AnaGate API 2

Programmer's Manual

Analytica GmbH

A. Schmidt, Analytica GmbH
S. Welisch, Analytica GmbH
J. Gossens, Analytica GmbH

AnaGate API 2: Programmer's Manual

Analytica GmbH
by A. Schmidt, S. Welisch, and J. Gossens

This document was generated with DocBook at 2016-04-07 11:01:06.

Hilfe-Datei (dtsch.): *Manual-AnaGateAPI.chm*

Hilfe-Datei (engl.): *Manual-AnaGateAPI-EN.chm*

PDF-Datei (dtsch.): *Manual-AnaGateAPI-2.2.pdf*

PDF-Datei (engl.): *Manual-AnaGateAPI-2.2-EN.pdf*

Publication date 2014-12-10

Abstract

The AnaGate Programmer's Manual includes the exact description of the programming interfaces to all models of *AnaGate* hardware series.

This manual bases on the actual *AnaGate* Application Programming Interface (API) in Version 2.2 and the AnaGate communication protocol V1.3 (see [TCP-2010]).

Analytica GmbH
Bannwaldallee 60
76185 Karlsruhe
Germany
Fon +49 (0) 721-43035-0
Fax +49 (0) 721-43035-20
<support@analytica-gmbh.de>

www.analytica-gmbh.de [http://www.analytica-gmbh.de]
www.anagate.de [http://www.anagate.de]

Revision History			
Revision 2.2	25.06.2015	ASc	Description for SocketCAN support added.
Revision 2.1	25.04.2014	SWe	Added description and example for user specified can baudrate
Revision 2.0	15.05.2013	JGo	Complete revision for AnaGate API 2
Revision 1.7	25.02.2013	SWe	LS_I2COpenDeviceEx added
Revision 1.6	09.11.2012	JGo	CANSetMaxSizePerQueue and CANGetMessage added
Revision 1.5	31.08.2012	JGo	I2CWriteEEPromPollAck added
Revision 1.4	01.10.2010	ASc	Complete revision of all chapters

Revision 1.3	12.07.2010	SWe	CAN UPD functions added (Lua only)
Revision 1.2	04.06.2010	SWe	I2C RAW functions added temporarily
Revision 1.1	01.04.2010	ASc	english version
Revision 1.0	08.06.2009	ASc	Manual changed to **DocBook** format

Table of Contents

List of Figures

List of Tables

List of Examples

Introduction

The AnaGate Programmer's Manual includes the exact description of the programming interfaces to all models of AnaGate hardware series.

The existing interfaces will be decribed below:

- Application Programming Interface (Part I, “AnaGate API”)
- Linux SocketCAN Interface(Part II, “ SocketCAN interface ”)
- Lua Scripting Interface (Part III, “ Scripting language Lua ”)

Part I. AnaGate API

Table of Contents

Chapter 1. The Programming interface of AnaGate product line

The *AnaGate* product line consist of several hardware devices which offer access to different bus systems (I2C, SPI, CAN) or processors (Renesas) via a standard network protocol.

The communication to the individual devices always is done through a documented and disclosed proprietary network protocol. Thus, all products which incorporates a socket interface (like personal computers, PLC, ...) are able to access the devices of the *AnaGate* product line.

Analytica provides a programming interface for users of Windows and Linux operating systems (X86) which implements the proprietary communication protocol and makes it available through simple function calls. The software API (Application Programming Interface) is available free of charge for Windows and Linux operating systems.

Device	Windows library 32-bit	64-bit
AnaGate CAN	AnaGateCan.dll	AnaGateCan64.dll
AnaGate CAN uno / duo / quattro	AnaGateCan.dll	AnaGateCan64.dll
AnaGate CAN USB	AnaGateCan.dll	AnaGateCan64.dll
AnaGate CAN X2 / X4 / X8	AnaGateCan.dll	AnaGateCan64.dll
AnaGate SPI	AnaGateSPI.dll	AnaGateSPI64.dll
AnaGate I2C / I2C X7	AnaGateI2C.dll	AnaGateI2C64.dll
AnaGate Universal Programmer	AnaGateSPI.dll, AnaGateI2C.dll	AnaGateSPI64.dll, AnaGateI2C64.dll

Table 1.1. Library files for Windows

Note

To provide a widespread support of different programming languages like C++, Visual Basic, Delphi and the programming languages of the .NET family, the cdecl calling convention is used in all function calls. Using this calling convention means that all function parameters are pushed on the stack in reverse order (from right to left) and that the caller is responsible for the stack handling. Most programming languages support this calling convention.

Device	Linux library (X86)	ARM9
AnaGate CAN	libCANDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a	-
AnaGate CAN uno / duo / quattro	libCANDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a	available

Device	Linux library (X86)	ARM9
AnaGate CAN USB	libCANDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a	available
AnaGate SPI	libSPIDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a	-
AnaGate I2C / I2C X7	libI2CDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a	-
AnaGate Universal Programmer	libSPIDLLStaticRelease.a, libI2CDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a	available

Table 1.2. Library files for 32-bit Linux

Device	Linux library (X86-64)
AnaGate CAN	libCANDLLStaticRelease64.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a
AnaGate CAN uno / duo / quattro	libCANDLLStaticRelease64.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a
AnaGate CAN USB	libCANDLLStaticRelease64.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a
AnaGate SPI	libSPIDLLStaticRelease64.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a
AnaGate I2C / I2C X7	libI2CDLLStaticRelease64.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a
AnaGate Universal Programmer	libSPIDLLStaticRelease64.a, libI2CDLLStaticRelease.a, libAnaGateExtStaticRelease.a, libAnaGateStaticRelease.a

Table 1.3. Library files for 64-bit Linux

The different libraries include common and specific functions which are necessary for accessing and controlling the devices of the *AnaGate* product line. In the following, all library functions of the software API are documented in detail.

Tip

It is possible to extend individually the newer device models with embedded Linux (kernel 2.6) and ARM9 processor. The complete software API is available in a cross-compiled version and can be used on the devices

itself to create individual device extensions. To do so is very easy because the programming interface on the personal computer and the device is completely identical.

A preconfigured virtual machine (Virtual-Box-Image) with Ubuntu Linux "READY-to-USE" with installed development environment (**Kdevelop**, **Eclipse**) and all necessary program libraries (**GCC**, cross compiler, libraries, **Lua**, ...) is available optionally.

Chapter 2. Notes concerning the communication protocol TCP

Access to the different models of the *AnaGate* product line is always done via the most frequently used network protocol TCP (Transmission Control Protocol).

TCP is connection-oriented packet-switched transport protocol which is located in layer 4 of the OSI reference model. In principle TCP is an end-to-end connection which allows exchange of data in both directions at the same time. An end-point is a pair formed of an IP address and a port number. Such a pair forms a bidirectional software interface and is called socket.

The *AnaGate* device offers its functionality as a so-called TCP server. It creates a socket with its IP address and a device-specific port number. On the models with CAN interface(s) a separate socket with a different port number is created for each existing CAN interface; on each socket up to 5 concurrent client connections are accepted. The SPI, I2C and Renesas interfaces accept only one connection at a time.

Device	Port number
AnaGate I2C, AnaGate Universal Programmer	5000
AnaGate CAN, AnaGate CAN uno	5001
AnaGate CAN duo, AnaGate CAN X2	5001, 5101
AnaGate CAN quattro, AnaGate CAN X4	5001, 5101, 5201, 5301
AnaGate CAN X8	5001, 5101, 5201, 5301, 5401, 5501, 5601, 5701
AnaGate SPI, AnaGate Universal Programmer	5002
AnaGate Renesas, AnaGate Universal Programmer	5008

Table 2.1. AnaGate devices and related port numbers

Important

Please ensure that all used ports are set active on the personal computer to grant access to the *AnaGate* device. Any existing firewalls are to be configured accordingly.

2.1. Important properties of the network protocol

In most cases TCP is based on the internet protocol (IP). IP is package-oriented, whereby it is possible that data packets are lost or the packets can be received in wrong order or perhaps more than once.

TCP eliminates this behaviour and ensures that the the data packets are received in correct order at the recipient. If a sent data packet is not confirmed by the recipient within a timeout limit, the packet is sent again. Duplicate packets are

recognized by the recipient and are deleted. During the lifetime of a connection the data transmission may be impaired, delayed or completely interrupted. A successful connection creation do not guarantee a permanently stable data transmission.

Detection and evaluation of network and line malfunctions can be difficult if there is only sporadic communication on the line. How is it possible to distinguish between a malfunction on the line or simply no data from the connected endpoint?

To amend this problematic nature TCP provides an internal keep alive mechanism. Keep-alives are special data packets which are sent in regular intervals between the two endpoints of an opened communication channel. The recipient of a keep-alive packet has to confirm the receipt to the sender within a certain period of time. When there are no keep-alives or confirmations of keep-alives received, the communication partner assumes that the channel is interrupted or the corresponding socket is malfunctioning.

The keep-alive mechanism of TCP is not active by default and has to be activated via the `setsockopt` function for each connection. The API functions which establish a connection to an *AnaGate* - like the `CANOpenDevice()` function - strictly activate the keep-alive mechanism of TCP.

Note

On Windows operating systems some settings concerning keep-alives can be set individually. These settings are valid for all network connection on this computer and can not be set individually for dedicated connections.

To do so the Windows registry keys **KeepAliveTime** and **KeepAliveInterval** of the node `\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Tcpip\Parameters` have to be adjusted (requires administrator rights).

Especially the CAN-Ethernet gateways can be affected by the above described problems, for example if customer-specific needs ask for faster detection of connection aborts than possible via the standard mechanism. So, for the AnaGate models with CAN interface and linux OS an application-specific keep-alive algorithm is integrated in the device firmware to enhance connection control. Based on a predefined time period additional data packets are exchanged between the AnaGate hardware and the controlling unit/personal computer which have to be confirmed by the corresponding endpoint (ALIVE_REQ, see [TCP-2010]). This integrated alive mechanism can be activated individually on each connection with a different timeout interval.

Note

Users of the AnaGate API do not have to implement the application-specific alive mechanism to use it. With a simple call to the API function `CANStartAlive` a concurrent thread is started which automatically monitors the communication channel time-controlled.

Chapter 3. Common function reference

The library interface of the AnaGate API is implemented in the programming language C. The default data types of this programming language aren't set to fixed value ranges across platforms which can complicate accessing C functions from other programming languages. Fixed-width data types were only introduced in newer C versions beginning with C99. However, these C versions aren't supported by all, especially older compilers.

To establish compatibility to such compilers which do not support this version of the C standard fully or at all, the AnaGate API defines and uses its own aliases for fixed-width range data types. These data types allow easy access to API functions from other programming languages by explicitly defining the memory sizes of individual parameters.

Bits width	Signed type	Unsigned type
8	AnaInt8	AnaUInt8
16	AnaInt16	AnaUInt16
32	AnaInt32	AnaUInt32
64	AnaInt64	AnaUInt64

Table 3.1. Fixed width data types

DLLInfo

DLLInfo — Determines the current version information of the AnaGate DLL.

Syntax

```
#include <AnaGateDLL.h>

AnaInt32 DLLVersion(char * pcMessage, AnaInt32 nMessageLen);
```

Parameter

pcMessage Data buffer that is to accept the version reference number of the AnaGate DLL.

nMessageLen Size in bytes of the transferred data buffer.

Return value

Actual size of the returned version reference number.

Remarks

If the version reference number is too large for the transferred data buffer, it is abbreviated to the given number of characters (*nMessageLen*).

Chapter 4. CAN API reference

The CAN API can be used with all CAN gateway models of the AnaGate series. The programming interface is identical for all devices and generally uses the network protocol TCP or UDP.

