQueryNumberSubdevices	Querying the number of subdevices of a device	88
meQuerySubdeviceType	Querying the type of a subdevice	88
meQueryNumberChannels	Querying the number of channels of a subdevice	89
meQueryNumberRanges	Querying the number of measurement ranges of an analog subdevice	90

meQueryRangeInfo	Querying the limits of the ranges of an analog measurement range	91
meQuerySubdeviceCaps	Querying special capabilities of a subdevice	92
meQuerySubdeviceCapsArgs	Querying details of the special capabilities of a subdevice	95
meQuerySubdeviceByType	Querying the index of a subdevice of the type wanted	96
meQueryRangeByMinMax	Querying the proper analog measurement range by passing the limits of the range	98

Function	Short description	Page
Property Functions		
mePropertyGetInt(A/W)	Determine properties of type integer	100
mePropertyGetDouble(A/W)	Determine properties of type double	101
mePropertyGetString(A/W)	Determine properties of type string	103
mePropertySetInt(A/W)	Set properties of type integer	103
mePropertySetDouble(A/W)	Set properties of type double	104
mePropertySetString(A/W)	Set properties of type string	105
Input/Output Functions		
meIOResetDevice	The whole device is reset	107
meIOResetSubdevice	The subdevice is reset	111
meIOIrqStart	Enabling interrupt operation	107
meIOIrqStop	Disabling interrupt operation	110
meIOIrqWait	Waiting for interrupt event	111
meIOIrqSetCallback	Callback function installing an IRQ	112

meIOSetChannelOffset	Adjusting the offset for analog inputs (available only with streaming operation at the moment)	116
meIOSingleConfig	Configuring a channel for input/output of a single value	118
meIOSingle	Input/output of a single value	118
meIOSingleTicksToTime	Converting ticks into time [s]	126
meIOSingleTimeToTicks	Converting time [s] into ticks	124
meIOStreamConfig	Preparing a continuously running input/output operation	130
meIOStreamTimeToTicks	Converting period [s] into ticks	141
meIOStreamFrequencyToTicks	Converting frequency [Hz] into ticks	143
meIOStreamStart	Starting streaming operation	145
meIOStreamStop	Stopping streaming operation	147
meIOStreamRead	Timer-controlled acquisition	150
meIOStreamWrite	Timer-controlled output	151
meIOStreamStatus	Request on state during streaming	154
meIOStreamNewValues	Checking the status of streaming operation	155
meIOStreamSetCallbacks	Installing callback functions	156

Function	Short description	Page
Auxiliary Functions		
meOpen	Initializing the ME-iDS	158
meClose	Closing the ME-iDS	159
meLockDriver	Locking the driver	160
meLockDevice	Locking a device	160

meLockSubdevice	Locking a subdevice	161
meErrorGetLast	Function returns the last error code	162
meErrorGetLastMessage	Assign error string to the last error	162
meErrorGetMessage	Assign error string to a error number	163
meErrorSetDefaultProc	Install predefined global error routine	164
meErrorSetUserProc	Install user-defined global error routine	164
meUtilityDigitalToPhysical	Converting a standardized digital value into a physical value	165
meUtilityDigitalToPhysicalV	…see meUtilityDigitalToPhysical, however to be used for an array	165
meUtilityPhysicalToDigital	Converting a physical value into a standardized digital value	170
meUtiltiyPhysicalToDigitalV	…see meUtiltiyPhysicalToDigital, however to be used for an array	170
meUtiltiyExtractValues	Extracting the values for one channel from the data buffer	165
meUtilityPWMStart	Starting PWM operation for 8254	173
meUtilityPWMStop	Stopping PWM operation	174
meUtilityPWMRestart	Restart PWM operation without reset	175

4.2.1 Query-Functions

meQueryInfoDevice

Description

Detailed information of the specified device. This is a PCI orientated function and some parameters have no meaning for ME-Synapse USB or ME-Synapse LAN.

Function Declaration:

int meQueryInfoDevice(int iDevice, int *piVendorId, int *piDeviceId, int *pi-SerialNo, int *piBusType, int *piBusNo, int *piDevNo, int*piFuncNo, int *piPlugged);

<iDevice>

Index of the device to be accessed.

<piVendorId> (r)

Pointer returns vendor ID of the device.

- 0x1402 for Meilhaus PCI boards.
- 0x1B04 for USB devices like e.g. ME-1 (ME-Synapse USB).

<piDeviceId> (r)

Pointer returns the device ID (e.g. 0x6034).

<piSerialNo> (r)

Pointer returns the serial number of the device.

<piBusType> (r)

Pointer returns the bus type by which the device is connected with the PC (PCI/cPCI, USB).

- ME_BUS_TYPE_INVALID invalid return value
- ME_BUS_TYPE_PCI PCI/cPCI bus
- ME_BUS_TYPE_USB Universal Serial Bus (USB)

<piBusNo> (r)

- PCI only: Pointer returns the PCI bus number, if several PCI buses are available in your system (if one bus it is always „0“)

<piDevNo> (r)

- PCI only: Slot number of the board to be accessed.

<piFuncNo> (r)

- PCI: Function number.

<piPlugged> (r)

Pointer indicates whether a device is physically available.

- ME_PLUGGED_INVALID invalid return value
- ME_PLUGGED_IN Device physically available.
- ME_PLUGGED_OUT

Device registered with the ME-iDC (ME-Config-Tool) but not con-nected with the PC.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.

meQueryNameDeviceDriver

Description:

Function determines the name of the device specific driver module. Example: “ME-6000” (Windows).

Function Declaration:

int meQueryNameDeviceDriver(int iDevice, char *pcName, int iCount);

<iDevice>

Index of the device to be accessed.

<pcName> (r)

Pointer to a string with the name of the driver module.

<iCount>

Buffer size in bytes for the driver module. Recommended: ME_DEVICE_DRIVER_NAME_MAX_COUNT.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_USER_BUFFER_SIZE: Buffer size of <pcName> too small.

meQueryNameDevice

Description:

Function determines the device code name. *Example:* „ME-6000ISLE/16“.

Function Declaration:

int meQueryNameDevice(int iDevice, char *pcName, int iCount);

<iDevice>

Index of the device to be accessed.

<pcName>

Buffer for the device name.

<iCount>

Buffer size in bytes for device name. Recommended: ME_DEVICE_NAME_MAX_COUNT.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME ERRNO_USER_BUFFER_SIZE: Buffer size of <pcName> too small.

meQueryDescriptionDevice

Description:

Device description. Example: “ME-6000ISLE/4 isle device, 4 analog outputs”.

Function Declaration:

int meQueryDescriptionDevice(int iDevice, char *pcDescription, int iCount);

<iDevice>

Index of the device to be accessed.

<pcDescription> (r)

Buffer for device description.

<iCount>

Buffer size in bytes for device description. Recommended: ME_DEVICE_DESCRIPTION_MAX_COUNT.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_USER_BUFFER_SIZE: Buffer size of <pcDescription> too small.

meQueryVersionLibrary

Description:

Function determines the version number of the library.

Function Declaration:

int meQueryVersionLibrary(int *piVersion);

<piVersion> (r)

Version number of the library (hexadecimal).

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meQueryVersionMainDriver

Description:

Function determines the version number of the main driver.

Function Declaration:

int meQueryVersionMainDriver(int *piVersion);

<piVersion> (r)

Version number of the main driver (hexadecimal). The two higher si-gnificant bytes (main version, sub version) must be the same one as the version number of the device specific driver module (see *meQueryVersionDeviceDriver()*). The lower significant bytes (build number can differ.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meQueryVersionDeviceDriver

Description:

Version number of the device specific driver module.

Function Declaration:

int meQueryVersionDeviceDriver(int iDevice, int *piVersion);

<iDevice>

Index of the device to be accessed.

<piVersion> (r)

Version number of the device specific driver module (hexadecimal). The two higher significant bytes (main version, sub version) must be the same one as the version number of the device specific driver module (see *meQueryVersionMainDriver()*). The lower significant bytes (build number) can differ.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.

meQueryNumberDevices

Description:

Returns the number of devices recognized by the ME-iDS.

Function Declaration:

int meQueryNumberDevices(int *piNumber);

`<piNumber> (r)`

Number of recognized devices.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meQueryNumberSubdevices

Description:

Returns the number of subdevices on a queried device.

Function Declaration:

int meQueryNumberSubdevices(int iDevice, int *piNumber);

`<iDevice>`

Index of the device to be accessed.

`<piNumber> (r)`

Number of subdevices on the device.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.

meQuerySubdeviceType

Description:

Returns type and subtype of the specified subdevice.

Function Declaration:

int meQuerySubdeviceType(int iDevice, int iSubdevice, int *piType, int*pi-Subtype);

`<iDevice>`

Index of the device to be accessed.

`<iSubdevice>`

Index of the subdevice to be queried.

`<piType> (r)`

Returns the subdevice type:

- ME_TYPE_AO Analog output
- ME_TYPE_AI Analog input
- ME_TYPE_DIO Digital input/output (bi-direct.)
- ME_TYPE_DO Digital output
- ME_TYPE_DI Digital input
- ME_TYPE_FIO Frequency input/output
- ME_TYPE_FO Frequency output
- ME_TYPE_FI Frequency input
- ME_TYPE_CTR Counter
- ME_TYPE_EXT_IRQ External interrupt
- ME_TYPE_FPGA FPGA - planned!

`<piSubtype> (r)`

Returns the subtype of the subdevice:

- ME_SUBTYPE_SINGLE
  Subdevice is able to acquire resp. output single values.
- ME_SUBTYPE_STREAMING
  The subdevice can acquire values continuously resp. output a data stream.
- ME_SUBTYPE_CTR_8254
  Subdevice with a counter of type 8254.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.

meQueryNumberChannels

Description:

Function determines the number of channels of a subdevice.

Function Declaration:

int meQueryNumberChannels(int iDevice, int iSubdevice, int *piNumber);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice that is queried.

<piNumber> (r)

Returns the number of channels of the specified subdevice.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.

meQueryNumberRanges

Description:

Returns the number of measurement ranges of a subdevice. Parameter <iUnit> allows a restriction of the query to a specific physical unit.

Function Declaration:

int meQueryNumberRanges(int iDevice, int iSubdevice, int iUnit, int

*piNumber);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice that is queried.

<iUnit>

Measurement ranges with the specified physical unit should be included in the query (see also *meQueryRangeByMinMax()*):

- ME_UNIT_VOLT Query only for voltage ranges
- ME_UNIT_AMPERE Query only for current ranges
- ME_UNIT_ANY Query for all ranges

<piNumber> (r)

Returns the number of ranges supporting the specified unit(s).

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_NOT_SUPPORTED: function is not supported by subdevice.

meQueryRangeInfo

Description:

Function determines details of the range: limits, resolution and physical unit of the specified measurement range.

Function Declaration:

int meQueryRangeInfo(int iDevice, int iSubdevice, int iRange, int *piUnit, double *pdMin, double *pdMax, int *piMaxData);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice that is queried.

<iRange>

Index of measurement range that is queried.

<piUnit> (r)

Pointer, which returns the physical unit of the specified measurement range.

- ME_UNIT_VOLT Voltage range
- ME_UNIT_AMPERE Current range
- ME_UNIT_INVALID Invalid return value

<pdMin> (r)

Returns the lower limit of the requested range. It applies to the physical unit specified in parameter <piUnit>.

<pdMax> (r)

Returns the upper limit of the requested range. It applies to the physical unit specified in parameter <piUnit>.

<piMaxData> (r)

Returns the maximum resolution of the measurement range (e.g. for 16-bit resolution the value 65535 (0xFFFF) is returned).

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_RANGE: on requested subdevice no requested range available.
- ME_ERRNO_NOT_SUPPORTED: function is not supported by subdevice.

meQuerySubdeviceCaps

Description:

Function determines the special capabilities of a subdevice which are returned by the parameter <piCaps> of the function *meQuerySubdeviceCaps().* Further details can be determined by the function *meQuery-SubdeviceCapsArgs().*

Function Declaration:

int meQuerySubdeviceCaps(int iDevice, int iSubdevice, int *piCaps);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice that is queried.

<piCaps> (r)

Pointer to a bit-coded integer value which returns the special capabilities of the specified subdevice. A bit which is set indicates that the subdevice provides the appropriate capability. If several capabilities apply, the values are ORed bit by bit.

Example: a subdevice provides a digital trigger input which triggers alternatively on a rising, falling or any (i.e. rising or falling) edge. The returned value is: 0x000E8000.

Note: You find a table of all capabilities which can be queried here in appendix B1 on page 187.

- ME_CAPS_NONE (0x00000001) Subdevice has no special capabilities.

Applies for subdevices of type ME_TYPE_AI, ME_TYPE_AO, ME_TYPE_DI, ME_TYPE_DO and ME_TYPE_DIO:

Replace xx depending on the subdevice type by AI, AO or DIO (see also table 15 from page 190).

- ME_CAPS_xx_TRIG_DIGITAL (0x00008000) Subdevice provides a digital trigger input.
- ME_CAPS_xx_TRIG_ANALOG (0x00010000) Subdevice provides an analog trigger input.
- ME_CAPS_xx_TRIG_EDGE_RISING (0x00020000) Subdevice can trigger specifically on a rising edge.
- ME_CAPS_xx_TRIG_EDGE_FALLING (0x00040000) Subdevice can trigger specifically on a falling edge.
- ME_CAPS_xx_TRIG_EDGE_ANY (0x00080000) Subdevice can trigger on a rising or falling edge.

Applies for subdevices of type ME_TYPE_AI:

- ME_CAPS_AI_TRIG_SYNCHRONOUS (0x00000001)
  Subdevice can be started synchronously.
- ME_CAPS_AI_FIFO (0x00000002)
  Subdevice provides FIFO for acquired values.
- ME_CAPS_AI_FIFO_THRESHOLD (0x00000004)
  Possibility to set the threshold (number of values) at which the values should be retrieved from the AI-FIFO. Can be set in parameter <`iFIFOIrqThreshold`> of the function *meIOStreamConfig().*
- ME_CAPS_AI_SAMPLE_HOLD (0x00000008)
  Subdevice provides a „Sample & Hold“ unit for simultaneous acquisition.

Applies for subdevices of type ME_TYPE_AO:

- ME_CAPS_AO_TRIG_SYNCHRONOUS (0x00000001)
  Subdevice can be started synchronously.
- ME_CAPS_AO_FIFO (0x00000002)
  Subdevice provides FIFO for values to be output.
- ME_CAPS_AO_FIFO_THRESHOLD (0x00000004)

Possibility to set the threshold (number of values) at which the AO-FIFO should be reloaded. Can be set in parameter <iFIFOIrqThreshold> of the function *meIOStreamConfig().*

Applies for subdevices of type ME_TYPE_DI, ME_TYPE_DO and ME_TYPE_DIO:

- ME_CAPS_DIO_DIR_BIT (0x00000001)
  Direction can be set per bit (1 bit block).
- ME_CAPS_DIO_DIR_BYTE (0x00000002)
  Direction can be set per byte (8-bit block).
- ME_CAPS_DIO_DIR_WORD (0x00000004)
  Direction can be set per word (16-bit block).
- ME_CAPS_DIO_DIR_DWORD (0x00000008)
  Direction can be set per long-word (32-bit block).
- ME_CAPS_DIO_SINK_SOURCE (0x00000010)
  Output driver of the specified subdevice can be switched between sink and source operation.
- ME_CAPS_DIO_BIT_PATTERN_IRQ (0x00000020)
  On bit-pattern match an interrupt can be triggered.
- ME_CAPS_DIO_BIT_MASK_IRQ_EDGE_RISING (0x00000040)
  On a rising edge of at least one of the active bits an interrupt can be triggered.
- ME_CAPS_DIO_BIT_MASK_IRQ_EDGE_FALLING (0x00000080)
  On a falling edge of at least one of the active bits an interrupt can be triggered.
- ME_CAPS_DIO_BIT_MASK_IRQ_EDGE_ANY (0x00000100)
  On a rising or falling edge of at least one of the active bits an interrupt can be triggered.
- ME_CAPS_DIO_OVER_TEMP_IRQ (0x00000200)
  On overheating of the driver chip an interrupt can be triggered (e.g.: ME-5810/8200).

Applies for subdevices of type ME_TYPE_CTR:

- ME_CAPS_CTR_CLK_PREVIOUS (0x00000001)
  Possibility to connect the clock input (CLK) of a counter with the counter output (OUT) of the previous counter.
- ME_CAPS_CTR_CLK_INTERNAL_1 MHZ (0x00000002)
  Possibility to source the counter with an internal clock of 1 MHz.
- ME_CAPS_CTR_CLK_INTERNAL_10 MHZ (0x00000004) Possibility to source the counter with an internal clock of 10 MHz.
- ME_CAPS_CTR_CLK_EXTERNAL (0x00000008)
  Possibility to source the counter with an external clock source.

Applies for subdevices of type ME_TYPE_EXT_IRQ:

- ME_CAPS_EXT_IRQ_EDGE_RISING (0x00000001)

Triggers an interrupt on a rising edge at the IRQ input.
- ME_CAPS_EXT_IRQ_EDGE_FALLING (0x00000002)
  Triggers an interrupt on a falling edge at the IRQ input.
- ME_CAPS_EXT_IRQ_EDGE_ANY (0x00000004)
  Triggers an interrupt on any edge (rising or falling) at the IRQ input.

Applies for subdevices of type ME_TYPE_FO:

- ME_CAPS_FIO_SINK_SOURCE (0x00000010)
  Output driver of the specified subdevice can be switched between sink and source operation.

See also appendix B1 on page 187 for more information.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.

meQuerySubdeviceCapsArgs

Description:

This function determines detailed information of a specific capability of a subdevice. See also function *meQuerySubdeviceCaps().*

Function Declaration:

int meQuerySubdeviceCapsArgs(int iDevice, int iSubdevice, int iCap, int*piArgs, int iCount);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be queried.

<iCap>

Select the capability to be queried (only an extract):

- ME_CAP_AI_FIFO_SIZE (0x001D0000)
  Query the size (number of values) of the AI-FIFO.
- ME_CAP_AO_FIFO_SIZE (0x001F0000)
  Query the size (number of values) of the AO-FIFO.

- ME_CAP_CTR_WIDTH (0x00200000) Query the width of the counter (in bits).

See appendix B2 on page 190 for a complete list of capabilities which can be queried here.

<piArgs> (r)

Pointer to an array of type integer, which returns the queried values.

<iCount>

Number of values in parameter <piArgs>.

As a rule „1“, if the returned values fit into an integer.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER_ passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRN0_INVALID_CAP: Passed code is invalid or not supported on subdevice.
- ME_ERRNO_INVALID CAP_ARG COUNT: Parameter <Count> doesnot fit for queried capability code.

meQuerySubdeviceByType

Description

Function determines the index of the first available subdevice which matches the features searching for. The query always starts with the subdevice index <iStartSubdevice>.

Note: Internally used hardware resources are not reported.

Function Declaration:

int meQuerySubdeviceByType(int iDevice, int iStartSubdevice, int iType, int iSubtype, int *piSubdevice);

<iDevice>

Index of the device to be accessed.

<iStartSubdevice>

Index of the subdevice the query starts with.

