SysToMath IO C++ Libraries Interface Reference Manual

Version 1.07-r348

Generated by Doxygen 1.5.4-20071203

Thu Jan 3 20:42:21 2008

Contents

1 SysToMath IO C++ Libraries Interface Main Page

1.1 Introduction

This documentation describes the C++ libraries contained in the SysToMath IO C++ Libraries package:

- SysToMath Device C++ Library (Headers only)
- SysToMath W32Device C++ Library
- SysToMath UsbDevice C++ Library

1.2 Supported Tool Families

The C++ libraries contained in the SysToMath IO C++ Libraries package are designed to support the tool families:

- Microsoft Visual Studio Tool Family
- GNU Tool Family

2 SysToMath IO C++ Libraries Interface Module Index

2.1 SysToMath IO C++ Libraries Interface Modules

Here is a list of all modules:

3 SysToMath IO C++ Libraries Interface Class Index

3.1 SysToMath IO C++ Libraries Interface Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

4 SysToMath IO C++ Libraries Interface Class Index

4.1 SysToMath IO C++ Libraries Interface Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

5 SysToMath IO C++ Libraries Interface File Index

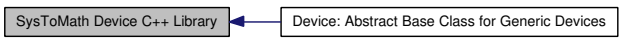
5.1 SysToMath IO C++ Libraries Interface File List

Here is a list of all documented files with brief descriptions:

6 SysToMath IO C++ Libraries Interface Module Documentation

6.1 SysToMath Device C++ Library

Collaboration diagram for SysToMath Device C++ Library:



6.1.1 Detailed Description

SysToMath Device C++ Library (stmdevice).

The SysToMath Device C++ Library consists of C++ header files providing the abstract base class stm::Device forming the ANSI-C++ API for generic devices.

Modules

- Device: Abstract Base Class for Generic Devices

  Abstract base class for generic devices neutralizing the native operating system interfaces.

6.2 Device: Abstract Base Class for Generic Devices

Collaboration diagram for Device: Abstract Base Class for Generic Devices:

6.2.1 Detailed Description

Abstract base class for generic devices neutralizing the native operating system interfaces.

Files

- file device.hpp

  Abstract base class stm::Device forming the ANSI-C++ API for generic devices.

Classes

- struct stm::Uuid

  Universal unique identifier storing its fields in little endian format.

- class stm::Device

  Abstract base class defining the C++ API for a generic device.

- struct stm::Device::Version

  Device driver version.

Functions

- std::ostream & stm::operator<< (std::ostream &os, const Device &device)

  Insert a description of the Device device *into* os.

6.2.2 Function Documentation

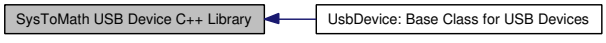
6.2.2.1 std::ostream& stm::operator<< (std::ostream & *os*, const Device & *device*)

Insert a description of the Device *device* into *os*.

The function inserts a verbal description of the Device *device* into the output stream *os* and returns *os*.

6.3 SysToMath USB Device C++ Library

Collaboration diagram for SysToMath USB Device C++ Library:



6.3.1 Detailed Description

SysToMath USB Device C++ Library (stmusbdevice).

The SysToMath USB Device C++ Library consists of a library object providing the base class stm::UsbDevice forming the ANSI-C++ API for USB devices.

Modules

- UsbDevice: Base Class for USB Devices

  Base class for USB devices neutralizing the native operating system interfaces.

6.4 UsbDevice: Base Class for USB Devices

Collaboration diagram for UsbDevice: Base Class for USB Devices:

6.4.1 Detailed Description

Base class for USB devices neutralizing the native operating system interfaces.

Files

- file usbdevice.hpp

  Base class stm::UsbDevice forming the ANSI-C++ API for libusb controlled USB devices.

Classes

- struct stm::UsbPipe

  Type specifying the configuration, the interface, the alternate setting, and the endpoint of the USB pipe to be used for a USB bulk or interrupt transfer.

- struct stm::UsbCtrl

  Type specifying the request type encoding the transfer direction, the value and the index of a USB control request.

- class stm::UsbDevice

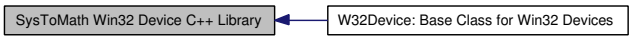
  Class defining the C++ API for a libusb controlled USB device inheriting the generic stm::Device C++ API.

- struct stm::UsbDevice::InterfaceClass

  Type describing a libusb controlled USB device interface class.

6.5 SysToMath Win32 Device C++ Library

Collaboration diagram for SysToMath Win32 Device C++ Library:



6.5.1 Detailed Description

SysToMath Win32 Device C++ Library (stmw32device).

The SysToMath Win32 Device C++ Library consists of a library object providing the base class stm::W32Device forming the ANSI-C++ API for Win32 devices.

Modules

- W32Device: Base Class for Win32 Devices

  Base class for Win32 devices neutralizing the native operating system interfaces.

6.6 W32Device: Base Class for Win32 Devices

Collaboration diagram for W32Device: Base Class for Win32 Devices:

6.6.1 Detailed Description

Base class for Win32 devices neutralizing the native operating system interfaces.

Files

- file w32device.hpp

  Base class stm::W32Device forming the ANSI-C++ API for Win32 devices.

Classes

- class stm::W32Device

  Class defining the C++ API for a Windows device inheriting the generic stm::Device C++ API.

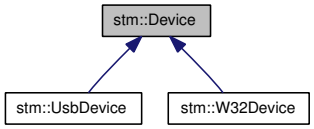
- struct stm::W32Device::InterfaceClass

  Type describing a Windows device interface class.

7 SysToMath IO C++ Libraries Interface Class Documentation

7.1 stm::Device Class Reference

Inheritance diagram for stm::Device:



7.1.1 Detailed Description

Abstract base class defining the C++ API for a generic device.

This class is intended to serve as base class of a special device class which shall implement the interface of stm::Device.

Definition at line 398 of file device.hpp.

Public Types

- enum OpenMode {
  NoAccess = 0x00000000,
  ReadAccess = 0x00000001,
  WriteAccess = 0x00000002,
  ReadWriteAccess = ReadAccess | WriteAccess }
  Open mode flags (bitwise orable).

- enum ErrorState {
  NoError = 0,
  ReadError = 1,
  WriteError = 2,
  ControlError = 3,
  ResourceError = 4,
  OpenError = 5,
  CloseError = 6,
  SeekError = 7,
  ArgumentError = 8,
  UnknownError = 9 }
  Error state values.

- enum ErrorFlags {
  ErrorFlagMask = 0x7fff0000,
  SystemError = 0x00010000 }
  Error flags.

- enum {
  NoFlags = 0x00000000,
  AcceptTimeout = 0x00000001 }
  Bitwise orable operation flag bits for read(), write() and control().

- enum DescribeFlags {
  IndentMask = 0x0000000f,
  IndentFirst = IndentMask + 1,
  NoPropertyNames = IndentFirst << 1,
  DefaultProperties = NoPropertyNames << 1,
  VerboseProperties = DefaultProperties << 1 | DefaultProperties,
  AllProperties = ~ ((DefaultProperties << 2) - 1),
  DeviceType = DefaultProperties << 2,
  DeviceUuid = DeviceType << 1,
  DriverVersion = DeviceUuid << 1 }
  Describe flags (bitwise orable).

- enum Timeout {
  DefaultTimeout = -1,
  Forever = INT_MAX }
  Special timeout values.

Public Member Functions

- Device (const Uuid &uuid=Uuid(), const std::string type=std::string(), int defaultTimeout=Forever, const Descriptor &descr=Descriptor())
  Constructor optionally setting the Uuid of the Device to uuid*, the type string of the Device to* type*, the default operation timeout of the Device to* defaultTimeout *milliseconds and the Descriptor of the Device to* descr.

- Device (int defaultTimeout, const Descriptor &descr=Descriptor())
  Constructor setting the default operation timeout of the Device to defaultTimeout *milliseconds and the Descriptor to* descr.

- Device (const Uuid &uuid, const std::string type, const Version &version, int defaultTimeout=Forever, const Descriptor &descr=Descriptor())
  Constructor setting the Uuid of the Device to uuid*, the type string and version of the Device to* type *and* version*, respectively, and the default operation timeout of the Device to* defaultTimeout *milliseconds which defaults to infinite.*

- Device (const Device &other)
  Copy constructor.

- Device & operator= (const Device &other)
  Copy assignment operator.

- virtual ~Device ()

  Destructor.

- virtual const Uuid & uuid () const

  The virtual method shall return the Uuid of this Device.

- virtual void setUuid (const Uuid &uuid)

  The virtual method shall set the Uuid of this Device to uuid.

- const std::string & property (const std::string &name) const

  The method returns the string value of property name *of this Device.*

- void setProperty (const std::string &name, const std::string &value)

  The method sets the property name *of this Device to* value.

- bool unsetProperty (const std::string &name)

  The method unsets the property name *of this Device to* value *and returns true, if it were set, else false.*

- bool hasProperty (const std::string &name)

  The method returns true, if the property name *is set for this Device, else false.*

- virtual const std::string & type () const

  The virtual method shall return the type string of this Device.

- virtual void setType (const std::string &type)

  The virtual method shall set the type string of this Device to type.

- virtual Version version () const

  The virtual method shall return the Version of this Device.

- virtual void setVersion (const Version &version)

  The virtual method shall set the Version of this Device to version.

- virtual int defaultTimeout () const

  The virtual method shall return the default timeout of this Device.

- virtual void setDefaultTimeout (int defaultTimeout=Forever)

  The virtual method shall set the default operation timeout in milliseconds of this Device to defaultTimeout.

- virtual const Descriptor & descr () const

  Return the Device::Descriptor of this Device.

- virtual void setDescr (const Descriptor &descr)=0

  Set the Device::Descriptor of this Device to descr.

- virtual bool canRead () const

  Return the read capability of this Device.

- virtual bool canWrite () const

  Return the write capability of this Device.

- virtual bool canControl () const

  Return the control capability of this Device.

- virtual bool canSeek () const

  Return the seek capability of this Device.

- virtual int error () const

  Return the error state of this Device.

- virtual void setError (int error, const std::string &msg=std::string()) const

  Set the error state and error string of this Device according to error *and* msg.

- virtual void clearError () const

  Clear the error state and error string of this Device.

- virtual std::string errorString (bool msgOnly=false) const

  Return the error string of this Device.

- virtual void augmentErrorString (const std::string &prefix, const std::string &suffix=std::string()) const

  Augment the error string of this device.

- virtual bool isOpen () const

  The virtual method shall return true, if this Device is open, else false.

- virtual unsigned int openMode () const

  The virtual method shall return the open mode of this Device.

- virtual bool open (unsigned int openMode=ReadWriteAccess)

  The virtual method shall open this Device in openMode*, if possible and return true on success, else false.*

- virtual bool close ()

  The virtual method shall close this Device, if possible and return true on success, else false.