The following devices can be addressed via the CAN API interface:

- AnaGate CAN
- AnaGate CAN uno
- AnaGate CAN duo
- AnaGate CAN quattro
- AnaGate CAN USB
- AnaGate CAN X2
- AnaGate CAN X4
- AnaGate CAN X8

CANOpenDevice, CANOpenDeviceEx

CANOpenDevice, CANOpenDeviceEx — Opens an network connection (TCP or UDP) to an AnaGate CAN device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANOpenDevice(AnaInt32 *pHandle, AnaInt32 bSendDataConfirm,
AnaInt32 bSendDataInd, AnaInt32 nCANPort, const char * pcIPAddress,
AnaInt32 nTimeout);

AnaInt32 CANOpenDeviceEx(AnaInt32 *pHandle, AnaInt32 bSendDataConfirm,
AnaInt32 bSendDataInd, AnaInt32 nCANPort, const char * pcIPAddress,
AnaInt32 nTimeout, AnaInt32 nSocketType);
```

Parameter

pHandle	Pointer to a variable in which the access handle is saved in case of a successful connection to the AnaGate device.
bSendDataConfirm	If set to TRUE, all incoming and outgoing data requests are confirmed by the internal message protocol. Without confirmations a better transmission performance is reached.
bSendDataInd	If set to FALSE, all incoming telegrams are discarded.
nCANPort	CAN port number. Allowed values are: `0` for port 1/A (all AnaGate CAN models) `1` for port 2/B (AnaGate CAN duo, AnaGate CAN quattro, AnaGate CAN X2/X4/X8) `2` for port 3/C (AnaGate CAN quattro,AnaGate CAN X4/X8) `3` for port 4/D (AnaGate CAN quattro, AnaGate CAN X4/X8) `4` for port 5/E (AnaGate CAN X8) `5` for port 6/F (AnaGate CAN X8) `6` for port 7/G (AnaGate CAN X8) `7` for port 8/H (AnaGate CAN X8)
pcIPAddress	Network address of the AnaGate partner.
nTimeout	Default timeout for accessing the AnaGate in milliseconds. A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

nSocketType	Specifies the socket type (ethernet layer 4) which is to be used for the new connection. Only two different types are supported: TCP and UDP. The function `CANOpenDevice` alyways uses TCP sockets. Use the following constants for the parameter: 1 (SOCK_STREAM) TCP (Transmission Control Protocol) 2 (SOCK_DGRAM) UDP (User Datagram Protocol)

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Opens a TCP/IP connection to an CAN interface of an AnaGate CAN device. By using `CANOpenDeviceEx` it is possible to set the ethernet layer 4 protocol (TCP or UDP). If the connection is established, CAN telegrams can be sent and received.

The connection should be closed with the function `CANCloseDevice` if not longer needed.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See the following example for the initial programming steps.

```
#include <AnaGateDllCan.h>
int main()
{
   AnaInt32 hHandle;
   AnaInt32 nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // ... now do something
      CANCloseDevice(hHandle);
   }
   return 0;
}
```

Remarks

The *`CANOpenDeviceEx`* function is supported for library versions 1.5-1.10 or higher and firmware version 1.3.7 or higher.

Device models of type AnaGate CAN (hardware version 1.1.A) do not listen for UPD connection requests. If trying to connect such a device via UPD, the `CANOpenDeviceEx` function returns with a timeout error.

See also

CANCloseDevice

CANRestart

CANCloseDevice

CANCloseDevice — Closes an open network connection to an AnaGate CAN device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANCloseDevice(AnaInt32 hHandle);
```

Parameter

hHandle Valid access handle.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Closes an open network connection to an AnaGate CAN device. The *hHandle* parameter is a return value of a succesfull call to the function `CANOpenDevice`.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See also

CANOpenDevice, CANOpenDeviceEx

CANSetGlobals

CANSetGlobals — Sets the global settings which are to be used on the CAN bus.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANSetGlobals(AnaInt32 hHandle, AnaUInt32 nBaudrate, AnaUInt8
nOperatingMode, AnaInt32 bTermination, AnaInt32 bHighSpeedMode,
AnaInt32 bTimeStampOn);
```

Parameter

hHandle Valid access handle.

nBaudrate The baud rate to be used. The following values are supported:

- 10.000 für 10kBit
- 20.000 für 20kBit
- 50.000 für 50kBit
- 62.500 für 62,5kBit
- 100.000 für 100kBit
- 125.000 für 125kBit
- 250.000 für 250kBit
- 500.000 für 500kBit
- 800.000 für 800kBit (not AnaGate CAN)
- 1.000.000 für 1MBit

It is also possible, to set individual baud rates on the device. In this case the registers CNF1, CNF2 and CNF3 of the CAN controller MCP2515 have to be set directly. If bit 24 of the baudrate parameter is set, the lower 3 bytes of the parameter value are copied directly in the control registers in the following way:

	MCP2515 CNF1	**MCP2515 CNF2**	**MCP2515 CNF3**	**Baudrate parameter**
	0xAA	0xBB	0xCC	0x01AABBCC
For example: 1MHz	0x41	0x89	0x00	0x01418900
For example: 250KHz	0x43	0xA1	0x03	0x0143A103

Table 4.1. Example configuration of can baudrate

Note

Please refer to the manual of the MCP2515 for a description of the configuration registers CNF1, CNF2, CNF3. The oscillator frequency of the AnaGate CAN is FOSC=20MHz, all other models use the frequency FOSC= 24MHz.

nOperatingMode The operating mode to be used. The following values are allowed.

- 0 = default mode.
- 1 = loop back mode: No telegrams are sent via CAN bus. Instead they are received as if they had been transmitted over CAN by a different CAN device.
- 2 = listen mode: Device operates as a passive bus partner, meaning no telegrams are sent to the CAN bus (nor ACKs for incoming telegrams).
- 3 = offline mode: No telegrams are sent or received on the CAN bus. Thus no error frames are generated on the bus if other connected CAN devices send telegrams with a different baud rate.

bTermination Use integrated CAN bus termination (TRUE=yes, FALSE=no). This setting is not supported by all AnaGate CAN models.

bHighSpeedMode Use high speed mode (TRUE=yes, FALSE=no). This setting is not supported by all AnaGate CAN models.

The high speed mode was created for large baud rates with continuously high bus load. In this mode telegrams are not confirmed on the protocol layer and the software filters defined via `CANSetFilter` are ignored.

bTimeStampOn Use time stamp mode (TRUE=yes, FALSE=no). This setting is not supported by all AnaGate CAN models.

In activated time stamp mode an additional timestamp is sent with the CAN telegram. This timestamp indicates when the incoming message was received by the CAN controller or when the outgoing message was confirmed by the CAN controller.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Sets the global settings of the used CAN interface. These settings are effective for all concurrent connections to the CAN interface. The settings are not saved permanently on the device and are reset after every device restart.

Remarks

The settings of the integrated CAN bus termination, the high speed mode and the time stamp are not supported by the AnaGate CAN (hardware version 1.1.A). These settings are ignored by the device.

The offline mode is only supported by firmware versions 1.3.12 and higher. The AnaGate CAN doesn't support this mode.

See also

CANGetGlobals

CANGetGlobals

CANGetGlobals — Returns the currently used global settings on the CAN bus.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANGetGlobals(AnaInt32 hHandle, AnaUInt32 * pnBaudrate,
AnaUInt8 * pnOperatingMode, AnaInt32 * pbTermination, AnaInt32 *
pbHighSpeedMode, AnaInt32 * pbTimeStampOn);
```

Parameter

hHandle Valid access handle.

pnBaudrate The baud rate currently used on the CAN bus.

pnOperatingMode The operating mode to be used. The following values are returned.

- 0 = default mode.
- 1 = loop back mode: No telegrams are sent via CAN bus. Instead they are received as if they had been transmitted over CAN by a different CAN device.
- 2 = listen mode: Device operates as a passive bus partner, meaning no telegrams are sent to the CAN bus (nor ACKs for incoming telegrams).
- 3 = offline mode: No telegrams are sent or received on the CAN bus. Thus no error frames are generated on the bus if other connected CAN devices send telegrams with a different baud rate.

pbTermination Is the integrated CAN bus termination used? (TRUE=yes, FALSE=no). This setting is not supported by all AnaGate CAN models.

pbHighSpeedMode Is the high speed mode switched on? (TRUE=yes, FALSE=no). This setting is not supported by all AnaGate CAN models.

The high speed mode was created for large baud rates with continuously high bus load. In this mode telegrams are not confirmed on the protocol layer and the software filters defined via `CANSetFilter` are ignored.

pbTimeStampOn Is a timestamp mode activated on the current network connection? (TRUE=yes, FALSE=no). This setting is not supported by all AnaGate CAN modells.

In activated time stamp mode an additional timestamp is sent with the CAN telegram. This timestamp indicates when the incoming

message was received by the CAN controller or when the outgoing message was confirmed by the CAN controller.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Returns the global settings of the used CAN interface. These settings are effective for all concurrent connections to the CAN interface.

Remarks

The settings of the integrated CAN bus termination, the high speed mode and the time stamp are not supported by the AnaGate CAN (hardware version 1.1.A). These settings are ignored by the device.

The offline mode is only supported by firmware versions 1.3.12 and higher. The AnaGate CAN doesn't support this mode.

See also

CANGetGlobals

CANSetFilter

CANSetFilter — Sets the current filter settings for the connection.

Syntax

#include <AnaGateDllCan.h>

```
AnaInt32 CANSetFilter(AnaInt32 hHandle, const AnaUInt32 * pnFilter);
```

Parameter

hHandle Valid access handle.

pnFilter Pointer to an array of 8 filter entries (4 mask and 4 range filter entries). A filter entry contains of two 32-bit values. Unused mask filter entries must be initialized with 0 values. Unused range filter entries must be initialized with 0 for the start value and 0x1FFFFFFF for the end value.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

This function sets the current filter settings for the current connection. Filters can be used to only transmit messages with specific CAN message IDs.

A mask filter contains of a mask value which defines the bits of the CAN identifier to examine, and the appropriate filter value. If the CAN identifier matches the filter value in the indicated filter mask, the incoming CAN telegram is sent to the PC, otherwise not.

A range filter defines an address range with an appropriate start and end address. If the CAN identifier do not lie in the indicated filter range, the incoming CAN telegram is not sent to the PC.

By default no filters are set so all CAN IDs are transmitted. If the parameter *bHighSpeedMode* of the `CANSetGlobals` function is set all filters are ignored to increase the data throughput.

CAN ID	mask value	filter value	result
0x0F	0x0E	0x0C	suppressed
0x0C	0x0E	0x0C	ok
0x5D	0x0E	0x0C	ok

Table 4.2. mask filter examples for CAN identifier

See the following example for setting some filters.

```
#include <AnaGateDllCan.h>
```



```
int main()
{
   AnaUInt32 anFilter[16] = {
      0xFF, 0x0F,    // mask  filter 1: mask = 0xFF, value = 0x0F: route only 0x*0F values
      0, 0,          // mask  filter 2: unused
      0, 0,          // mask  filter 3: unused
      0, 0,          // mask  filter 4: unused
      0, 0x00000FFF, // range filter 1: all IDs greater than 0xFFF are discarded
      0, 0x1FFFFFFF, // range filter 2: unused
      0, 0x1FFFFFFF, // range filter 3: unused
      0, 0x1FFFFFFF, // range filter 4: unused
   };
   AnaInt32 hHandle;
   AnaInt32 nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      nRC = CANSetFilter( hHandle, &anFilter );
      // ... now do something
      CANCloseDevice(hHandle);
   }
   return 0;
}
```

See also

CANGetFilter

CANGetFilter

CANGetFilter — Returns the current filter settings for the connection.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANGetFilter(AnaInt32 hHandle, AnaUInt32 * pnFilter);
```

Parameter

hHandle Valid access handle.

pnFilter Pointer to an array of 8 filter entries (4 mask and 4 range filter entries). A filter entry contains of two 32-bit values. Unused mask filter entries are initialized with 0 values. Unused range filter entries are initialized with (0,0x1FFFFFFF) value pairs.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

This function retrieves the current filter settings for the current connection. Filter can be used to suppress messages with specific CAN message IDs.

See also

CANSetFilter

CANSetTime

CANSetTime — Sets the current system time on the AnaGate device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANSetTime(AnaInt32 hHandle, AnaUInt32 nSeconds, AnaUInt32 nMicroseconds);
```

Parameter

hHandle Valid access handle.

nSeconds Time in seconds from 01.01.1970.

nMicroseconds Micro seconds.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The `CANSetTime` function sets the system time on the AnaGate hardware.

If the time stamp mode is switched on by the `CANSetGlobals` function, the AnaGate hardware adds a time stamp to each incoming CAN telegram and a time stamp to the confirmation of a telegram sent via the API (only if confirmations are switched on for data requests).

Remarks

The `CANSetTime` function is supported by library version 1.4-1.8 or higher.

The setting of the base time for the time stamp mode is not supported by the AnaGate CAN (hardware version 1.1.A). This setting is ignored by the device.

See also

CANGetTime

CANGetTime

CANGetTime — Gets the current system time from the AnaGate CAN device.

Syntax

#include <AnaGateDllCan.h>

```
AnaInt32 CANGetTime(AnaInt32 hHandle, AnaInt32 * pbTimeWasSet, AnaUInt32
* pnSeconds, AnaUInt32 * pnMicroseconds);
```

Parameter

hHandle Valid access handle.

pbTimeWasSet True if the system time was set via `CANSetTime`.

pnSeconds Time in seconds from 01.01.1970.

pnMicroseconds Micro seconds.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Gets the current system time from the AnaGate CAN device.

If the time stamp mode is switched on by the `CANSetGlobals` function, the AnaGate hardware adds a time stamp to each incoming CAN telegram and a time stamp to the confirmation of a telegram sent via the API (only if confirmations are switched on for data requests).

Remarks

The `CANGetTime` function is supported by library version 1.7-1.13 or higher and firmware version 1.3.16 or higher.

The setting of the base time for the time stamp mode cannot be read from the AnaGate CAN (hardware version 1.1.A). The command is ignored by the device.