`<iType>`

Type of the subdevice searched for:

- ME_TYPE_AI Analog acquisition
- ME_TYPE_AO Analog output
- ME_TYPE_DIO Digital input/output (bi-direct.)
- ME_TYPE_DO Digital output
- ME_TYPE_DI Digital input
- ME_TYPE_FIO Frequency input/output
- ME_TYPE_FO Frequency output
- ME_TYPE_FI Frequency input
- ME_TYPE_CTR Counter
- ME_TYPE_EXT_IRQ External interrupt
- ME_TYPE_FPGA „FPGA“ - planned!
  - only for advanced users

`<iSubtype>`

The capabilities of the subdevice searched for can be specified more detailed if necessary:

- ME_SUBTYPE_ANY Sub-type doesn´t matter.
- ME_SUBTYPE_SINGLE
  Acquisition resp. output of a single value.
- ME_SUBTYPE_STREAMING
  Continuous acquisition resp. output by using special hardware capabilities (e.g. FIFOs).
- ME_SUBTYPE_CTR_8254
  Subdevice with a counter of type 8254.

`<piSubdevice> (r)`

Returns the index of the first matching subdevice.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_NO_MORE_SUBDEVICE_TYPE: Matching subdevice not found.

meQueryRangeByMinMax

Description:

Function determines the appropriate measurement range when the range and the limits of the range are given.

Function Declaration:

int meQueryRangeByMinMax(int iDevice, int iSubdevice, int iUnit, double*pdMin, double *pdMax, int *piMaxData, int *piRange);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice that is queried.

<iUnit>

Restrict the range query to the physical unit chosen here:

- ME_UNIT_VOLT Only voltage ranges
- ME_UNIT_AMPERE Only current ranges
- ME_UNIT_ANY All ranges

<pdMin> (r/w)

(w): Passing the lower limit of the searched range.

(r): Returns the lower limit for the range determined (see parameter.

<piRange>).

<pdMax> (r/w)

(w): Passing the upper limit of the searched range.

(r): Returns the upper limit for the range determined (see parameter

<piRange>).

<piMaxData> (r)

Returns the maximum resolution for the range determined (e.g. 65535 (0xFFFF) for 16-bit resolution).

<piRange> (r)

The index of the best fitting measurement range is returned. Always the smallest range is chosen, which includes the range limits searched for.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.

- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subde-vice mapped to requested ID.
- ME_ERRNO_INVALID_UNIT: unit's type not supported on subde-vice.
- ME_ERRNO_INVALID_MIN_MAX: lower limit is higher than up-per.
- ME_ERRNO_NO_RANGE: fitted range not found.

4.2.2 Property Functions

With the so-called property functions you have the possibility to read and if applicable to write all general and hardware specific properties and attributes. The entity of all properties is like a tree structure. By the so-called property path you can access to a device, a subdevice, a channel or a range and so on.

All property functions are implemented as ANSI and Unicode (UTF-16) versions. The ANSI versions have suffix 'A' and use NULL terminated ANSI strings (char*). The unicode versions with the suffix 'W' use wide character strings (wchar_t*).

Depending on the value to be passed (integer, double, string) the appropriate function must be used to read resp. write the value (see also parameter <`piValue`> of the function *mePropertyGetIntA()* on page 115).

Note: The property functions are available with ME-iDS 2.0 and higher and completely implemented for the ME-5000 series under Windows. For all other devices only the general properties are supported at the moment. In future releases of the ME-iDS this will be extended and completed more and more.

mePropertyGetIntA

mePropertyGetIntW

Description:

By this function you can query properties of type ME_PROPERTY_TYPE_INT via the property path. You can use this function also to query for an unknown type of property (PropertyType).

For more details concerning the properties supported by your hardware please refer to the ME-iDS help file (*.chm).

Note: This function is available with ME-iDS 2.0 and higher.

Function Declaration:

int mePropertyGetIntA(char* pcPropertyPath, int* piValue);

int mePropertyGetIntW(wchar_t* pcPropertyPath, int* piValue);

`<pcPropertyPath> (w)`

Pointer to the property path to be accessed.

`<piValue> (r)`

Pointer to buffer which returns the property as integer value.

Beside the normal return values the following constants are returned if quering for the property PropertyType:

- ME_PROPERTY_TYPE_CONTAINER: contains further properties.
- ME_PROPERTY_TYPE_BOOL: Queried property is of type boolean: 0 (FALSE) or „1“ (TRUE) for off/on resp. inactive/active.
- ME_PROPERTY_TYPE_INT: Queried property is of type integer.
- ME_PROPERTY_TYPE_DOUBLE: Queried property is of type double.
- ME_PROPERTY_TYPE_STRING: Queried property is of type string.
- ME_PROPERTY_TYPE_DEFINE: Queried property returns a predefined constant as an integer value. See also function *meProperty-GetString* on page 119.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_PROPERTY_CONTAINER: A value from a container should be read.
- ME_ERRNO_PROPERTY_DATA_TYPE: The value in the given property path cannot be read as an integer.
- ME_ERRNO_PROPERTY_PATH: The given property path is not valid.

- ME_ERRNO_PROPERTY_INDEX: The given index in property path is outside the valid range.
- ME_ERRNO_PROPERTY_UNKNOWN: The given property is unknown.
- ME_ERRNO_PROPERTY_ATTRIBUTE_UNKNOWN: The given attribute is unknown.
- ME_ERRNO_PROPERTY_ATTRIBUTE_UNSUPPORTED: The given attribute is not supported.

mePropertyGetDoubleA

mePropertyGetDoubleW

Description:

By this function you can query properties of type ME_PROPERTY_TYPE_DOUBLE via the property path. Use the function *mePropertyGetInt()* first, if you want to determine the unknown type of a property (PropertyType).

For more details concerning the properties supported by your hardware please refer to the ME-iDS help file (*.chm).

Note: This function is available with ME-iDS 2.0 and higher.

Function Declaration:

int mePropertyGetDoubleA(char* pcPropertyPath, double* pdValue); int mePropertyGetDoubleW(wchar_t*pcPropertyPath, double* pdValue);

`<pcPropertyPath> (w)`

Pointer to the property path to be accessed.

`<pdValue> (r)`

Pointer to buffer which returns the property as double value.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_PROPERTY_CONTAINER: A value from a container should be read.
- ME_ERRNO_PROPERTY_DATA_TYPE: The value in the given property path cannot be read as an integer.
- ME_ERRNO_PROPERTY_PATH: The given property path is not valid.
- ME_ERRNO_PROPERTY_INDEX: The given index in property path is outside the valid range.
- ME_ERRNO_PROPERTY_UNKNOWN: The given property is unknown.

- ME_ERRNO_PROPERTY_ATTRIBUTE_UNKNOWN: The given attribute is unknown.
- ME_ERRNO_PROPERTY_ATTRIBUTE_UNSUPPORTED: The given attribute is not supported.

mePropertyGetStringA

mePropertyGetStringW

Description:

By this function you can query properties of type ME_PROPERTY_TYPE_STRING via the property path. Use the function *mePropertyGetInt()* first, if you want to determine the unknown type of a property (PropertyType).

For more details concerning the properties supported by your hardware please refer to the ME-iDS help file (*.chm).

Note: This function is available with ME-iDS 2.0 and higher.

Function Declaration:

int mePropertyGetStringA(char* pcPropertyPath, char* pcValue, int iBufferLength); int mePropertyGetStringW(wchar_t* pcPropertyPath, wchar_t* pcValue, int iBufferLength);

`<pcPropertyPath> (w)`

Pointer to the property path to be accessed.

`<pcValue> (r)`

Pointer to a buffer, which contains the value as a NULL terminated string, after successful calling the function.

`<iBufferLength>`

Buffer length in number of characters (not bytes) including terminating NULL character. The length required can be queried by the attribute Length.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_PROPERTY_CONTAINER: A value from a container should be read.
- ME_ERRNO_PROPERTY_DATA_TYPE: The value in the given property path cannot be read as an integer.
- ME_ERRNO_PROPERTY_PATH: The given property path is not valid.

- ME_ERRNO_PROPERTY_INDEX: The given index in property path is outside the valid range.
- ME_ERRNO_PROPERTY_BUFFER_TOO_SMALL: The size of the given buffer is too small for the string.
- ME_ERRNO_PROPERTY_UNKNOWN: The given property is unknown.
- ME_ERRNO_PROPERTY_ATTRIBUTE_UNKNOWN: The given attribute is unknown.
- ME_ERRNO_PROPERTY_ATTRIBUTE_UNSUPPORTED: The given attribute is not supported.

mePropertySetIntA

mePropertySetIntW

Description:

By this function you can set properties of type ME_PROPERTY_TYPE_INT via the property path. Use the *function mePropertyGetInt()* first, if you want to determine the unknown type of a property (PropertyType).

For more details concerning the properties supported by your hardware please refer to the ME-iDS help file (*.chm).

Note: This function is available with ME-iDS 2.0 and higher.

Function Declaration:

int mePropertySetIntA(char* pcPropertyPath, int iValue); int mePropertySetIntW(wchar_t* pcPropertyPath, int iValue);

`<pcPropertyPath> (w)`

Pointer to the property path to be accessed.

`<iValue>`

The property is passed as an integer value.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_PROPERTY_CONTAINER: A value from a container should be read.
- ME_ERRNO_PROPERTY_DATA_TYPE: The value in the given property path cannot be read as an integer.
- ME_ERRNO_PROPERTY_PATH: The given property path is not valid.

- ME_ERRNO_PROPERTY_INDEX: The given index in property path is outside the valid range.
- ME_ERRNO_PROPERTY_UNKNOWN: The given property is unknown.
- ME_ERRNO_PROPERTY_SELECTION_INVALID: The given value is not part of a valid define.
- ME_ERRNO_PROPERTY_VALUE_INVALID: The value is outside of the valid range. You can query the valid minimum and maximum values by the attributes **MinValue** and **MaxValue**.
- ME_ERRNO_PROPERTY_READ_ONLY: The given property is read only.

mePropertySetDoubleA

mePropertySetDoubleW

Description:

By this function you can set properties of type ME_PROPERTY_TYPE_ DOUBLE via the property path. Use the function *mePropertyGetInt()* first, if you want to determine the unknown type of a property (PropertyType).

For more details concerning the properties supported by your hardware please refer to the ME-iDS help file (*.chm).

Note: This function is available with ME-iDS 2.0 and higher.

Function Declaration:

int mePropertySetDoubleA(char* pcPropertyPath, double dValue); int mePropertySetDoubleW(wchar_t* pcPropertyPath, double dValue);

`<pcPropertyPath> (w)`

Pointer to the property path to be accessed.

`<dValue>`

The property is passed as a double value.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_PROPERTY_CONTAINER: A value from a container should be read.
- ME_ERRNO_PROPERTY_DATA_TYPE: The value in the given property path cannot be read as an integer.
- ME_ERRNO_PROPERTY_PATH: The given property path is not valid.

- ME_ERRNO_PROPERTY_INDEX: The given index in property path is outside the valid range.
- ME_ERRNO_PROPERTY_UNKNOWN: The given property is unknown.
- ME_ERRNO_PROPERTY_VALUE_INVALID: The value is outside of the valid range. You can query the valid minimum and maximum values by the attributes **MinValue** and **MaxValue.**
- ME_ERRNO_PROPERTY_READ_ONLY: The given properties read only.

mePropertySetStringA

mePropertySetStringW

Description:

By this function you can set properties of type ME_PROPERTY_TYPE_ STRING via the property path. Use the function *mePropertyGetInt()* first, if you want to determine the unknown type of a property (PropertyType).

For more details concerning the properties supported by your hardware please refer to the ME-iDS help file (*.chm).

Note: This function is available with ME-iDS 2.0 and higher.

Function Declaration:

int mePropertySetStringA(char* pcPropertyPath, char* pcValue); int mePropertySetStringW(wchar_t* pcPropertyPath, wchar_t* pcValue);

`<pcPropertyPath> (w)`

Pointer to the property path to be accessed.

`<pcValue> (w)`

A pointer to the property as a NULL terminated string must be passed.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_PROPERTY_CONTAINER: A value from a container should be read.
- ME_ERRNO_PROPERTY_DATA_TYPE: The value in the given property path cannot be read as an integer.
- ME_ERRNO_PROPERTY_PATH: The given property path is not valid.
- ME_ERRNO_PROPERTY_INDEX: The given index in property path is outside the valid range.

- ME_ERRNO_PROPERTY_UNKNOWN: The given property is unknown.
- ME_ERRNO_PROPERTY_VALUE_INVALID: The string is too long. By the attribute MaxLength the required buffer length in characters (not bytes) including the terminating NULL character can be queried.
- ME_ERRNO_PROPERTY_READ_ONLY: The given properties read only.

4.2.3 Input/Output Functions

meIOResetDevice

Description:

The device will be reset. All currently running operations of the specific device are cancelled:

- All hardware actions are stopped.
- Hardware is set to default (idle) state.
- Internal states are cleared.
- Buffers are flushed (emptied).
- Interrupt counters are set to zero.

Function Declaration:

int meIOResetDevice(int iDevice, int iFlags);

<iDevice>

Index of the device to be reset.

<iFlags>

- ME_IO_RESET_DEVICE_NO_FLAGS

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_LOCKED: device or some of subdevices are protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOIrqStart

Description:

This function starts the interrupt handler for the interrupt subdevice wanted. You can choose interrupt source, trigger edge, reference bit pattern, etc.

On demand you can install an user-defined callback function by the function *meIOIrqSetCallback()* which is called on each interrupt.

Function Declaration:

int meIOIrqStart(int iDevice, int iSubdevice, int iChannel, int iIrqSource, int iIrqEdge, int iIrqArg, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iChannel>

Index of the interrupt channel within the selected subdevice, otherwise it should to be set to „0“.

<iIrqSource>

Selection of interrupt source:

- ME_IRQ_SOURCE_DIO_LINE
  Interrupt source is a dedicated external interrupt input. <iIrqArg> is not in use and has to be set to „0“. <iIrqEdge> is in use.
- ME_IRQ_SOURCE_DIO_PATTERN (only for digital input ports) Operation mode „Bit-Pattern Match“ (e.g.: ME-5810, ME-8100/8200): When the current bit pattern at the digital port matches the reference bit pattern to be passed in <iIrqArg> an interrupt is triggered. <iIrqEdge> is not in use and has to be set to ME_IRQ_EDGE_NOT_USED.
- ME_IRQ_SOURCE_DIO_MASK (for digital input ports only) Operation mode „Bit-Pattern Change“ (e.g.: ME-5100/5810, ME-8100/8200):
- On change of at least one bit, masked as „sensitive“ an interrupt is triggered. The reference bit pattern is passed by the <iIrqArg> parameter. <iIrqEdge> is in use.
- ME_IRQ_SOURCE_DIO_OVER_TEMP
  On overheating of the driver chip an interrupt is triggered (e.g.: ME-5810/8200). <iIrqEdge> is not in use and has to be set to ME_IRQ_EDGE_NOT_USED. <iIrqArg> is not in use and has to be set to “0”.

<iIrqEdge>

Selection of the edge on which an interrupt should be triggered.

- ME_IRQ_EDGE_NOT_USED: Choosing an edge is not supported.
- ME_IRQ_EDGE_RISING: Interrupt on rising edge.
- ME_IRQ_EDGE_FALLING: Interrupt on falling edge.
- ME_IRQ_EDGE_ANY: Interrupt on rising or falling edge.

<iIrqArg>

Argument to configure the modes „Bit-Pattern Match“ (ME_IRQ_ SOURCE_DIO_PATTERN) and „Bit-Pattern Change“ (ME_IRQ_ SOURCE_DIO_MASK). In other cases pass the value „0“ here.

It applies to the boards of type ME-8100/8200 for example: Depending on the interrupt source in parameter <`iIrqSource`> a reference bit pattern is written into the appropriate register. The width of the bit pattern is determined by the parameter <`iFlags`>.

- Interrupt on bit-pattern match“:
  Argument (reference bit pattern) is written into the comparison register. If the current bit pattern at the digital port matches the reference bit pattern an interrupt is triggered.
- “Interrupt on bit-pattern change“:
  Argument (reference bit pattern) is written into the mask register. When the state of at least one bit set to „1“ in the mask register toggles (0 → 1 or 1 → 0), an interrupt occurs. The bit pattern of the digitalport which triggered the interrupt can be checked by the parameter <`piValue`> of the function *meIOIrqWait()* („BLOCKING“ mode) or with the callback function *meIOIrqSetCallback().*

<iFlags>

- ME_IO_IRQ_START_NO_FLAGS
  No flags in use. Default settings will be used.
- ME_IO_IRQ_START_DIO_BIT
  The reference bit pattern is one bit wide.
- ME_IO_IRQ_START_DIO_BYTE
  The reference bit pattern is one byte wide (8 bit).
- ME_IO_IRQ_START_DIO_WORD
  The reference bit pattern is one word wide (16 bit).
- ME_IO_IRQ_START_DIO_DWORD
  The reference bit pattern is one long-word wide (32 bit).
- ME_IO_IRQ_START_PATTERN_FILTERING (for „Bit-Pattern Match“ only): Enables filtering of the result.
- Possibly the „-Match“ mode can generate many false interrupts. In theory changing of more than one bit on a multi-bit port can happen at the same moment. In practice there is always a (smaller or bigger) shift between toggling the bits. Because pattern match works asynchronously and very fast some interstates are cached. *Example:* when two bits are changing from 00b to 11b states 01b and 10b can be detected by comparator. If the filter is enabled in interrupt handling routine current port state is compared with requested value.
- ME_IO_IRQ_START_EXTENDED_STATUS (for „Bit-Pattern Change“ only): Set extended IRQ status format as default. See also function *meIOIrqWait().*

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_INVALID_IRQ_SOURCE: wrong mode or not suppor-ted by subdevice.
- ME_ERRNO_INVALID_IRQ_EDGE: set edge not supported.
- ME_ERRNO_INVALID_IRQ_ARG: wrong configuration argument / argument not supported.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRN0_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRN0_START_THREAD: creating callback thread failed. (Windows only).

meIOIrqStop

Description:

With this function the interrupt handler is stopped:

- Cancels any pending action.
- Interrupts will be disabled (in hardware and operating system).

Function Declaration:

int meIOIrqStop(int iDevice, int iSubdevice, int iChannel, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iChannel>

Index of the interrupt channel within the selected subdevice, otherwise it should be set to „0“.

<iFlags>

- ME_IO_IRQ_STOP_NO_FLAGS

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOIrqWait

Description:

This function waits as long as an interrupt occurs and serves for analyzing the interrupt event. This is completely independent from *meIOIrqStart().* In multi-threading applications there is no need for synchronization. *meIOIrqWait()* can be called before or after *meIOIrqStart()* and waits for the first interrupt event. If an interrupt occurs before *meIOIrqWait()* is called the function returns immediately and reports it. You must decide whether you want to install a callback function by *meIOIr-qSetCallback()* before calling the function *meIOIrqStart().*

Function Declaration:

int meIOIrqWait(int iDevice, int iSubdevice, int iChannel, int *piIrqCount, int *piValue, int iTimeOut, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the interrupt subdevice.

<iChannel>

Index of the interrupt channel within the selected subdevice of type ME_TYPE_EXT_IRQ otherwise it has to be set to „0“.

<piIrqCount> (r)

Parameter returns the number of interrupts from the specified channel since starting. *meIOResetDevice()* and *meIOResetSubdevice()* clear that counter.