- virtual int64_t read (void *data, int64_t maxLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  The virtual method shall read maximal maxLen *bytes from this Device and store them in the buffer pointed to by* data*.*

- virtual int64_t write (const void *data, int64_t len, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  The virtual method shall write len *bytes from the buffer pointed to by* data *to this Device.*

- virtual int64_t control (unsigned int request, const void *inData, int64_t inLen, void *outData, int64_t maxOutLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags) const

  The virtual method shall perform the control operation specified by request *for this Device with input data of length* inLen *pointed to by* inData *producing output data of maximal length* maxOutLen *in the buffer pointed to by* outData*.*

- virtual int64_t size () const

The virtual method shall return the size of this Device, if it is an opened random access device, else -1.

- virtual int64_t pos () const

  The virtual method shall return the access position of this Device, if it is an opened random access device, else -1.

- virtual bool seek (int64_t pos)

  The virtual method shall set the access position of this Device to pos *and return true on success, if it is an opened random access device, else it shall return false.*

- virtual bool reset ()

  The virtual method shall set the access position of this Device to 0 and return true on success, if it is an opened random access device, else it shall return false.

- virtual std::ostream & describe (std::ostream &os, unsigned int flags=DefaultProperties) const

  Insert a description of this Device into os.

Classes

- struct Descriptor

  Objects of type Device::Descriptor describe system dependent aspects of a Device as a pair of a void pointer and a void function pointer.

- struct Version

  Device driver version.

7.1.2 Member Enumeration Documentation

7.1.2.1 enum stm::Device::OpenMode

Open mode flags (bitwise orable).

The enumerators of Device::OpenMode specify the possible access modes of a Device.

Enumerator:

NoAccess No acces, Device not open.

ReadAccess Read access, Device readable.

WriteAccess Write access, Device writeable.

ReadWriteAccess Read and write access.

Definition at line 405 of file device.hpp.

7.1.2.2 enum stm::Device::ErrorState

Error state values.

The enumerators of Device::ErrorState indicate the reason of the last Device error occurred.

Enumerator:

NoError No error.

ReadError Error during read().

WriteError Error during write().

ControlError Error during control().

ResourceError Resource error.

OpenError Error during open().

CloseError Error during close().

SeekError Error during seek(), reset(), pos() or size().

ArgumentError Invalid argument.

UnknownError Unknown error.

Definition at line 416 of file device.hpp.

7.1.2.3 enum stm::Device::ErrorFlags

Error flags.

Enumerator:

ErrorFlagMask All error flags shall be covered by that mask.

SystemError Indicates that the error string shall be augmented by a system error description, if available.

Definition at line 432 of file device.hpp.

7.1.2.4 anonymous enum

Bitwise orable operation flag bits for read(), write() and control().

Enumerator:

NoFlags No flag bits set.

AcceptTimeout Timeout is no error.

Definition at line 442 of file device.hpp.

7.1.2.5 enum stm::Device::DescribeFlags

Describe flags (bitwise orable).

The enumerators of Device::DescribeFlags specify output format and extent produced by describe().

Enumerator:

IndentMask Mask for indentation field.

If the value masked is not 0, each property is output on a new line indented by this value. If it is 0, all properties are enumerated on one line separated by commas.

IndentFirst If set, also the first property is indented, else not and also its name is omitted.

This flag has no effect, if no indentation is performed.

NoPropertyNames If set and no indentation is performed, all property names are omitted.

DefaultProperties If set, the device specific default properties are included.

VerboseProperties If set, the device specific verbose properties are included which include the device specific default properties.

AllProperties If set, all device specific properties are included.

DeviceType If set, the device type property is included, else not.

DeviceUuid If set, the device UUID property is included, else not.

DriverVersion If set, the driver version property is included, else not.

Reimplemented in stm::UsbDevice, and stm::W32Device.

Definition at line 451 of file device.hpp.

7.1.2.6 enum stm::Device::Timeout

Special timeout values.

Enumerator:

DefaultTimeout Use default timeout of this Device.

Forever Wait forever.

Definition at line 489 of file device.hpp.

7.1.3 Constructor & Destructor Documentation

7.1.3.1 stm::Device::Device (const Uuid & *uuid* = Uuid(), const std::string *type* = std::string(), int *defaultTimeout* = Forever, const Descriptor & *descr* = Descriptor())

Constructor optionally setting the Uuid of the Device to *uuid*, the type string of the Device to *type*, the default operation timeout of the Device to *defaultTimeout* milliseconds and the Descriptor of the Device to *descr*.

The default Uuid is null as is the default type string, whereas the default *defaultTimeout* is infinite and the default *descr* is invalid.

7.1.3.2 stm::Device::Device (int *defaultTimeout*, const Descriptor & *descr* = Descriptor())

Constructor setting the default operation timeout of the Device to *defaultTimeout* milliseconds and the Descriptor to *descr*.

7.1.3.3 stm::Device::Device (const Uuid & *uuid*, const std::string *type*, const Version & *version*, int *defaultTimeout* = Forever, const Descriptor & *descr* = Descriptor())

Constructor setting the Uuid of the Device to *uuid*, the type string and version of the Device to *type* and *version*, respectively, and the default operation timeout of the Device to *defaultTimeout* milliseconds which defaults to infinite.

Moreover, the Descripor is set to *descr* defaulting to invalid.

7.1.3.4 stm::Device::Device (const Device & *other*)

Copy constructor.

Only copyable, if the Device::Descriptor of *other* is invalid. Then the constructed Device object is a copy of *other* with the exception that its Uuid is null.

7.1.3.5 virtual stm::Device::~Device () [virtual]

Destructor.

If this Device is open, it is closed first.

7.1.4 Member Function Documentation

7.1.4.1 Device& stm::Device::operator= (const Device & *other*)

Copy assignment operator.

Only copyable, if the Device::Descriptor of this Device and of *other* are invalid. Then this Device is replaced with a copy of *other* with the exception that its Uuid is null.

7.1.4.2 virtual const Uuid& stm::Device::uuid () const [virtual]

The virtual method shall return the Uuid of this Device.

The default implementation returns the Uuid of this Device set by the constructor or by setUuid().

7.1.4.3 virtual void stm::Device::setUuid (const Uuid & *uuid*) [virtual]

The virtual method shall set the Uuid of this Device to *uuid*.

The default implementation sets the Uuid of this device to *uuid*.

7.1.4.4 const std::string& stm::Device::property (const std::string & *name*) const

The method returns the string value of property *name* of this Device.

If property *name* was not set, an empty string is returned.

7.1.4.5 void stm::Device::setProperty (const std::string & *name*, const std::string & *value*)

The method sets the property *name* of this Device to *value*.

7.1.4.6 bool stm::Device::unsetProperty (const std::string & *name*)

The method unsets the property *name* of this Device to *value* and returns true, if it were set, else false.

7.1.4.7 bool stm::Device::hasProperty (const std::string & *name*)

The method returns true, if the property *name* is set for this Device, else false.

7.1.4.8 virtual const std::string& stm::Device::type () const [virtual]

The virtual method shall return the type string of this Device.

The default implementation returns the type string of this Device set by the constructor or by setType() and is implemented as property.

7.1.4.9 virtual void stm::Device::setType (const std::string & *type*) [virtual]

The virtual method shall set the type string of this Device to *type*.

The default implementation sets the type string of this device to *type* and is implemented as property.

7.1.4.10 virtual Version stm::Device::version () const [virtual]

The virtual method shall return the Version of this Device.

The default implementation returns the Version of this Device set by the constructor or by setVersion().

7.1.4.11 virtual void stm::Device::setVersion (const Version & *version*) [virtual]

The virtual method shall set the Version of this Device to *version*.

The default implementation sets the Version of this device to *version*.

7.1.4.12 virtual int stm::Device::defaultTimeout () const [virtual]

The virtual method shall return the default timeout of this Device.

The default implementation returns the default operation timout of this Device in milliseconds set by the constructor or by setDefaultTimeout().

7.1.4.13 virtual void stm::Device::setDefaultTimeout (int *defaultTimeout* = Forever) [virtual]

The virtual method shall set the default operation timeout in milliseconds of this Device to *defaultTimeout*.

The default implementation sets the default timeout of this Device to *defaultTimeout* which defaults to infinite.

7.1.4.14 virtual const Descriptor& stm::Device::descr () const [virtual]

Return the Device::Descriptor of this Device.

Returns:

An invalid Device::Descriptor, if this Device does not represent a device.
A valid Device::Descriptor of this Device represents a device.

Note:

It is not necessary that this UsbDevice is open.

See also:

setDescr().

7.1.4.15 virtual void stm::Device::setDescr (const Descriptor & *descr*) [pure virtual]

Set the Device::Descriptor of this Device to *descr*.

Parameters:

← ***descr*** A valid Device::Descriptor of the device to be represented by this Device object or an invalid Device::Descriptor.

Effects:

If this Device is open, it is closed. Then *descr* is defined for it. If *descr* is valid, this Device is ready to be opened, else it does not represent a device.

See also:

descr(), isOpen(), close(), open().

Implemented in stm::UsbDevice.

7.1.4.16 virtual bool stm::Device::canRead () const [virtual]

Return the read capability of this Device.

Returns:

`false`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device cannot be successfully opended in open mode Device::ReadAccess.
It is not necessary that this Device is open.

See also:

canWrite(), canControl(), canSeek(), open().

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.17 virtual bool stm::Device::canWrite () const [virtual]

Return the write capability of this Device.

Returns:

`false`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device cannot be successfully opended in open mode Device::WriteAccess.
It is not necessary that this Device is open.

See also:

canRead(), canControl(), canSeek(), open().

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.18 virtual bool stm::Device::canControl () const [virtual]

Return the control capability of this Device.

Returns:

`false`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device does not support the method control().
It is not necessary that this Device is open.

See also:

canRead(), canWrite(), canSeek(), control().

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.19 virtual bool stm::Device::canSeek () const [virtual]

Return the seek capability of this Device.

Returns:

`false`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device does not support the methods size(), pos(), seek () and reset().
It is not necessary that this Device is open.

See also:

canRead(), canWrite(), canControl(), size(), pos(), seek(), reset ().

7.1.4.20 virtual int stm::Device::error () const [virtual]

Return the error state of this Device.

Returns:

The error state of this Device as one of the enumerators of Device::ErrorState.

Note:

It is not necessary that this Device is open.

See also:

setError(), clearError(), errorString(), augmentErrorString().

Reimplemented in stm::W32Device.

7.1.4.21 virtual void stm::Device::setError (int *error*, const std::string & *msg* = `std::string()`) const [virtual]

Set the error state and error string of this Device according to *error* and *msg*.

Parameters:

← ***error*** Error state as one of the enumerators of Device::ErrorState optionally ored with one ore more of the enumerators of Device::ErrorFlags.

← ***msg*** Error string.

Effects:

If the error state part of *error* is one of the enumerators of Device::ErrorState, the error state of this Device is set to that state and its error string to *msg*, else to Device::UnknownError. If the error flag Device::SystemError is set in *error*, the error string is augmented by a system error description, if available.

Note:

Despite of being const, the method can change the error state and error string.
It is not necessary that this Device is open.

See also:

error(), clearError(), errorString(), augmentErrorString().

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.22 virtual void stm::Device::clearError () const [virtual]

Clear the error state and error string of this Device.

Effects:

The error state and error string of this Device are cleared, that means set to Device::NoError and the empty string.

Note:

Despite of being const, the method can change the error state and error string.
It is not necessary that this Device is open.