See also

CANSetTime

CANWrite, CANWriteEx

CANWrite, CANWriteEx — Sends a CAN telegram to the CAN bus via the AnaGate device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANWrite(AnaInt32 hHandle, AnaInt32 nIdentifier, const char * pcBuffer, AnaInt32 nBufferLen, AnaInt32 nFlags);

AnaInt32 CANWriteEx(AnaInt32 hHandle, AnaInt32 nIdentifier, const char * pcBuffer, AnaInt32 nBufferLen, AnaInt32 nFlags, AnaInt32 * pnSeconds, AnaInt32 * pnMicroSeconds);
```

Parameter

hHandle	Valid access handle.
nIdentifier	CAN identifier of the sender. Parameter *nFlags* defines whether the address is in extended format (29-bit) or standard format (11-bit).
pcBuffer	Pointer to the data buffer.
nBufferLen	Length of data buffer (max. 8 bytes).
nFlags	The format flags are defined as follows. • Bit 0: If set, the CAN identifier is in extended format (29 bit), otherwise not (11 bit). • Bit 1: If set, the telegram is marked as remote frame. • Bit 2: If set, the telegram has a valid timestamp. This bit is only set for incoming data telegrams and doesn't need to be set for the `CANWrite` and `CANWriteEx` functions.
pnSeconds	Timestamp of the confirmation of the CAN controller (seconds from 01.01.1970).
pnMicroSeconds	Micro seconds portion of the timestamp.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Both functions send a CAN telegram to the CAN bus via the AnaGate device.

The CANWriteEx additionallly returns a timestamp of the time at which the telegram was sent.

Note

With remote frames (RTR = remote transmission request) a destination node can request data from a source node. The data length is to be set to the number of requested bytes - on the CAN bus no data is sent, only the data size information.

When using the CANWrite or CANWriteEx functions to send remote frames the data buffer and the buffer size equal to the number of requested bytes have to be set correctly.

See the following example for sending a data telegram to the connected CAN bus.

```
#include <AnaGateDllCan.h>
int main()
{
   char cMsg[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
   AnaInt32 hHandle = 0;
   AnaInt32 nRC = 0;
   AnaInt32 nFlags = 0x0; //  11bit address + standard (not remote frame)
   AnaInt32 nIdentifier = 0x25; // send with CAN ID 0x25;

   AnaInt32 nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // send 8 bytes with CAN ID 37
      nRC = CANWrite( hHandle, nIdentifier, cMsg, 8, nFlags );

      // send a remote frame to CAN ID 37 (request 4 data bytes)
      nRC = CANWrite( hHandle, nIdentifier, cMsg, 4, 0x02 );

      CANCloseDevice(hHandle);
   }
   return 0;
}
```

Remarks

The CANWriteEx function is supported by library version 1.4-1.8 or higher.

For devices of type AnaGate CAN (hardware version 1.1.A) the function CANWriteEx is equal to CANWrite, the return values *pnSeconds* and *pnMicroSeconds* will remain unchanged.

CANSetCallback, CANSetCallbackEx

CANSetCallback, CANSetCallbackEx — Defines an asynchronous callback function which is called for each incoming CAN telegram.

Syntax

```
#include <AnaGateDllCan.h>

typedef void (WINAPI * CAN_PF_CALLBACK)(AnaInt32 nIdentifier, const char * pcBuffer, AnaInt32 nBufferLen, AnaInt32 nFlags, AnaInt32 hHandle);

AnaInt32 CANSetCallback(AnaInt32 hHandle, CAN_PF_CALLBACK pCallbackFunction);

typedef void (WINAPI * CAN_PF_CALLBACK_EX)(AnaInt32 nIdentifier, const char * pcBuffer, AnaInt32 nBufferLen, AnaInt32 nFlags, AnaInt32 hHandle, AnaInt32 nSeconds, AnaInt32 nMicroseconds);

AnaInt32 CANSetCallbackEx(AnaInt32 hHandle, CAN_PF_CALLBACK_Ex pCallbackFunctionEx);
```

Parameter

hHandle	Valid access handle.
pCallbackFunction	Function pointer to the private callback function. Set this parameter to `NULL` to deactivate the callback function. The parameters of the callback function are described in the documentation of the `CANWrite` function.
pCallbackFunctionEx	Function pointer to the private callback function. Set this parameter to `NULL` to deactivate the callback function. The parameters of the callback function are described in the documentation of the `CANWriteEx` function.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Incoming CAN telegrams can bei received via a callback function which can be set by a simple API call. If a callback function is set it will be called by the API asynchronously.

Caution

The callback function is called from a thread which is started by the CAN API and which reads data from the socket. Because of this behaviour the callback code is executed by the thread context of the API and therefore it

uses it the heap memeory of the API DLL and not the application program. So programming code should not use functions like `new`, `delete`, `malloc` or `free` which allocate, free or reallocate heap memory inside the callback.

See the following example for using a callback.

```
#include <AnaGateDllCan.h>

// Definition of a callback which writes incoming CAN data with timestamp to console
void WINAPI MyCallbackEx(AnaInt32 nIdentifier, const char * pcBuffer, AnaInt32 nBufferLen,
                         AnaInt32 nFlags, AnaInt32 hHandle,
                         AnaInt32 nSeconds, AnaInt32 nMicroseconds)
{
   std::cout << "CAN-ID=" << nIdentifier << ", Data=";
   for ( AnaInt32 i = 0; i < nBufferLen; ++i )
   {
      std::cout << " 0x" << std::hex << int((unsigned char)(pcBuffer[i]));
   }
   time_t tTime = nSeconds;
   struct tm * psLocalTime = localtime(&tTime );
   std::cout << " " << std::setw(19) << asctime( psLocalTime ) << " ms(" << std::dec
             << std::setw(3) << nMicroseconds/1000 << "." << nMicroseconds%1000 << ")" << std::endl;
}

int main()
{
   AnaInt32 hHandle = 0;
   AnaInt32 nRC = 0;

   AnaInt32 nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // deactivate callback
      nRC = CANSetCallbackEx( hHandle, MyCallbackEx );

      getch();  // wait for keyboard input

      // deactivate callback
      nRC = CANSetCallbackEx( hHandle, 0 );

      CANCloseDevice(hHandle);
   }
   return 0;
}
```

Remarks

The two different callback functions have to be used depending on the active setting of the global timestamp option (`CANSetGlobals`). Only one of the callbacks can be activated at the same time.

See also

CANWrite, CANWriteEx

CANSetMaxSizePerQueue

CANSetMaxSizePerQueue — Sets the maximum size of the queue that buffers received CAN telegrams.

Syntax

#include <AnaGateDllCan.h>

AnaInt32 **CANSetMaxSizePerQueue**(AnaInt32 *hHandle*, AnaUInt32 *nMaxSize*);

Parameter

hHandle Valid access handle from a successful CANOpenDevice call.

nMaxSize Maximum size of the receive buffer.

Return value

Returns Null if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Sets the maximum size of the queue that buffers received CAN telegrams. No telegrams are buffered before this function is called. Once received telegrams have been added to the buffer they can be read with CANGetMessage. If the queue is full while a new telegram is received then it gets discarded.

if the queue size is set to 0 then all previously queued telegrams are deleted. However, if the queue size is reduced to a value different from 0 then excess telegrams are not discarded. Instead newly received telegrams don't get queued until the queue has been freed enough via CANGetMessage calls, or until the queue size has been increased again.

See the following example for setting the queue size.

```
#include <AnaGateDllCan.h>
int main()
{
   AnaInt32 hHandle;
   AnaInt32 nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      nRC = CANSetMaxSizePerQueue( hHandle, 1000 );
      // ... now do something
      CANCloseDevice(hHandle);
   }
   return 0;
}
```

Remarks

Received telegrams are only buffered if no callback function was registered via CANSetCallback or CANSetCallbackEx. Once a call-back function has been enabled,

previously buffered telegrams can still be read via `CANGetMessage`. Newly received telegrams are not added to the queue though.

See also

CANSetCallback, CANSetCallbackEx

CANGetMessage

CANGetMessage

CANGetMessage — Returns a received CAN telegram from the receive queue.

Syntax

#include <AnaGateDllCan.h>

```
AnaInt32 CANGetMessage(AnaInt32 hHandle, AnaUInt32 * pnAvailMsgs,
AnaInt32 * pnID, char * pcData, AnaUInt8 * pnDataLen, AnaInt32 * pnFlags,
AnaInt32 * pnSeconds, AnaInt32 * pnMicroseconds);
```

Parameter

hHandle	Valid access handle from a successful CANOpenDevice call.
pnAvailMsgs	Pointer to a variable to which the number of telegrams which are still available after this function call is saved. If no telegram is available when this function is called then this variable is set to -1. In this case all following parameters are invalid.
pnID	Pointer to a variable to which the CAN ID of the telegram is saved unless the pointer is NULL.
pcData	Pointer to a buffer to which the data bytes of the telegram are saved unless the pointer is NULL. The buffer must be large enough to hold 8 bytes.
pnDataLen	Pointer to a variable to which the number of data bytes of the telegram is saved unless the pointer is NULL.
pnFlags	Pointer to a variable to which the flags of the telegram are saved unless the pointer is NULL.
pnSeconds	Pointer to a variable to which the receive time of the telegram (in seconds since 1.1.1970) is saved unless the pointer is NULL.
pnMicroseconds	Pointer to a variable to which the microseconds part of the receive time of the telegram is saved unless the pointer is NULL.

Return value

Returns Null if successful, or an error value otherwise (Appendix A, *API return codes*).

Description

Returns a received CAN telegram from the receive queue. The caller needs to supply memory buffers for the telegram parameters he is interested in. Parameters for unneeded values can be NULL pointers.

The function returns the number of telegrams that are still in the queue after the function call via the pnAvailMsgs parameter. This variable is set to -1 if no telegram was available in the queue. In that case all telegram parameters are invalid.

See the following example for getting a telegram from the queue.

```
#include <AnaGateDllCan.h>
int main()
{
   AnaInt32 hHandle;
   AnaInt32 nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      nRC = CANSetMaxSizePerQueue( hHandle, 1000 );
      // wait so a received telegram can be queued
      Sleep( 5 );
      AnaUInt32 nAvailMsgs;
      AnaInt32 nID;
      char acData[8];
      AnaUInt8 nDataLen;
      AnaInt32 nFlags;
      AnaInt32 nSeconds, nMicroseconds;
      nRC = CANGetMessage( hHandle, &nAvailMsgs, &nID, acData, &nDataLen,
                           &nFlags, &nSeconds, &nMicroseconds );
      if( nAvailMsgs != (AnaUInt32)(-1) && nAvailMsgs > 0 )
      {
         // ... now do something
      }
      CANCloseDevice(hHandle);
   }
   return 0;
}
```

Remarks

Received telegrams are only buffered if no call-back function was registered via `CANSetCallback` or `CANSetCallbackEx`. Once a call-back function has been enabled, previously buffered telegrams can still be read via `CANGetMessage`. Newly received telegrams are not added to the queue though.

See also

CANSetCallback, CANSetCallbackEx

CANSetMaxSizePerQueue

CANReadDigital

CANReadDigital — Reads the current values of digital input and output registers of the AnaGate device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANReadDigital(AnaInt32 hHandle, AnaUInt32 * pnInputBits,
AnaUInt32 * pnOutputBits);
```

Parameter

hHandle	Valid access handle.
pnInputBits	Pointer to the current value of the digital input register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.
pnOutputBits	Pointer to the current value of the digital output register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The current values of the digital inputs and outputs can be retrieved with the `CANReadDigital` function.

See the following example for setting and reading the digital IOs.

```
#include <AnaGateDllCan.h>
int main()
{
   AnaInt32 hHandle = 0;
   AnaInt32 nRC = 0;
   AnaUInt32 nInputs;
   AnaUInt32 nOutputs = 0x03;

   nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // set the digital output register (PIN 0 and PIN 1 to HIGH value)
      nRC = CANWriteDigital( hHandle, nOutputs );
```



```
      // read all input and output registers
      nRC = CANReadDigital( hHandle, &nInputs, &nOutputs );

      CANCloseDevice(hHandle);
   }
   return 0;
}
```

See also

CANWriteDigital

CANWriteDigital

CANWriteDigital — Writes a new value to the digital output register of the AnaGate device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANWriteDigital(AnaInt32 hHandle, AnaUInt32 nOutputBits);
```

Parameter

hHandle	Valid access handle.
nOutputBits	New register value. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The digital outputs can be written with the `CANWriteDigital` function.

A simple example for reading/writing of the IOs can be found at the description of `CANReadDigital`.

See also

CANReadDigital

CANReadAnalog

CANReadAnalog — Reads the current values of analog inputs of the AnaGate device.

Syntax

#include <AnaGateDllCan.h>

AnaInt32 **CANReadAnalog**(AnaInt32 *hHandle*, AnaUInt32 * *pnPowerSupply*, AnaUInt32 *anAnalogInputs*[], AnaUInt16 * *pnInputCount*);

Parameter

hHandle	Valid access handle.
pnPowerSupply	Pointer to a variable to which the current supply voltage in millivolt is saved unless the pointer is NULL.
anAnalogInputs	Array of variables to which the current analog input values in millivolt are saved.
pnInputCount	Pointer to a variable with the number of elements of anAnalogInputs. After the function call the variable contains the number of actually received values.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The AnaGate CAN X series models have connectors for 4 analog inputs and 4 analog outputs at the top side.

The current values of the analog inputs and the current supply voltage can be retrieved with the `CANReadAnalog` function.

See the following example for reading the analog inputs.