<piValue> (r)

Parameter returns the interrupt status. There are two formats (configured by parameter <iFlags>):

- **“Simple format“:** One status bit per IRQ line. The status bit is set when an interrupt was generated by corresponding line. Also several status bits (b15…0) can be set, e.g. in the operation modes “bit pattern match“ and „bit-pattern change“.
- **“Extended format“:** Two status bits per IRQ line. One status bit is for a rising edge (0 → 1) and one for a falling edge (1 → 0). The falling edge bits are in the lower word (b15…0) and the rising edge bits are in the upper word (b31…16). Interesting e.g. in the operation mode „bit-pattern change“.

<iTimeOut>

Time-out value in milliseconds. If no interrupt was detected within the defined time, the operation will be cancelled. If no time-out value should be used, pass the value „0“.

<iFlags>

- ME_IO_IRQ_WAIT_NO_FLAGS
  No flags in use. Default settings will be used.
- ME_IO_IRQ_WAIT_NORMAL_STATUS
  Use „simple format“ for interrupt status (see <piValue>)
- ME_IO_IRQ_WAIT_EXTENDED_STATUS
  Use „extended format“ for interrupt status (see <piValue>)

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_CANCELLED: Subdevice was reset.
- ME_ERRNO_SIGNAL: Driver was unloaded.
- ME_ERRNO_PREVIOUS_CONFIG: Subdevice wrongly configured.

meIOIrqSetCallback

Description:

By this function you can install a callback function which waits for an interrupt in the background.

- **Windows:**
  - Callback function is dependent on *meIOIrqStart()* and *meIOIrqStop().*
  - Background task is created within *meIOIrqStart()* execution and cancelled in meIOIrqStop(). Therefore *meIOIrqSetCallback()* can only be called *before meIOIrqStart().*
  - Works exactly like meIOIrqWait() without any flags set (ME_IO_IRQ_WAIT_NORMAL_STATUS and ME_IO_IRQ_WAIT_EXTENDED_STATUS are NOT supported).
  - Only one callback function can be installed per subdevice.

To deinstall/cancel the callback function (all registered instances) for the selected subdevice call *meIOIrqSetCallback()* and pass NULL in <pCallback>.

Function Declaration:

int meIOIrqSetCallback(int iDevice, int iSubdevice, meIOIrqCB_t pCallback, void *pCallbackContext, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the interrupt subdevice.

<pCallback>

Pointer to a user-defined callback function. This function is called when an interrupt occurred. If the function exits with a return value different than ME_ERNNO_SUCCESS (0x00) an interrupt is instantly stopped (*meIOIrqStop()* is called).

<pCallbackContext> (w)

User-defined pointer passed to the callback function. This parameter is optional. If you don´t want to use it, pass NULL.

<iFlags>

- ME_IO_IRQ_SET_CALLBACK_NO_FLAGS No flags in use. Default settings will be used.
- ME_IO_IRQ_WAIT_NORMAL_STATUS (only Linux)
  Use „simple format“ for interrupt status (see parameter <iValue>)

- ME_IO_IRQ_WAIT_EXTENDED_STATUS (only Linux) Use „extended format“ for interrupt status (see parameter <`iValue`>)

Type Definition `meIOIrqCB_t`

```
typedef int (*meIOIrqCB_t)(
   int iDevice,
   int iSubdevice,int iChannel,int iIrqCount, int
   iValue,
   void *pvContext,
   int iErrorCode);
```

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the interrupt subdevice.

<iChannel>

Index of the interrupt channel within the selected subdevice of type

ME_TYPE_EXT_IRQ otherwise it has to be set to „0“.

<iIrqCount>

Parameter returns the number of interrupts from the specified channel since starting. See also parameter <piIrqCount> of function *meIOIrqWait().*

<iValue>

Parameter returns the interrupt status. There are two formats (see also parameter <iFlags> of function *meIOIrqWait()*):

- **“Simple format“:** each bit represents one bit. It is set when an interrupt was generated by this bit. More than one bit can be set.
- **“Extended format“:** Each line is represented by two bits. One is for a rising edge (0 → 1) and one for a falling edge (1 → 0). The falling edge bits are in the lower word (b15…0) and the rising edge bits are in the upper word (b31…16).

<pvContext> (w)

User-defined pointer <pCallbackContext>. If you do not want to use this parameter, pass NULL.

<iErrorCode>

Error code: see error reported by *meIOIrqWait().*

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.

- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_THREAD_RUNNING: callback thread already is running. (Windows only).

meIOSetChannelOffset

Description:

Using the function *meIOSetChannelOffset()* the anlog input ranges can be adjusted. If you use the function *meIOSetChannelOffset()* to change an offset range you must adjust the final result accordingly by adding this same offset.

Note: Currently, the offset adjustment only applies to streaming mode. In single mode the ranges have a fixed offset of 0.0 Volts.

Detailed informations regarding the MEphisto Scope can be found in appendix B4 on page 221.

Function Declaration:

int meIOSetChannelOffset(int iDevice, int iSubdevice, int iChannel, int iRange, double *pdOffset, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iChannel>

Index of the channel whose offset should be adjusted.

<iRange>

Index (0…6) of the measurement range to be used for the measurement. See also functions *meQueryNumberRanges(),* meQueryRangeByMinMax() and meQueryRangeInfo().

<pdOffset> (r/w)

(w): Pointer to a double value, passing the asked offset [V].

(r): If the asked offset value cannot be realized exactly by the hardware the currently adjusted value will be returned.

<iFlags>

Flag for extended options:

- ME_IO_SET_CHANNEL_OFFSET_NO_FLAGS No flags used. Default settings.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOSingleConfig

Description:

This function prepares a subdevice (AI, AO, digital-I/O, frequency-I/O or counter) for a „single operation“. Basically, the operation starts after calling the function meIOSingle() corresponding to the trigger conditions described here.

Note: In case of differential measuring only bi-polar input ranges can be used! Of course the variously trigger modes are only available if the hardware offers the appropriate capabilities.

Function Declaration:

int meIOSingleConfig(int iDevice, int iSubdevice, int iChannel, int iSingleConfig, int iRef, int iTrigChan, int iTrigType, int iTrigEdge, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iChannel>

Channel index. See chapter „Single Operation“ on page 42 for details.

<iSingleConfig>

Configuration of measurement ranges, digital ports and counters. See also chapter 3.4.1 „Single Operation“ on page 37 for details.

- Analog input range. Pass the range returned by the query functions.
- Analog output range. Pass the range returned by the query functions.

Configuration of digital ports, if supported by the respective hardware (see hardware manual):

- ME_SINGLE_CONFIG_DIO_INPUT Configuring the specified digital port as an input.
- ME_SINGLE_CONFIG_DIO_OUTPUT Configuring the specified digital port as an output.

Note: With the ME-5810/8100 you must choose one of the following constants instead of the above one:

- ME_SINGLE_CONFIG_DIO_HIGH_IMPEDANCE
  Setting the specified digital output port in high impedance state (e.g. ME-5810/8100).

- ME_SINGLE_CONFIG_DIO_SINK
  Enabling the sink drivers (low-active) for the specified digital output port (e.g. ME-5810/8100).
- ME_SINGLE_CONFIG_DIO_SOURCE
  Enabling the source drivers (high-active) for the specified digital output port (e.g. ME-5810/8100).
- ME_SINGLE_CONFIG_DIO_BIT_PATTERN
  Configuring the specified digital output port (subdevice of type „ME_TYPE_DO“ or „ME_TYPE_DIO“) for timer-controlled bit-pattern output. See also parameter <iRef> of this function and parameter <iFlags> of the function *meIOStreamConfig().* Refer to appendix A3 on page 182 for a detailed description.

The following both constants are only required, if the ME-MultiSig system should not be registered with the ME-iDC (in the pipeline):

- ME_SINGLE_CONFIG_DIO_MUX32M
  Specified digital output port will be used for the timer-controlled MUX operation (streaming operation) of the ME-MultiSig system in combination with the ME-4680. See also parameter <iRef> of this function.
- ME_SINGLE_CONFIG_DIO_DEMUX32
  Specified digital output port will be used for the timer-controlled DEMUX operation (streaming operation) of the ME-MultiSig system in combination with the ME-4680. See also parameter <iRef> of this function.

Configuration of frequency input/output:

- ME_SINGLE_CONFIG_FIO_INPUT
  Configuring the specified channel (subdevice of type „ME_TYPE_FI“ or „ME_TYPE_FIO“) as an input for frequency measurement.
- ME_SINGLE_CONFIG_FIO_OUTPUT Configuring the specified channel (subdevice of type
  „ME_TYPE_FO“ or „ME_TYPE_FIO“) as a frequency generator output.

Operation mode for the counters of type 8254 (a detailled description of the modes can be found from page 177):

- ME_SINGLE_CONFIG_CTR_ 8254_MODE_0 "Change state at zero".
- ME_SINGLE_CONFIG_CTR_8254_MODE_1 "Retriggerable One-Shot".
- ME_SINGLE_CONFIG_CTR_8254_MODE_2 "Asymmetric divider".
- ME_SINGLE_CONFIG_CTR_8254_MODE_3 "Symmetric divider".
- ME_SINGLE_CONFIG_CTR_8254_MODE_4 "Counter start by software trigger".
- ME_SINGLE_CONFIG_CTR_8254_MODE_5 "Counter start by hardware trigger".

<iRef>

For some channels the ground reference must be defined explicitly (see also chapter „Single Operation“ on page 37 for details):

- ME_REF_NONE
  Default setting (e.g. for standard digital I/O and frequency I/O)
- ME_REF_AI_GROUND
  Single ended measurement with reference to ground of AI-section.
- ME_REF_AI_DIFFERENTIAL
  Differential measurement without direct ground reference.
- ME_REF_AO_GROUND
  Analog output with reference to ground of AO-section.
- ME_REF_AO_DIFFERENTIAL
  Differential output without direct ground reference.

The following constants define the clock source of the counters:

- ME_REF_CTR_PREVIOUS
  Clock source is the output of previous counter.
- ME_REF_CTR_INTERNAL_1 MHZ
  Clock source is the internal 1 MHz crystal oscillator.
- ME_REF_CTR_INTERNAL_10 MHZ
  Clock source is the internal 10 MHz crystal oscillator.
- ME_REF_CTR_EXTERNAL
  Clock source is an external oscillator.

The following both constants are only required for bit-pattern output and ME-MultiSig operation if the system should not be registered with the ME-iDC (in the pipeline). They serve the assignment of low-byte and high-byte of the 16-bit wide FIFO values to the 8-bit-wide digital ports of the ME-4680 (only for output ports). See also parameter

<iSingleConfig>:

- ME_REF_DIO_FIFO_LOW Low-byte of the FIFO (Bit 7…0).
- ME_REF_DIO_FIFO_HIGH
  High-byte of the FIFO (Bit 15…8).

<iTrigChan>

Trigger channel, if supported by the subdevice, else pass ME_TRIG_CHAN_DEFAULT (see also chapter 3.4.3.3 „Synchro-nous Start“ on page 80 for details).

- ME_TRIG_CHAN_DEFAULT
  Triggering is done separatly for each channel.
- ME_TRIG_CHAN_SYNCHRONOUS
  Including this channel into the sync-list. All channels start synchro-nously in dependency of further trigger options (e.g. software start or external trigger).

<iTrigType>

Trigger type for starting of input/output (if supported by subdevice, else pass ME_TRIG_TYPE_SW). Basically conversion will be started by calling the function *meIOSingle()* in accordance with the trigger conditions defined in this function *(meIOSingleConfig()*).

- ME_TRIG_TYPE_SW
  Start directly after calling the function *meIOSingle().*
- ME_TRIG_TYPE_EXT_DIGITAL Start by external, digital trigger signal.
- ME_TRIG_TYPE_EXT_ANALOG Start by external, analog trigger signal.

<iTrigEdge>

Choose the appropriate trigger edge (if supported by the subdevice, else pass ME_VALUE_NOT_USED):

- ME_TRIG_EDGE_ABOVE
  Trigger if the level is above the threshold.
- ME_TRIG_EDGE_UNDER
  Trigger if the level is below the threshold.
- ME_TRIG_EDGE_ENTRY
  Trigger, if the signal enters a defined window.
- ME_TRIG_EDGE_EXIT
  Trigger, if the value leaves a defined window.
- ME_TRIG_EDGE_RISING
  Trigger on a rising edge.
- ME_TRIG_EDGE_FALLING
  Trigger on a falling edge.
- ME_TRIG_EDGE_ANY
  Trigger on a rising or falling edge.

<iFlags>

Flag for extended options:

- ME_IO_SINGLE_CONFIG_NO_FLAGS
  No flags used. Default settings.
- ME_IO_SINGLE_CONFIG_DIO_BIT
  Digital input/output operation by bit.
- ME_IO_SINGLE_CONFIG_DIO_BYTE
  Digital input/output operation by byte (8 bit).
- ME_IO_SINGLE_CONFIG_DIO_WORD
  Digital input/output operation by word (16 bit).
- ME_IO_SINGLE_CONFIG_DIO_DWORD
  Digital input/output operation by long-word (32 bit).
- ME_IO_SINGLE_CONFIG_CONTINUE
  „Helper“ flag for applying the settings made in this function for the channel passed in parameter <iChannel> and all channels beyond. E.g.: a subdevice for analog acquisition should have 32 channels. In

<iChannel> the index 4 is passed and this flag is set, i.e. the channels 4… 31 will be configured identically. This has the advantage, that significant less write accesses to the device are necessary (especially for USB devices).

Both of the following constants serve for switching the address LEDs ON and OFF on the MUX base boards (ME-MUX32-M/S) – see also the ME-MultiSig manual. In parameter <iChannel> any channel of that one base board must be selected, whose address LED should be switched:

- ME_IO_SINGLE_CONFIG_MULTISIG_LED_ON Switching the address LED on.
- ME_IO_SINGLE_CONFIG_MULTISIG_LED_OFF Switching the address LED off.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_INVALID_REF: parameter <iRef> is not correct.
- ME_ERRNO_INVALID_TRIG_CHAN: parameter <iTrigChan> is not correct.
- ME_ERRNO_INVALID_TRIG_TYPE: parameter <iTrigType> is not correct.
- ME_ERRNO_INVALID_TRIG_EDGE: parameter <iTrigEdge> is not correct.
- ME_ERRNO_INVALID_SINGLE_CONFIG: parameter <iSingleConfig> is not correct.

meIOSingle

Description:

With this function one or more read/write operations can be processed by a list. See also chapter 3.4.1 „Single Operation“ on page 37 for details.

Note: If for one or more list entries an external trigger source has been selected, the function waits until the appropriate trigger signal occurs. I.e. if

operation 1 with external trigger is running in blocking mode, operation 2 waits until the external trigger pulse of operation 1 occurs.

Function Declaration:

int meIOSingle(meIOSingle_t *pSingleList, int iCount, int iFlags);
<pSingleList> (r/w)

Pointer to a list of type meIOSingle_t. Each entry represents a single read/write operation. If for one or more list entries an external trigger source has been selected, the function waits until the appropriate trigger signal occurs.

<iCount>

Number of entries in <pSingleList>.

<iFlags>

- ME_IO_SINGLE_NO_FLAGS
  No extended options. Default settings. Execution is stopped on first error. Return value corresponds with <iErrno> field in the last processed <pSingleList> entry.
- ME_IO_SINGLE_NONBLOCKING (Linux only)
  Processes the whole single list. When this flag is set execution is processed although an error occured for some entries. The function returns ME_ERRNO_SUCCESS when no global error was detected. <iErrno> fields have to be checked.

Type Definition meIOSingle

```
typedef struct meIOSingle {
   int iDevice;
   int iSubdevice; int iChannel; int iDir;
   int iValue;
   int iTimeOut; int iFlags; int iErrno;
   } meIOSingle_t;
```

<iDevice> (w)

Index of the device to be accessed.

<iSubdevice> (w)

Index of the subdevice to be accessed.

<iChannel> (w)

Channel index resp. index of the group. The size of a group depends of the flag used in parameter <iFlags>.

In combination with the ME-MultiSig system the MUX resp. DEMUX channels are chosen by this parameter.

If you use a subdevice with only one channel pass the value „0“. Note: The number of channels of digital I/O ports depends on the parameter <iFlags> of this function. I.e. for a 32-bit port applies:

- Bit access (…DIO_BIT): channel index 0…31
- Byte access (…DIO_BYTE): channel index 0…3
- Word access (…DIO_WORD): channel index 0…1
- Langword access (…DIO_DWORD): channel index 0

Example: If you pass the flag ME_IO_SINGLE_TYPE_DIO_ BYTE for a digital subdevice with 32 bits there are four channels each one byte (8 bits) wide available.

<iDir> (w)

- ME_DIR_INPUT Read operation
- ME_DIR_OUTPUT Write operation
- ME_DIR_SET_OFFSET Set offset for current channels (see also ME-Axon manual)

<iValue> (r/w)

- AI: Measurement value is returned as standardized digital value. Use the function *meUtilityDigitalToPhysical()* to convert the digital value into the correct physical unit.
- AO: Passing voltage/current as standardized digital value. Use the function *meUtilityPhysicalToDigital()* to convert the voltage resp. current into the correct digital value.
- Digital IO: Read resp. output a 32-bit digital value. Depending on port width always the lower significant bits are relevant.
- Frequency IO: To read resp. write the period and the duration of the first phase of the period you have to call the function *meIOSingle()* twice. Depending on the value in parameter <iFlags>, either the period (ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL) or the duration of the first phase of the period (ME_IO_SINGLE_TYPE_ FIO_TICKS_FIRST_PHASE) will be returned resp. passed in ticks by <iValue> (see also chap. 3.4.1.4.1 on page 42). Use the functions *meIOSingleTicksToTime()* and *meIOSingleTimeToTicks()* for easy conversion of ticks to seconds and reverse.
- Counter: Writing a start value resp. reading the counter value.

<iTimeOut> (w)

Time-out value in milliseconds. If no external trigger pulse was detected within the defined time interval, the operation will be cancelled. If no external trigger is being used or no time-out value is used, pass ME_VALUE_NOT_USED.

`<iFlags> (w)`

Extended settings:

- ME_IO_SINGLE_TYPE_NO_FLAGS
  No extended options – default settings will be used. For digital ports the “natural” size is used.
- ME_IO_SINGLE_TYPE_DIO_BIT Digital input/output operation by bit.
- ME_IO_SINGLE_TYPE_DIO_BYTE
  Digital input/output operation by byte (8 bit).
- ME_IO_SINGLE_TYPE_DIO_WORD
  Digital input/output operation by word (16 bit).
- ME_IO_SINGLE_TYPE_DIO_DWORD
  Digital input/output operation by long-word (32 bit).
- ME_IO_SINGLE_TYPE_TRIG_SYNCHRONOUS
  Synchronous trigger by software start. Useful for the last channel of <`pSingleList`> to start all channels configured for synchronous start by calling this function. See option ME_TRIG_CHAN_ SYNCHRONOUS in parameter <`iTrigChan`> of the function *meIO- SingleConfig()*.
- ME_IO_SINGLE_TYPE_NONBLOCKING
  (alias: ME_IO_SINGLE_TYPE_WRITE_NONBLOCKING) Operation will run in background. This means that the function doesnot wait for execution's end.
  Note: Not every subdevice supports nonblocking mode.
- ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL Read resp. output the period in <iValue>.
- ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_PHASE
  Read resp. output the duration of the first phase of the period in <`iValue`>.