See also:

error(), setError(), errorString(), augmentErrorString().

Reimplemented in stm::W32Device.

7.1.4.23 virtual std::string stm::Device::errorString (bool *msgOnly* = false) const [virtual]

Return the error string of this Device.

Parameters:

← ***msgOnly*** If true, return only error message, else precede it by a verbal description of the error state.

Returns:

A non empty string describing the error state of this Device, if that error state is not Device::NoError.
The empty string, if the error state of this Device is Device::NoError.

Note:

It is not necessary that this Device is open.

See also:

error(), setError(), clearError(), augmentErrorString().

7.1.4.24 virtual void stm::Device::augmentErrorString (const std::string & *prefix*, const std::string & *suffix* = `std::string()`) const `[virtual]`

Augment the error string of this device.

Parameters:

← ***prefix*** Error string prefix.

← ***suffix*** Error string suffix.

Effects:

If the error state of this Device is not Device::NoError and at least one of *prefix* or *suffix* is not empty, the current error string is augmented accordingly.

Note:

Despite of being const, the method can change the error string.
It is not necessary that this Device is open.

See also:

error(), setError(), clearError(), errorString().

7.1.4.25 virtual bool stm::Device::isOpen () const `[virtual]`

The virtual method shall return true, if this Device is open, else false.

The default implementation returns true, if this Device is open, that is if its open mode is not the Device::OpenMode enumerator Device::NoAccess, else false.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.26 virtual unsigned int stm::Device::openMode () const `[virtual]`

The virtual method shall return the open mode of this Device.

The default implementation returns the open mode of this Device as one of the enumerators of Device::OpenMode.

7.1.4.27 virtual bool stm::Device::open (unsigned int *openMode* = `ReadWriteAccess`) `[virtual]`

The virtual method shall open this Device in *openMode*, if possible and return true on success, else false.

The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::OpenError and returns false, if isOpen() does not return false or if *openMode* is not compatible with the results of canRead() and/or canWrite(). Else the method sets the open mode of this Device to *openMode* and returns true.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.28 virtual bool stm::Device::close () `[virtual]`

The virtual method shall close this Device, if possible and return true on success, else false.

The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::CloseError and returns false, if isOpen() returns false. Else the method sets the open mode of this Device to the Device::OpenMode enumerator Device::NoAccess and returns true.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.29 virtual int64_t stm::Device::read (void * *data*, int64_t *maxLen*, int *timeout* = DefaultTimeout, unsigned int *flags* = NoFlags) [virtual]

The virtual method shall read maximal *maxLen* bytes from this Device and store them in the buffer pointed to by *data*.

On error the method shall return -1, else the number of bytes actually read. The default implementation sets the error state of this Device to Device::ErrorState enumerator Device::ReadError and returns -1, if the result of openMode() does not contain the Device::OpenMode enumerator Device::ReadAccess or if *maxLen* is negative or *data* is the NULL pointer unless *maxLen* is also 0. Else the method returns *maxLen*.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.30 virtual int64_t stm::Device::write (const void * *data*, int64_t *len*, int *timeout* = DefaultTimeout, unsigned int *flags* = NoFlags) [virtual]

The virtual method shall write *len* bytes from the buffer pointed to by *data* to this Device.

On error the method shall return -1, else the number of bytes actually written. The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::WriteError and returns -1, if the result of openMode() does not contain the Device::OpenMode enumerator Device::WriteAccess or if *len* is negative or *data* is the NULL pointer unless *len* is also 0. Else the method returns len.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.31 virtual int64_t stm::Device::control (unsigned int *request*, const void * *inData*, int64_t *inLen*, void * *outData*, int64_t *maxOutLen*, int *timeout* = DefaultTimeout, unsigned int *flags* = NoFlags) const [virtual]

The virtual method shall perform the control operation specified by *request* for this Device with input data of length *inLen* pointed to by *inData* producing output data of maximal length *maxOutLen* in the buffer pointed to by *outData*.

On error the method shall return -1, else the number of bytes produced in *outData*. The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::ControlError and returns -1, if canControl() returns false, or if *inLen* is negative or *inData* is NULL unless *inLen* is also 0, or if *outLen* is negative or *outData* is NULL unless *maxOutLen* is also 0. Else the method returns *maxOutLen*.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.1.4.32 virtual int64_t stm::Device::size () const [virtual]

The virtual method shall return the size of this Device, if it is an opened random access device, else -1.

The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::SeekError and returns -1, if canSeek() or isOpen() return false. Else the method returns 0.

7.1.4.33 virtual int64_t stm::Device::pos () const [virtual]

The virtual method shall return the access position of this Device, if it is an opened random access device, else -1.

The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::SeekError and returns -1, if canSeek() or isOpen() return false. Else the method returns 0.

7.1.4.34 virtual bool stm::Device::seek (int64_t *pos*) [virtual]

The virtual method shall set the access position of this Device to *pos* and return true on success, if it is an opened random access device, else it shall return false.

The default implementation sets the error state of this Device to the Device::ErrorState enumerator Device::SeekError and returns false, if canSeek() or isOpen() return false, or if *pos* is negative. Else the method returns true.

7.1.4.35 virtual bool stm::Device::reset () [virtual]

The virtual method shall set the access position of this Device to 0 and return true on success, if it is an opened random access device, else it shall return false.

The default implementation returns seek(0).

7.1.4.36 virtual std::ostream& stm::Device::describe (std::ostream & *os*, unsigned int *flags* = DefaultProperties) const [virtual]

Insert a description of this Device into *os*.

The method shall insert a verbal description of this Device into the output stream *os* and return *os*. Format and extent of the description shall be controlled by the *flags* parameter according to the bitwise ored enumerators of Device::DescribeFlags. The default implementation handles the device properties device type and device Uuid for DefaultProperties and additionally the driver version property for VerboseProperties or AllProperties.

Reimplemented in stm::UsbDevice, and stm::W32Device.

7.2 stm::Device::Descriptor Struct Reference

7.2.1 Detailed Description

Objects of type Device::Descriptor describe system dependent aspects of a Device as a pair of a void pointer and a void function pointer.

If a Device::Descriptor object's first pointer is NULL the descriptor is called invalid else valid.

Definition at line 502 of file device.hpp.

Public Member Functions

- Descriptor (void *d=NULL, void(*f)()=NULL)

  Constructs a Device::Descriptor object called invalid, if d *is* NULL.

- operator const void * () const

  Returns the first pointer of the Descriptor.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 stm::Device::Descriptor::Descriptor (void * *d* = NULL, void(*)() *f* = NULL)

Constructs a Device::Descriptor object called invalid, if *d* is NULL.

7.2.3 Member Function Documentation

7.2.3.1 stm::Device::Descriptor::operator const void * () const

Returns the first pointer of the Descriptor.

7.3 stm::Device::Version Struct Reference

7.3.1 Detailed Description

Device driver version.

Definition at line 922 of file device.hpp.

Public Types

- enum Parts {
  Major = 3,
  Minor = 2,
  Micro = 1,
  Nano = 0 }
  Version part names.

Public Member Functions

- Version ()
  Construct the null Version object with all part words cleared.
- Version (unsigned short major, unsigned minor, unsigned micro=0, unsigned nano=0)
  Construct the Version object with all parts major, minor, micro *and* nano.
- Version (const Version &other)
  Copy constructor.
- Version & operator= (const Version &other)
  Assignment operator.
- std::string string () const
  Return the string representation of this Version.
- bool isNull () const
  Returns true, if this Version is null.

- bool operator== (const Version &other) const
  Equality comparison operator.
- bool operator!= (const Version &other) const
  Unequality comparison operator.
- bool operator< (const Version &other) const
  Less than comparison operator.
- bool operator> (const Version &other) const
  Greater than comparison operator.
- bool operator<= (const Version &other) const
  Less or equal comparison operator.
- bool operator>= (const Version &other) const
  Greater or equal comparison operator.

Public Attributes

- unsigned short part [4]
  Word array of version parts.

7.3.2 Member Enumeration Documentation

7.3.2.1 enum stm::Device::Version::Parts

Version part names.

They serve as indexes of the part array.

Enumerator:

Major Major version part.
Minor Minor version part.
Micro Micro version part.
Nano Nano version part.

Definition at line 926 of file device.hpp.

7.3.3 Constructor & Destructor Documentation

7.3.3.1 stm::Device::Version::Version ()

Construct the null Version object with all part words cleared.

7.3.3.2 stm::Device::Version::Version (unsigned short *major*, unsigned *minor*, unsigned *micro* = 0, unsigned *nano* = 0)

Construct the Version object with all parts *major*, *minor*, *micro* and *nano*.

7.3.3.3 stm::Device::Version::Version (const Version & *other*)

Copy constructor.

7.3.4 Member Function Documentation

7.3.4.1 Version& stm::Device::Version::operator= (const Version & *other*)

Assignment operator.

7.3.4.2 std::string stm::Device::Version::string () const

Return the string representation of this Version.

7.3.4.3 bool stm::Device::Version::isNull () const

Returns true, if this Version is null.

That means, if all part words are cleared.

7.3.4.4 bool stm::Device::Version::operator== (const Version & *other*) const

Equality comparison operator.

7.3.4.5 bool stm::Device::Version::operator!= (const Version & *other*) const

Unequality comparison operator.

7.3.4.6 bool stm::Device::Version::operator< (const Version & *other*) const

Less than comparison operator.

7.3.4.7 bool stm::Device::Version::operator> (const Version & *other*) const

Greater than comparison operator.

7.3.4.8 bool stm::Device::Version::operator<= (const Version & *other*) const

Less or equal comparison operator.

7.3.4.9 bool stm::Device::Version::operator>= (const Version & *other*) const

Greater or equal comparison operator.

7.3.5 Member Data Documentation

7.3.5.1 unsigned short stm::Device::Version::part[4]

Word array of version parts.

Its elements store the version parts indexed by the enumerators of Parts.

Definition at line 976 of file device.hpp.

7.4 stm::UsbCtrl Struct Reference

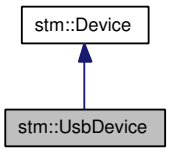
7.4.1 Detailed Description

Type specifying the request type encoding the transfer direction, the value and the index of a USB control request.

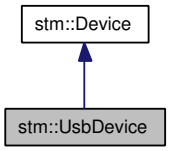
Definition at line 162 of file usbdevice.hpp.

7.5 stm::UsbDevice Class Reference

Inheritance diagram for stm::UsbDevice:



Collaboration diagram for stm::UsbDevice:



7.5.1 Detailed Description

Class defining the C++ API for a libusb controlled USB device inheriting the generic stm::Device C++ API.

This class can be instantiated or be used as base class of a special libusb controlled USB device class which may reimplement the interface of stm::UsbDevice as well as that of the abstract base class stm::Device.

Definition at line 199 of file usbdevice.hpp.