```
#include <AnaGateDllCan.h>
#include <algorithm>
#include <stdio.h>
int main()
{
   AnaInt32 hHandle = 0;
   AnaInt32 nRC = 0;

   nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      const AnaUInt16 INPUT_SIZE = 4;
      AnaUInt32 nPowerSupply;
      AnaUInt32 anAnalogInputs[INPUT_SIZE];
      AnaUInt16 nInputCount = INPUT_SIZE;
```



```
      nRC = CANReadAnalog(nHandle, &nPowerSupply, anAnalogInputs, &nInputCount);
      if ( nRC == 0 )
      {
         printf("Power supply: %d mV\n", nPowerSupply);
         for ( AnaUInt16 i = 0; i < std::min(INPUT_SIZE, nInputCount); ++i )
         {
            printf("Analog input %d: %d mV\n", i, anAnalogInputs[i]);
         }
      }
      CANCloseDevice(hHandle);
   }
   return 0;
}
```

See also

CANWriteAnalog

CANWriteAnalog

CANWriteAnalog — Writes new values to the analog outputs of the AnaGate device.

Syntax

#include <AnaGateDllCan.h>

AnaInt32 **CANWriteAnalog**(AnaInt32 *hHandle*, AnaUInt32 *anAnalogOutputs*[], AnaUInt16 *nOutputCount*);

Parameter

hHandle	Valid access handle.
anAnalogOutputs	Array of new analog output values in millivolt.
nOutputCount	Number of anAnalogOutputs values.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The AnaGate CAN X series models have connectors for 4 analog inputs and 4 analog outputs at the top side.

The analog outputs can be written with the `CANWriteAnalog` function. The upper output voltage is limited by the supply voltage of the AnaGate device. The current value of the supply voltage can be read with the `CANReadAnalog` function.

See the following example for writing the analog outputs.

```
#include <AnaGateDllCan.h>
int main()
{
   AnaInt32 hHandle = 0;
   AnaInt32 nRC = 0;

   nRC = CANOpenDevice(&hHandle, TRUE, TRUE, 0, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      const AnaUInt16 OUTPUT_SIZE = 4;
      AnaUInt32 anAnalogOutputs[OUTPUT_SIZE] = { 0, 9000, 24000, 0 };
      nRC = CANWriteAnalog(nHandle, anAnalogOutputs, OUTPUT_SIZE);

      CANCloseDevice(hHandle);
   }
   return 0;
}
```

See also

CANReadAnalog

CANRestart

CANRestart — Restarts an AnaGate CAN device.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANRestart(const char * pcIPAddress, AnaInt32 nTimeout );
```

Parameter

pcIPAddress	Network address of the AnaGate partner.
nTimeout	Default timeout for accessing the AnaGate in milliseconds. A timeout is reported if the AnaGate partner does not respond within the defined timeout period.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Restarts the AnaGate CAN device at the specified network address. It implicitly disconnects all open network connections to all existing CAN interfaces. The Restart command is even possible if the maximum number of allowed connections is reached.

Important

It is recommended to use this command only in emergency cases, if there is a need to connect even if the maximum number of concurrent connections is reached.

See also

CANOpenDevice

CANDeviceConnectState

CANDeviceConnectState — Retrieves the current network connection state of the current AnaGate connection.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANDeviceConnectState(AnaInt32 hHandle);
```

Parameter

hHandle Valid connection handle of a successful call to `CANOpenDevice`.

Return value

Returns the current network connection state. The following values are possible:

- `1 = DISCONNECTED`: The connection to the AnaGate is disconnected.
- `2 = CONNECTING`: The connection is connecting.
- `3 = CONNECTED` : The connection is established.
- `4 = DISCONNECTING`: The connection is disonnecting.
- `5 = NOT_INITIALIZED`: The network protocol is not successfully initialized.

Description

This function can be used to check if an already connected device is disconnected.

The detection period of a state change depends on the use of the internal AnaGate-ALIVE mechanism. This ALIVE mechanism has to be switched on explicitly via the `CANStartAlive` function. Once activated the connection state is periodically checked by the ALIVE mechanism.

Remarks

The `CANDeviceConnectState` function is supported by library version 1.4-1.10 or higher.

See also

CANStartAlive

CANStartAlive

CANStartAlive — Starts the ALIVE mechanism, which checks periodically the state of the network connection to the AnaGate hardware.

Syntax

```
#include <AnaGateDllCan.h>

AnaInt32 CANStartAlive(AnaInt32 hHandle, AnaUInt32 nAliveTime );
```

Parameter

hHandle	Valid access handle.
nAliveTime	Time out interval in seconds for the ALIVE mechanism.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The AnaGate communication protocol (see [TCP-2010]) supports an application specific connection control which allows faster detection of broken connection lines.

The `CANStartAlive` function starts a concurrent thread in the DLL in order to send defined alive telegrams (ALIVE_REQ) peridically (approx. every half of the given time out) to the Anagate device via the current network connection. If the alive telegram is not confirmed within the alive time the connection is marked as `disconnected` and the socket is closed if not already closed.

Use the `CANDeviceConnectState` function to check the current network connection state.

Remarks

The `CANStartAlive` function is supported by library version 1.4-1.10 or higher.

It requires firmware version 1.3.8 or higher installed on the hardware, devices of type AnaGate CAN (hardware version 1.1.A) don't support the application specific alive mechanism.

See also

CANDeviceConnectState

Section 2.1, " Important properties of the network protocol "

CANErrorMessage

CANErrorMessage — Returns a description of the given error code as a text string.

Syntax

#include <AnaGateDllCan.h>

AnaInt32 **CANErrorMessage**(AnaInt32 *nRetCode*, char * *pcMessage*, AnaInt32 *nMessageLen*);

Parameter

nRetCode Error code for which the error description is to be determined.

pcMessage Data buffer that is to accept the error description.

nMessageLen Size in bytes of the transferred data buffer.

Return value

Actual size of the returned description.

Description

Returns a textual description of the parsed error code (see Appendix A, *API return codes*). If the destination buffer is not large enough to store the text, the text is shortened to the specified buffer size.

See the following example in C/C++ language.

```
#include <AnaGateDllCan.h>
// ...
AnaInt32 nRC;
char cBuffer[200];
//... call an API function here
CANErrorMessage(nRC, cBuffer, sizeof cBuffer);
std::cout << "Fehler: " << cBuffer << std::endl;
```

Chapter 5. SPI API reference

The Serial Peripheral Interface (SPI) is a synchroneous data link standard named by Motorola which operates in full duplex mode. The SPI gateway models of the AnaGate series provide access to a SPI bus via standard networking.

With the SPI API these SPI gateways can be controlled easily. The programming interface is identical for all devices and generally uses the network protocol TCP.

The following devices can be addressed via the SPI API interface:

- AnaGate SPI
- AnaGate Universal Programmer

SPIOpenDevice

SPIOpenDevice — Opens a network connection to an AnaGate SPI device.

Syntax

#include <AnaGateDllSPI.h>

```
AnaInt32 SPIOpenDevice(AnaInt32 * pHandle, const char * pcIPAddress,
AnaInt32 nTimeout);
```

Parameter

pHandle	Pointer to a variable in which the access handle is saved in case of a successful connection to the AnaGate device.
pcIPAddress	Network address of the AnaGate partner.
nTimeout	Default timeout for accessing the AnaGate in milliseconds. A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Opens a TCP/IP connection to an AnaGate SPI (resp. AnaGate Universal Programmer). After the connection is established, access to the SPI bus is possible.

Note

The AnaGate SPI (resp. the SPI interface of an AnaGate Universal Programmer) does not allow more than one concurrent network connection. As long as a connection is active, all new connections are refused.

See the following example for the initial programming steps.

```
#include <AnaGateDllSPI.h>
int main()
{
   AnaInt32 hHandle;
   AnaInt32 nRC = SPIOpenDevice(&hHandle, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // ... now do something
      SPICloseDevice(hHandle);
```

```
   }
   return 0;
}
```

See also

SPICloseDevice

SPICloseDevice

SPICloseDevice — Closes an open network connection to an AnaGate SPI device.

Syntax

```
#include <AnaGateDllSPI.h>

AnaInt32 SPICloseDevice(AnaInt32 hHandle);
```

Parameter

hHandle Valid access handle.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Closes an open network connection to an AnaGate SPI device. The *hHandle* parameter is a return value of a successful call to the function `SPIOpenDevice`.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See also

SPIOpenDevice

SPISetGlobals

SPISetGlobals — Sets the global settings, which are to be used on the AnaGate SPI.

Syntax

```
#include <AnaGateDllSPI.h>

AnaInt32 SPISetGlobals(AnaInt32 hHandle, AnaInt32 nBaudrate, AnaUInt8
nSigLevel, AnaUInt8 nAuxVoltage, AnaUInt8 nClockMode);
```

Parameter

hHandle Valid access handle.

nBaudrate The baud rate to be used. The values can be set individually, like

- 500,000 for 500 kBit
- 1,000,000 for 1 MBit
- 5,000,000 for 5 MBit

Note

The required baud rate can be different from the value actually used because of internal hardware restrictions (frequency of the oscillator). If it is not possible to adjust the baud rate exactly to the parsed value, the nearest smaller possible value is used instead.

nSigLevel The voltage level for SPI signals to be used. The following values are allowed:

- 0 = Outputs in High Impedance Modus (Standard mode).
- 1 = +5.0 V for the signals.
- 2 = +3.3 V for the signals.
- 3 = +2.5 V for the signals.

nAuxVoltage The voltage level of the support voltage to be used. The following values are allowed:

- 0 = support voltage is +3.3 V.
- 1 = support voltage is 2.5 V.

nClockMode The phase and polarity of the clock signal. The following values are allowed:

- 0 = CPHA=0 and CPOL=0.
- 1 = CPHA=0 and CPOL=1.

- 2 = CPHA=1 and CPOL=0.
- 3 = CPHA=1 and CPOL=1.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Sets the global settings of SPI interface of the AnaGate SPI or the AnaGate Universal Programmer. These settings are not saved permanent on the device and are reset after every device restart.

See also

SPIGetGlobals

SPIGetGlobals

SPIGetGlobals — Returns the currently used global settings of the AnaGate SPI.

Syntax

#include <AnaGateDllSPI.h>

AnaInt32 **SPIGetGlobals**(AnaInt32 *hHandle*, AnaInt32 * *pnBaudrate*, AnaUInt8 * *pnSigLevel*, AnaUInt8 * *pnAuxVoltage*, AnaUInt8 * *pnClockMode*);

Parameter

hHandle Valid access handle.

pnBaudrate The baud rate currently used on the SPI bus in bits per second.

pnSigLevel The voltage level currently used by the AnaGate SPI. The following values are possible:

- 0 = Outputs in High Impedance Modus (Standard mode).
- 1 = +5.0 V for the signals.
- 2 = +3.3 V for the signals.
- 3 = +2.5 V for the signals.

pnAuxVoltage The voltage level of the support voltage currently used by the AnaGate SPI. The following values are possible:

- 0 = support voltage is +3.3 V.
- 1 = support voltage is 2.5 V.

pnClockMode The phase and polarity of the clock signal currently used by the AnaGate SPI. The following values are possible:

- 0 = CPHA=0 and CPOL=0.
- 1 = CPHA=0 and CPOL=1.
- 2 = CPHA=1 and CPOL=0.
- 3 = CPHA=1 and CPOL=1.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Returns the currently used global settings of the SPI interface of the AnaGate SPI or the AnaGate Universal Programmer.

See also

SPISetGlobals

SPIDataReq

SPIDataReq — Writes and reads data to/from SPI bus.

Syntax

#include <AnaGateDllSPI.h>

AnaInt32 **SPIDataReq**(AnaInt32 *hHandle*, const char * *pcBufWrite*, AnaInt32 *nBufWriteLen*, char * *pcBufRead*, AnaInt32 *nBufReadLen*);

Parameter

hHandle	Valid access handle.
pcBufWrite	Buffer with the data that is to be sent to the SPI partner.
nBufWriteLen	Length of the data buffer *pcBufWrite* (byte count).
pcBufRead	Byte buffer which holds the data received from the SPI partner.
nBufReadLen	Number of bytes to read. The buffer *pcBufRead* must be large enough to hold this data.

Description

Sends data to the SPI bus and receives data from the SPI bus.

On the SPI bus Data is transferred on two seperate data lines full duplex (SDO and SDI). The SPIDatReq has to split a single data transfer into two steps because of the spacial separation from the SPI bus. First the write data buffer is put into a TCP data telegram and sent to the AnaGate SPI. The AnaGate SPI performs the real data transfer on the SPI bus and sends a confirmation back including the data received from the bus.

Important

It is impossible to detect that no device is present at the SPI bus. So if no device is attached, the requested number of bytes are returned anyway - in this case the read buffer is filled with 0.

See the following example for sending a command to the connected SPI bus.