The frequency measurement (frequency counter) can be controlled by an appropriate combination of the following flags. It is done by bitwise OR-linking ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL resp. ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_PHASE with the following options:

- ME_IO_SINGLE_TYPE_FI_LAST_VALUE
  Flag for acquisition of low frequencies repeatedly (see also parameter <`iFlags`> of the function *meIOSingleConfig()*). It must be additionally OR-linked with the flag ME_IO_SINGLE_TYPE_ NON-BLOCKING bitwise.

Starting the frequency output (pulse generator) can be controlled by an appropriate combination of the following flags. It is done by bitwise OR-

linking ME_IO_SINGLE_TYPE_FIO_TICKS_TOTAL resp. ME_IO_SINGLE_TYPE_FIO_TICKS_FIRST_PHASE with one or more of the following options.

- ME_IO_SINGLE_TYPE_FO_UPDATE_ONLY
  The output value should be updated but not output at once. No linking with other flags possible. Default: the new value is output immediately.
- ME_IO_SINGLE_TYPE_FO_START_SOFT
  The value is output not until the end of the current period (if already running). Can be bitwise OR-linked with ME_IO_SINGLE_TYPE_FO_START_LOW. Default: the new value is output immediately.
- ME_IO_SINGLE_TYPE_FO_START_LOW
  By default the first phase of the rectangular signal is „high“. If the flag is set, the output starts with „low“ level. Can be bitwise OR-linked with ME_IO_SINGLE_TYPE_FO_START_SOFT.
- ME_IO_SINGLE_TYPE_TRIG_SYNCHRONOUS
  All subdevices, which have been added to the sync-list by parameter <`iTrigChan`> of the function *meIOSingleConfig()* will be started simultanously (see also chap. 3.4.3.3 on page 74). Default: subdevice starts independently.

`<iErrno>`

Error code returned by the particular entry.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL: no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_TIMEOUT: timeout condition occurred.
- ME_ERRNO_PREVIOUS_CONFIG: subdevice was not configured for required operation.
- ME_ERRNO_SUBDEVICE_BUSY: subdevice is performing other operation.

meIOSingleTicksToTime

Description:

Converts the number of ticks into the desired time, e.g. period [s] for further processing in your application (e.g. frequency measurement). In dependency of parameter <iTimer> of this function the return values from <piTicksLow> resp. <piTicksHigh> can be passed to the parameter <iValue> of the function *meIOSingle().*

Note: The conversion and the allowed value range depend on each subdevice and their properties. If hardware limits are exceeded, always the limit values are returned.

Tip: If you need the dimensions frequency and duty-cycle you can calculate them easily by the return values from <pdTime>. It applies:

- Frequency [Hz] = 1/period [s]
- Duty-cycle [%] = („Duration of the first phase of the period“ [s] /period [s]) × 100

Function Declaration:

int meIOSingleTicksToTime(int iDevice, int iSubdevice, int iTimer, int iTicksLow, int iTicksHigh, double *pdTime, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iTimer>

Ticks will be calculated in dependency of the subdevice type and the timer specified here.

- ME_TIMER_FIO_TOTAL
  The period is converted into seconds.
- ME_TIMER_FIO_FIRST_PHASE
  The duration of the first phase of the period is converted into seconds.

<iTicksLow>

The number of ticks (lower significant part, bits 31…0) from parameter

<iValue> of the function *meIOSingle()* are passed here.

<iTicksHigh>

The number of ticks (higher significant part, bits 63…32) from parameter <iValue> of the function *meIOSingle()* are passed here. This parameter is resevered for future enhancements.

<pdTime> (r)

`(r)`**:** Pointer to a double value, which returns the calculated time in seconds, e.g. period [s].

`<iFlags>`

- ME_IO_SINGLE_TIME_TO_TICKS_NO_FLAGS (default)

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_TIMER: not supported timer ID.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOSingeTimeToTicks

Description:

Converts a given period [s] into the number of „ticks“ to be passed to the timer in the function *meIOStreamConfig().* In dependency of parameter `<iTimer>` of this function the return values of <piTicksLow> and `<piTicksHigh>` can be passed to the parameter <iValue> of the function *meIOSingle().*

Note: The conversion and the allowed value range depend on each subdevice and their properties. If hardware limits are exceeded, an error message is returned.

Function Declaration:

int meIOSingleTimeToTicks(int iDevice, int iSubdevice, int iTimer, double*pdTime, int *piTicksLow, int *piTicksHigh, int iFlags);

`<iDevice>`

Index of the device to be accessed.

`<iSubdevice>`

Index of the subdevice to be accessed.

`<iTimer>`

Ticks will be calculated in dependency of the subdevice type and the timer specified here. The values `<piTicksLow>` and `<piTicks- High>` can be passed to the parameter `<iValue>` of the function *meIOSingle()* next.

- ME_TIMER_FIO_TOTAL
  The period is converted into ticks.
- ME_TIMER_FIO_FIRST_PHASE
  The duration of the first phase of the period is converted into ticks.

<pdTime> (r/w)

(w): Pointer to a double value, passing the asked time in seconds, e.g. the period [s] to be converted into ticks. If you pass invalid values a corresponding error code will be returned.

(r): If the asked time cannot be realized exactly by the hardware the value next lower to it will be returned here. The corresponding ticks are returned in the parameters <piTicksLow> and <piTicksHigh>.

<piTicksLow> (r)

Pointer to an integer value, which contains the lower significant 32 bits (31…0) of the calculated ticks. To be passed to the parameter <iValue> of the function *meIOSingle()*

<piTicksHigh> (r)

Pointer to an integer value, which contains the higher significant 32 bits (63…32) of the calculated ticks. This parameter is resevered for future enhancements.

<iFlags>

- ME_IO_SINGLE_TIME_TO_TICKS_NO_FLAGS (default)

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_TIMER: not supported timer ID.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOStreamConfig

Description:

This function configures the hardware for a timer-controlled streaming operation. See also chapter 3.4.2 „Stream Operation“ on page 51.

For analog and digital (bit-pattern output) streaming operations a channel-list must be created, which contains an entry for each channel (channel index, measurement range…) of type *meIOStreamConfig_t.*

Additionally a trigger structure (meIOStreamTrigger_t) is required, which defines numerous settings like start/stop conditions, timer settings, trigger sources and trigger edges valid for the whole operation.

The operation is always started by the function *meIOStreamStart()* either at once (software start) or in accordance with the start conditions defined in the function *meIOStreamConfig().* Stop the operation either according to the stop conditions defined in the trigger structure or by calling the function *meIOStreamStop().*

Note: Use the functions *meIOStreamFrequencyToTicks()* and *meIOStreamFrequencyToTicks()* (see page 169) to convert frequency resp. period into ticks easily in order to pass them to the timers.

Please note appendix A5 on page 185 if you want to use the ME-MultiSig system.

Tip: Initialize the trigger structure *meIOStreamTrigger_t* with „0“. In that way you must only take care of the parameters which are required. At the same time unused parameters are passed correctly and automatically.

Function Declaration:

int meIOStreamConfig(int iDevice, int iSubdevice, meIOStreamConfig_t*pConfigList, int iCount, meIOStreamTrigger_t *pTrigger, int iFIFOIrqThreshold, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<pConfigList>

Pointer to a list of type *meIOStreamConfig_t* (see below).

<iCount>

Number of entries in <pConfigList>.

<pTrigger>

Pointer to a structure of type *meIOStreamTrigger_t* (see below).

<iFIFOIrqThreshold>

Number of values to be read or written (reloaded) in one package. Refreshing the hardware buffer event is used for user – hardware synchronization. If not set hardware FIFO is read/loaded when the “HALF FIFO” flag is detected.

<iFlags>

- ME_IO_STREAM_CONFIG_NO_FLAGS (default setting) Pointer to a list with <iCount> entries of type meIOStreamConfig_t with the configuration of the single channels
  - Flag for continuous streaming operation. See chapter 3.4.2.6 on page 67.
  - MEphisto Scope: analog acquisition
- ME_IO_STREAM_CONFIG_BIT_PATTERN
  - MEphisto Scope: Logic analyzer mode, parameter <iCount> will be ignored.
  - Special functions bit-pattern output and FIFO redirection (e.g. to control the ME-MultiSig system). The DAC will be disconnected from the FIFO. For details see chapter 3.4.3.2 on page 73 and appendix A3 on page 182.

Note: Also the digital ports should be configured to accept this redirection (with ME_REF_DIO_FIFO_LOW or ME_REF_DIO_FIFO_HIGH).

- ME_IO_STREAM_CONFIG_WRAPAROUND
  Flag for wraparound mode (periodical output). See chapter 3.4.2.6.4 „Wraparound Option“ on page 78.
- ME_IO_STREAM_CONFIG_SAMPLE_AND_HOLD
  Enables “Sample&Hold” feature. For details see chapter 3.4.3.1 „Sample and Hold“ on page 72.

! For periodic bit-pattern output and for periodic DEMUX operation the constant ME_IO_STREAM_CONFIG_WRAPAROUND must be ORed with the constant ME_IO_STREAM_CONFIG_ BIT_PATTERN.

Type Definition meIOStreamConfig

```
typedef struct meIOStreamConfig {
int iChannel;
int iStreamConfig;
int iRef;
int iFlags;
} meIOStreamConfig_t;
```

`<iChannel> (w)`

Channel index. Depending on subdevice this can be analog inputs or outputs as well as a group of digital I/Os (e.g. in the operation mode bit-pattern output).

`<iStreamConfig> (w)`

Choosing the range for the channel-list entry (see chapter 3.4.2 „Streaming Operation“ on page 51 for details):

- Index for analog input range. Pass the index for the asked range returned by the query functions.
- Index for analog output range. Pass the index for the asked range returned by the query functions.
- On bit-pattern output pass ME_VALUE_NOT_USED.

`<iRef> (w)`

Defines the ground reference for analog inputs and outputs (see chapter 3.4.2 „Streaming Operation“ on page 51 for details):

- ME_REF_NONE
  Default setting (e.g. for bit-pattern output)
- ME_REF_AI_GROUND
  Single ended measurment with reference to ground of the AI section.
- ME_REF_AI_DIFFERENTIAL
  Differential measurement without direct ground reference.
- ME_REF_AO_GROUND
  Output with reference to ground of the AO section. Use this constant for analog output.
- ME_REF_AO_DIFFERENTIAL
  Differential measurement without direct ground reference.

`<iFlags>`

- ME_IO_STREAM_CONFIG_TYPE_NO_FLAGS (No flags available)

‘ **Type Deinition** `meIOStreamTrigger`

```
typedef struct meIOStreamTrigger {
        int iAcqStartTrigType;
        int iAcqStartTrigEdge;
        int iAcqStartTrigChan;
        int iAcqStartTicksLow;
        int iAcqStartTicksHigh;
        int iAcqStartArgs[10];
        int iScanStartTrigType;
```

```
        int iScanStartTicksLow;
        int iScanStartTicksHigh;
        int iScanStartArgs[10];
        int iConvStartTrigType;
        int iConvStartTicksLow;
        int iConvStartTicksHigh;
        int iConvStartArgs[10];
        int iScanStopTrigType;
        int iScanStopCount;
        int iScanStopArgs[10];
        int iAcqStopTrigType;
        int iAcqStopCount;
        int iAcqStopArgs[10];
        int iFlags;
} meIOStreamTrigger_t;
```

! **Note** the description of the trigger structure from page 58.

<iAcqStartTrigType>

This parameter defines the trigger type for start of the whole operation. Depending on the used hardware you can choose from the following options:

- ME_TRIG_TYPE_SW
  Start directly after calling the function. Pass „0“ in <iAcqStart-TrigEdge>.
- ME_TRIG_TYPE_EXT_ANALOG
  Start of the operation by an appropriate signal at the external analog trigger input.
- ME_TRIG_TYPE_THRESHOLD
  Start of the operation by a positive or negative deviation of a given threshold at the analog trigger channel (see <iAcqStart- Args[0]>) .
- ME_TRIG_TYPE_WINDOW
  Start of the operation when the signal at the analog trigger channel leaves or enters the defined window (see <iAcqStartArgs[0]> and <iAcqStartArgs[1]>).
- ME_TRIG_TYPE_EDGE
- Start of the operation by a falling or rising edge crossing a given level at the trigger channel (see <iAcqStartArgs[0]>). E.g. to get a still graph for repetitive AC signals.

- ME_TRIG_TYPE_SLOPE
  Start of the operation if the signal at the trigger channel increases or decreases faster as defined in <iAcqStartArgs[0]>.
- ME_TRIG_TYPE_EXT_DIGITAL
  Start of the operation by an appropriate signal at the external digital trigger input.
- ME_TRIG_TYPE_PATTERN
  Start of the operation if the bit-pattern at the trigger port equals the reference bit-pattern defined in <iAcqStartTrigChan>. Pass „0“ in <iAcqStartTrigEdge>.

<iAcqStartTrigEdge>

This parameter defines the edge to start a single conversion by an external trigger signal. Depending on the trigger type and the used hardware you can choose from different options:

- ME_TRIG_EDGE_NONE
  If you have choosen the option software start (ME_TRIG_TYPE _SW) in parameter <iAcqStartTrigType>.
- ME_TRIG_EDGE_RISING
  Start by a rising edge
- ME_TRIG_EDGE_FALLING
  Start by a falling edge
- ME_TRIG_EDGE_ANY
  Start either by a rising or falling edge.

…in combination with ME_TRIG_TYPE_THRESHOLD in

<iAcqStartTrigType>:

- ME_TRIG_EDGE_ABOVE
  Start if the level is above the trigger threshold.
- ME_TRIG_EDGE_BELOW
  Start if the level is below the trigger threshold.

…in combination with ME_TRIG_TYPE_WINDOW in

<iAcqStartTrigType>:

- ME_TRIG_EDGE_ENTRY
  Start if the level enters the defined window.
- ME_TRIG_EDGE_EXIT
  Start if the level leaves the defined window.

<iAcqStartTrigChan>

With this parameter you can choose whether triggering should be done separately for each channel (standard) or if a channel should be started synchronously with other channels (e.g. for analog acquisition with sample & hold option or synchronous start of several AO channels).

- ME_TRIG_CHAN_DEFAULT Independent start (default).
- ME_TRIG_CHAN_SYNCHRONOUS

This channel will be included in the „synchronous start list“.

- MEphisto Scope:
  - Selection of the analog trigger channel (0, 1) in the modes „Threshold“, „Window“, „Edge“ and „Slope“.
  - Passing the reference bit-pattern in trigger mode „Pattern“.
  - Else, pass „0“ here.

<iAcqStartTicksLow>

Offset time in number of ticks between „start“ of the measurement and the first conversion. **Note** that the settling time of the AI section may not fall below.

If supported by the hardware, combining of <iAcqStart-TicksLow> and <iAcqStartTicksHigh> allows values of up to 64-bit width. It applies:

AcqStartTicks = (AcqStartTicksHigh <<32) ∨ AcqStartTicksLow.

For standard applications we recommend to set the offset time to the minimum chan interval of the appropriate hardware (AcqStartTicks = min. ConvStartTicks).

<iAcqStartTicksHigh>

Higher significant part (bits 63…32) of the offset time, see <iAcq-Start-TicksLow>. If you don´t want use this parameter, pass „0“ here.

<iAcqStartArgs[0]> (r/w)

- If ME_TRIG_TYPE_THRESHOLD was passed in <iAcq-Start-TrigType> define the threshold value in [µV] here.
- If ME_TRIG_TYPE_WINDOW was passed in <iAcqStart-TrigType> define the upper threshold value of the window in [µV] here.
- If ME_TRIG_TYPE_EDGE was passed in <iAcqStartTrig-Type> define the threshold value in [µV] here.
- If ME_TRIG_TYPE_SLOPE was passed in <iAcqStartTrig-Type> define the slew rate in [µV/Sample].
  (r): In the above cases, on returning from *meIOStreamConfig()* the actual trigger value used is returned here. For other values of <iAcq-StartTrigType>, this parameter is not used and should be „0“.

<iAcqStartArgs[1]> (r/w)

- If ME_TRIG_TYPE_WINDOW was passed in <iAcqStart-TrigType> define the lower threshold value of the window in [µV] here.

(r): In the above cases, on returning from *meIOStreamConfig()* the actual trigger value used is returned here. For other values of <iAcq-Start-TrigType>, this parameter is not used and should be „0“.

<iScanStartTrigType>

This parameter defines the trigger type for start of a scan. Depending on the used hardware you can choose from the following options:

- ME_TRIG_TYPE_TIMER
  Start of the scan by the scan timer (e.g. channel-list processing).
- ME_TRIG_TYPE_FOLLOW
  Start will be triggered automatically by conversion of the last channel-list entry. The scan-timer will be disabled.
- ME_TRIG_TYPE_EXT_DIGITAL
  Start of the scan by an appropriate trigger signal at the external digital trigger input.
- ME_TRIG_TYPE_EXT_ANALOG
  Start of the scan by an appropriate trigger signal at the external analog trigger input.

<iScanStartTicksLow>

Time interval in ticks between the start of two consecutive scans (= channel-list processings). Usage is optional. If you don´t want to use the scan-timer pass „0“ here.

Note the following dependency when calculating the scan interval (see also the diagrams from page 58):

ScanStartTicks = (Number of channel-list entries x ConvStartTicks) + „Pause“ [Ticks].

If supported by the hardware, combining of <iScanStart-TicksLow> and <iScanStartTicksHigh> allows values of up to 64-bit width. It applies:

ScanStartTicks = (ScanStartTicksHigh <<32) v ScanStartTicksLow.

<iScanStartTicksHigh>

Higher significant part (bits 63…32) of the scan-time, see

<iScanStartTicksLow>. If you don´t want to use this parameter, pass „0“ here.

<iScanStartArgs[10]>

This parameter is reserved for future extensions.

<iConvStartTrigType>

This parameter defines the trigger type for start of a single conversion. Depending on the used hardware you can choose from the following options:

- ME_TRIG_TYPE_TIMER
  Start of the conversion by the chan-timer.
- ME_TRIG_TYPE_EXT_DIGITAL
  Start of the conversion by an appropriate trigger signal at the external digital trigger input.
- ME_TRIG_TYPE_EXT_ANALOG
  Start of the conversion by an appropriate trigger signal at the external analog trigger input.

<iConvStartTicksLow>

Chan interval in number of ticks between two conversions (Sample resp. output rate). The value range for the ME-4600 series is between 66 (42Hex) and 232-1 (FFFFFFFFHex) ticks.

If supported by the hardware, combining of <iConvStart-TicksLow> and <iConvStartTicksHigh> allows values of up to 64-bit width. It applies:

ConvStartTicks = (ConvStartTicksHigh <<32) & ConvStartTicksLow

<iConvStartTicksHigh>

Higher significant part (bits 63…32) of the chan interval, see <iConvStartTicksLow>. If you don´t want to use this parameter, pass „0“ here.

<iConvStartArgs[10]>

This parameter is reserved for future extensions.

<iScanStopTrigType>

This parameter defines the trigger type for ending the scan. Depending on the used hardware you can choose from the following options:

- ME_TRIG_TYPE_NONE No trigger source given.
- ME_TRIG_TYPE_COUNT

Acquisition/output will be ended after the total number of conversion defined in <iScanStopCount>.