Public Types

- enum DescribeFlags {
  DeviceReleaseNumber = DriverVersion << 1,
  DeviceBusNumber = DeviceReleaseNumber << 1,
  DeviceManufacturer = DeviceBusNumber << 1,
  DeviceProduct = DeviceManufacturer << 1,
  DeviceSerialNumber = DeviceProduct << 1,
  DeviceConfigurations = DeviceSerialNumber << 1,
  DeviceInterfaces = DeviceConfigurations << 1,
  DeviceAltSettings = DeviceInterfaces << 1,

DeviceEndpoints = DeviceAltSettings << 1,

DeviceChildren = DeviceEndpoints << 1 }

Describe flags (bitwise orable).

Public Member Functions

- UsbDevice (int defaultTimeout=Forever, const Descriptor &descr=Descriptor())

  Constructor of a UsbDevice object representing a libusb controlled USB device described by descr.

- virtual ~UsbDevice ()

  Destructor.

- virtual bool canRead () const

  Return the read capability of this UsbDevice.

- virtual bool canWrite () const

  Return the write capability of this UsbDevice.

- virtual bool canControl () const

  Return the control capability of this UsbDevice.

- virtual void setError (int error, const std::string &msg=std::string()) const

  Set the error state and error string of this UsbDevice according to error *and* msg.

- virtual void setDescr (const Descriptor &descr)

  Set the Device::Descriptor of this UsbDevice to descr.

- UsbPipe pipe () const

  Return a copy of the UsbPipe object configured for this UsbDevice.

- UsbPipe::Type pipeType () const

  Return the type of the UsbPipe object configured for this UsbDevice as an enumerator of the enumeration UsbPipe::Type.

- UsbPipe::Type setPipe (UsbPipe pipe)

  Set the UsbPipe object configured for this UsbDevice to a copy of pipe *and return its type.*

- virtual bool isOpen () const

  Determine, if this UsbDevice is open.

- virtual bool open (unsigned int openMode)

  Open this UsbDevice in openMode.

- virtual bool close ()

  Close this UsbDevice.

- virtual int64_t read (void *data, int64_t maxLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

Read maxLen *bytes from the currently confiured USB pipe of this UsbDevice into the* data *buffer.*

- virtual int64_t readPipe (UsbPipe pipe, void *data, int64_t maxLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  Atomically set the UsbPipe object configured for this UsbDevice to a copy of pipe *and read* maxLen *bytes from that pipe into the* data *buffer.*

- virtual int64_t write (const void *data, int64_t len, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  Write len *bytes from the* data *buffer to the currently configured USB pipe of this UsbDevice.*

- virtual int64_t writePipe (UsbPipe pipe, const void *data, int64_t len, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  Atomically set the UsbPipe object configured for this UsbDevice to a copy of pipe *and write* len *bytes from the* data *buffer to that pipe.*

- virtual int64_t control (unsigned int request, const void *ctrl, int64_t ctrlLen, void *data, int64_t dataLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags) const

  Perform the control operation request *over the default control pipe of this UsbDevice.*

- virtual std::ostream & describe (std::ostream &os, unsigned int flags=DefaultProperties) const

  Insert a description of this UsbDevice into os.

- bool isA (const InterfaceClass &interfaceClass) const

  The method returns true, if this UsbDevice is a device supporting the libusb controlled USB device interface class described by interfaceClass.

- template<class ForwardIterator>
  bool isA (ForwardIterator beginInterfaceClass, ForwardIterator endInterfaceClass) const

  *The method template returns true, if this UsbDevice is a device supporting one of the the libusb controlled USB device interface classes whose description is contained in the half open interval [*beginInterfaceClass*,* endInterfaceClass*).*

Static Public Member Functions

- static size_t enumerate (std::vector< Descriptor > &descriptors, const InterfaceClass &interfaceClass, bool append=false, bool quick=false)

  Enumerate all Device::Descriptor objects describing libusb controlled USB devices supporting the libusb controlled USB device interface class described by interfaceClass.

- template<class ForwardIterator>
  static size_t enumerate (std::vector< Descriptor > &descriptors, ForwardIterator beginInterfaceClass, ForwardIterator endInterfaceClass, bool append=false)

  *Enumerate all Device::Descriptor objects describing libusb controlled USB devices supporting one of the the libusb controlled USB device interface classes whose description is contained in the half open interval [*beginInterfaceClass*,* endInterfaceClass*).*

- static size_t enumerateAll (std::vector< Descriptor > &descriptors)

  Enumerate all Device::Descriptor objects describing libusb controlled USB devices.

Classes

- struct InterfaceClass

  Type describing a libusb controlled USB device interface class.

7.5.2 Member Enumeration Documentation

7.5.2.1 enum stm::UsbDevice::DescribeFlags

Describe flags (bitwise orable).

The enumerators of UsbDevice::DescribeFlags augment the Device::DescribeFlags further specifying the extent produced by describe().

Enumerator:

DeviceReleaseNumber If set, the device release number property is included, else not.

DeviceBusNumber If set, the device bus number property is included, else not.

DeviceManufacturer If set, the device manufacturer property is included, else not.

DeviceProduct If set, the device product property is included, else not.

DeviceSerialNumber If set, the device serial number property is included, else not.

DeviceConfigurations If set, the device configurations property is included, else not.

DeviceInterfaces If set and property names are configured, the device interfaces property is included, else not.

DeviceAltSettings If set and property names are configured, the device alternate settings property is included, else not.

DeviceEndpoints If set and property names are configured, the device endpoints property is included, else not.

DeviceChildren If set, the device children property is included, else not.

Reimplemented from stm::Device.

Definition at line 208 of file usbdevice.hpp.

7.5.3 Constructor & Destructor Documentation

7.5.3.1 stm::UsbDevice::UsbDevice (int *defaultTimeout* = `Forever`, const Descriptor & *descr* = Descriptor `()`)

Constructor of a UsbDevice object representing a libusb controlled USB device described by *descr*.

Parameters:

← ***defaultTimeout*** Timeout in milliseconds used by default for all operations of this UsbDevice.

← ***descr*** A valid Device::Descriptor for the libusb controlled USB device to be represented by the UsbDevice object to be constructed or an invalid Device::Descriptor. A valid Device::Descriptor is typically yielded by one of the static methods enumerate() or enumerateAll().

Effects:

Constructs a UsbDevice object with the Device::Descriptor *descr* defined for it. If *descr* is valid, the constructed UsbDevice is ready to be opened.

See also:

descr(), open(), enumerate(), enumerateAll().

7.5.3.2 virtual stm::UsbDevice::~UsbDevice () [virtual]

Destructor.

Effects:

If this UsbDevice is open, it is closed first.

See also:

isOpen(), close().

7.5.4 Member Function Documentation

7.5.4.1 virtual bool stm::UsbDevice::canRead () const [virtual]

Return the read capability of this UsbDevice.

Returns:

`true`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device can be successfully opended in open mode Device::ReadAccess.
It is not necessary that this UsbDevice is open.

See also:

canWrite(), canControl(), canSeek(), open().

Reimplemented from stm::Device.

7.5.4.2 virtual bool stm::UsbDevice::canWrite () const [virtual]

Return the write capability of this UsbDevice.

Returns:

`true`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device can be successfully opended in open mode Device::WriteAccess.
It is not necessary that this UsbDevice is open.

See also:

canRead(), canControl(), canSeek(), open().

Reimplemented from stm::Device.

7.5.4.3 virtual bool stm::UsbDevice::canControl () const [virtual]

Return the control capability of this UsbDevice.

Returns:

`true`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device does support the method control().
It is not necessary that this UsbDevice is open.

See also:

canRead(), canWrite(), canSeek(), control().

Reimplemented from stm::Device.

7.5.4.4 virtual void stm::UsbDevice::setError (int *error*, const std::string & *msg* = `std::string()`) const [virtual]

Set the error state and error string of this UsbDevice according to *error* and *msg*.

Parameters:

← ***error*** Error state as one of the enumerators of Device::ErrorState optionally ored with one ore more of the enumerators of Device::ErrorFlags.

← ***msg*** Error string.

Effects:

If the error state part of *error* is one of the enumerators of Device::ErrorState, the error state of this UsbDevice is set to that state and its error string to *msg*, else to Device::UnknownError. If the error flag Device::SystemError is set in *error*, the error string is augmented by a system error description, if available.

Note:

Despite of being const, the method can change the error state and error string.
It is not necessary that this UsbDevice is open.

See also:

error(), clearError(), errorString(), augmentErrorString().

Reimplemented from stm::Device.

7.5.4.5 virtual void stm::UsbDevice::setDescr (const Descriptor & *descr*) [virtual]

Set the Device::Descriptor of this UsbDevice to *descr*.

Parameters:

← ***descr*** A valid Device::Descriptor of the libusb controlled USB device to be represented by this UsbDevice or an invalid Device::Descriptor. A valid descriptor is typically yielded by one of the static methods enumerate() or enumerateAll().

Effects:

If this UsbDevice is open, it is closed. Then *descr* is defined for it. If *descr* is valid, this UsbDevice is ready to be opened, else it does not represent a libusb controlled USB device.

See also:

descr(), isOpen(), close(), open(), enumerate(), enumerateAll().

Implements stm::Device.

7.5.4.6 UsbPipe stm::UsbDevice::pipe () const

Return a copy of the UsbPipe object configured for this UsbDevice.

Returns:

an invalid UsbPipe object, if this UsbDevice does not represent a libusb controlled USB device.
a copy of the UsbPipe object configured for the libusb controlled USB device represented by this UsbDevice.

Note:

It is not necessary that this UsbDevice is open.

See also:

pipeType(), setPipe(), isOpen().

7.5.4.7 UsbPipe::Type stm::UsbDevice::pipeType () const

Return the type of the UsbPipe object configured for this UsbDevice as an enumerator of the enumeration UsbPipe::Type.

Returns:

UsbPipe::Invalid, if this UsbDevice does not represent a libusb controlled USB device, or if its configured pipe is invalid.
UsbPipe::Bulk or UsbPipe::Interrupt, if this UsbDevice represents a libusb controlled USB device with a pipe configured for bulk or interrupt data transfer.

Note:

It is not necessary that this UsbDevice is open.

See also:

pipe(), setPipe(), isOpen().

7.5.4.8 UsbPipe::Type stm::UsbDevice::setPipe (UsbPipe *pipe*)

Set the UsbPipe object configured for this UsbDevice to a copy of *pipe* and return its type.

Parameters:

← ***pipe*** A UsbPipe object a copy of which is to be configured for the libusb controlled USB device represented by this UsbDevice.

Returns:

The type UsbPipe::Bulk or UsbPipe::Interrupt, if a copy of *pipe* can be configured for this UsbDevice, else `UsbPipe::Invalid`, in which case the UsbPipe object configured for this UsbDevice stays unchanged.

Note:

It is not necessary that this UsbDevice is open.

See also:

pipe(), pipeType(), isOpen().

7.5.4.9 virtual bool stm::UsbDevice::isOpen () const [virtual]

Determine, if this UsbDevice is open.

Returns:

`true`, if this UsbDevice is open, that is if its open mode is not the Device::OpenMode enumerator Device::NoAccess.
`false`, if this UsbDevice is not open, that is if its open mode is the Device::OpenMode enumerator Device::NoAccess.

See also:

open(), close().

Reimplemented from stm::Device.

7.5.4.10 virtual bool stm::UsbDevice::open (unsigned int *openMode*) [virtual]

Open this UsbDevice in *openMode*.