```
#include <AnaGateDllSPI.h>
int main()
{
   char cBufWrite[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   char cBufReceive[100];
   AnaInt32 hHandle = 0;
   AnaInt32 nRC = 0;

   AnaInt32 nRC = SPIOpenDevice(&hHandle, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
```



```
      // send 1 byte and receive 1 byte
      nRC = SPIDataReq( hHandle, cBufWrite, 1, cBufReceive, 1 );
      // send 1 byte and receive 5 byte
      nRC = SPIDataReq( hHandle, cBufWrite, 1, cBufReceive, 5 );
      // send 2 byte and receive 1 byte
      nRC = SPIDataReq( hHandle, cBufW2ite, 2, cBufReceive, 1 );

      SPICloseDevice(hHandle);
   }
   return 0;
}
```

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

SPIReadDigital

SPIReadDigital — Reads the current values of digital input and output registers of the AnaGate device.

Syntax

#include <AnaGateDllSPI.h>

AnaInt32 **SPIReadDigital**(AnaInt32 *hHandle*, AnaUInt32 * *pnInputBits*, AnaUInt32 * *pnOutputBits*);

Parameter

hHandle	Valid access handle.
pnInputBits	Pointer to the current value of the digital input register. Currently only bits 0 to 3 are used; the other bits are reserved for future use and are set to 0.
pnOutputBits	Pointer to the current value of the digital output register. Currently only bits 0 to 3 are used; the other bits are reserved for future use and are set to 0.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel.

The current values of the digital inputs and outputs can be retrieved with the `SPIReadDigital` function.

See the following example for setting and reading the digital IO.

```
#include <AnaGateDllSPI.h>
int main()
{
   AnaInt32 hHandle = 0;
   AnaUInt32 nInputs;
   AnaUInt32 nOutputs = 0x03;

   AnaInt32 nRC = SPIOpenDevice(&hHandle, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // set the digital output register (PIN 0 and PIN 1 to HIGH value)
      nRC = SPIWriteDigital( hHandle, nOutputs );

      // read all input and output registers
      nRC = SPIReadDigital( hHandle, &nInputs, &nOutputs );
```



```
      SPICloseDevice(hHandle);
   }
   return 0;
}
```

See also

SPIWriteDigital

SPIWriteDigital

SPIWriteDigital — Write a new value to the digital output register of the AnaGate device.

Syntax

```
#include <AnaGateDllSPI.h>

AnaInt32 SPIWriteDigital(AnaInt32 hHandle, AnaUInt32 nOutputBits);
```

Parameter

hHandle	Valid access handle.
nOutputBits	New register value. Currently only bits 0 to 3 are used; the other bits are reserved for future use.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel.

The digital outputs can be written with the `SPIWriteDigital` function.

A simple example for reading/writing of the IOs can be found at the description of `SPIReadDigital`.

See also

SPIReadDigital

SPIErrorMessage

SPIErrorMessage — Returns a description of the given error code as a text string.

Syntax

#include <AnaGateDllSPI.h>

AnaInt32 **SPIErrorMessage**(AnaInt32 *nRetCode*, char * *pcMessage*, AnaInt32 *nMessageLen*);

Parameter

nRetCode Error code for which the error description is to be determined.

pcMessage Data buffer that is to accept the error description.

nMessageLen Size in bytes of the transferred data buffer.

Return value

Actual size of the returned description.

Description

Returns a textual description of the parsed error code (see Appendix A, *API return codes*). If the destination buffer is not big enough to store the text, the text is shortened to the specified buffer size.

See the following example in C/C++ language.

```
#include <AnaGateDllSPI.h>
// ...
AnaInt32 nRC;
char cBuffer[200];
//... call a API function here
SPIErrorMessage(nRC, cBuffer, sizeof cBuffer);
std::cout << "Fehler: " << cBuffer << std::endl;
```

Chapter 6. I2C API reference

Philips Semiconductors (now NXP Semiconductors) has developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter-IC or I2C-bus. Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL). Serial, 8-bit oriented, bidirectional data transfers can be performed at up to 100 kbit/s in Standard mode, up to 400 kbit/s in Fast mode, up to 1 Mbit/s in Fast mode Plus (Fm+) or up to 3.4 Mbit/s in High speed mode. [NXP-I2C].

The I2C gateway models of the AnaGate series provides access to a I2C bus via a standard networking. With the I2C API these I2C gateways can be easily controlled. The programming interface is identical for all devices and used the network protocol TCP in general.

Following devices can be addressed via the I2C API interface:

- AnaGate I2C
- AnaGate Universal Programmer

I2COpenDevice

I2COpenDevice — Opens a network connection to an AnaGate I2C or an AnaGate Universal Programmer).

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2COpenDevice(AnaInt32 * pHandle, AnaUInt32 nBaudrate, const
char * pcIPAddress, AnaInt32 nTimeout);
```

Parameter

pHandle — Pointer to a variable in which the access handle is saved in the event of a successful connection to the AnaGate device.

nBaudrate — Baud rate to be used for the I2C bus. The value can be set individually, like

- 100000 for 100 kBit (standard mode)
- 400000 for 400 kBit (fast mode)

Note

Values above 400 kBit are ignored by the AnaGate SPI.

pcIPAddress — Network address of the AnaGate partner.

nTimeout — Default timeout for accessing the AnaGate in milliseconds.

A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Opens a TCP/IP connection to an AnaGate I2C (resp. AnaGate Universal Programmer). After the connection is established, access to the I2C bus is possible.

Note

The AnaGate I2C (resp. the I2C interface of an AnaGate Universal Programmer) does not allow more than one concurrent network connection. During an established network connection all new connections are refused.

See the following example for the initial programming steps.

```
#include <AnaGateDllI2C.h>
int main()
{
   AnaInt32 hHandle;
   AnaInt32 nRC = I2COpenDevice(&hHandle, 100000, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // ... now do something
      I2CCloseDevice(hHandle);
   }
   return 0;
}
```

See also

I2CCloseDevice

I2CCloseDevice

I2CCloseDevice — Closes an open network connection to an AnaGate I2C device.

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2CCloseDevice(AnaInt32 hHandle);
```

Parameter

hHandle Valid access handle.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Closes an open network connection to an AnaGate I2C device. The *hHandle* parameter is a return value of a successful call to the function `I2COpenDevice`.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See also

I2COpenDevice

I2CReset

I2CReset — Resets the I2C Controller in an AnaGate I2C device.

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2CReset(AnaInt32 hHandle);
```

Parameter

hHandle Valid access handle.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Resets the I2C Controller in an AnaGate I2C device.

I2CRead

I2CRead — Reads data from an I2C partner.

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2CRead(AnaInt32 hHandle, AnaUInt16 nSlaveAddress, const char * pcBuffer, AnaInt32 nBufferLen);
```

Parameter

hHandle	Valid access handle.
nSlaveAddress	Slave address of the I2C partner. The slave address can represent a so-called 7-bit or 10-bit address.(siehe Appendix B, *I2C slave address formats*).
pcBuffer	Byte buffer in which the data received from the I2C partner is to be stored.
nBufferLen	Number of bytes to read.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Reads data from an I2C device (slave), the AnaGate hardware is operating as I2C master.

The R/W bit of the slave address does not have to be explicitly set by the user.

See also

I2CWrite

I2CWrite

I2CWrite — Writes data to an I2C partner.

Syntax

#include <AnaGateDllI2C.h>

```
AnaInt32 I2CWrite(AnaInt32 hHandle, AnaUInt16 nSlaveAddress, const char
* pcBuffer, AnaInt32 nBufferLen, AnaInt32 * pnErrorByte);
```

Parameter

hHandle	Valid access handle.
nSlaveAddress	Slave address of the I2C partner. The slave address can represent a so-called 7-bit or 10-bit address.(siehe Appendix B, *I2C slave address formats*).
pcBuffer	Byte buffer with the data that is to be sent to the I2C partner.
nBufferLen	Size of bytes to be read.
pnErrorByte	Pointer to a variable where the byte position of an error in the data buffer is saved.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Writes data to an I2C device (slave), the AnaGate hardware is operating as I2C master.

The R/W bit of the slave address does not have to be explicitly set by the user.

See also

I2CRead

I2CSequence

I2CSequence — This command is used to write a sequence of write and read commands to an I2C device.

Syntax

#include <AnaGateDllI2C.h>

AnaInt32 **I2CSequence**(AnaInt32 *hHandle*, const char * *pcWriteBuffer*, AnaInt32 *nNumberOfBytesToWrite*, char * *pcReadBuffer*, AnaInt32 *nNumberOfBytesToRead*, AnaInt32 * *pnNumberOfBytesRead*, AnaInt32 * *pnByteNumberLastError*);

Parameter

hHandle — Valid access handle.

pcWriteBuffer — Byte buffer that contains the commands which are to be sent to the AnaGate I2C. The single commands are stored sequentially in this byte buffer.

A read command is defined as follows:

Structure of read command for I2CSequence

Read command	**Description**
2 bytes (LSB,MSB)	slave address in 7- or 10-bit format, the R/W bit must be set to explicitly to 1.
2 bytes (LSB,MSB)	Bit 0-14: number of data bytes to be read from the I2C device. The successfully read data bytes are stored in the *pcReadBuffer* buffer. Bit 15: If this bit is set then the stop bit at the end of the read command is omitted.

A write command is defined as follows:

Structure write command for I2CSequence

Write command	**Description**
2 bytes (LSB,MSB)	slave address in 7- or 10-bit format, the R/W bit must be set to explicitly to 1.
2 bytes (LSB,MSB)	Bit 0-14: number of data bytes to be written to the I2C device. Bit 15: If this bit is set then the stop bit at the end of the write command is omitted.

Write command	Description
N bytes	data bytes.

nNumberOfBytesToWrite	byte size of the data to write
pcReadBuffer	Byte buffer in which the received data is to be stored. The received data from different commands are stored in the buffer sequentially (first the data of the first command, then the data of the second command, ...).
nNumberOfBytesRead	byte size of the read buffer (must be big enough for all included read requests)
pnNumberOfBytesRead	number of bytes which are read from I2C.
pnByteNumberLastError	byte offset in the *pcWriteBuffer* buffer where an error occurred.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The user must ensure that the setup of the data buffer and the address of the I2C partner are correct.

I2CReadDigital

I2CReadDigital — Reads the current values of digital input and output registers of the AnaGate device.

Syntax

#include <AnaGateDllI2C.h>

AnaInt32 **I2CReadDigital**(AnaInt32 *hHandle*, AnaUInt32 * *pnInputBits*, AnaUInt32 * *pnOutputBits*);

Parameter

hHandle	Valid access handle.
pnInputBits	Pointer to the current value of the digital input register. Currently only bits 0 to 3 are used; the other bits are reserved for future use and are set to 0.
pnOutputBits	Pointer to the current value of the digital output register. Currently only bits 0 to 3 are used; the other bits are reserved for future use and are set to 0.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel.

The current values of the digital inputs and outputs can be retrieved with the `I2CReadDigital` function.

See the following example for setting and reading the digital IO.

```
#include <AnaGateDllI2C.h>
int main()
{
   AnaInt32 hHandle = 0;
   AnaUInt32 nInputs;
   AnaUInt32 nOutputs = 0x03;

   AnaInt32 nRC = I2COpenDevice(&hHandle, 400000, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      // set the digital output register (PIN 0 and PIN 1 to HIGH value)
      nRC = I2CWriteDigital( hHandle, nOutputs );

      // read all input and output registers
      nRC = I2CReadDigital( hHandle, &nInputs, &nOutputs );
```



```
      CANCloseDevice(hHandle);
   }
   return 0;
}
```

See also

I2CWriteDigital

I2CWriteDigital

I2CWriteDigital — Writes a new value to the digital output register of the AnaGate device.

Syntax

```
AnaInt32 I2CWriteDigital(AnaInt32 hHandle, AnaUInt32 nOutputBits);
```

Parameter

hHandle	Valid access handle.
nOutputBits	New register value. Currently only bits 0 to 3 are used; the other bits are reserved for future use.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel.

The digital outputs can be written with the `I2CWriteDigital` function.

A simple example for reading/writing of the IOs can be found at the description of `I2CReadDigital`.

See also

I2CReadDigital

I2CErrorMessage

I2CErrorMessage — Returns a description of the given error code as a text string.

Syntax

#include <AnaGateDllI2C.h>

AnaInt32 **I2CErrorMessage**(AnaInt32 *nRetCode*, char * *pcMessage*, AnaInt32 *nMessageLen*);

Parameter

nRetCode Error code for which the error description is to be determined.

pcMessage Data buffer that is to accept the error description.

nMessageLen Size in bytes of the transferred data buffer.

Return value

Actual size of the returned description.

Description

Returns a textual description of the parsed error code (see Appendix A, *API return codes*). If the destination buffer is not large enough to store the text, the text is shortened to the specified buffer size.

See the following example in C/C++ language.