! Use ME_TRIG_TYPE_COUNT only alternatively either in

<iScanStopTrigType> or <iAcqStopTrigType>.

<iScanStopCount>

Total number of conversions after which the scans will be ended and at the same time the measurement as a whole. If you want to run the measurement for an undefined time, pass „0“ here.

! Use this parameter only alternatively either in

<iScanStopCount> or <iAcqStopCount>.

<iScanStopArgs[0]>

- MEphisto Scope in oscilloscope mode: trigger point in percent between 0 %…100 %. On returning from *meIOStreamConfig()* the actual trigger point as a percentage is returned here. If the MEphisto Scope is used in data-logging mode, then this parameter is not required and should be „0“.

<iAcqStopTrigType>

On demand, this parameter defines the trigger type for ending the whole operation. The following options are available:

- ME_TRIG_TYPE_NONE No trigger source defined.
- ME_TRIG_TYPE_COUNT

The operation will be ended after the number of scans (channel-list processings) defined in <iAcqStopCount>.

! Use ME_TRIG_TYPE_COUNT only alternatively either in

<iScanStopTrigType> or <iAcqStopTrigType>.

- ME_TRIG_TYPE_FOLLOW
  The measurement will be ended, as soon as

<iAcqStopCount>

Number of scans (channel-list processings) after which the complete operation should be ended. If you want to run the operation for an undefined time, pass „0“ here.

! Use this parameter only alternatively either in

<iScanStopCount> or <iAcqStopCount>.

<iAcqStopArgs[10]>

This parameter is reserved for future extensions.

<iFlags>

- ME_IO_STREAM_TRIGGER_TYPE_NO_FLAGS (default) (no flags available).

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.

- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_CHANNEL:no channel available on subdevice.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_INVALID_REF: parameter <`iRef`> is not correct.
- ME_ERRNO_INVALID_ACQ_START_TRIG_CHAN: parameter. <`iAcqStartTrigChan`> is not correct.
- ME_ERRNO_INVALID_ACQ_START_TRIG_EDGE: parameter <`iAcqStartTrigEdge`> is not correct.
- ME_ERRNO_INVALID_STREAM_CONFIG: parameter <`iStreamConfig`> is not correct.
- ME_ERRNO_TIMEOUT: timeout condition occurred.
- ME_ERRNO_PREVIOUS_CONFIG: subdevice was not configured for required operation.
- ME_ERRNO_SUBDEVICE_BUSY: subdevice is performing other operation.
- ME_ERRNO_INVALID_FIFO_IRQ_THRESHOLD: parameter <`iFifoIrqThreshold`> is not valid (too big).
  ME_ERRNO_INVALID_CONFIG_LIST_COUNT: Wrong<`iCount`> of configuration list.
- ME_ERRNO_INVALID_ACQ_START_TRIG_TYPE: parameter <`iAcqStartTrigType`> is not correct.
- ME_ERRNO_INVALID_ACQ_START_ARG: interval <`iAcq-Start-Ticks`> is not correct.
- ME_ERRNO_INVALID_SCAN_START_ARG: interval <`iScan-StartTicks`> is not correct.
- ME_ERRNO_INVALID_CONV_START_ARG: interval <`iConv-StartTicks`> is not correct.
- ME_ERRNO_INVALID_ACQ_STOP_TRIG_TYPE: parameter <`iConvStartTrigType`> is not correct.
- ME_ERRNO_INVALID_SCAN_STOP_TRIG_TYPE: parameter <`iConvStartTrigType`> is not correct.
- ME_ERRNO_INVALID_ACQ_STOP_ARG: parameter <`iAcqStopCount`> is not correct.
- ME_ERRNO_INVALID_SCAN_STOP_ARG: parameter <`iScan-StopCount`> is not correct.

meIOStreamTimeTo Ticks

Description:

Converts a given period [s] into the number of „ticks“ to be passed to the timer in the function *meIOStreamConfig().*

Note: The conversion and the allowed value range depend on each subdevice and their timers. If hardware limits are exceeded, always the limit values are returned.

Tip: Passing „0“ in parameter <pdTime> of this function returns the minimum frequency allowed. In dependency of parameter <iTimer> of this function the return values of <piTicksLow> and <piTicks-High> can be passed to the corresponding parameters <iAcqStart- Ticks…>, <iConvStartTicks…> and <iScanStartTicks…> in the trigger structure of function *meIOStreamConfig().*

Function Declaration

int meIOStreamTimeToTicks(int iDevice, int iSubdevice, int iTimer, double *pdTime, int *piTicksLow, int *piTicksHigh, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iTimer>

Ticks will be calculated in dependency of the subdevice and the timer specified here. The values <piTicksLow> and <piTicksHigh> are passed in the trigger structure of the function *meIOStreamConfig()* (see page 58ff).

- ME_TIMER_ACQ_START
  <iAcqStartTicks> should be calculated for passing to the parameter of the same name.
- ME_TIMER_SCAN_START
  <iScanStartTicks> should be calculated for passing to the parameter of the same name.
- ME_TIMER_CONV_START
  <iConvStartTicks> should be calculated for passing to the parameter of the same name.

<pdTime> (r/w)

`(w)`: Pointer to a double value, passing the asked period [s] to be converted into ticks. If you pass „0“ the minimum period will be returned.

`(r)`: If the asked period cannot be realized exactly by the hardware the period next lower to it will be returned as an approximation. The corresponding ticks are returned in the parameters <`piTicksLow`> and <`piTicksHigh`>.

<`piTicksLow`> `(r)`

Pointer to an integer value, which contains the lower significant 32 bits (31…0) of the calculated ticks. Will be passed to the appropriate parameter <…`StartTicksLow`> of the function *meI-StreamConfig().*

<`piTicksHigh`> `(r)`

Pointer to an integer value, which contains the higher significant 32 bits (63…32) of the calculated ticks. Will be passed to the appropriate parameter <…`StartTicksHigh`> of the function *meIOStreamConfig().*

<`iFlags`>

- ME_IO_TIME_TO_TICKS_NO_FLAGS (default) MEphisto Scope: data-logging mode.
- ME_IO_TIME_TO_TICKS_MEPHISTO_SCOPE_ OSCILLOSCOPE MEphisto Scope: oscilloscope mode.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_TIMER: not supported timer ID.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOStreamFrequencyToTicks

Description:

Converts a given frequency [Hz] into the number of „ticks“ to be passed to the timer in the function meIOStreamConfig().

Note: The conversion and the allowed value range depend on each subdevice and their timers. If hardware limits are exceeded, always the limit values are returned.

Tip: Passing „0“ in parameter <pdFrequency> of this function returns the maximum frequency allowed. In dependency of parameter <iTimer> of this function the return values of <piTicksLow> and <piTicks-High> can be passed to the corresponding parameters <iAcqStart-Ticks…>, <iConvStartTicks…> and <iScanStartTicks…> in the trigger structure of *function meIOStreamConfig().*

Function Declaration:

int meIOStreamFrequencyToTicks(int iDevice, int iSubdevice, int iTimer, double *pdFrequency, int *piTicksLow, int *piTicksHigh, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iTimer>

Ticks will be calculated in dependency of the subdevice to which the specified timer belongs to. The values <piTicksLow> and <piTicksHigh> are passed in the trigger structure of the function *meIOStreamConfig()* (see page 58ff).

- ME_TIMER_ACQ_START
  <iAcqStartTicks> should be calculated for passing to the parameter of the same name.
- ME_TIMER_SCAN_START
  <iScanStartTicks> should be calculated for passing to the parameter of the same name.
- ME_TIMER_CONV_START
  <iConvStartTicks> should be calculated for passing to the parameter of the same name.

<pdFrequency> (r/w)

Pointer to a double value, passing the asked frequency [Hz] to be converted into ticks. If you pass „0“ the maximum frequency will be returned. If

the asked frequency cannot be realized exactly by the hardware the frequency next higher to it will be returned as an approximation. The corresponding ticks are returned in the parameters <piTicksLow> and <piTicksHigh>.

<piTicksLow> (r)

Pointer to an integer value, which contains the lower significant 32 bits (31…0) of the calculated ticks. Will be passed to the appropriate parameter <…StartTicksLow> of the function *meIOStreamConfig().*

<piTicksHigh> (r)

Pointer to an integer value, which contains the higher significant 32 bits (63…32) of the calculated ticks. Will be passed to the appropriate parameter <…StartTicksHigh> of the function *meIOStreamConfig().*

<iFlags>

- ME_IO_FREQUENCY_TO_TICKS_NO_FLAGS (default) MEphisto Scope: data-logging mode.
- ME_IO_FREQUENCY_TO_TICKS_MEPHISTO_SCOPE_ OSCILLOSCOPE.
  MEphisto Scope: oscilloscope mode

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_TIMER: not supported timer ID.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meIOStreamStart

Description:

Function starts streaming operations. Either immediately (software start) or in accordance to the start conditions defined in the function *meIOStreamConfig().*

Ending a streaming operation is either done in accordance to the stop conditions defined in the function *meIOStreamConfig()* or by the function *meIOStreamStop().*

If a streaming operation was not ended by the function *meIOResetDevice()* (hardware configuration will be deleted) you can start a new operation by calling this function without configuring newly.

Note: Returning of the function depends on the <iStartmode> (BLOCKING or NONBLOCKING) and trigger conditions defined in the function *meIOStreamConfig().*

Function Declaration:

int meIOStreamStart(meIOStreamStart_t *pStartList, int iCount, int iFlags);

<pStartList>

Pointer to a list of type meIOStreamStart_t, by which one or more streaming operations can be started. The start is done immediately after calling the function corresponding to the start conditions. If for one or more list entries the <iStartMode> ME_START_MODE_BLOCKING and an external trigger source (see *meIOStreamConfig())* has been selected, the function waits until the trigger signal occurs.

<iCount>

Number of entries in <pStartList>.

<iFlags>

- ME_IO_STREAM_START_NO_FLAGS
  Default settings. Execute list up to the first error. Returned value corresponds with the first non-zero <iErrno> field.
- ME_IO_STREAM_START_NONBLOCKING (Linux only) Execute whole start list. When this flag is set execution is processed although an error occured for some entries. The function returns ME_ERRNO_SUCCESS when no global error was detected. <iErrno> fields have to be checked.

Type Definition meIOStreamStart

```
typedef struct meIOStreamStart {
        int iDevice;
```

```
        int iSubdevice;
        int iStartMode;
        int iTimeOut;
        int iFlags;
        int iErrno;
} meIOStreamStart_t;
```

<iDevice> (w)

Index of the device to be accessed.

<iSubdevice> (w)

Index of the subdevice to be accessed.

<iStartMode> (w)

- ME_START_MODE_BLOCKING
  When using an external trigger, the function waits until the proper trigger signal occurs.
- ME_START_MODE_NONBLOCKING
  Function returns immediately. Starting the hardware runs in background. <pStartList> is processed independently of external trigger signals. Function returns ME_ERRNO_SUCCESS when no global error was detected.

<iTimeOut> (w)

Optionally, you can determine a time interval in milliseconds within the first trigger pulse must occur in accordance to the conditions defined in function *meIOStreamConfig().* Else the operation will be cancelled. If no external trigger and no time-out value is used, pass „0“ here.

<iFlags> (w)

- ME_IO_STREAM_START_TYPE_NO_FLAGS
  This constant is valid, if no other constant was choosen.
- ME_IO_STREAM_START_TYPE_TRIG_SYNCHRONOUS Synchronous start by „synchronous start list“.

<iErrno>

If an error occurs, an error code will be returned.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested

ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_INVALID_TIMEOUT: timeout lower than 0.
- ME_ERRNO_INVALID_START_MODE: not supported start mode detected.
- ME_ERRNO_PREVIOUS_CONFIG: subdevice is not configured correctly to proceed streaming operation.
- ME_STATUS_ERROR: previous operation ended with an error. Reset has to be called to clear this state.
- ME_ERRNO_TIMEOUT: Timeout. Operation didn't start on time.
- ME_ERRNO_START_THREAD: creating callback thread failed. (Windows only).

meIOStreamStop

Description:

By this function an „infinite“ operation either can be cancelled at once or stopped in a defined way (see parameter <`iStopMode`>). With that you have the possibility to stop an output operation by the last entry in the FIFO which is a known value.

If in the parameters <`iAcqStopCount`> resp. <`iScanStopCount`> of the function *meIOStreamConfig()* stop conditions have been defined, calling the function *meIOStreamStop()* is not necessary.

Configuration of the subdevice remains preserved (channel-list, timer…) so that a restart with the function *meIOStreamStart()* is possible without new configuration.

In opposite to this using the function *meIOResetDevice()* deletes the whole configuration of the device.

Function Declaration:

int meIOStreamStop(meIOStreamStop_t *pStopList, int iCount, int iFlags);

<`pStopList`>

Pointer to a list of type `meIOStreamStop_t` to end one or several input/output operations. Stopping is done in accordance to parameter <`iStopMode`>.

<`iCount`>

Number of entries in <`pStopList`>.

<iFlags>

- ME_IO_STREAM_STOP_NO_FLAGS
  Default settings. Execute list up to the first error. Returned value corresponds with the first non-zero <iErrno> field.
- ME_IO_STREAM_STOP_NONBLOCKING (Linux only) Execute whole stop list one by one. When this flag is set execution is processed although an error occured for some entries. The function returns ME_ERRNO_SUCCESS when no global error was detected.

<iErrno> fields have to be checked.

Type Definition meIOStreamStop

```
typedef struct meIOStreamStop {
        int iDevice;
        int iSubdevice;
        int iStopMode;
        int iFlags;
        int iErrno;
} meIOStreamStop_t;
```

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iStopMode>

- ME_STOP_MODE_IMMEDIATE
  Streaming operation is stopped at once. In case of an analog output 0V is attached to the pin.
- ME_STOP_MODE_LAST_VALUE
  - Output subdevice: operation is stopped on the last entry in the DA-FIFO which is a defined value.
  - Input subdevice: operation is stopped on the last entry from the channel-list.

<iFlags>

- ME_IO_STREAM_STOP_TYPE_NO_FLAGS Default settings.
  - Output subdevice: all buffers are cleared.
  - Input subdevice: Hardware buffer is cleared. No synchronization at all. In mode ME_STOP_MODE_ IMMEDIATE some data can be lost.

- ME_IO_STREAM_STOP_TYPE_PRESERVE_BUFFERS (Linux only)
  - Output subdevice: all buffers are preserved. Streaming can be continued.
  - Input subdevice: Synchronization is done before hardware buffer is cleared. No data are lost.

`<iErrno>`

If an error occurs, an error code will be returned.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_INVALID_STOP_MODE: not supported stop mode detected.
- ME_ERRNO_PREVIOUS_CONFIG: subdevice is not configured correctly to proceed streaming operation.
- ME_STATUS_ERROR: previous operation ended with an error. Reset has to be called to clear this state.

meIOStreamRead

Description:

With this function you can read values from the data buffer during a streaming operation (timer-controlled acquisition).

The user has to allocate a data buffer, to which the measurement values are written to. Using the execution mode „BLOCKING“ the function meIOStreamRead() returns when the last value has been read. In „NON-BLOCKING“ mode the function returns at once with the available measurement values.

For reading the data a callback function can also be used (for installation of callback functions see function *meIOStreamSetCallbacks()* on page 186).

Note: For details please note the chapter „Streaming Operation“ from page 51.

Function Declaration

int meIOStreamRead(int iDevice, int iSubdevice, int iReadMode, int*piValues, int *piCount, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iReadMode>

- ME_READ_MODE_BLOCKING
  The function waits until the number of measurement values specified in parameter <piCount> has been acquired.
  Important: Call can block for ever!
- ME_READ_MODE_NONBLOCKING
  The function returns immediately either with the number of measurement values available when calling the function or with the number specified in <piCount>. Each time the lower value is valid.

<piValues> (r)

Pointer to a data buffer (data stream) to be filled up. Use the function *meUtilityDigitalToPhysical()* for easy conversion into voltage values.

<piCount> (r/w)

(w): Size of the data buffer to be read in number of measurement values.
(r): The pointer returns the number of values actually read from the data

buffer. If in BLOCKING mode the acquisition has been cancelled, the return value can also be less than the value passed.

- BLOCKING mode:
  Number of values to be read – as a rule a multiple of the channel-list length, but this is not necessary (see also *meUtilityExtractValues()* on page 207).
- NONBLOCKING mode:
  Number of values to be read – if you want to read a multiple of channel-list length use the constant ME_IO_STREAM_READ_ FRAMES in the parameter <iFlags>.

<iFlags>

- ME_IO_STREAM_READ_NO_FLAGS Default settings.
- ME_IO_STREAM_READ_FRAMES
  Reading a multiple of channel-list length in NONBLOCKING mode.

Example: If the channel-list has 5 entries and in parameter <piCount> the value „14“ has been passed:

- ME_IO_STREAM_READ_NO_FLAGS: 14 values are returned.
- ME_IO_STREAM_READ_FRAMES: 10 values are returned.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_INVALID_VALUE_COUNT: <piCount> is lower than zero.
- ME_ERRNO_INVALID_READ_MODE: not supported read mode detected.
- ME_ERRNO_SUBDEVICE_NOT_RUNNING: buffer is empty and subdevice is not running a streaming operation.
- ME_ERRNO_HARDWARE_FIFO_OVERFLOW: error during acquisition. Reading data from FIFO was too slow.
- ME_ERRNO_RING_BUFFER_OVERFLOW: no place in buffer for new data. Reading data from buffer was too slow.

meIOStreamWrite

Description:

This function is for writing data to a buffer for timer-controlled analog resp. digital output in the operation mode „streaming“. Allocate for every subdevice to be used a data buffer of defined size for the values to be output.

In dependency of parameter <iFlags> of the function *meIO- StreamConfig()* you can choose between the following options:

a. With the constant ME_IO_STREAM_CONFIG_NO_FLAGS you can output any analog signals continuously. The data buffer must be reloaded periodically with new values which can also be changed after starting the output operation. When loading for the first time use the option ME_WRITE_MODE_PRELOAD in the parameter <iWriteMo-de>.

b. The constant ME_IO_STREAM_CONFIG_BIT_PATTERN is used for bit-pattern output and timer-control of the ME-MultiSig system if it should not be registered with the ME-iDC (in the pipeline). By this constant you connect the timer-control with the digital ports of the ME-4680. See also parameter <iSingleConfig> and <iRef> of the function *meIOSingle-Config().*

c. By the constant ME_IO_STREAM_CONFIG_WRAPAROUND you can output analog signals as well as digital bit patterns periodically. The data buffer must be loaded once with the values to be output. Therefore use the option ME_WRITE_MODE_PRELOAD in the parameter <iWriteMode>. If the number of values in the data buffer does not exceed the FIFO size (depends on hardware) the output operation is running on firmware level. I.e. it is no additional load for the host computer (ME-4680: 4096 values, ME-6100/6300: 8192 values). For periodic bit-pattern output and for periodic DEMUX operation this constant must be ORed with the constant ME_IO_STREAM_CONFIG_BIT_PATTERN.

Note: AO channels with FIFO, which are suitable for timer-controlled output (ME_SUBTYPE_STREAMING) have to be accessed as independent subdevices.