Parameters:

← ***openMode*** Open mode to be set.

Effects:

The method sets the error state of this UsbDevice to the Device::ErrorState enumerator Device::OpenError, if *openMode* is the Device::OpenMode enumerator Device::NoAccess, if isOpen() does not return false or if the libusb controlled USB device represented by this UsbDevice cannot be opened conforming to *openMode*. Else the method sets the open mode of this UsbDevice to *openMode*.

Returns:

`true`, if this UsbDevice could be opened in *openMode*.
`false`, if this UsbDevice could not be opened in *openMode*. Then the error state of this UsbDevice is set to Device::OpenError.

See also:

isOpen(), close(), error ().

Reimplemented from stm::Device.

7.5.4.11 virtual bool stm::UsbDevice::close () [virtual]

Close this UsbDevice.

Effects:

The method sets the error state of this UsbDevice to the Device::ErrorState enumerator Device::CloseError, if isOpen() returns false or if the libusb controlled USB device represented by this UsbDevice cannot be closed successfully. Else the method sets the open mode of this UsbDevice to the Device::OpenMode enumerator Device::NoAccess.

Returns:

`true`, if this UsbDevice could be closed successfully.
`false`, if this UsbDevice could not be closed successfully. Then the error state of this UsbDevice is set to Device::CloseError.

See also:

isOpen(), open(), error ().

Reimplemented from stm::Device.

7.5.4.12 virtual int64_t stm::UsbDevice::read (void * *data*, int64_t *maxLen*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) [virtual]

Read *maxLen* bytes from the currently confiured USB pipe of this UsbDevice into the *data* buffer.

Parameters:

→ ***data*** Buffer for the data to be read.

← ***maxLen*** Maximal number of bytes to be read.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this UsbDevice is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method sets the error state of this UsbDevice to the Device::ErrorState enumerator Device::ReadError, if the result of openMode() does not contain the Device::OpenMode enumerator Device::ReadAccess, or if *maxLen* is negative or *data* is the NULL pointer unless *maxLen* is also, 0 or if the read operation described below fails. During the read operation maximal *maxLen* bytes from the libusb controlled USB device represented by this UsbDevice are read through the USB pipe currently configured and are stored in the buffer pointed to by *data*.

Returns:

The number of bytes actually read, if the read operation was successful.
`-1`, if the read operation was not successful. Then the error state of this UsbDevice is set to Device::ReadError.

Note:

Be aware that in multithreaded applications configuring the USB pipe to be used and the reading from the pipe must occur atomically. To ensure this, better use readPipe() in those situations.

See also:

write(), control(), openMode(), error (), defaultTimeout(), pipe(), setPipe(), readPipe().

Reimplemented from stm::Device.

7.5.4.13 virtual int64_t stm::UsbDevice::readPipe (UsbPipe *pipe*, void * *data*, int64_t *maxLen*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) `[virtual]`

Atomically set the UsbPipe object configured for this UsbDevice to a copy of *pipe* and read *maxLen* bytes from that pipe into the *data* buffer.

Parameters:

← ***pipe*** A UsbPipe object a copy of which is to be configured for the libusb controlled USB device represented by this UsbDevice.

→ ***data*** Buffer for the data to be read.

← ***maxLen*** Maximal number of bytes to be read.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this UsbDevice is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method sets the error state of this UsbDevice to the Device::ErrorState enumerator Device::ReadError, if it cannot set the *pipe* successfully, if the result of openMode() does not contain the Device::OpenMode enumerator Device::ReadAccess, or if *maxLen* is negative or *data* is the NULL pointer unless *maxLen* is also, 0 or if the read operation described below fails. During the read operation maximal *maxLen* bytes from the libusb controlled USB device represented by this UsbDevice are read through *pipe* and are stored in the buffer pointed to by *data*.

Returns:

The number of bytes actually read, if the read operation was successful.
`-1`, if the read operation was not successful. Then the error state of this UsbDevice is set to Device::ReadError.

Note:

The *timeout* only applies to the read operation not to the setting of the *pipe*.

See also:

write(), control(), openMode(), error (), defaultTimeout(), pipe(), setPipe(), read().

7.5.4.14 virtual int64_t stm::UsbDevice::write (const void * *data*, int64_t *len*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) `[virtual]`

Write *len* bytes from the *data* buffer to the currently configured USB pipe of this UsbDevice.

Parameters:

← ***data*** Buffer containing the data to be written.

← ***len*** Number of bytes to be written.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this UsbDevice is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method sets the error state of this UsbDevice to the Device::ErrorState enumerator Device::WriteError, if the result of openMode() does not contain the Device::OpenMode enumerator Device::WriteAccess, or if *len* is negative or *data* is the NULL pointer unless *len* is also 0, or if the write operation described below fails. During the write operation *len* bytes from the buffer pointed to by *data* are written through the USB pipe currently configured to the libusb controlled USB device represented by this UsbDevice.

Returns:

len, if the write operation was successful.
`-1`, if the write operation was not successful. Then the error state of this UsbDevice is set to Device::WriteError.

Note:

Be aware that in multithreaded applications configuring the USB pipe to be used and the writing to the pipe must occur atomically. To ensure this, better use writePipe() in those situations.

See also:

read(), control(), openMode(), error (), defaultTimeout(), pipe(), setPipe(), writePipe().

Reimplemented from stm::Device.

7.5.4.15 virtual int64_t stm::UsbDevice::writePipe (UsbPipe *pipe*, const void * *data*, int64_t *len*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) `[virtual]`

Atomically set the UsbPipe object configured for this UsbDevice to a copy of *pipe* and write *len* bytes from the *data* buffer to that pipe.

Parameters:

← ***pipe*** A UsbPipe object a copy of which is to be configured for the libusb controlled USB device represented by this UsbDevice.

← ***data*** Buffer containing the data to be written.

← ***len*** Number of bytes to be written.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this UsbDevice is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method sets the error state of this UsbDevice to the Device::ErrorState enumerator Device::WriteError, if it cannot set the *pipe* successfully, if the result of openMode() does not contain the Device::OpenMode enumerator Device::WriteAccess, or if *len* is negative or *data* is the NULL pointer unless *len* is also 0, or if the write operation described below fails. During the write operation *len* bytes from the buffer pointed to by *data* are written through *pipe* to the libusb controlled USB device represented by this UsbDevice.

Returns:

len, if the write operation was successful.
`-1`, if the write operation was not successful. Then the error state of this UsbDevice is set to Device::WriteError.

Note:

The *timeout* only applies to the write operation not to the setting of the *pipe*.

See also:

read(), control(), openMode(), error (), defaultTimeout(), pipe(), setPipe(), write().

7.5.4.16 virtual int64_t stm::UsbDevice::control (unsigned int *request*, const void * *ctrl*, int64_t *ctrlLen*, void * *data*, int64_t *dataLen*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) const `[virtual]`

Perform the control operation *request* over the default control pipe of this UsbDevice.

Parameters:

← ***request*** Specifies the particular control operation to be performed.

← ***ctrl*** Pointer to a UsbCtrl object specifying the request type (in which the transfer direction is encoded), the value and the index of the request.

← ***ctrlLen*** sizeof(UsbCtrl).

↔ ***data*** Buffer for the input or output data.

← ***dataLen*** Byte length of the data buffer.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this UsbDevice is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method performs the control operation characterized by *request* for the libusb controlled USB device represented by this UsbDevice using its default control pipe. The parameter *ctrl* shall be the address of a UsbCtrl object specifying the request type encoding the transfer direction, the value and the index of the request. The parameter ctrlLen shall be sizeof(UsbCtrl). If any data transfer is required, *data* shall not be NULL and point to a buffer of size *dataLen*.

Returns:

The number of bytes transferred to or from *data*, if the operation was successful. This is 0 in the case of an accepted timeout.
`-1`, if the operation was not successful. Then the error state of this UsbDevice is set to the Device::ErrorState enumerator Device::ControlError.

See also:

write(), read(), openMode(), error (), defaultTimeout().

Reimplemented from stm::Device.

7.5.4.17 virtual std::ostream& stm::UsbDevice::describe (std::ostream & *os*, unsigned int *flags* = DefaultProperties) const [virtual]

Insert a description of this UsbDevice into *os*.

Parameters:

← ***os*** The output stream to insert the description.

← ***flags*** Description flags.

Effects:

The method inserts a verbal description of this UsbDevice into the output stream *os*. Format and extent of the description is controlled by the *flags* parameter according to the bitwise ored enumerators of Device::DescribeFlags and UsbDevice::DescribeFlags.

Returns:

The output stream *os*.

Note:

This UsbDevice need not be open.

Reimplemented from stm::Device.

7.5.4.18 bool stm::UsbDevice::isA (const InterfaceClass & *interfaceClass*) const

The method returns true, if this UsbDevice is a device supporting the libusb controlled USB device interface class described by *interfaceClass*.

That means it returns true, if the Device::Descriptor of this UsbDevice describes a libusb controlled USB device supporting the libusb controlled USB device interface class described by *interfaceClass*, else false.

7.5.4.19 template<class ForwardIterator> bool stm::UsbDevice::isA (ForwardIterator *beginInterfaceClass*, ForwardIterator *endInterfaceClass*) const

The method template returns true, if this UsbDevice is a device supporting one of the the libusb controlled USB device interface classes whose description is contained in the half open interval [∗*beginInterfaceClass*, ∗*endInterfaceClass*).

That means it returns true, if the Device::Descriptor of this UsbDevice describes a libusb controlled USB device supporting one of the libusb controlled USB device interface classes described by that interval, else false.

7.5.4.20 static size_t stm::UsbDevice::enumerate (std::vector< Descriptor > & *descriptors*, const InterfaceClass & *interfaceClass*, bool *append* = false, bool *quick* = false) [static]

Enumerate all Device::Descriptor objects describing libusb controlled USB devices supporting the libusb controlled USB device interface class described by *interfaceClass*.

The static method clears the vector *descriptors*, scans the system for all libusb controlled USB devices supporting the libusb controlled USB device interface class described by *interfaceClass*, stores the Device::Descriptor objects describing those devices in the vector *descriptors* and returns the size of that vector. If *quick* is true, the system is not scanned for new hardware.

7.5.4.21 template<class ForwardIterator> static size_t stm::UsbDevice::enumerate (std::vector< Descriptor > & *descriptors*, ForwardIterator *beginInterfaceClass*, ForwardIterator *endInterfaceClass*, bool *append* = false) [static]

Enumerate all Device::Descriptor objects describing libusb controlled USB devices supporting one of the the libusb controlled USB device interface classes whose description is contained in the half open interval [∗*beginInterfaceClass*, ∗*endInterfaceClass*).

The static method template clears the vector *descriptors*, scans the system for all libusb controlled USB devices supporting one of the libusb controlled USB device interface classes described by that interval, stores the Device::Descriptor objects describing those devices in the vector *descriptors* and returns the size of that vector.

7.5.4.22 static size_t stm::UsbDevice::enumerateAll (std::vector< Descriptor > & *descriptors*) [static]

Enumerate all Device::Descriptor objects describing libusb controlled USB devices.