```
#include <AnaGateDllI2C.h>
// ...
AnaInt32 nRC;
char cBuffer[200];
//... call a API function here
I2CErrorMessage(nRC, cBuffer, sizeof cBuffer);
std::cout << "Fehler: " << cBuffer << std::endl;
```


I2CReadEEProm

I2CReadEEProm — Reads data from an EEPROM on the I2C bus.

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2CReadEEProm(AnaInt32 hHandle, AnaUInt16 nSubAddress,
AnaUInt32 nOffset, char * pcBuffer, AnaInt32 nBufferLen, AnaUInt32
nOffsetFormat);
```

Parameter

hHandle	Valid access handle.
nSubAddress	Sub address of the EEPROM to communicate with. The valid values for *nSubAddress* are governed by the setting used in the parameter *nOffsetFormat* (bits 8-10). If the EEPROM type needs to use bits of the *Chip Enable Address* to address the internal memory, only the remaining bits can be used to select the device itself. • No bit is used for addressing: 0 to 7 • 1 bit is used for addressing: 0 to 3 • 2 bits are used for addressing: 0 to 1 • 3 bits are used for addressing: 0
nOffset	Data offset on the EEPROM from which the transferred data is to be read.
pcBuffer	Character string buffer in which the received data is to be stored.
nBufferLen	Length of the data buffer.
nOffsetFormat	Defines how a memory address of EEPROM has to be specified when accessing the device. Bits 0-7 indicate the number of bits which are used in the address byte (word) of the I2C command for addressing the device memory. Bits 8-10 indicate how much and which bits of the *Chip Enable Bits* are used for addressing the device memory (see Table C.1, " Usage of the Chip Enable Bits of I2C EEPROMs " for allowed values).

Note

The maximum addressable size of an EEPROM is derived from the sum of all the bits. For example a M24C08 uses 8 bits of the address byte and an extra bit in the slave address. The total 9 bits can address up to 512 bytes.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The `I2CReadEEProm` function reads data from an I2C EEPROM.

Of course all access to the memory of an EEPROM is done by standard I2C read or write commands. So when reading from the memory only the matching slave address, the memory offset address and the data has to be sent to the I2C bus.

`I2CReadEEProm` translates the given memory address on the chip by means of the sub address and the addressing mode of the present EEPROM type. The slave address of the EEPROM is determined automatically and not necessary for the function call.

A programming example which clears a **ST24C1024** can be found at the description of I2WriteEEPROM.

See also

I2CWriteEEProm

Appendix C, *Programming I2C EEPROM*

I2CWriteEEProm

I2CWriteEEProm — Writes data to an I2C EEPROM.

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2CWriteEEProm(AnaInt32 hHandle, AnaUInt16 nSubAddress,
AnaUInt32 nOffset, const char * pcBuffer, AnaInt32 nBufferLen, AnaUInt32
nOffsetFormat);
```

Parameter

hHandle	Valid access handle.
nSubAddress	Sub address of the EEPROM to communicate with. The valid values for *nSubAddress* are governed by the setting used in the parameter *nOffsetFormat* (bits 8-10). If the EEPROM type needs to use bits of the *Chip Enable Address* to address the internal memory, only the remaining bits can be used to select the device itself. • No bit is used for addressing: 0 to 7 • 1 bit is used for addressing: 0 to 3 • 2 bits are used for addressing: 0 to 1 • 3 bits are used for addressing: 0
nOffset	Data offset on the EEPROM to which the transferred data is to be written.
pcBuffer	Character string buffer with the data that is to be written.
nBufferLen	Length of the data buffer.
nOffsetFormat	Defines how a memory address of EEPROM has to be specified when accessing the device. Bits 0-7 indicate the number of bits which are used in the address byte (word) of the I2C command for addressing the device memory. Bits 8-10 indicate how much and which bits of the *Chip Enable Bits* are used for addressing the device memory (see Table C.1, " Usage of the Chip Enable Bits of I2C EEPROMs " for allowed values).

Note

The maximum addressable size of an EEPROM is derived from the sum of all the bits. For example a M24C08 uses 8 bits of the address byte and an extra bit in the slave address. The total 9 bits can address up to 512 bytes.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The `I2CWriteEEProm` function writes data to an I2C EEPROM.

Of course all access to the memory of an EEPROM is done by standard I2C read or write commands. So when writing to the memory only the matching slave address, the memory offset address and the data has to be sent to the I2C bus.

`I2CWriteEEProm` translates the given memory address on the chip by means of the sub address and the addressing mode of the present EEPROM type. The slave address of the EEPROM is determined automatically and not necessary for the function call.

Tip

It is important to note that an EEPROM is divided into memory pages, and that a single write command can only program data within a page. Users of `I2CWriteEEProm` must ensure to do not write across page limits. The page size depends on the EEEPOM type.

See the following example for writing data to a **ST24C1024**.

```
#include <AnaGateDllSPI.h>
int main()
{
   char      cBufferPage[256];
   int       hHandle = 0;
   int       nRC = 0;
   AnaUInt16 nSubAddress = 0;1
   AnaUInt32 nOffsetFormat = 0x10|0x0F;2

   int nRC = I2COpenDevice(&hHandle, 400000, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      memset(cBufferPage,0,256);  // clear page buffer
      for (int i=0; i<512;i++)
      {
         I2CWriteEEProm( hHandle, nSubAddress, i*256, cBufferPage, 256, nOffsetFormat );3
      }
      I2CCloseDevice(hHandle);
   }
   return 0;
}
```

1. It is possible to address 4 individual ST24C1024 on a single I2C bus. By selection of sub address `0` the control pins E2 and E1 have to be LOW.
2. 17 address bits are used to address the 128KB of a ST24C1024. 16 bits are set via the address bytes of the write command: `16=0x0F`. The address bit A16 is set via the E0 bit of the *Chip Enable Address*, therefore addressing mode `1` (E2-E1-A0) must be set: `0x10`.
3. The page size of a ST24C1024 is 256 bytes, every page is programmed fully within the for loop.

See also

I2CReadEEProm, I2CWriteEEPromPollAck

Appendix C, *Programming I2C EEPROM*

I2CWriteEEPromPollAck

I2CWriteEEPromPollAck — Writes data to an I2C EEPROM and checks if the write operation finishes within a given time period.

Syntax

```
#include <AnaGateDllI2C.h>

AnaInt32 I2CWriteEEPromPollAck(AnaInt32 hHandle, AnaUInt16 nSubAddress,
AnaUInt32 nOffset, const char * pcBuffer, AnaInt32 nBufferLen, AnaUInt32
nOffsetFormat, AnaUInt16 nTimeout);
```

Parameter

hHandle	Valid access handle.
nSubAddress	Sub address of the EEPROM to communicate with. The valid values for *nSubAddress* are governed by the setting used in the parameter *nOffsetFormat* (bits 8-10). If the EEPROM type needs to use bits of the *Chip Enable Address* to address the internal memory, only the remaining bits can be used to select the device itself. • No bit is used for addressing: 0 to 7 • 1 bit is used for addressing: 0 to 3 • 2 bits are used for addressing: 0 to 1 • 3 bits are used for addressing: 0
nOffset	Data offset on the EEPROM to which the transferred data is to be written.
pcBuffer	Character string buffer with the data that is to be written.
nBufferLen	Length of the data buffer.
nOffsetFormat	Defines how a memory address of EEPROM has to be specified when accessing the device. Bits 0-7 indicate the number of bits which are used in the address byte (word) of the I2C command for addressing the device memory. Bits 8-10 indicate how much and which bits of the *Chip Enable Bits* are used for addressing the device memory (see Table C.1, " Usage of the Chip Enable Bits of I2C EEPROMs " for allowed values).

Note

The maximum addressable size of an EEPROM is derived from the sum of all the bits. For example a M24C08 uses 8 bits of the address byte and an extra bit in the slave address. The total 9 bits can address up to 512 bytes.

nTimeout Timeout in milliseconds for the write operation.

Return value

Returns `Null` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The `I2CWriteEEPromPollAck` function writes data to an I2C EEPROM and checks if the write operation finishes within a given time period.

Of course all access to the memory of an EEPROM is done by standard I2C read or write commands. So when writing to the memory only the matching slave address, the memory offset address and the data has to be sent to the I2C bus.

`I2CWriteEEPromPollAck` translates the given memory address on the chip by means of the sub address and the addressing mode of the present EEPROM type. The slave address of the EEPROM is determined automatically and not necessary for the function call.

Tip

It is important to note that an EEPROM is divided into memory pages, and that a single write command can only program data within a page. Users of `I2CWriteEEPromPollAck` must ensure to do not write across page limits. The page size depends on the EEEPOM type.

See the following example for writing data to a **ST24C1024**.

```
#include <AnaGateDllSPI.h>
int main()
{
   char      cBufferPage[256];
   AnaInt32  hHandle = 0;
   AnaUInt16 nSubAddress = 0;1
   AnaUInt32 nOffsetFormat = 0x10|0x0F;2
   AnaUInt16 nTimeout = 100;

   int nRC = I2COpenDevice(&hHandle, 400000, "192.168.1.254", 5000);
   if ( nRC == 0 )
   {
      memset(cBufferPage,0,256);  // clear page buffer
      for (int i=0; i<512;i++)
      {
         I2CWriteEEPromPollAck( hHandle, nSubAddress, i*256,
                                cBufferPage, 256, nOffsetFormat, nTimeout );3
      }
      I2CCloseDevice(hHandle);
   }
   return 0;
}
```

1 It is possible to address 4 individual ST24C1024 on a single I2C bus. By selection of sub address `0` the control pins E2 and E1 have to be LOW.

2. 17 address bits are used to address the 128KB of a ST24C1024. 16 bits are set via the address bytes of the write command: `16=0x0F`. The address bit A16 is set via the E0 bit of the *Chip Enable Address*, therefore addressing mode `1` (E2-E1-A0) must be set: `0x10`.
3. The page size of a ST24C1024 is 256 bytes, every page is programmed fully within the for loop.

See also

I2CWriteEEProm

Appendix C, *Programming I2C EEPROM*

Chapter 7. Programming examples

7.1. Programming language C/C++

The AnaGate programming API can be used on Windows systems as well as on linux systems (X86). All available API functions are coded operating system independently, so source code once created can be used on both operating systems. Only the way the libraries are linked on the different operating systems or different compilers has to be customized.

Windows operating systems

There are basically two ways to access the API functions for the C/C++ programmer:

- When directly accessing the library all functions have to made known by preceding calls to the Win32 methods `LoadLibrary` and `GetProcAddress`.
- The functions can be alternatively accessed easily via an import library, which automatically loads all DLL functions and makes them available implicitly.

  The import libaries for MS VC8 are included and can be used directly. The libraries are named after their corresponding DLLs, so for example the name of the import library of `AnaGateCAN.dll` is `AnaGateCANDll.lib`.

In the following examples it is assumed that the second option is used and the corresponding import library is linked.

7.1.1. CAN Console application C/C++ (MSVC)

This programming example for C++ demonstrates how to connect to an AnaGate CAN and how to process received CAN data via a callback function.

Note

The source code of the example can be found in directory `Samples/CAN-VC6` resp. `Samples/CAN-VC7` on the CD.