See also chapter 3.4.2 „Streaming Operation“ from page 51

Function Declaration:

int meIOStreamWrite(int iDevice, int iSubdevice, int iWriteMode, int*piValues, int *piCount, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iWriteMode>

- ME_WRITE_MODE_BLOCKING

The function waits until the number of values specified in parameter.<piCount> can be written to the internal buffer. Important: Call can block for ever!

- ME_WRITE_MODE_NONBLOCKING
  With this option the function writes as many values to the internal buffer as there is space at the moment the function was called (max. number specified in <piCount>).
- ME_WRITE_MODE_PRELOAD
  Pre-loading the data buffer for the first time. Data will be written directly to the hardware buffer. If there are more data as there is space in the FIFO the rest is stored in an internal buffer.
  Note: This is non-blocking writing.

<piValues> (w)

Pointer to a data buffer (data stream) with the values resp. bit patterns to be output. Use the function *meUtilityPhysicalToDigital* for easy conversion of physical values (e.g. voltage) into digital values.

<piCount> (r/w)

(w): Number of values to be loaded into the data buffer.

(r): The pointer returns the number of values which could be written into the data buffer actually. The number will never be greater, but can also be less, if there is less memory for the number of values passed.

<iFlags>

- ME_IO_STREAM_WRITE_NO_FLAGS Default settings – no flags available.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_INVALID_VALUE_COUNT: <piCount> is lower than zero.
  Note: When <piCount> is set to zero ME_ERRNO_SUCCESS is returned.
- ME_ERRNO_INVALID_WRITE_MODE: not supported write mode detected.
- ME_ERRNO_PREVIOUS_CONFIG: device is configured to work in single mode.

- ME_ERRNO_HARDWARE_FIFO_UNDERFLOW:
  - General: error during streaming. Writing data to FIFO was too slow.
  - Hardware wraparound: More data than space in FIFO.
- ME_ERRNO_RING_BUFFER_UNDERFLOW: data buffer is empty. Writing to data buffer was too slow.
- ME_ERRNO_SUBDEVICE_NOT_RUNNING: The internal hardware state machine is stopped but logical status show that should be working. No data in FIFO but software buffer is not empty.

meIOStreamStatus

Description:

Checking the status of streaming operation. Depending on input or output operation this function is used to check whether all measurement values have been acquired or whether an output operation is still running.

With the parameter <iWait> you can control, whether the function should return the current state at once, or whether you want to wait until the input resp. output operation has been ended.

Function Declaration:

int meIOStreamStatus(int iDevice, int iSubdevice, int iWait, int *piStatus, int *piCount, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iWait>

Behaviour of return of this function:

- ME_WAIT_NONE
- Check current status. Function returns the current state of operation immediately in parameter <piStatus>.
- ME_WAIT_IDLE
  In case of an output operation the function waits until all values have been output. The function returns with ME_STATUS_IDLE in parameter <piStatus>.
  Important: Call can block for ever!
- ME_WAIT_BUSY (Linux only)
  The function blocks while the status is ME_STATUS_BUSY
  Important: Call can block for ever!

<piStatus> (r)

Pointer which returns the current state of operation of the specified subdevice:

- ME_STATUS_IDLE
  Streaming operation has finished.
- ME_STATUS_BUSY
  Streaming operation is still running.
- ME_STATUS_ERROR
  Error occured, e.g. data stream was interrupted.

<piCount> (r)

- Input subdevice: Number of values which can be read.
- Output subdevice: Free memory in the output buffer (in number of values).

<iFlags>

- ME_IO_STREAM_STATUS_NO_FLAGS Default settings – no flags available.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_INVALID_WAIT: not supported wait mode detected.

meIOStreamNewValues

Description:

Checking the status of streaming operation. Function returns when:

- Input subdevice: there are some values in the buffer.
- Output subdevice: there is empty space in the buffer.

With the parameter <iTimeOut> you can avoid that the function blocks for ever.

Function Declaration

int meIOStreamNewValues(int iDevice, int iSubdevice, int iTimeOut, int*piCount, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iTimeOut>

Time-out value in milliseconds. The function returns within a certain time if the buffer status could not be determined. If you donot want to use a time-out value, pass „0“.

<piCount> (r)

- Input subdevice: Free memory in buffer (number of values).
- Output subdevice: Free memory in buffer (number of values).

<iFlags>

- ME_IO_STREAM_NEW_VALUES_NO_FLAGS Default settings – no flags available.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_TIMEOUT: Timeout.

meIOStreamSetCallbacks

Description:

background for a event on a streaming subdevice. The functions can be called in dependency of the data stream:

- <pStartCB> – executed when streaming starts.
- <pNewValuesCB> – executed when data can be read (input subdevice) or written (output subdevice)
- <pEndCB> – executed when streaming stops.

Note: To deinstall/cancel all registered callback instances for the selected subdevice call *meIOStreamSetCallbacks()* with all callback pointers (<pStartCB>,<pNewValuesCB> and <pEndCB>) set to NULL. It is the same as calling the function *meIOStreamStop().*

Function Declaration

int meIOStreamSetCallbacks(int iDevice, int iSubdevice, meIOStreamCB_t pStartCB, void *pStartCBContext, meIOStreamCB_t pNewValuesCB, void *pNewValuesCBContext, meIOStreamCB_t pEndCB, void *pEndCBContext, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<pStartCB>

Pointer to an user-defined function. This function is called when streaming operation starts. If the function exits with a return value different than ME_ERNNO_SUCCESS (0x00) streaming is instantly stopped (*meIOStreamStop()* is executed).

<pStartCBContext>

User-defined pointer passed to start callback function. This parameter is optional. If you don´t want to use this functionality pass NULL.

<pNewValuesCB>

Pointer to an user-defined function. This function is called when the buffer status is changing. If the function exits with a return value different than ME_ERNNO_SUCCESS (0x00) streaming is instantly stopped *(meIOStreamStop()* is executed.

<pNewValuesCBContext>

User-defined pointer passed to new values callback function. This parameter is optional. If you donot want to use this functionality pass NULL.

<pEndCB>

Pointer to a user-defined function. This function is called when streaming operation stops.

<pEndCBContext>

User-defined pointer passed to stop callback function. This parameter is optional. If you don´t want to use this functionality pass NULL.

<iFlags>

- ME_IO_STREAM_SET_CALLBACKS_NO_FLAGS (no flags available)

‘ **Type Definition** meIOStreamCB_t

```
typedef int (*meIOStreamCB_t)(
```

```
        int iDevice,
        int iSubdevice,
        int iCount,
        void *pvContext,
        int iErrorCode);
```

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iCount>

- Input subdevice: number of values, which can be read.
- Output subdevice: Free memory in the output buffer.

<pvContext> (w)

User-defined pointer to exactly that value, which was passed to this function in the parameter <p…CBContext>. If you don't want to use this parameter pass NULL.

<iErrorCode>

If an error occurs an error code will be returned.

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: no subdevice mapped to requested ID.
- ME_ERRNO_LOCKED: subdevice is protected.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.
- ME_ERRNO_START_THREAD: creating callback thread failed. (Li-nux only).

4.2.4 Auxiliary Functions

meOpen

Description:

- This function initializes the function library:
- Reserve memory.
- Set internal variables.
- Establish connection to driver(s).
- Map detected resources to logical structures.

Else, there is no access to the ME-iDS possible.

Function Declaration:

int meOpen(int iFlags);

<iFlags>

- ME_OPEN_NO_FLAGS

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_OPEN: ME-iDS cannot be properly opened. Usually driver is not loaded.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meCloser

Description:

This function closes the connection to the function library:

- Free used memory.
- Disconnect from driver.

Function Declaration:

int meClose(int iFlags);

<iFlags>

- ME_CLOSE_NO_FLAGS

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_CLOSE: ME-iDS cannot be properly closed.
- ME_ERRNO_INVALID_FLAGS: some of passed flags are not supported.

meLockDriver

Description:

The entire driver system (ME-iDS) will be locked resp. unlocked for access to other threads. If another thread wants to access to the driver system an error message is returned.

Function Declaration:

int meLockDriver(int iLock, int iFlags);

<iLock>

- ME_LOCK_SET
  The driver system will be locked for other threads.
- ME_LOCK_RELEASE
  The locked driver system will be released.
- ME_LOCK_CHECK
  Check the current locking status.

<iFlags>

- ME_LOCK_DRIVER_NO_FLAGS

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_LOCKED: some resources are locked by other application/task.
- ME_ERRNO_USED: some resources are currently in use therefore lock cannot be set.
- ME_ERRNO_INVALID_FLAGS: passed flags are not supported.

meLockDevice

Description:

A device will be locked resp. unlocked as a whole. If another thread wants to access to a locked device an error message is returned.

Function Declaration:

int meLockDevice(int iDevice, int iLock, int iFlags);

<iDevice>

Index of the device to be accessed.

<iLock>

- ME_LOCK_SET
  The device will be locked for access to other threads.

- ME_LOCK_RELEASE
  The locked device will be released.
- ME_LOCK_CHECK
  Check the current locking status.

<iFlags>

- ME_LOCK_DEVICE_NO_FLAGS

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_LOCKED: some resources are locked by other application/task.
- ME_ERRNO_USED: some resources are currently in use therefore lock cannot be set.
- ME_ERRNO_INVALID_FLAGS: passed flags are not supported.

meLockSubdevice

Description:

A subdevice will be locked resp. released. If another thread wants to access to a locked subdevice an error message is returned.

Function Declaration:

int meLockSubdevice(int iDevice, int iSubdevice, int iLock, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice>

Index of the subdevice to be accessed.

<iLock>

- ME_LOCK_SET
  The subdevice will be locked for other threads.
- ME_LOCK_RELEASE
  The locked subdevice will be released.
- ME_LOCK_CHECK
  Check the current locking status.

<iFlags>

- ME_LOCK_SUBDEVICE_NO_FLAGS

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.

- ME_ERRNO_LOCKED: resource is locked by other application/task.
- ME_ERRNO_USED: resource is currently in use therefore lock cannot be set.
- ME_ERRNO_INVALID_FLAGS: passed flags are not supported.

meErrorGetLast

Description:

This function returns the last error code.

Function Declaration:

int meErrorGetLast(int *piErrorCode, int iFlags);

<piErrorCode>

Pointer to the error code.

<iFlags>

- ME_NO_FLAGS: Default setting.
- ME_ERRNO_CLEAR_FLAGS: Do not report this error again.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_INVALID_FLAGS: passed flags are not supported.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meErrorGetLastMessage

Description:

This function returns the last error caused by an API function. A corresponding error text can be displayed.

Function Declaration:

int meErrorGetLastMessage(char *pcErrorMsg, int iCount);

<pcErrorMsg>

Pointer to the error description text.

<iCount>

Buffer size in bytes for the error description text. Use the constant ME_ERROR_MSG_MAX_COUNT.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_INVALID_ERROR_MSG_COUNT: reserved buffer is too small for description.
  Note: This error can be ignored in some cases. Whole available space will be filled with description string.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meErrorGetMessage

Description:

This function converts an error code returned from an API function to plain text.

Function Declaration:

int meErrorGetMessage(int iErrorCode, char *pcErrorMsg, int iCount)

<iErrorCode>

The error code from the API function.

<pcErrorMsg>

Pointer to the error description text.

<iCount>

Buffer size in bytes for the error description text. Use the constant ME_ERROR_MSG_MAX_COUNT.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_INVALID_ERROR_NUMBER: provided error code is not valid in ME-iDS.
- ME_ERRNO_INVALID_ERROR_MSG_COUNT: reserved buffer is too small for description.
  Note: This error can be in some cases ignored. Whole available space will be filled with description string.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meErrorSetDefaultProc

Description:

This function can be used to install a standard (predefined) global error logging routine for the entire ME-iDS. This global error routine is automatically called if any function call returns an error. The following info are returned:

- Name of the function which caused the error.
- Short error description.
- Error code.

Note: Only one global error routine can be installed (…*ErrorSetDefaultProc* or …*ErrorSetUserProc*).

Function Declaration:

int meErrorSetDefaultProc(int iSwitch);

<iSwitch>

- ME_SWITCH_ENABLE
  Installing the predefined error routine for global error logging.
- ME_SWITCH_DISABLE
  Uninstall the predefined error routine.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_INVALID_SWITCH: passed action code is not supported.

meErrorSetUserProc

Description:

This function is used to install a global user-defined error logging routine for the ME-iDS. This function is automatically called if any function call returns an error. The following info are returned:

- Name of the function which caused the error.
- Error code.

Use the function …ErrorGetMessage() to assign an error description to the error code.

Note: Only one global error routine can be installed (…*ErrorSetDefaultProc* or …*ErrorSetUserProc*).

Function Declaration:

int meErrorSetUserProc(meErrorCB_t pErrorProc);

<pErrorProc>

Pointer to a user-defined error logging routine. The name of the faulty function and the error code will be passed to the callback function installed there. Passing a NULL will uninstall a previously installed error routine.

Type Definition meErrorCB_t

```
typedef int (*meErrorCB_t)
        (char *pcFunctionName,
        int iErrorCode);
```

<pcFunctionName>

String with the name of the function where the error was detected.

<iErrorCode>

Error code.

Return Value

No error possible.

meUtilityDigitalToPhysical

Description:

Auxiliary function for easy conversion of the standardized digital values into the appropriate physical unit (voltage, current or temperature). Using this function is optional.

If you read data from an input subdevice in streaming mode, you should apply the function *meUtiltiyExtractValues()* to the array of values before calling this function. Only that way is itguarenteed that different gain factors and the usage of different plug-on modules in combination with the ME-MultiSig system can be taken into account when calculating.

The temperature for:

…RTDs is calculated in accordance to DIN EN 60751

…Thermocouples is calculated in accordance to DIN EN 60584.

If you want to apply the function to a whole array of values we recommend the function *meUtilityDigitalToPhysicalV().*

Note: The parameters <dMin> and <dMax> must correspond with the limits of the measurement range choosen in the functions *meIOSingleConfig()* resp. *meIOStreamConfig().*

The parameters <dMin>, <dMax> and <pdPhysical> must be given always in the same decimal power of the respective base unit (e.g. either „mV“ or „V“).

The physical unit is not relevant for calculation.

Function Declaration:

int meUtilityDigitalToPhysical(double dMin, double dMax, int iMaxData, int iData, int iModuleType, double dRefValue, double *pdPhysical);

<dMin>

The lower limit of the range (from *meQueryRangeInfo()*), e.g. -10[V]. See also note above.

<dMax>

The upper limit of the range (from *meQueryRangeInfo()*), e.g. +10[V]. See also note above.

<iMaxData>

The max. resolution of the range (from *meQueryRangeInfo()*), e.g. 65535 (0xFFFF) at 16-bit resolution.

<iData>

Digital value (0…65535) to be converted.

<iModuleType>

Note: Support for ME-MultiSig is in preparation. Please contact our sales team for further details.

If you are using the ME-MultiSig system in combination with a plug-on module for signal conditioning choose the module type here. In parameter <dMin> pass „-10“, in parameter <dMax> „+10“ and in parameter <iMaxData> „65535“. This is important for a correct calculation of the measurement value. If for the current calculation no plug-on module must be taken into account, pass the constant:

- ME_MODULE_TYPE_MULTISIG_NONE
  No plug-on module used (standard).
- ME_MODULE_TYPE_MULTISIG_DIFF16_10 V
  Plug-on module ME-Diff16 with input range 10 V.
- ME_MODULE_TYPE_MULTISIG_DIFF16_20V Plug-on module ME-Diff16 with input range 20 V.
- ME_MODULE_TYPE_MULTISIG_DIFF16_50 V Plug-on module ME-Diff16 with input range 50 V.
- ME_MODULE_TYPE_MULTISIG_CURRENT16_0_20 MA
  Plug-on module ME-Current16 with input range 0…20 mA.
- ME_MODULE_TYPE_MULTISIG_RTD8_PT100

Plug-on module ME-RTD8 for RTDs of type Pt100 (0,4 Ω/K).

- ME_MODULE_TYPE_MULTISIG_RTD8_PT500
  Plug-on module ME-RTD8 for RTDs of type Pt500 (2,0 Ω/K).
- ME_MODULE_TYPE_MULTISIG_RTD8_PT1000
  Plug-on module ME-RTD8 for RTDs of type Pt1000 (4,0 Ω/K).
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_B
  Plug-on module ME-TE8, channel with thermocouple type B.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_E
  Plug-on module ME-TE8, channel with thermocouple type E.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_J
  Plug-on module ME-TE8, channel with thermocouple type J.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_K
  Plug-on module ME-TE8, channel with thermocouple type K.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_N
  Plug-on module ME-TE8, channel with thermocouple type N.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_R
  Plug-on module ME-TE8, channel with thermocouple type R.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_S
  Plug-on module ME-TE8, channel with thermocouple type S.
- ME_MODULE_TYPE_MULTISIG_TE8_TYPE_T
- Plug-on module ME-TE8, channel with thermocouple type T.
- ME_MODULE_TYPE_MULTISIG_TE8_TEMP_SENSOR
- Plug-on module ME-TE8, channel for cold junction compensation at the terminal of the module.
- ME_MODULE_TYPE_MULTISIG_BA4_DMS120
  Plug-on module ME-BA4, channel with 120 Ω nominal strain gauge resistance.
- ME_MODULE_TYPE_MULTISIG_BA4_DMS350
  Plug-on module ME-BA4, channel with 350 Ω nominal strain gauge resistance.
- ME_MODULE_TYPE_MULTISIG_BA4_DMS1000
  Plug-on module ME-BA4, channel with 1 kΩ nominal strain gauge resistance.

<dRefValue>

- Default setting: ME_VALUE_NOT_USED.
- If you have choosen a RTD module in parameter <iModuleType>:
  For exact calculation of temperature you must pass the constant measurement current IM in ampere [A]. It must be measured before with a high-precision amperemeter (see manual ME-MultiSig system).
- If you want to ignore the measurement tolerance the function calculates with a typical constant measurement current of $I_M = 500 \times 10^{-6}$ A.
  In that case pass the constant:
  ME_REFVALUE_MULTISIG_I_MEASURED_DEFAULT.

- If you have choosen a thermo-couple module in parameter <iModule-Type>:
  Because the calculation refers to a cold-junction temperature of 0 °C you must measure the temperature at the connector STM1 of the plug-on module with the integrated sensor before you start the series of measurements (see parameter <iModuleType>). Next, the determined value is passed by this parameter (in °C). Parameter returns a pointer to the compensated temperature value in °C.

<pdPhysical>

Result in the corresponding physical unit [V], [A], [°C].

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_VALUE_OUT_OF_RANGE: passed value is lower than “0” or bigger than <iMaxData>.
- ME_ERRNO_INVALID_MIN_MAX: passed <iMaxData> is not valid.
- ME_ERRNO_INVALID_MODULE_TYPE: passed moduletype is not supported.
- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meUtilityDigitalToPhysicalIV

Description:

In opposite to the function *meUtilityDigitalToPhysical()* this function can be applied to a whole array of values. For it the parameter <iCount> was added to the function declaration. Else the description of the function *meUtilityDigitalToPhysical()* is valid in an analogue way.