The static method template clears the vector *descriptors*, scans the system for all libusb controlled USB devices, stores the Device::Descriptor objects describing those devices in the vector *descriptors* and returns the size of that vector.

7.6 stm::UsbDevice::InterfaceClass Struct Reference

7.6.1 Detailed Description

Type describing a libusb controlled USB device interface class.

Such a libusb controlled USB device interface class is characterized by its vendor and product IDs.

Definition at line 750 of file usbdevice.hpp.

Public Member Functions

- InterfaceClass (unsigned short vendorId=0, unsigned short productId=0)

  Constructor yielding a UsbDevice::InterfaceClass object describing the libusb controlled USB device interface class characterized by its vendor and product IDs.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 stm::UsbDevice::InterfaceClass::InterfaceClass (unsigned short *vendorId* = 0, unsigned short *productId* = 0)

Constructor yielding a UsbDevice::InterfaceClass object describing the libusb controlled USB device interface class characterized by its vendor and product IDs.

7.7 stm::UsbPipe Struct Reference

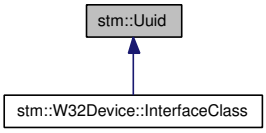
7.7.1 Detailed Description

Type specifying the configuration, the interface, the alternate setting, and the endpoint of the USB pipe to be used for a USB bulk or interrupt transfer.

Definition at line 123 of file usbdevice.hpp.

7.8 stm::Uuid Struct Reference

Inheritance diagram for stm::Uuid:



7.8.1 Detailed Description

Universal unique identifier storing its fields in little endian format.

Definition at line 307 of file device.hpp.

Public Member Functions

- Uuid ()

  Construct the null Uuid object with all octet bytes cleared.

- Uuid (const unsigned char array[Size])

  Construct the Uuid object defined by the octet bytes array.

- Uuid (unsigned long l, unsigned short w1, unsigned short w2, unsigned char b1, unsigned char b2, unsigned char b3, unsigned char b4, unsigned char b5, unsigned char b6, unsigned char b7, unsigned char b8)

  Construct an Uuid object from the components l, w1, w2, b1, b2, b3, b4, b5, b6, b7 and b8.

- Uuid (const Uuid &other)

  Copy constructor.

- Uuid & operator= (const Uuid &other)

  Assignment operator.

- std::string string () const

  Return the string representation of this Uuid.

- bool isNull () const

  Returns true, if this Uuid is null.

- bool operator== (const Uuid &other) const

  Equality comparison operator.

- bool operator!= (const Uuid &other) const

  Unequality comparison operator.

- bool operator< (const Uuid &other) const

Less than comparison operator.

- bool operator> (const Uuid &other) const

  Greater than comparison operator.

- bool operator<= (const Uuid &other) const

  Less or equal comparison operator.

- bool operator>= (const Uuid &other) const

  Greater or equal comparison operator.

Public Attributes

- unsigned char octet [Size]

  Structured byte array.

Static Public Attributes

- static const size_t Size = 16

  Number of bytes of a Uuid object.

7.8.2 Constructor & Destructor Documentation

7.8.2.1 stm::Uuid::Uuid ()

Construct the null Uuid object with all octet bytes cleared.

7.8.2.2 stm::Uuid::Uuid (const unsigned char *array*[Size]) [explicit]

Construct the Uuid object defined by the octet bytes *array*.

7.8.2.3 stm::Uuid::Uuid (unsigned long *l*, unsigned short *w1*, unsigned short *w2*, unsigned char *b1*, unsigned char *b2*, unsigned char *b3*, unsigned char *b4*, unsigned char *b5*, unsigned char *b6*, unsigned char *b7*, unsigned char *b8*)

Construct an Uuid object from the components l, w1, w2, b1, b2, b3, b4, b5, b6, b7 and b8.

7.8.2.4 stm::Uuid::Uuid (const Uuid & *other*)

Copy constructor.

7.8.3 Member Function Documentation

7.8.3.1 Uuid& stm::Uuid::operator= (const Uuid & *other*)

Assignment operator.

7.8.3.2 std::string stm::Uuid::string () const

Return the string representation of this Uuid.

7.8.3.3 bool stm::Uuid::isNull () const

Returns true, if this Uuid is null.

That means, if all octet bytes are cleared.

7.8.3.4 bool stm::Uuid::operator== (const Uuid & *other*) const

Equality comparison operator.

7.8.3.5 bool stm::Uuid::operator!= (const Uuid & *other*) const

Unequality comparison operator.

7.8.3.6 bool stm::Uuid::operator< (const Uuid & *other*) const

Less than comparison operator.

7.8.3.7 bool stm::Uuid::operator> (const Uuid & *other*) const

Greater than comparison operator.

7.8.3.8 bool stm::Uuid::operator<= (const Uuid & *other*) const

Less or equal comparison operator.

7.8.3.9 bool stm::Uuid::operator>= (const Uuid & *other*) const

Greater or equal comparison operator.

7.8.4 Member Data Documentation

7.8.4.1 const size_t stm::Uuid::Size = 16 [static]

Number of bytes of a Uuid object.

Definition at line 310 of file device.hpp.

7.8.4.2 unsigned char stm::Uuid::octet[Size]

Structured byte array.

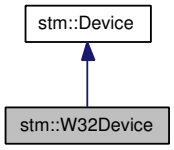
Its fields are stored in little endian format. In the description below the layout of its hexadecimal bytes hh is shown together with the corresponding *octet* array indexes.

```
{hh.hh.hh.hh-hh.hh-hh.hh-hh.hh-hh-hh-hh-hh-hh-hh}
  3  2  1  0  5  4  7  6  8  9  a  b  c  d  e  f
```

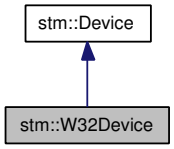
Definition at line 373 of file device.hpp.

7.9 stm::W32Device Class Reference

Inheritance diagram for stm::W32Device:



Collaboration diagram for stm::W32Device:



7.9.1 Detailed Description

Class defining the C++ API for a Windows device inheriting the generic stm::Device C++ API.

This class can be instantiated or be used as base class of a special Windows device class which may reimplement the interface of stm::W32Device as well as that of the abstract base class stm::Device.

Definition at line 128 of file w32device.hpp.

Public Types

- enum DescribeFlags {
  DevicePath = DriverVersion << 1,
  DeviceInstance = DevicePath << 1,
  DeviceIfClass = DeviceInstance << 1 }
  Describe flags (bitwise orable).

Public Member Functions

- W32Device (int defaultTimeout=Forever, const Descriptor &descr=Descriptor())
  Constructor of a W32Device object representing a Windows device described by descr.
- virtual ~W32Device ()
  Destructor.
- virtual bool canRead () const
  Return the read capability of this W32Device.
- virtual bool canWrite () const
  Return the write capability of this W32Device.

- virtual bool canControl () const

  Return the control capability of this W32Device.

- virtual int error () const

  Return the error state of this W32Device.

- virtual void setError (int error, const std::string &msg=std::string()) const

  Set the error state and error string of this W32Device according to error *and* msg.

- virtual void clearError () const

  Clear the error state and error string of this W32Device.

- virtual bool isOpen () const

  Determine, if this W32Device is open.

- virtual bool open (unsigned int openMode)

  Open this W32Device in openMode.

- virtual bool close ()

  Close this W32Device.

- virtual int64_t read (void *data, int64_t maxLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  Read maxLen *bytes from this W32Device into the* data *buffer.*

- virtual int64_t write (const void *data, int64_t len, int timeout=DefaultTimeout, unsigned int flags=NoFlags)

  Write len *bytes from the* data *buffer to this W32Device.*

- virtual int64_t control (unsigned int request, const void *inData, int64_t inLen, void *outData, int64_t maxOutLen, int timeout=DefaultTimeout, unsigned int flags=NoFlags) const

  Perform the control operation request *on this W32Device.*

- virtual std::ostream & describe (std::ostream &os, unsigned int flags=DefaultProperties) const

  Insert a description of this W32Device into os.

- bool isA (const InterfaceClass &interfaceClass) const

  The method returns true, if this W32Device is a device supporting the Windows device interface class described by interfaceClass.

- template<class ForwardIterator>
  bool isA (ForwardIterator beginInterfaceClass, ForwardIterator endInterfaceClass) const

  *The method template returns true, if this W32Device is a device supporting one of the the Windows device interface classes whose description is contained in the half open interval [*beginInterfaceClass*,* endInterfaceClass*).*

Static Public Member Functions

- static size_t enumerate (std::vector< Descriptor > &descriptors, const InterfaceClass &interfaceClass, bool append=false)

  Enumerate all Device::Descriptor objects describing Windows devices supporting the Windows device interface class described by interfaceClass.

- template<class ForwardIterator>
  static size_t enumerate (std::vector< Descriptor > &descriptors, ForwardIterator beginInterfaceClass, ForwardIterator endInterfaceClass, bool append=false)

  Enumerate all Device::Descriptor objects describing Windows devices supporting one of the the Windows device interface classes whose description is contained in the half open interval [∗beginInterfaceClass*,* ∗endInterfaceClass*).*

Classes

- struct InterfaceClass

  Type describing a Windows device interface class.

7.9.2 Member Enumeration Documentation

7.9.2.1 enum stm::W32Device::DescribeFlags

Describe flags (bitwise orable).

The enumerators of W32Device::DescribeFlags augment the Device::DescribeFlags further specifying the extent produced by describe().

Enumerator:

DevicePath If set, the device path property is included, else not.

DeviceInstance If set, the device instance property is included, else not.

DeviceIfClass If set, the device interface class property is included, else not.

Reimplemented from stm::Device.

Definition at line 137 of file w32device.hpp.

7.9.3 Constructor & Destructor Documentation

7.9.3.1 stm::W32Device::W32Device (int *defaultTimeout* = `Forever`, const Descriptor & *descr* = Descriptor `()`)

Constructor of a W32Device object representing a Windows device described by *descr*.

Parameters:

← ***defaultTimeout*** Timeout in milliseconds used by default for all operations of this W32Device.

← ***descr*** A valid Device::Descriptor of the Windows device to be represented by the W32Device object to be constructed or an invalid Device::Descriptor. A valid Device::Descriptor is typically yielded by one of the static methods enumerate() or enumerateAll().

Effects:

Constructs a W32Device object with the Device::Descriptor *descr* defined for it. If *descr* is valid, the constructed W32Device is ready to be opened.

See also:

descr(), open(), enumerate(), enumerateAll().

7.9.3.2 virtual stm::W32Device::~W32Device () [virtual]

Destructor.

Effects:

If this W32Device is open, it is closed first.

See also:

isOpen(), close().

7.9.4 Member Function Documentation

7.9.4.1 virtual bool stm::W32Device::canRead () const [virtual]

Return the read capability of this W32Device.

Returns:

`true`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device can be successfully opended in open mode Device::ReadAccess.
It is not necessary that this W32Device is open.

See also:

canWrite(), canControl(), canSeek(), open().

Reimplemented from stm::Device.

7.9.4.2 virtual bool stm::W32Device::canWrite () const [virtual]

Return the write capability of this W32Device.

Returns:

`true`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device can be successfully opended in open mode Device::WriteAccess.
It is not necessary that this W32Device is open.