```
#include <AnaGateDllCan.h>

WINAPI void MyCallback(AnaInt32 nIdentifier, const char * pcBuffer,
                       AnaInt32 nBufLen, AnaInt32 nFlags, AnaInt32 nHandle)
{
   std::cout << "CAN-ID=" << nIdentifier << ", Data=";
   for ( AnaInt32 i = 0; i < nBufLen; i++ )
   {
      std::cout " 0x" << std::hex << pcBuffer[i];
   }
   std::cout << std::endl;
}

int main( )
{
```

```
   AnaInt32 hHandle = NULL;

   // opens AnaGate CAN duo device on port A, timeout after 1000 milliseconds
   AnaInt32 nRC = CANOpenDevice(&hHandle, FALSE, TRUE, 0, "192.168.1.254", 1000);

   if ( nRC == 0 )
   {
      nRC = CANSetCallback(hHandle, MyCallback);
      getch();  // wait for keyborad input
   }
   if ( nRC == 0 )
   {
      nRC = CANCloseDevice(hHandle); // close device
   }
}
```

7.2. Programming language Visual Basic 6

As already described in the previous chapters, the libraries of the *AnaGate*-API use the cdecl calling convention to parse function parameters to the program stack. Unfortunately this is generally not supported by the programming language *Visual Basic 6*.

To work around this limitation, the libraries for the AnaGate devices are available in a specific version for programming VB6 applications. In this versions the stdcall calling convention is used, which is the only one supported by VB6. Except for the way the parameters are pushed on the stack these specific VB6 versions of the API libraries are exactly identical to the standard versions.

Note

Use the library *AnaGateCAN.dll* instead of the *AnaGateCANVB6.dll* library.

Use the library *AnaGateSPI.dll* instead of the *AnaGateSPIVB6.dll* library.

7.2.1. SPI Example with user interface for VB6

This programming example for Visual Basisc 6 demonstrates how to connect to an AnaGate SPI and how to execute a command on the SPI bus:

- Getting the global device settings like baud rate for example
- Sending a single command to the SPI bus

Note

The source code of the example can be found in the `Samples/SPI-VB6` directory on the CD.

7.2.1.1. User interface

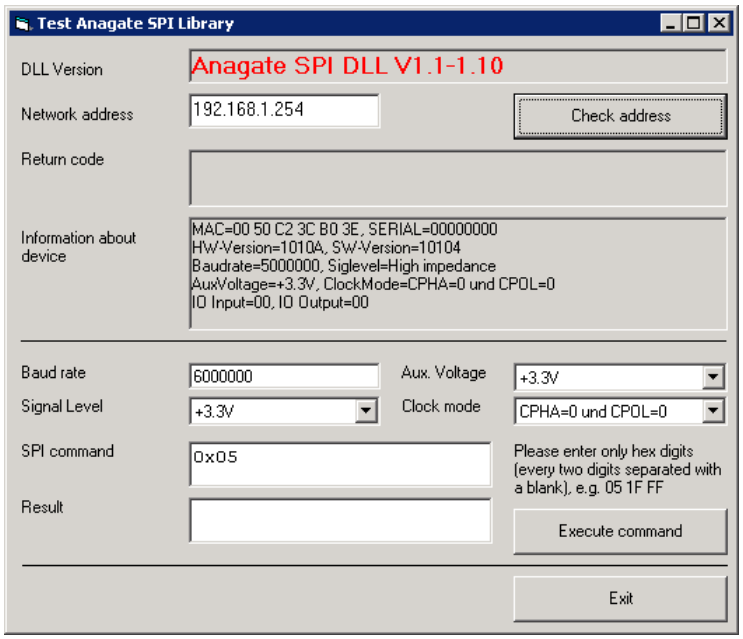


Figure 7.1. Input form of SPI example (VB6)

Dialog fields

Network address	Network address of the AnaGate SPI.
Check address	Establishes a connection to the AnaGate SPI with the specified network address and reads back some device information and global device settings.
Baud rate	The baud rate to be used. The value can be set individually.
Signal Level	The voltage level for SPI signals to be used.
Aux. Voltage	The voltage level of the support voltage to be used.
Clock mode	The phase and polarity of the clock signal.
SPI command	SPI command to be sent to the connected SPI device. The command has to be entered as blank-separated hexadecimal byte groups ("`05 1F 3A`" for example).
Execute command	Executes a SPI command and displays the result in the *`Result`* dialogue field. Please keep in mind that the SPI bus is used as full duplex line; this means that data is written and received in parallel. Make sure that you write the same number of bytes to the bus as you want to receive (in this case add padding bytes to the SPI command). For example the **Read Status Register** command of a **M25P80** is defined as `0x05`. The result of the command is a single byte (8 bit) representing the current value of the Status Register.

7.2.1.2. Getting global device settings

All SPI related functions of the AnaGate API are declared for Visual Basic users in `AnaGateSPI.bas` and are ready-to-use. The following code snippet includes some exemplary declarations of the API functions used below.

```
Public Declare Function SPIOpenDevice Lib "AnaGateSPIVB6" _
                Alias "_SPIOpenDevice@12" (ByRef Handle As Long, _
                                           ByVal TCPAddress As String, _
                                           ByVal Timeout As Long) As Long

Public Declare Function SPICloseDevice Lib "AnaGateSPIVB6" _
                Alias "_SPICloseDevice@4" (ByVal Handle As Long) As Long

Public Declare Function SPIGetGlobals Lib "AnaGateSPIVB6" _
               Alias "_SPIGetGlobals@20" (ByVal hHandle As Long, _
                                          ByRef nBaudrate As Long, _
                                          ByRef SigLevel As Byte, _
                                          ByRef nAuxVoltage As Byte, _
                                          ByRef nClockMode As Byte) As Long
```

The event procedure `btnCheckAddress_Click` is called on click of the ***Check Address*** button.

```
Private Sub btnCheckAddress_Click()
    Dim nRC As Long, Data As String, sText As String, I As Long

    nRC = SPIOpenDevice(hHandle, Me.IPAddresse.Text, 2000) 1
    If nRC <> 0 Then
        Me.lblErrorMsg.Caption = "Fehler bei SPIOpenDevice: " & GetErrorMsg(nRC)
    Else
        Me.lblErrorMsg.Caption = GetAnagateInfo(hHandle)
    End If
    nRC = SPICloseDevice(hHandle) 2
End Sub
```

1 A call to `SPIOpenDevice` establishes a network connection to the device. If the function fails a textual error description is returned via function `GetErrorMsg`.
2 The connection to the device is closed with the `SPICloseDevice` function.

Reading the device settings and creation of the textual presentation of the data is done by the `GetAnagateInfo` function.

```
Private Function GetAnagateInfo(hHandle As Long) As String
    Dim nRC As Long, sText As String
    Dim nBaudrate As Long, nSigLevel As Byte, nAuxVoltage As Byte, nClockMode As Byte
    Dim nDigitalOutput As Long, nDigitalInput As Long

    nRC = SPIGetGlobals(hHandle, nBaudrate, nSigLevel, nAuxVoltage, nClockMode)
    If (nRC = 0) Then
        sText = sText & "Baudrate=" & CStr(nBaudrate) & ", Siglevel="
        Select Case nSigLevel
            Case 1: sText = sText & "+5.0V"
            Case 2: sText = sText & "+3.3V"
            Case 3: sText = sText & "+2.5V"
            Case Else: sText = sText & "High impedance"
        End Select
        sText = sText & vbCrLf & "AuxVoltage="
        Select Case nAuxVoltage
```



```
            Case 1: sText = sText & "+2.5V"
            Case Else: sText = sText & "+3.3V"
        End Select
        sText = sText & ", ClockMode="
        Select Case nClockMode
            Case 1: sText = sText & "CPHA=0 und CPOL=1"
            Case 2: sText = sText & "CPHA=1 und CPOL=0"
            Case 3: sText = sText & "CPHA=1 und CPOL=1"
            Case Else: sText = sText & "CPHA=0 und CPOL=0"
        End Select
    Else
       sText = sText & "Fehler bei SPIGetGlobals: " & GetErrorMsg(nRC) & vbCrLf
    End If
    GetAnagateInfo = sText
End Function
```

7.2.1.3. Executing a command on the SPI bus

The AnaGate SPI can send arbitrary commands to the connected SPI bus. To write and read data by the PC only the `SPIDataReq` function is required.

```
Public Declare Function SPIDataReq Lib "AnaGateSPIVB6"
                Alias "_SPIDataReq@20" (ByVal hHandle As Long, _
                                        ByVal lpBufferWrite As Any, _
                                        ByVal nBufferWriteLen As Long, _
                                        ByVal lpBufferRead As Any, _
                                        ByVal nBufferReadlLen As Long) As Long
```

The event procedure `btnStart_Click` is called on click of the ***Execute command*** button.

```
Private Sub btnStart_Click()
    Dim nRC As Long, sText As String, I As Integer, sByteText As String
    Dim nBaudrate As Long, nSigLevel As Byte, nAuxVoltage As Byte, nClockMode As Byte
    Dim nBufferWriteLen As Long, nBufferReadLen As Long
    Dim arrWrite(1 To 255) As Byte, arrRead(1 To 255) As Byte

    nRC = SPIOpenDevice(hHandle, Me.IPAddresse.Text, 2000)
    If nRC <> 0 Then
        sText = "Fehler bei SPIOpenDevice: " & GetErrorMsg(nRC)
    Else
        nBaudrate = CLng(Me.txtBaudrate)
        nSigLevel = CLng(Me.cmbSigLevel.ListIndex)
        nAuxVoltage = CLng(Me.cmbAuxVoltage.ListIndex)
        nClockMode = CLng(Me.cmbClockMode.ListIndex)
        nRC = SPISetGlobals(hHandle, nBaudrate, nSigLevel, nAuxVoltage, nClockMode) 1

        If nRC <> 0 Then
              sText = sText & "Fehler bei SPISetGlobals: " & GetErrorMsg(nRC) & vbCrLf
        End If
        Me.lblDeviceInfo.Caption = GetAnagateInfo(hHandle)

        nBufferWriteLen = GetCommand(arrWrite)  2
        nBufferReadLen = nBufferWriteLen

        nRC = SPIDataReq(hHandle, VarPtr(arrWrite(1)), nBufferWriteLen, _
                                  VarPtr(arrRead(1)), nBufferReadLen)      3
        If nRC = 0 Then
            For I = 1 To nBufferReadLen
               sByteText = sByteText & "0x" & ToHex(arrRead(I)) & " "
            Next I
            Me.txtBufferRead = sByteText
            sText = sText & "SPIDataReq OK: " & vbCrLf
        Else
```

```
            sText = sText & "Fehler bei SPIDataReq: " & GetErrorMsg(nRC) & vbCrLf
        End If

        nRC = SPICloseDevice(hHandle)
    End If

End Sub
```

1. A call to `SPISetGlobals` sets the global settings on the device. The parameter values of the input fields in the dialogue form are used.
2. The `GetCommand` function converts the texual SPI command entered in the input field of the form to a byte array structure.
3. To process the data in the read and receive buffers, a byte array is used as VB6 data type for both buffers. For this to work, the real memory address of the array data has to be parsed to the DLL function. This will be done by using the `VarPtr` function on the first byte array element.

.

7.3. Programming language VB.NET

Of course it is also possible to use the functions of the AnaGate API with the .NET programming languages. For these languages the functions have only to be declared correctly in one of the .NET languages. Loading and unloading of the declared API functions is done automatically by the .NET framework.

7.3.1. CAN Console application VB.NET

This programming example for VB.NET demonstrates how to connect to an AnaGate CAN and how to process received CAN data via a callback function.

```
Declare Function CANOpenDevice Lib "AnaGateCAN" (ByRef Handle As Int32, _
                                                 ByVal ConfirmData As Int32, _
                                                 ByVal MonitorOn As Int32, _
                                                 ByVal PortNumber As Int32, _
                                                 ByVal TCPAddress As String, _
                                                 ByVal Timeout As Int32) As Int32
Declare Function CANCloseDevice Lib "AnaGateCAN" (ByVal Handle As Int32) As Int32
Public Delegate Sub CAN_CALLBACK(ByVal ID As Int32, ByVal Buffer As IntPtr, _
                                 ByVal Bufferlen As Int32, ByVal Flags as Int32, _
                                 ByVal Handle as Int32)
Declare Function CANSetCallback Lib "AnaGateCAN" (ByVal Handle As Int32, _
                                                  ByVal MyCB As CAN_CALLBACK) As Int32

Sub CANCallback( ByVal ID As Int32, ByVal Buffer As IntPtr, _
                 ByVal Bufferlen As Int32, ByVal Flags as Int32, _
                 ByVal Handle as Int32)
  Dim Bytes as Byte(8)

  System.Runtime.InteropServices.Marshal.Copy(Buffer, Bytes, 0, Bufferlen )
  Console.Out.Write( "CAN-ID=" )
  Console.Out.Write( ID )
  Console.Out.Write( ",Data=" )
  For I As Int32 = 0 To BufferLen - 1
    Console.Out.Write( Bytes(I) )
  Next

End Sub
```



```
Function Main(ByVal CmdArgs() As String) As Integer

  'Opens the single CAN port of a AnaGate CAN
  Dim RC as Int32 = CANOpenDevice(Handle, 0, 1, 400, 0, "192.168.1.254", 1000)
  If RC = 0 Then
    CANSetCallback( Handle, AddressOf CANCallback )
  End If
  If RC = 0 Then
    CANCloseDevice( Handle )
  End If
End Function
```

7.3.2. SPI Console application VB.NET

This programming example for VB.NET demonstrates how to connect to an AnaGate SPI and how to execute SPI commands.

Note

The source code of the example can be found in directory `Samples/SPI-VB.NET` on the CD.

```
Sub Main()

  Dim hHandle As Int32, nIndex As Integer
  Dim BufferWrite(100) As Byte, BufferRead(100) As Byte
  Dim nBaudrate As Int32 = 5000000 ' 500kBit
  Dim nSigLevel As Byte = 2   ' +3.3V for the signals.
  Dim nAuxVoltage As Byte = 0 ' support voltage is +3.3V.
  Dim nClockMode As Byte = 3  ' CPHA=1 and CPOL=1.