Function Declaration:

int meUtilityDigitalToPhysicalV(double dMin, double dMax, int iMaxData, int *piDataBuffer, int iCount, int iModuleType, double dRefValue, double *pdPhysicalBuffer);

<dMin>

The lower limit of the range (from *meQueryRangeInfo()).*

<dMax>

The upper limit of the range (from *meQueryRangeInfo()).*

<iMaxData>

The max. resolution of the range (from *meQueryRangeInfo()).*

<piDataBuffer> (w)

Pointer to an array of digital values to be converted.

<iCount>

Number of values in the array.

<iModuleType>

See function *meUtilityDigitalToPhysical().*

<dRefValue>

See function *meUtilityDigitalToPhysical().*

<pdPhysicalBuffer> (r)

Pointer to an array for the results in the appropriate physical unit [V], [A], [°C]. Has to be able to store <iCount> values.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_VALUE_OUT_OF_RANGE: passed value is lower than “0” or bigger than <iMaxData>.
- ME_ERRNO_INVALID_MIN_MAX: <iMaxData> not valid.
- ME_ERRNO_INVALID_MODULE_TYPE: passed module type is not supported.
- • ME_ERRNO_INVALID_POINTER: passed pointers are NULL.

meUtilityPhysicalToDigital

Description:

Auxiliary function allows simple conversion of values (voltage or current) to be output in standardized digital values which are appropriate for the converter. Using this function is optional.

If you want to apply the function to a whole array of values we recommend the function *meUtilityPhysicalToDigitalV().*

Note: The parameters <dMin> and <dMax> must correspond with the limits of the measurement range choosen in the functions *meIOSingleConfig()* resp. *meIOStreamConfig().*

The parameters <dMin>, <dMax> and <dPhysical> must be given always in the same decimal power of the respective base unit (e.g. either „mV“ or „V“).

The physical unit is not relevant for calculation.

Function Declaration:

int meUtilityPhysicalToDigital(double dMin, double dMax, int iMaxData, double dPhysical, int *piData);

<dMin>

The lower limit of the range (from *meQueryRangeInfo()),* e.g. -10[V]. See also note above.

<dMax>

The upper limit of the range (from *meQueryRangeInfo()),* e.g. +10[V]. See also note above.

<iMaxData>

The maximum resolution of the range (from *meQueryRangeInfo()),* e.g. 65535 (0xFFFF) at 16-bit resolution.

<dPhysical>

Voltage or current value to be converted, e.g. +0,75[V].

<piData>

Result as a digital value to be output.

Return Value

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_VALUE_OUT_OF_RANGE: passed value is lower than <dMin> or bigger than <dMax>.
- ME_ERRNO_INVALID_MIN_MAX: <iMaxData> not valid.

- ME_ERRNO_INVALID_POINTER: passed pointer is NULL.

meUtilityPhysicalToDigitalV

Description:

In opposite to the function *meUtilityPhysicalToDigital()* this function can be applied to a whole array of values. For it the parameter <iCount> was added to the function declaration. Else the description of the function *meUtilityPhysicalToDigital()* is valid in an analogue way.

Function Declaration:

int meUtilityPhysicalToDigitalV(double dMin, double dMax, int iMaxData, double *pdPhysicalBuffer, int iCount, int *piDataBuffer);

<dMin>

The lower limit of the range (from *meQueryRangeInfo()).*

<dMax>

The upper limit of the range (from *meQueryRangeInfo()).*

<iMaxData>

The maximum resolution of the range (from *meQueryRangeInfo()).*

<pdPhysicalBuffer> (w)

Pointer to an array with the voltage or current values to be converted, e.g. +0,75 [V].

<iCount>

Number of values in the array.

<piDataBuffer> (r)

Pointer to an array with the digital values to be output. Has to be able to store <iCount> values.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_VALUE_OUT_OF_RANGE: passed value is lower than <dMin> or bigger than <dMax>.
- ME_ERRNO_INVALID_MIN_MAX: <iMaxData> not valid.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.

meUtilityExtractValues

Description:

Auxiliary function extracts the values of the specified channel from an array of values, allocated by the function *meIOStreamRead()* taking into account the channel-list. To extract the channels for several channels the function must be called separately.

Note: If channel is not on list the function returns ME_ERRNO_ SUCCESS and <piChanBufferCount> is set to “0”.

Function Declaration:

int meUtilityExtractValues(int iChannel, int *piAIBuffer, int iAIBufferCount, meIOStreamConfig_t *pConfigList, int iConfigListCount, int *piChanBuffer, int *piChanBufferCount);

<iChannel>

Channel index whose values should be extracted.

<piAIBuffer> (w)

Pointer to the data buffer allocated by the function *meIOStreamRead().*

<iAIBufferCount>

Number of measurement values in data buffer <piAIBuffer>.

<pConfigList> (w)

Pointer to the channel-list, which was passed to the function *meIOStreamConfig().*

<iConfigListCount>

Number of channel-list entries in <pConfigList>.

<piChanBuffer> (r)

Pointer to an array with the extracted values of the specified channel.

<piChanBufferCount> (r/w)

(w) : Passing the size of the array <piChanBuffer> in number of values.

(r) : The function returns the number values actually written to <piChanBuffer>.

Type Definition meIOStreamConfig_t

Type definition see function *meIOStreamConfig()* from page 156.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_INVALID_POINTER: passed pointers are NULL.

meUtilityPWMStart

Description:

This auxiliary function configures the counter device 8254 (ME_SUB-TYPE_CTR_8254) for the operation mode „pulse width modulation“ (PWM) and starts the operation. A further usage of the counters 0…2 is not possible in this operation mode. The signal is available at OUT_2 of the specified counter device. Depending on device type the base clock (max.10 MHz) must be provided externally or (if supported by the hardware) an on-board crystal oszillator can be used. Counter 0 is used as a prescaler. The frequency of the output signal is max. 50 kHz and can be calculated as follows:

$$f_{OUT\_2} = \frac{\text{Base clock}}{\texttt{<iPrescaler>} \cdot 100} \quad \text{(with <iPrescaler> = 2…(}2^{16}\text{– 1))}$$

The duty cycle can be set between 1…99 % in steps of 1 % (see diagram 32 on page 181).

Note: Using this function is only meaningful in combination with the external switching shown in diagram 31 on page 217.

Funktion Declaration:

int meUtilityPWMStart(int iDevice, int iSubdevice1, int iSubdevice2, int iSubdevice3, int iRef, int iPrescaler, int iDutyCycle, int iFlags);

<iDevice>

Index of the device to be accessed.

<iSubdevice1>

Index of the subdevice counter 0 (used as prescaler).

<iSubdevice2>

Index of the subdevice counter 1.

<iSubdevice3>

Index of the subdevice counter 2.

<iRef>

Defines the clock source for counter 0 (CLK_0):

- ME_REF_CTR_INTERNAL_1 MHZ
  Clock source is the internal 1 MHz crystal oszillator.
- ME_REF_CTR_INTERNAL_10 MHZ

Clock source is the internal 10 MHz crystal oszillator.

- ME_REF_CTR_EXTERNAL
  Clock source is an external oszillator.

`<iPrescaler>`

Value for the prescaler (counter 0) in the range 2…65535.

`<iDutyCycle>`

Duty cycle of the output signal from 1 % –99 % adjustable in steps of 1 %.

`<iFlags>`

Flag for extended options:

- ME_PWM_START_NO_FLAGS: default settings
- ME_PWM_START_CONNECT_INTERNAL
  If supported by hardware (e.g. ME-1400 series), connect OUT_1 with GATE_2 internally. This reduces the number of external connections.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_INVALID_FLAGS: not supported flag detected.
- ME_ERRNO_INVALID_REF: used signal source not available.
- ME_ERRNO_INVALID_DUTY_CYCLE: value outside of supported range.
- ME_ERRNO_NOT_SUPPORTED:
  - used subdevice is not a counter.
  - internal connection can not be done.

meUtilityPWMStop

Description:

With this function a PWM operation started by the function *meUtilityPWMStart()* is ended.

Function Declaration:

int meUtilityPWMStop(int iDevice, int iSubdevice1);

`<iDevice>`

Index of the device to be accessed.

<iSubdevice1>

Index of the subdevice counter 0 used as prescaler.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_NOT_SUPPORTED: used subdevice is no counter.

meUtilityPWMRestart

Description:

Auxiliary function to restart a PWM operation stopped by *meUtilityPWMStop()*. The prescaler can be loaded with a new value to set another frequency. The duty cycle cannot be changed here.

Note: Using this function is only meaningful in combination with the external switching shown in diagram 31 on page 180. The operation starts where it was stopped. No reset!

Function Declaration:

int meUtilityPWMRestart(int iDevice, int iSubdevice1, int iRef, int iPrescaler);

<iDevice>

Index of the device to be accessed.

<iSubdevice1>

Index of the subdevice counter 0 (used as prescaler).

<iRef>

Defines the clock source for counter 0 (CLK_0):

- ME_REF_CTR_INTERNAL_1MHZ
  Clock source is the internal 1 MHz crystal oszillator.
- ME_REF_CTR_INTERNAL_10MHZ
  Clock source is the internal 10 MHz crystal oszillator.
- ME_REF_CTR_EXTERNAL
  Clock source is an external oszillator.

<iPrescaler>

Value for the prescaler (counter 0) in the range 2…65535.

Return Value:

- ME_ERRNO_SUCCESS: Function returned successfully.
- ME_ERRNO_NOT_OPEN: ME-iDS is not properly open.
- ME_ERRNO_INVALID_DEVICE: no device mapped to requested ID.
- ME_ERRNO_INVALID_SUBDEVICE: on requested device no subdevice mapped to requested ID.
- ME_ERRNO_NOT_SUPPORTED: used subdevice is no counter.
- ME_ERRNO_INVALID_REF: used signal source not available.

5 Appendix

A Special Operation Modes

A1 Operation Modes 8254

The ME-iDS supports the standard counter chip of type 8254 providing three 16-bit counters which can be configured independently of each other for the following 6 operation modes (see also data-sheet of the manufacturer). Each counter is a subdevice of type ME_TYPE_CTR, sub-type ME_SUBTYPE_CTR_8254 betrachtet.

- Mode 0: Change state at zero (ME_SINGLE_CONFIG_CTR_8254_MODE_0)
- Mode 1: Retriggerable „One Shot“ (ME_SINGLE_CONFIG_CTR_8254_MODE_1)
- Mode 2: Asymmetric divider (ME_SINGLE_CONFIG_CTR_8254_MODE_2)
- Mode 3: Symmetric divider (ME_SINGLE_CONFIG_CTR_8254_MODE_3)
- Mode 4: Counter start by software trigger (ME_SINGLE_CONFIG_CTR_8254_MODE_4)
- Mode 5: Counter start by hardware trigger (ME_SINGLE_CONFIG_CTR_8254_MODE_5)

Note: The real voltage level at the inputs/outputs of the counters depends on the respective hardware. E.g. on opto-isolated versions of the ME-4600 series a high level at the output corresponds to the „high-impedance“ state and a low-level to the state „driving“. Please note the corresponding hardware manual. The logic levels in the following description apply for the counter chip without further circuitry.

Mode 0: Change State at Zero

This mode of operation can be used e.g. to trigger an interrupt when the counter meets zero. The counter output (OUT_0…2) is set to low when the counter is initialised or when a new start value is loaded. To enable counting, a high level must be applied to the GATE input. As soon as the start value is loaded and the counter is enabled, the counter begins counting downwards and the output remains low.

Upon zero axis crossing, the output is set to high and remains there until the counter is reloaded or initialised again. The counter continues to count down, even after zero is met. If a counter register is loaded during a count in progress the following occurs:

1. when the first byte is written, the count process is stopped.
2. when the second byte is written, the count process begins again.

Mode 1: Retriggerable „One-Shot“

The counter output (OUT_0…2) is set to high when the counter is initialised. When a start value is loaded the output becomes low on the next clock following to the first trigger pulse at the GATE input (positive edge). Upon zero axis crossing, the counter output is set to high again.

On a positive edge at the GATE input, the counter can be reset (re-triggered) to the start value. The output remains low until the counter meets zero.

The counter value can be read at any time without effecting the count process.

Mode 2: Asymmetric Divider

In this mode, the counter is used as a frequency divider. The counter output (OUT_0…2) is set to high after initialisation. When the counter is enabled by applying a high level to the GATE input, the counter is counting downwards and the output remains high. When the count meets the value 0001Hex, the output becomes low for one clock cycle. This process will be repeated periodically as long as the GATE input is enabled (high level), else the output is set to high immediately.

If the counter register is reloaded between two output pulses, the current counter state is not affected. However the new value is used on the next period.

Mode 3: Symmetric Divider

This mode of operation is similar to mode 2 with the difference that the divided frequency is symmetric (only for even count values). The counter output (OUT_0…2) is set to high after initialisation. When the GATE input is enabled (high level), the counter is counting downwards in steps of 2. The output will toggle its state on a half of the start value number of periods referenced to the input clock (starting with high level). This process will be repeated periodically as long as the GATE input is enabled (high level), else the output is set to high immediately.

If the counter is reloaded between two output pulses, the current counter state is not affected. The new value is used on the next period.

Mode 4: Counter Start by Software Trigger

The counter output (OUT_0…2) is set to high when the counter is initialised. To enable the counter the GATE input must be enabled (high level). When the counter is loaded (software trigger) and enabled, the counter starts counting downwards, while the output remains high.

Upon zero axis crossing the output becomes low for one clock cycle. Afterwards the output becomes high again and remains there until the counter is initialised and a new start value is loaded.

If the counter is reloaded during a count process, the new start value is used in the next cycle.

Mode 5: Counter Start by Hardware Trigger

The counter output (OUT_0…2) is set to high when the counter is initialised. After loading a start value to the counter, counting starts on the next clock following to the first trigger pulse at the GATE input (positive edge). Upon zero axis crossing, the output becomes low for one clock cycle. Afterwards, the output becomes high again and remains there until the next trigger pulse occurs.

If the count register is reloaded between two trigger pulses, the new start value is used after the next trigger pulse.

The counter can be reset to the start value (re-triggered) at any time by applying a positive edge to the GATE input. The output remains high until zero axis crossing is met

A2 Pulse Width Modulation

A special application for the counters of type 8254 is the output of a rectangular signal with a variable duty cycle (operation mode „PWM“). With that you can output a rectangular signal of maximum 50 kHz with a variable duty cycle to OUT_2. Condition is a correct switching of inputs and outputs (CLK, GATE, OUT) by the external circuitry (see diagram 31 for TTL I/Os). For opto-isolated counters read the appropriate chapter for PWM switching of opto-isolated counters in the hardware manual.

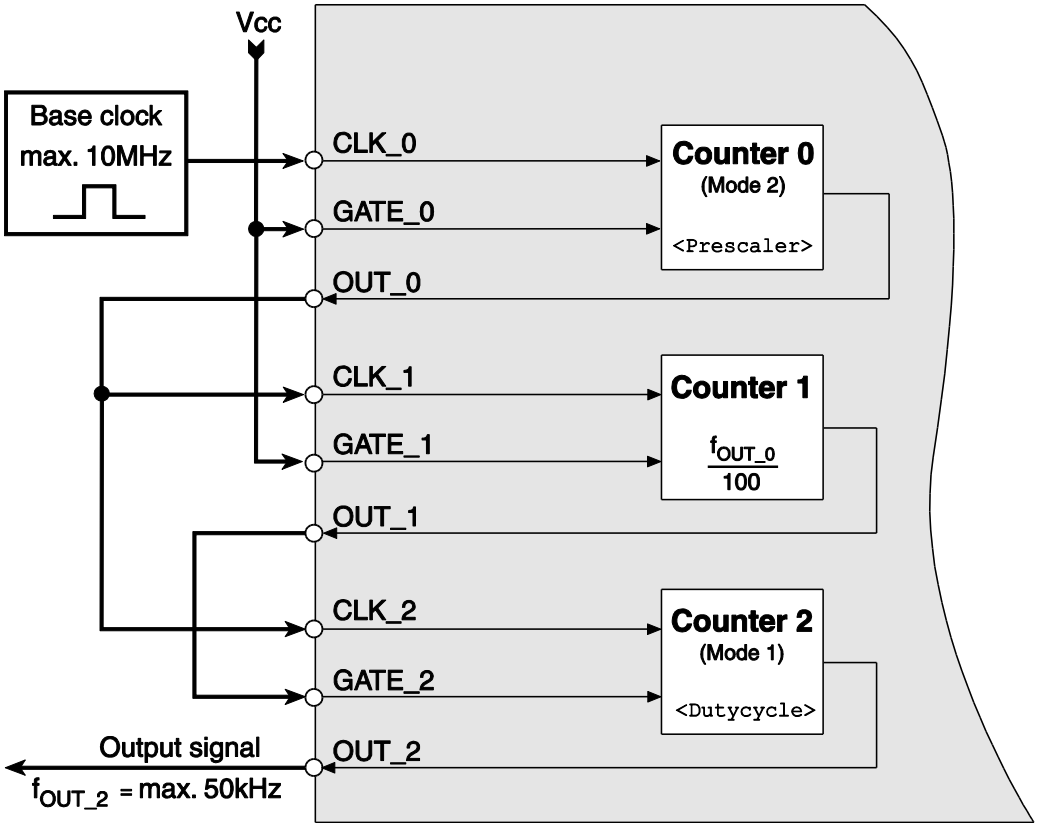

Diagram 31: PWM switching for TTL I/Os

Counter 0 is used as a prescaler for the externally driven base clock. Using the parameter <`iPrescaler`> you can vary the frequency f_{OUT_2} as follows:

$$f_{OUT\_2} = \frac{\text{Base clock}}{\texttt{<iPrescaler>} \cdot 100} \quad \text{(with <iPrescaler> = 2…(}2^{16}\text{– 1))}$$

By the parameter <`iDutyCycle`> you can set the duty cycle between 1…99 % in steps of 1 % (see diagram 32). The operation is started immediately after calling the function *meUtiltiyPWMStart()* and stopped by the function meUtiltiyPWMStop(). No further programming of the counters is required.

On opto-isolated devices the output OUT_2 is an open collector output as a rule. It means a logical „1“ is driving the output and a logical „0“ sets the output in a high impedance state (see also hardware manual).

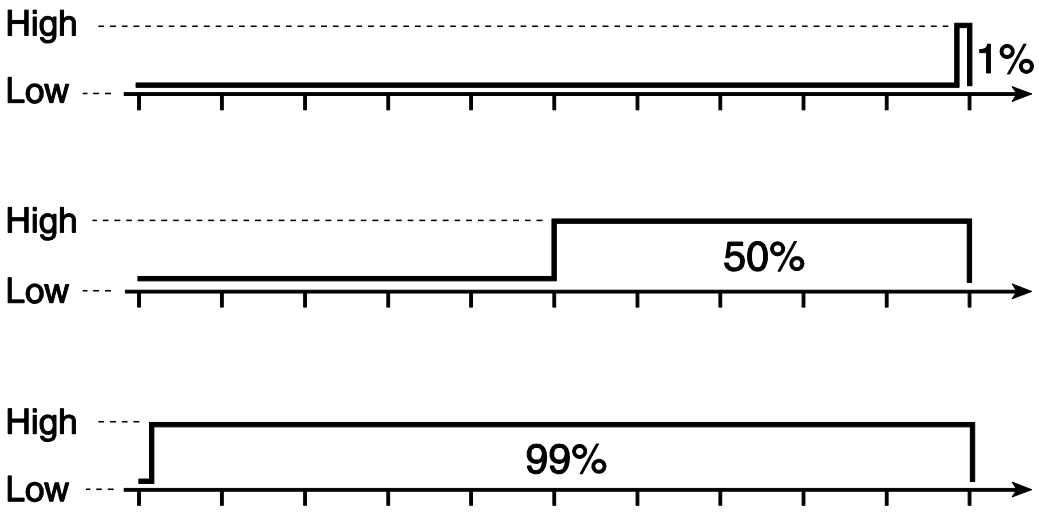

Diagram 32: Duty cycle PWM signal

A3 Bit-Pattern Output of ME-4680

This chapter describes some particularities of the timer controlled bit-pattern output as it is implemented on the ME-4680. The FIFO from AO channel 3 serves a special purpose for doing this. Separated into low byte and high byte, the 16-bit-wide FIFO values (= bit patterns) can be assigned by byte to the 8-bit-wide digital ports (subdevice 0, 1, 2, 3). See diagram 33 on page 220. A port used for bit-pattern output is automatically configured as output. The input port B (subdevice 1) on opto-isolated boards cannot be used for bit-pattern output.