See also:

canRead(), canControl(), canSeek(), open().

Reimplemented from stm::Device.

7.9.4.3 virtual bool stm::W32Device::canControl () const [virtual]

Return the control capability of this W32Device.

Returns:

`true`.

Note:

If this virtual method is not reimplemented by a derived class, this means that the device does support the method control().
It is not necessary that this W32Device is open.

See also:

canRead(), canWrite(), canSeek(), control().

Reimplemented from stm::Device.

7.9.4.4 virtual int stm::W32Device::error () const [virtual]

Return the error state of this W32Device.

Effects:

If the error state of this W32Device is Device::NoError, the Windows last-error code is cleared.

Returns:

The error state of this W32Device as one of the enumerators of Device::ErrorState.

Note:

It is not necessary that this W32Device is open.

See also:

setError(), clearError(), errorString().

Reimplemented from stm::Device.

7.9.4.5 virtual void stm::W32Device::setError (int *error*, const std::string & *msg* = `std::string()`) const [virtual]

Set the error state and error string of this W32Device according to *error* and *msg*.

Parameters:

← ***error*** Error state as one of the enumerators of Device::ErrorState optionally ored with one ore more of the enumerators of Device::ErrorFlags.

← ***msg*** Error string.

Effects:

If the error state part of *error* is one of the enumerators of Device::ErrorState, the error state of this W32Device is set to that state and its error string to *msg*, else to Device::UnknownError. If the error flag Device::SystemError is set in *error*, the error string is augmented by a system error description, if available.

Note:

Despite of being const, the method can change the error state and error string.
It is not necessary that this W32Device is open.

See also:

error(), clearError(), errorString(), augmentErrorString().

Reimplemented from stm::Device.

7.9.4.6 virtual void stm::W32Device::clearError () const [virtual]

Clear the error state and error string of this W32Device.

Effects:

The error state and error string of this W32Device are cleared, that means set to Device::NoError and the empty string. Moreover, the Windows last-error code is cleared.

Note:

Despite of being const, the method can change the error state and error string.
It is not necessary that this W32Device is open.

See also:

error(), setError(), errorString(), augmentErrorString().

Reimplemented from stm::Device.

7.9.4.7 virtual bool stm::W32Device::isOpen () const [virtual]

Determine, if this W32Device is open.

Returns:

`true`, if this W32Device is open, that is if its open mode is not the Device::OpenMode enumerator Device::NoAccess.
`false`, if this W32Device is not open, that is if its open mode is the Device::OpenMode enumerator Device::NoAccess.

See also:

open(), close().

Reimplemented from stm::Device.

7.9.4.8 virtual bool stm::W32Device::open (unsigned int *openMode*) [virtual]

Open this W32Device in *openMode*.

Parameters:

← ***openMode*** Open mode to be set.

Effects:

The method sets the error state of this W32Device to the Device::ErrorState enumerator Device::OpenError, if *openMode* is the Device::OpenMode enumerator Device::NoAccess, if isOpen() does not return false or if the Windows device represented by this W32Device cannot be opened conforming to *openMode*. Else the method sets the open mode of this W32Device to *openMode*.

Returns:

`true`, if this W32Device could be opened in *openMode*.
`false`, if this W32Device could not be opened in *openMode*. Then the error state of this W32Device is set to Device::OpenError.

See also:

isOpen(), close(), error ().

Reimplemented from stm::Device.

7.9.4.9 virtual bool stm::W32Device::close () [virtual]

Close this W32Device.

Effects:

The method sets the error state of this W32Device to the Device::ErrorState enumerator Device::CloseError, if isOpen() returns false or if the Windows device represented by this W32Device cannot be closed successfully. Else the method sets the open mode of this W32Device to the Device::OpenMode enumerator Device::NoAccess.

Returns:

`true`, if this W32Device could be closed successfully.
`false`, if this W32Device could not be closed successfully. Then the error state of this W32Device is set to Device::CloseError.

See also:

isOpen(), open(), error ().

Reimplemented from stm::Device.

7.9.4.10 virtual int64_t stm::W32Device::read (void * *data*, int64_t *maxLen*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) [virtual]

Read *maxLen* bytes from this W32Device into the *data* buffer.

Parameters:

→ ***data*** Buffer for the data to be read.

← ***maxLen*** Maximal number of bytes to be read.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this W32Device is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method sets the error state of this W32Device to the Device::ErrorState enumerator Device::ReadError, if the result of openMode() does not contain the Device::OpenMode enumerator Device::ReadAccess, or if *maxLen* is negative or *data* is the NULL pointer unless *maxLen* is also, 0 or if the read operation described below fails. During the read operation maximal *maxLen* bytes from the Windows device represented by this W32Device are read and stored in the buffer pointed to by *data*.

Returns:

The number of bytes actually read, if the read operation was successful.
`-1`, if the read operation was not successful. Then the error state of this W32Device is set to Device::ReadError.

See also:

write(), control(), openMode(), error (), defaultTimeout().

Reimplemented from stm::Device.

7.9.4.11 virtual int64_t stm::W32Device::write (const void * *data*, int64_t *len*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) `[virtual]`

Write *len* bytes from the *data* buffer to this W32Device.

Parameters:

← ***data*** Buffer containing the data to be written.

← ***len*** Number of bytes to be written.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this W32Device is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method sets the error state of this W32Device to the Device::ErrorState enumerator Device::WriteError, if the result of openMode() does not contain the Device::OpenMode enumerator Device::WriteAccess, or if *len* is negative or *data* is the NULL pointer unless *len* is also 0, or if the write operation described below fails. During the write operation *len* bytes from the buffer pointed to by *data* are written to the Windows device represented by this W32Device.

Returns:

len, if the write operation was successful.
`-1`, if the write operation was not successful. Then the error state of this W32Device is set to Device::WriteError.

See also:

read(), control(), openMode(), error (), defaultTimeout().

Reimplemented from stm::Device.

7.9.4.12 virtual int64_t stm::W32Device::control (unsigned int *request*, const void * *inData*, int64_t *inLen*, void * *outData*, int64_t *maxOutLen*, int *timeout* = `DefaultTimeout`, unsigned int *flags* = `NoFlags`) const `[virtual]`

Perform the control operation *request* on this W32Device.

Parameters:

← ***request*** Specifies the particular control operation to be performed.

← ***inData*** Buffer containing the input data for the control operation.

← ***inLen*** Number of bytes of the input data.

→ ***outData*** Buffer for the output data generated by the control operation.

← ***maxOutLen*** Maximal number of the output data.

← ***timeout*** Operation timeout in milliseconds. If the value is Device::Forever, no timeout occurs. The default value Device::DefaultTimeout means, that the default timeout of this W32Device is used.

← ***flags*** If the flag bit Device::AcceptTimeout is set, a timeout is no error.

Effects:

The method performs the control operation characterized by *request* for the Windows device represented by this W32Device. If any input data are required, the parameter *inData* shall point to a buffer of length *inLen* containing them, else *inData* shall be NULL and *inLen* 0. If any data result is expected, *outData* shall not be NULL and point to a buffer of size *maxOutLen*.

Returns:

The number of bytes available in *outData*, if the operation was successful. This is 0 in the case of an accepted timeout.
`-1`, if the operation was not successful. Then the error state of this W32Device is set to the Device::ErrorState enumerator Device::ControlError.

See also:

write(), read(), openMode(), error (), defaultTimeout().

Reimplemented from stm::Device.

7.9.4.13 virtual std::ostream& stm::W32Device::describe (std::ostream & *os*, unsigned int *flags* = `DefaultProperties`) const `[virtual]`

Insert a description of this W32Device into *os*.

Parameters:

← ***os*** The output stream to insert the description.

← ***flags*** Description flags.

Effects:

The method inserts a verbal description of this W32Device into the output stream *os*. Format and extent of the description is controlled by the *flags* parameter according to the bitwise ored enumerators of Device::DescribeFlags and W32Device::DescribeFlags.

Returns:

The output stream *os*.

Note:

This W32Device need not be open.

Reimplemented from stm::Device.

7.9.4.14 bool stm::W32Device::isA (const InterfaceClass & *interfaceClass*) const

The method returns true, if this W32Device is a device supporting the Windows device interface class described by *interfaceClass*.

That means it returns true, if the Device::Descriptor of this W32Device describes a Windows device supporting the Windows device interface class described by *interfaceClass*, else false.

7.9.4.15 template<class ForwardIterator> bool stm::W32Device::isA (ForwardIterator *beginInterfaceClass*, ForwardIterator *endInterfaceClass*) const

The method template returns true, if this W32Device is a device supporting one of the the Windows device interface classes whose description is contained in the half open interval [∗*beginInterfaceClass*, ∗*endInterfaceClass*).

That means it returns true, if the Device::Descriptor of this W32Device describes a Windows device supporting one of the Windows device interface classes described by that interval, else false.

7.9.4.16 static size_t stm::W32Device::enumerate (std::vector< Descriptor > & *descriptors*, const InterfaceClass & *interfaceClass*, bool *append* = `false`) `[static]`

Enumerate all Device::Descriptor objects describing Windows devices supporting the Windows device interface class described by *interfaceClass*.

The static method clears the vector *descriptors*, scans the system for all Windows devices supporting the Windows device interface class described by *interfaceClass*, stores the Device::Descriptor objects describing those devices in the vector *descriptors* and returns the size of that vector.

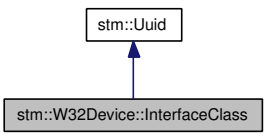
7.9.4.17 template<class ForwardIterator> static size_t stm::W32Device::enumerate (std::vector< Descriptor > & *descriptors*, ForwardIterator *beginInterfaceClass*, ForwardIterator *endInterfaceClass*, bool *append* = `false`) `[static]`

Enumerate all Device::Descriptor objects describing Windows devices supporting one of the the Windows device interface classes whose description is contained in the half open interval [∗*beginInterfaceClass*, ∗*endInterfaceClass*).

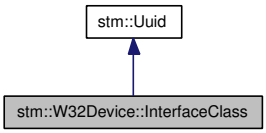
The static method template clears the vector *descriptors*, scans the system for all Windows devices supporting one of the Windows device interface classes described by that interval, stores the Device::Descriptor objects describing those devices in the vector *descriptors* and returns the size of that vector.

7.10 stm::W32Device::InterfaceClass Struct Reference

Inheritance diagram for stm::W32Device::InterfaceClass:



Collaboration diagram for stm::W32Device::InterfaceClass:



7.10.1 Detailed Description

Type describing a Windows device interface class.

Such a Windows device interface class is characterized by their GUID.

Definition at line 505 of file w32device.hpp.

Public Member Functions

- InterfaceClass ()

  Default constructor yielding an invalid W32Device::InterfaceClass object.

- InterfaceClass (const GUID &interfaceClassGuid)

  Constructor yielding a W32Device::InterfaceClass object describing the Windows device interface class characterized by their GUID interfaceClassGuid.

7.10.2 Constructor & Destructor Documentation

7.10.2.1 stm::W32Device::InterfaceClass::InterfaceClass ()

Default constructor yielding an invalid W32Device::InterfaceClass object.

7.10.2.2 stm::W32Device::InterfaceClass::InterfaceClass (const GUID & *interfaceClassGuid*)

Constructor yielding a W32Device::InterfaceClass object describing the Windows device interface class characterized by their GUID *interfaceClassGuid*.

8 SysToMath IO C++ Libraries Interface File Documentation

8.1 device.hpp File Reference

8.1.1 Detailed Description

Abstract base class stm::Device forming the ANSI-C++ API for generic devices.

Version:

1.03-r41

Date:

2007-07-31 09:31:16 (Tom)

Author:

Tom Michaelis
SysToMath
Wittelsbacherstr. 7
D-80469 Munich

Contact:

http://www.SysToMath.com
mailto:Tom.Michaelis@SysToMath.com

This header file declares the abstract base class stm::Device.

Definition in file device.hpp.

Namespaces

- namespace **stm**

Classes

- struct stm::Uuid

  Universal unique identifier storing its fields in little endian format.

- class stm::Device

  Abstract base class defining the C++ API for a generic device.

- struct stm::Device::Descriptor

  Objects of type Device::Descriptor describe system dependent aspects of a Device as a pair of a void pointer and a void function pointer.

- struct stm::Device::Version

  Device driver version.

Functions

- std::ostream & stm::operator<< (std::ostream &os, const Device &device)

  Insert a description of the Device device *into* os.

8.2 usbdevice.hpp File Reference

8.2.1 Detailed Description

Base class stm::UsbDevice forming the ANSI-C++ API for libusb controlled USB devices.

Version:

1.03-r41

Date:

2007-07-31 09:31:17 (Tom)

Author:

Tom Michaelis
SysToMath
Wittelsbacherstr. 7
D-80469 Munich

Contact:

http://www.SysToMath.com
mailto:Tom.Michaelis@SysToMath.com

This header file declares the base class stm::UsbDevice.

Definition in file usbdevice.hpp.

Namespaces

- namespace **stm**

Classes

- struct stm::UsbPipe

  Type specifying the configuration, the interface, the alternate setting, and the endpoint of the USB pipe to be used for a USB bulk or interrupt transfer.

- struct stm::UsbCtrl

  Type specifying the request type encoding the transfer direction, the value and the index of a USB control request.

- class stm::UsbDevice

  Class defining the C++ API for a libusb controlled USB device inheriting the generic stm::Device C++ API.

- struct stm::UsbDevice::InterfaceClass

  Type describing a libusb controlled USB device interface class.

8.3 w32device.hpp File Reference

8.3.1 Detailed Description

Base class stm::W32Device forming the ANSI-C++ API for Win32 devices.

Version:

1.01-r28

Date:

2007-06-13 01:28:56 (Tom)

Author:

Tom Michaelis
SysToMath
Wittelsbacherstr. 7
D-80469 Munich

Contact:

http://www.SysToMath.com
mailto:Tom.Michaelis@SysToMath.com

This header file declares the base class stm::W32Device.

Definition in file w32device.hpp.

Namespaces

- namespace **stm**

Classes

- class stm::W32Device

  Class defining the C++ API for a Windows device inheriting the generic stm::Device C++ API.

- struct stm::W32Device::InterfaceClass

  Type describing a Windows device interface class.

9 SysToMath IO C++ Libraries Interface Page Documentation

9.1 Microsoft Visual Studio Tool Family

The Microsoft Visual Studio tool family consists of the tool sets:

- `Microsoft Visual Studio .NET 2003 (vc71)`
- `Microsoft Visual Studio 2005 (vc80)`

9.1.1 Automatic Linking with Microsoft Visual Studio

On Microsoft Visual Studio .NET 2003 (`vc71`) and Microsoft Visual Studio 2005 (`vc80`) the necessary libraries are linked automatically when one of the main library interface header files stm/w32device.hpp or stm/usbdevice.hpp , unless this mechanism is suppressed by the definition of the preprocessor symbol `STM_NO_LIB`, `STM_W32DEVICE_NO_LIB` or `STM_USBDEVICE_NO_LIB` before that inclusion.

The choice of the libraries depends on the tool set used (`vc71` or `vc80`) and on the system runtime library selected for the executable to be built. The SysToMath IO C++ Libraries package provides for each of its non-header-only library modules four static library configurations (`lib` files) and two dynamic ones (`dll` files). In the following list *`module`* stands for `w32device` or `usbdevice` and *`vcnn`* for `vc71` or `vc80`:

- **DebugMt**: Static debug library libstm*module*-*vcnn*-mt-gd.lib together with its debug database file libstm*module*-*vcnn*-mt-gd.pdb chosen when linking against the multithreaded debug DLL runtime (Compiler switch /MDd).
- **DebugDllMt**: Dynamic debug library stm*module*-*vcnn*-mt-gd.dll together with its import library stm*module*-*vcnn*-mt-gd.lib and its debug database file stm*module*-*vcnn*-mt-gd.pdb chosen when linking against the multithreaded debug DLL runtime (Compiler switch /MDd) and defining the preprocessor variable STM_DYN_LINK before the SysToMath C++ Libraries header file inclusion.
- **DebugMtStaticRt**: Static debug library libstm*module*-*vcnn*-mt-sgd.lib together with its debug database file libstm*module*-*vcnn*-mt-sgd.pdb chosen when linking against the multithreaded static debug runtime(Compiler switch /MTd).
- **ReleaseMt**: Static release library libstm*module*-*vcnn*-mt.lib chosen when linking against the multithreaded release DLL runtime (Compiler switch /MD).
- **ReleaseDllMt**: Dynamic release library stm*module*-*vcnn*-mt.dll together with its import library stm*module*-*vcnn*-mt.lib chosen when linking against the multithreaded release DLL runtime (Compiler switch /MD) and defining the preprocessor variable STM_DYN_LINK before the SysToMath C++ Libraries header file inclusion.
- **ReleaseMtStaticRt**: Static release library libstm*module*-*vcnn*-mt-s.lib chosen when linking against the multithreaded static release runtime (Compiler switch /MT).

9.1.2 Environment

It is recommended that all static libraries (lib files) and their debug database files (pdb files) are located in a directory contained in the compiler system library search path. Moreover, to satisfy the application runtime requirements, it is recommended that all dynamic link libraries (dll files) and their debug database files (pdb files) are located in the application directory or in a directory contained in the system executable search path.

If you used the installation program LibStmIoSetup.exe to install the SysToMath IO C++ Libraries with the installation root directory, say C:\Program Files\SysToMath and you use Microsoft Visual Studio .NET 2003 (vc71) or Microsoft Visual Studio 2005 (vc80), then the aforementioned compiler system directory recommendations are satisfied, if you add the following entries in Visual Studio, menu Tools, Options, Projects, VC++ Directories:

- Executable Files: Add C:\Program Files\SysToMath\bin\w32
- Library Files: Add C:\Program Files\SysToMath\lib\w32
- Include Files: Add C:\Program Files\SysToMath\include

9.2 GNU Tool Family

The GNU tool family consists of the tool sets:

- GNU gcc for POSIX environment (cygwin)
- GNU gcc for Microsoft environment (cygming)

The tool set cygwin (chosen by gcc or g++ without option -mno-cygwin) produces libraries depending on the dynamic library cygwin1.dll and therefore is restricted to free open source software projects, whereas the tool set cygming (chosen by gcc or g++ with option -mno-cygwin) produces libraries only depending on the Microsoft runtime libraries and thus also allows commercial closed source software projects.

9.2.1 Linking with the GNU Tool Family

The libraries for the GNU tool family have to be linked explicitely.

The choice of the libraries depends on the tool set used (cygwin or cygming) and on the system runtime library selected for the executable to be built. The SysToMath IO C++ Libraries package provides for each of its library modules two static library configurations (a files) and two dynamic ones (dll files). In the following list *module* stands for w32device or usbdevice and *cygxxx* for cygwin or cygming:

- **DebugMt**: Static debug library libstm*module*-*cygxxx*-mt-d.a to be chosen for multithreaded debug enabled statically linked executables (the preprocessor variables _MT and _DEBUG should be defined).
- **DebugDllMt**: Dynamic debug library stm*module*-*cygxxx*-mt-d.dll together with its import library libstm*module*-*cygxxx*-mt-d.dll.a to be chosen for multithreaded debug enabled dynamically linked executables (the preprocessor variables _MT, _DEBUG and STM_DYN_LINK should be defined).
- **ReleaseMt**: Static release library libstm*module*-*cygxxx*-mt.a to be chosen for multithreaded not debug enabled statically linked executables (the preprocessor variables _MT and NDEBUG should be defined).
- **ReleaseDllMt**: Dynamic release library stm*module*-*cygxxx*-mt.dll together with its import library libstm*module*-*cygxxx*-mt.dll.a to be chosen for multithreaded not debug enabled dynamically linked executables (the preprocessor variables _MT, NDEBUG and STM_DYN_LINK should be defined).

9.2.2 Environment

It is recommended that all static libraries (a files) are located in a directory contained in the compiler system library search path. Moreover, to satisfy the application runtime requirements, it is recommended that all dynamic link libraries (dll files) are located in the application directory or in a directory contained in the system executable search path.

If you used the installation program LibStmIoSetup.exe to install the SysToMath C++ Libraries with the installation root directory, say C:\Program Files\SysToMath, and you have added /cygdrive/c/Program Files/SysToMath/bin/w32 to your PATH environment variable, then the aforementioned compiler system directory recommendations are satisfied, if you use the following options in your gcc or g++ command lines:

- Include Files: Use -I"/cygdrive/c/Program Files/SysToMath/include"
- Library Files: Use -L"/cygdrive/c/Program Files/SysToMath/lib/w32"

9.3 SysToMath Device C++ Library (Headers only)

9.3.1 Content

The SysToMath Device C++ Library consits of header files only.

9.3.2 Usage

To compile an application which uses the SysToMath Device C++ Library, the public main interface header file

- stm/device.hpp

shall be included and be located in a directory contained in the compiler system include search path. Moreover, the implementation header file

- `stm/impl/xdevice.hpp`

shall also be located in a directory contained in the compiler system include search path.

9.4 SysToMath UsbDevice C++ Library

9.4.1 Content

The SysToMath UsbDevice C++ Library consits of the following object:

- UsbDevice: Base Class for USB Devices

9.4.2 Usage

To compile an application which uses the SysToMath USB Device C++ Library, the public main library interface header file

- stm/usbdevice.hpp

shall be included and be located in a directory contained in the compiler system include search path. Moreover, the implementation header files

- `stm/impl/usbdeviceconfig.hpp`
- `stm/impl/xusbdevice.hpp`

shall also be located in a directory contained in the compiler system include search path.

9.5 SysToMath W32Device C++ Library

9.5.1 Content

The SysToMath W32Device C++ Library consits of the following object:

- W32Device: Base Class for Win32 Devices

9.5.2 Usage

To compile an application which uses the SysToMath Win32 Device C++ Library, the public main library interface header file

- stm/w32device.hpp

shall be included and be located in a directory contained in the compiler system include search path. Moreover, the implementation header files

- `stm/impl/w32deviceconfig.hpp`
- `stm/impl/xw32device.hpp`

shall also be located in a directory contained in the compiler system include search path.

Index