Programming is done in operation mode streaming. A digital port used for bit-pattern output must be a subdevice of type ME_TYPE_DO or ME_TYPE_DIO, subtype ME_SUBTYPE_STREAMING. The following parameters can be configured by the functions *meIOSingleConfig()* and *meIOStreamConfig().*

- Configure one or several digital output ports for the timer-controlled bit-pattern output with the constant ME_SINGLE_CONFIG_ DIO_BIT_PATTERN in parameter <`iSingleConfig`> of function *meIOSingleConfig().*
- Assignment of low-byte and high-byte of the 16-bit-wide FIFO values to the specified digital port with the constants ME_REF_FIFO_LOW resp. ME_REF_FIFO_HIGH in parameter <`iRef`> of function *meIOSingleConfig().*
- The subdevice of AO channel 3 (subdevice with index 11 of type ME_TYPE_AO) is configured for bit-pattern output with the constant ME_IO_STREAM_CONFIG_BIT_PATTERN in parameter <`iFlags`> of the function *meIOStreamConfig().*
- As a reference the constant ME_REF_AO_GROUND must be used in parameter <`iRef`> of the function *meIOStreamConfig().* However use the ground pins of the digital I/O section (PC_GND resp. DIO_GND) for ground reference (not the AO section´s ground).
- Trigger channel, trigger type and trigger edge are defined by the trigger structure `meIOStreamTrigger` of the function *meIOStreamConfig().*
- A programmable counter serves as timer which is configured by the trigger structure `meIOStreamTrigger`. The 32-bit counter uses a 33 MHz base frequency. This results in a period of 30.30 ns, which is the smallest time unit available. This will be referred to as “1 Tick” in the following sections. The functions *meIOStreamFrequencyTo-Ticks()* and *meIOStreamTimeToTicks()* offer a convenient way to convert the frequency resp. the period into ticks for programming the timer.

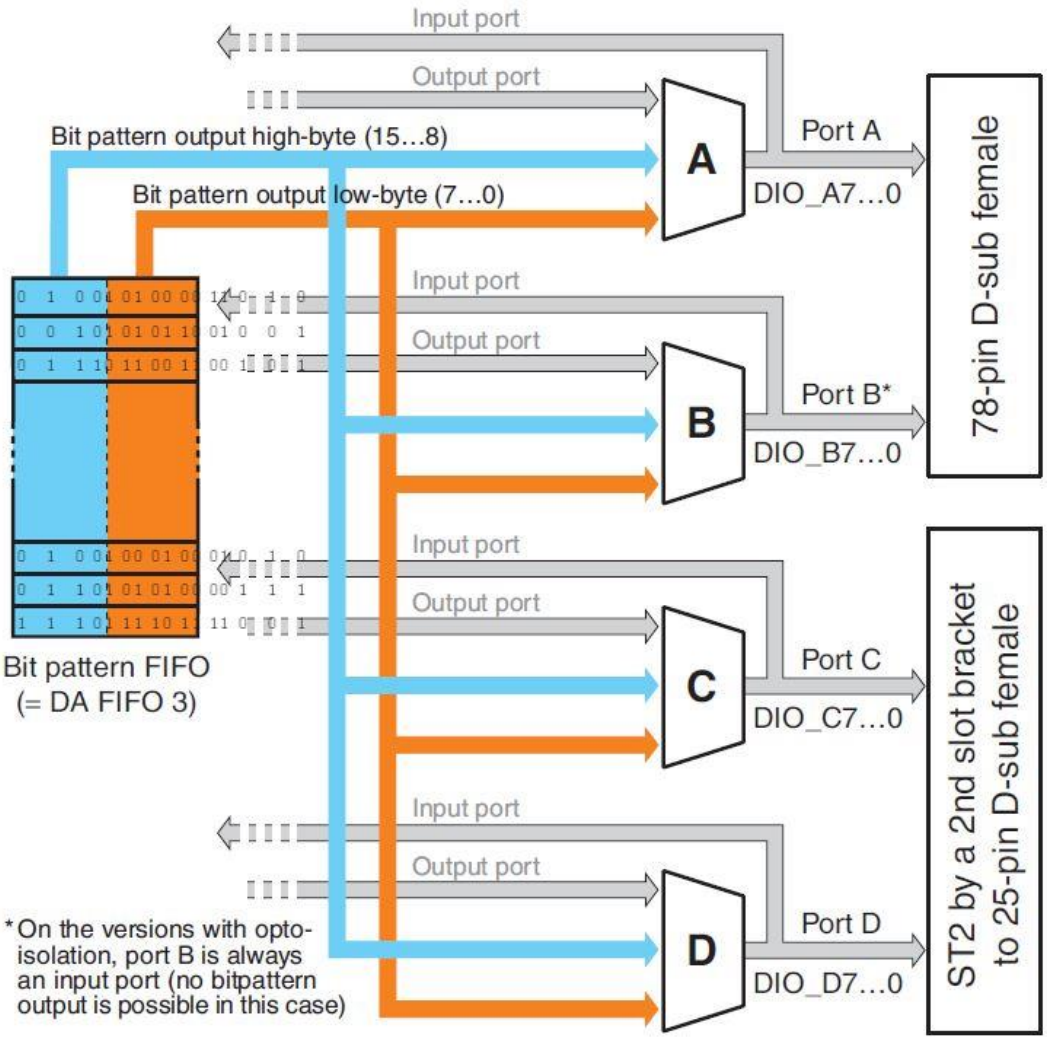

Diagram 33: Port mapping

A4 MEphisto Scope

Special features and limitations

For a demonstration of how to use the special features of the MEphisto Scope in streaming mode, please see the source code of the sample program 'Con_meIOMEphistoScopeStreamRead' (C++ test program) included in the SDK.

Using the function *meIOSetChannelOffset() the* anlog input ranges can be adjusted. This makes it possible to measure an incoming potential very accurately if you know roughly in advance the minimum and maximum values which are likely to occur. Currently this only applies to streaming mode, in single mode the ranges have a fixed offset of 0.0 Volts. If you use the function *meIOSetChannelOffset()* to change an offset range for a streaming operation then you must adjust the final result accordingly by adding this same offset.

In contrast to other ME-iDS devices, the MEphisto Scope can only be used by one application at a time. If one or more MEphisto Scopes (UM202, UM203) are present in a system, then the first application using the ME-iDS which is started will 'see' these in a listing of all devices. As long as this first application is running, the MEphisto Scope devices will be invisible to subsequent applications and cannot be addressed by them. Only when the first application is terminated will the MEphisto Scope devices be visible to the next ME-iDS application which starts.

The digital input/output subdevice (subdevice 0) and the analog input subdevice (subdevice 1) are not completely independent of one another. If a streaming operation is running on the analog subdevice, then an attempt to use the digital input/output subdevice will cause an error (subdevice busy, error 33). During any other operation on one of the subdevices, any other thread trying to use either of the subdevices will be queued and will only gain accesss to the subdevice after the first operation has terminated.

In single mode the result returned on calling *meIOSingle()* is the average of a large number of measurements taken over a period of approximately 0.9 seconds. During this period, any other thread trying to use either subdevice will be queued as explained in the paragraph above.

A5 ME-MultiSig Control

An extensive support of the ME-MultiSig system is in the pipeline (please ask our sales team for further details). Because of that the description in this chapter is to be considered as preliminary.

To understand the ME-MultiSig system it is strongly recommended to fully read the ME-MultiSig manual beforehand! In the following paragraphs you find some notes concerning the programming in combination with the ME-iDS:

- With the ME-iDS the ME-MultiSig system can be programmed fully transparently. „Transparent“ means, e.g. you can access to a MUX-chain which consists of one master and one slave board similar to an AI subdevice (type ME_TYPE_AI) with 64 channels.
- Condition for the transparent programming is the „registration“ of the used base boards (ME-MUX32-M(aster), ME-MUX32-S(lave), ME-DE-MUX32) within the ME-iDC. There you determine number and type of the base boards. A MUX-chain can consist of maximum one ME-MUX32-M and up to seven ME-MUX32-S or one ME-DEMUX32.
- Make sure that the ME-MultiSig base boards of type ME-MUX32-M/S are configured for “Single-Mux” operation. See manual ME-MultiSig system.
- The AI-channel of the board to be used for the MUX-chain must be set in the ME-iDC (default: AI-channel 0). The channel number must correspond with the solder-bridge “A” on the master board (ME-MUX32-M). See manual ME-MultiSig system.
- For using the full feature range in MUX-operation a subdevice of type „DIO“ or „DO“ with a minimum of 16 digital output lines is required. The output line can be selected of a pool of appropriate ports within the ME-iDS. **Note** that the wiring must correspond with it.
- The conversion of the digital values in the accordingly physical unit (voltage, current, temperature) and reverse is done by the functions meUtilityDigitalToPhysical resp. meUtilityPhysicalToDigital in view of the used plug-on modules.
- **Note** when configuring, that the amplification factor within a channel group of the base boards ME-MUX32-M(aster) and ME-MUX32-S(lave) must be the same (default: G = 1). See functions *meIOSingleConfig* resp. *meIOStreamConfig.*
- The address LED of the base boards ME-MUX32-M and ME-MUX32-S can be controlled by the parameter <`Flags`> of the function *meIOSingleConfig.*
- All master and slave boards can be reset to their default state with the functions meIOResetDevice resp. meIOResetSubdevice (Gain G=1, address LED off).

- Using the operation modes „Stream-Input“ resp. „Stream-Output“ in combination with the ME-MultiSig system is only possible after registration of the base boards choosing the option „Use streaming operation mode“ in the ME-iDC. For this operation mode you need a multi-I/O board of type ME-4680. In that case AO channel 3 is locked for timer-controlled output.
- If you register a MUX base board of type Typ ME-MUX32-M with the ME-iDC you cannot access to the remaining AI channels of the board. In that case the AI subdevice includes only the channels of the MUX-chain (max. 256 channels).
- Connection and transparent programming of a base board of type ME-SIG32 is possible without registration at the ME-iDC.
- If you don’t want to program transparently you can program all operation modes with the ME-iDS as described in the ME-MultiSig manual „on foot“.

B Subdevice Caps

B1 Caps in meQuerySubdeviceCaps()

The capabilities of the queried subdevices are returned by the parameter <piCaps> of the function *meQuerySubdeviceCaps().* If several „caps“ apply, the values are ORed bit by bit. E.g.: a subdevice provides a digital trigger input which triggers alternatively on a rising, falling or any (i.e. rising or falling) edge. The returned value is: 0x000E8000.

Definitions	Description	Hex-Value
ME_CAPS_NONE	No special capabilites	0x00000000
Analog Acquisition		
ME_CAPS_AI_TRIG_SYNCHRONOUS	Analog acquisition can be started synchronously	0x00000001
ME_CAPS_AI_TRIG_SIMULTANEOUS		0x00000002
ME_CAPS_AI_FIFO	AI-FIFO available	0x00000004
ME_CAPS_AI_FIFO_THRESHOLD	Threshold for reading the AI-FIFO	0x00000008
ME_CAPS_AI_SAMPLE_HOLD	adjustable	0x00008000
ME_CAPS_AI_TRIG_DIGITAL	„Sample & Hold“ unit available	0x00010000
ME_CAPS_AI_TRIG_ANALOG	Digital trigger input	0x00020000
ME_CAPS_AI_TRIG_EDGE_ RISING	Analoger Triggereingang	0x00040000
ME_CAPS_AI_TRIG_EDGE_ FALLING	Trigger on rising edge	0x00080000
ME_CAPS_AI_TRIG_EDGE_ ANY	Trigger on falling edge	0x00000010
ME_CAPS_AI_DIFFERENTIAL	Trigger on any edge	0x00000001
Analog Output		
ME_CAPS_AO_TRIG_ SYNCHRONOUS	Analog output can be started synchronously	0x00000001
ME_CAPS_AO_TRIG_ SIMULTANEOUS		
ME_CAPS_AO_FIFO	AO-FIFO available	0x00000002

ME_CAPS_AO_FIFO_THRESHOLD	Threshold for re-loading the AO-FIFO adjustable	0x00000004
ME_CAPS_AO_TRIG_DIGITAL	Digital trigger input	0x00008000
ME_CAPS_AO_TRIG_ANALOG	Analog trigger input	0x00010000
ME_CAPS_AO_TRIG_EDGE_ RISING	Trigger on rising edge	0x00020000
ME_CAPS_AO_TRIG_EDGE_ FALLING	Trigger on falling edge	0x00040000
ME_CAPS_AO_TRIG_EDGE_ANY	Trigger on any edge	0x00080000
ME_CAPS_AO_DIFFERENTIAL	Differential output possible	0x00000010
Digital I/O		
ME_CAPS_DIO_DIR_BIT	Direction can be set per bit	0x00000001
ME_CAPS_DIO_DIR_BYTE	Direction can be set per byte (8-bit block)	0x00000002
ME_CAPS_DIO_DIR_WORD	Direction can be set per word (16-bit block)	0x00000004
ME_CAPS_DIO_DIR_DWORD	Direction can be set per long-word (32-bit block)	0x00000008
ME_CAPS_DIO_SINK_SOURCE	Sink/Source switch-over	0x00000010
ME_CAPS_DIO_BIT_PATTERN_ IRQ	IRQ on bit-pattern match	0x00000020
ME_CAPS_DIO_BIT_MASK_IRQ_ EDGE_RISING	IRQ on a rising edge of at least one active bit	0x00000040
ME_CAPS_DIO_BIT_MASK_IRQ_ EDGE_FALLING	IRQ on a falling edge of at least one active bit	0x00000080
ME_CAPS_DIO_BIT_MASK_IRQ_ EDGE_ANY	IRQ on any edge of at least one active bit	0x00000100
ME_CAPS_DIO_OVER_TEMP_IRQ	IRQ on over-heating of the driver chip	0x00000200
ME_CAPS_DIO_NORMAL_TEMP_ IRQ	IRQ on cooling down to normal temperature of driver chip	0x00000400

ME_CAPS_DIO_TRIG_ SYNCHRONOUS	Digital input/output can be started synchronously	0x00004000
ME_CAPS_DIO_TRIG_DIGITAL	Digital trigger input	0x00008000
ME_CAPS_DIO_TRIG_ANALOG	Analog trigger input	0x00010000
ME_CAPS_DIO_TRIG_EDGE_ RISING	Trigger on rising edge	0x00020000
ME_CAPS_DIO_TRIG_EDGE_ FALLING	Trigger on falling edge	0x00040000
ME_CAPS_DIO_TRIG_EDGE_ANY	Trigger on any edge	0x00080000
Frequency I/O		
ME_CAPS_FIO_SINK_SOURCE	Sink/Source switch-over	0x00000010
Counter		
ME_CAPS_CTR_CLK_PREVIOUS	CLK can be sourced by OUT of the previous counter	0x00000001
ME_CAPS_CTR_CLK_INTERNAL_1MHZ	Counter can be sourced by an internal 1 MHz clock	0x00000002
ME_CAPS_CTR_CLK_INTERNAL_10MHZ	Counter can be sourced by an internal 10 MHz clock	0x00000004
ME_CAPS_CTR_CLK_EXTERNAL	Counter can be sourced via an external clock input	0x00000008
External Interrupt		
ME_CAPS_EXT_IRQ_EDGE_ RISING	External IRQ can be triggered on a rising edge	0x00000001
ME_CAPS_EXT_IRQ_EDGE_ FALLING	External IRQ can be triggered on a rising edge	0x00000002
ME_CAPS_EXT_IRQ_EDGE_ANY	External IRQ can be triggered on any edge	0x00000004

Table 14: Caps in meQuerySubdeviceCaps

B2 Caps in meQuerySubdeviceCapsArgs()

The value of the queried “cap” of a subdevice will be returned in parameter <piArgs> of the function *meQuerySubdeviceCapsArgs().*

Definitions	Description	Hex-Value
Analog Acquisition		
ME_CAP_AI_FIFO_SIZE	Query the AI-FIFO size	0x001D0000
ME_CAP_AI_BUFFER_SIZE	Query the size of the AI buffer allocated by the driver	0x001D0001
ME_CAP_AI_CHANNEL_LIST_SIZE	Query the AI channel-list size	0x001D0002
ME_CAP_AI_MAX_THRESHOLD_ SIZE	Maximum number of values in the AI-FIFO	0x001D0003
Analog Output		
ME_CAP_AO_FIFO_SIZE	Query the AI-FIFO size	0x001F0000
ME_CAP_AO_BUFFER_SIZE	Query the size of the AI buffer allocated by the driver	0x001F0001
ME_CAP_AO_CHANNEL_LIST_SIZE	Query the AI channel-list size	0x001F0002
ME_CAP_AO_MAX_THRESHOLD_ SIZE	Maximum number of values in the	0x001F0003
Counter		
ME_CAP_CTR_WIDTH	Query the bid-width of the counter	0x00200000

Table 15: Capabilities for meQuerySubdeviceCapsArgs

C Properties

An overview of the reserved keywords for access to the properties can be found in the ME-iDS help file.

Note: Install ME-iDS 2.0 or higher for using the properties.

D Error Codes

A table with all error codes can be found in the ME-iDS help file.

E Accessories

We recommend to use high-quality connector cables with single-shielded lines per channel.

For further accessories please refer to the current Meilhaus Electronic catalog and the internet:
www.meilhaus.de/en/pc-boards/accessories/

F Technical Questions

F1 Hotline

Should you have questions or inquiries concerning your Meilhaus device, please contact us:

Meilhaus Electronic GmbH
Repair & Service
Am Sonnenlicht 2
D-82239 Alling

Sales:		**Support:**	
Tel.:	(08141) 52 71 – 0	Tel.:	(08141) 52 71 – 188
Fax:	(08141) 52 71 – 129	Fax:	(08141) 52 71 – 169
eMail:	sales@meilhaus.de	eMail:	support@meilhaus.de

Download-Server and Driver Update:
To download current driver versions for Meilhaus Electronic devices as well as manuals in PDF format, please go to:
www.meilhaus.org/driver

Service Department with RMA Process:
In case you need to return a board for repair purposes, we strongly ask you attach a detailed description of the error as well as information regarding your computer/system and the software used. Please register online using our RMA process:
www.meilhaus.de/en/infos/service/rma.htm.

G Index

N

O

P

Q

R

S

T

U

W