  Dim nRC = SPIOpenDevice(hHandle, "192.168.1.254", 5000) 1
  If nRC <> 0 Then
    Console.WriteLine("Error SPIOpenDevice: " & GetErrorMsg(nRC) & vbCrLf)
  Else
    nRC = SPISetGlobals(hHandle, nBaudrate, nSigLevel, nAuxVoltage, nClockMode) 2
    nRC = SPIGetGlobals(hHandle, nBaudrate, nSigLevel, nAuxVoltage, nClockMode)

    For nIndex = 0 To 100  ' init buffers with some data
      BufferWrite(nIndex) = 69
      BufferRead(nIndex) = 96
    Next nIndex

    BufferWrite(0) = 5 * 16 ' 0x50 = READ STATUS (M25P80)
    BufferWrite(1) = 5 * 16 ' 0x50 = READ STATUS (M25P80)

    nRC = SPIDataReq(hHandle, BufferWrite, 2, BufferRead, 2) 3
    If nRC <> 0 Then
      Console.WriteLine("Error SPIDatReg: " & GetErrorMsg(nRC) & vbCrLf)
    Else
      Console.Write("Result: DATAREQ")
      For nIndex = 0 To 1  ' init buffers with some data
        Console.Write(BufferRead(nIndex) & " ")
      Next
      Console.WriteLine()
    End If

    SPICloseDevice(hHandle) 4
  End If
End Sub
```

1. A call to `SPIOpenDevice` establishes a network connection to the device. If the function fails, a textual error description is returned using the `GetErrorMsg` function.
2. `SPISetGlobals` sets the global parameters of the device (baud rate, signal level, voltage level of the support voltage, clock mode).
3. Via the `SPIDataReq` function data is written to the SPI bus. If the command is successful, the data read from the SPI partner is returned in the receive buffer.
4. The connection to the device is closed with the `SPICloseDevice` function.

The functions of the programming API are defined in a wrapper module. In the following you can see a part of the wrapper module which includes the declarations of all API functions.

```
Imports System.Runtime.InteropServices

Namespace Analytica.AnaGate
  Public Module AnaGateAPI

    Declare Function SPIOpenDevice Lib "AnaGateSPI" (ByRef Handle As Int32, _
                                                     ByVal TCPAddress As String, _
                                                     ByVal Timeout As Int32) As Int32

    Declare Function SPICloseDevice Lib "AnaGateSPI" (ByVal Handle As Int32) As Int32

    Declare Function SPISetGlobals Lib "AnaGateSPI" (ByVal Handle As Int32, _
                                                     ByVal Baudrate As Int32, _
                                                     ByVal SigLevel As Byte, _
                                                     ByVal AuxVoltage As Byte, _
                                                     ByVal ClockMode As Byte) As Int32

    Declare Function SPIGetGlobals Lib "AnaGateSPI" (ByVal Handle As Int32, _
                                                     ByRef Baudrate As Int32, _
                                                     ByRef SigLevel As Byte, _
                                                     ByRef AuxVoltage As Byte, _
                                                     ByRef ClockMode As Byte) As Int32

    Declare Function SPIDataReq Lib "AnaGateSPI" (ByVal Handle As Int32, _
                                                  <MarshalAs(UnmanagedType.LPArray)> _
                                                    ByVal BufferWrite() As Byte, _
                                                  ByVal BufferWriteLen As Int32, _
                                                  <MarshalAs(UnmanagedType.LPArray)> _
                                                    ByVal BufferRead() As Byte, _
                                                  ByVal BufferReadLen As Int32) As Int32

    Declare Function SPIErrorMessage Lib "AnaGateSPI" (ByVal RC As Int32, _
                                                       ByVal Buffer As IntPtr, _
                                                       ByVal Bufferlen As Int32) As Int32
  End Module
End Namespace
```


Part II. SocketCAN interface

Table of Contents

Chapter 8. The *SocketCAN* interface of the *AnaGate* product line

> *SocketCAN*is a set of open source CAN drivers and a networking stack contributed by Volkswagen Research to the Linux kernel. Formerly known as Low Level CAN Framework (LLCF).
>
> —Wikipedia, *socketCAN*

All devices of the *AnaGate CAN* series are controlled by a propriatary network protocol. To use the standard *SocketCAN* interface of the linux kernel, the communication to the AnaGate hardware must be transmitted over a virtual SocketCAN network device. The data interconnection between the AnaGate hardware and the virtual network device is provided by a gateway software component, the so called *SocketCANGateway*, which is described in the following.

Chapter 9. Description of SocketCAN gateway

The *SocketCANGateway* of Analytica GmbH is designed to connect a CAN interface of a Linux system to a CAN port of an *AnaGate CAN* gateway. The *SocketCANGateway* provides easy access to the proprietary TCP/IP protocol of the *AnaGate CAN* via a standard Linux SocketCAN interface.

The SocketCAN concept maps a CAN bus to a network device. Multiple programs can access the bus concurrently via this network device. A network device can be connected to a local CAN port of the Linux system. Alternatively it is possible to create virtual CAN network devices without a physical CAN interface which can then be used by applications to exchange CAN telegrams.

SocketCANGateway is an application that establishes connections both to a SocketCAN network device and to the CAN port of a *AnaGate CAN* gateway. All telegrams that are received from one connection are forwarded to the other connection. Thus applications can use a local virtual SocketCAN network device as if it was directly connected to the CAN hardware.

Chapter 10. Usage of the SocketCAN gateway

A SocketCAN network device is required to run the *SocketCANGateway* application. Usually a virtual network device is used which allows Linux applications to access a CAN port of a *AnaGate CAN* gateway. However, it is also possible to connect a physical CAN device of the Linux system to an *AnaGate CAN* via *SocketCANGateway*.

10.1. Virtual SocketCAN network device

Initially the necessary driver module for virtual SocketCAN network devices must be loaded:

```
$ sudo modprobe vcan
```

Next a virtual SocketCAN network device can be created and enabled with the following commands:

```
$ sudo ip link add dev vcan0 type vcan
$ sudo ip link set up vcan0
```

The command **ip link show vcan0** can be used to show the network device:

```
$ ip link show vcan0
3: vcan0: <NOARP,UP,LOWER_UP> mtu 16 qdisc noqueue state UNKNOWN
    link/can
```

A list of all currently existing network devices can be shown via **ip link show**:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:4d:39:d9 brd ff:ff:ff:ff:ff:ff
3: vcan0: <NOARP,UP,LOWER_UP> mtu 16 qdisc noqueue state UNKNOWN
    link/can
```

If multiple buses should be accessed via different CAN ports of one or more *AnaGate CAN* gateways, a separate network device should be created for each bus:

```
$ sudo ip link add dev vcan1 type vcan
$ sudo ip link set up vcan1
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:4d:39:d9 brd ff:ff:ff:ff:ff:ff
3: vcan0: <NOARP,UP,LOWER_UP> mtu 16 qdisc noqueue state UNKNOWN
    link/can
4: vcan1: <NOARP,UP,LOWER_UP> mtu 16 qdisc noqueue state UNKNOWN
    link/can
```

10.2. SocketCANGateway

SocketCANGateway

SocketCANGateway — A Linux console application which connects a SocketCAN network device to a CAN port of an *AnaGate CAN* gateway. The application is available in 32 and 64 bit versions for x86 CPUs.

Syntax

```
SocketCANGateway {interface}
[ -i | --ipaddress= ipaddress]
[ -b | --baudrate= baudrate]
[ -p | --canport= canport]
[ -t | --termination= { 0 | 1 }]
[ -s | --highspeed= { 0 | 1 }]
[ -d | --timestamp= { 0 | 1 }]
[ -h | --help | --version ]
```

Parameter

interface	Name of the SocketCAN network device.
-i, --ipaddress	IP address of the *AnaGate CAN* (default 192.168.1.254).
-b, --baudrate	CAN bus baud rate in bps (default 1000000=1 MBit/s). Currently the following baud rates are supported: 10000 for kBit/s, 20000 for 20 kBit/s, 50000 for 50 kBit/s, 62500 for 62.5 kBit/s, 100000 for 100 kBit/s, 125000 for 125 kBit/s, 250000 for 250 kBit/s, 500000 for 500 kBit/s and 1000000 for 1 MBit/s. The devices *AnaGate CAN uno/duo/quattro/USB/ X2/X4/X8* also support 800000 for 800 kBit/s.
-p, --canport	CAN port of the *AnaGate CAN* (default 0=A).
-t, --termination	bus termination (default 0=off).
-s, --highspeed	High speed mode (default 0=off).
-d, --timestamp	Time stamps in messages (default 0=off).
-h, --help	Displays a help message and exits.
--version	Displays version information and exits.

Description

SocketCANGateway establishes connections both to a SocketCAN network device and to the CAN port of a *AnaGate CAN* gateway. All telegrams that are received from one connection are forwarded to the other connection.

When connecting to the *AnaGate CAN* the CAN port is configured with the given parameter values. If no parameters are set the default values are used.

See the following example for connecting the virtual SocketCAN network device vcan0 to Port A of an *AnaGate CAN* gateway using the IP address 192.168.1.254 while explicitly setting its configuration values.


```
$ sudo modprobe vcan
$ sudo ip link add dev vcan0 type vcan
$ sudo ip link set up vcan0
$ SocketCANGateway vcan0 --ipaddress=192.168.1.254 --baudrate=1000000 \
> --canport=0 --termination=1 --highspeed=1 --timestamp=0
****IP:192.168.1.254****
****Baudrate:1000000 1000000****
****Port:0****
****Termination:1****
****HighSpeed:1****
****Date/Time:0****
SocketCANGateway (Connection between SocketCAN and AnaGate
CAN), version 0.1.0 of Jul  9 2014
Copyright (C) 2014 Analytica GmbH

Hardware=vcan0...OK!

Press ^C to abort.
```

Example 10.1. Program call

10.3. SocketCAN example application

The following C code is an example of using the SocketCAN API. It sends a CAN telegram using the vcan0 interface.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <net/if.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#include <linux/can.h>
#include <linux/can/raw.h>

int
main(void)
{
 int s;
 int nbytes;
 struct sockaddr_can addr;
 struct can_frame frame;
 struct ifreq ifr;

 char *ifname = "vcan0";

 if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
  perror("Error while opening socket");
  return -1;
 }

 strcpy(ifr.ifr_name, ifname);
 ioctl(s, SIOCGIFINDEX, &ifr);
```



```
 addr.can_family  = AF_CAN;
 addr.can_ifindex = ifr.ifr_ifindex;

 printf("%s at index %d\n", ifname, ifr.ifr_ifindex);

 if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
  perror("Error in socket bind");
  return -2;
 }

 frame.can_id  = 0x123;
 frame.can_dlc = 2;
 frame.data[0] = 0x11;
 frame.data[1] = 0x22;

 nbytes = write(s, &frame, sizeof(struct can_frame));

 printf("Wrote %d bytes\n", nbytes);

 return 0;
}
```


Part III. Scripting language Lua

Table of Contents

Chapter 11. The Lua scripting interface of the *AnaGate* product line

> **Lua** [http://www.lua.org] (from Portuguese: lua meaning moon; explicitly not "LUA") is a lightweight multi-paradigm programming language designed as a scripting language with "extensible semantics" as a primary goal. Lua is cross-platform since it is written in ISO C. Lua has a relatively simple C API, thus "Lua is especially useful for providing end users with an easy way to program the behavior of a software product without getting too far into its innards."
>
> —Wikipedia, *Lua*

In order to be able to solve simple programming problems concerning the *AnaGate* devices with the scripting language *Lua*, the *Lua* interpreter is extended by several functions to operate the different *AnaGate* devices. These additional functions are described in detail in the following chapters and are closely related to the functions of the *AnaGate API* libraries.

Source files for *Lua* (called scripts) are created and edited on a personal computer (Windows or Linux) in a standard text editor. Then the script is simply executed in the command shell via a free *Lua* interpreter. The full standard functionality of the *Lua* language can be used as well as additional functional extensions for access of the *AnaGate* hardware.

The scripting language *Lua* is very well-suited for use on *embedded systems* because of its good performance and small runtime size. For this reason, the *Lua* interpreter is integrated into the firmware of the *AnaGate* hardware [1]. Thus it is possible to execute scripts not only on the personal computer, but also on the *AnaGate* device itself.

Note

Please refer to the printed paperbacks *Lua Reference Manual* ([LuaRef2006-EN]) and *Programming in Lua* ([LuaProg2013-EN]) for detailed information about Lua. The reference manual is also available online at Lua.org [http://www.lua.org].

11.1. Creating scripts

Creating and editing script files for the scripting language *Lua* is exceptionally easy, because a standard text editor is sufficient to do that. On Windows operating systems Notepad or Wordpad can be used for example, on Linux systems vi or other text tools are available.

[1] only AnaGate CAN uno, AnaGate CAN duo, AnaGate CAN quattro, AnaGate CAN USB and AnaGate Universal Programmer

In the meantime some text editors, partly free of charge, support syntax-highlighting for *Lua*, which makes it much easier for a programmer to develop.

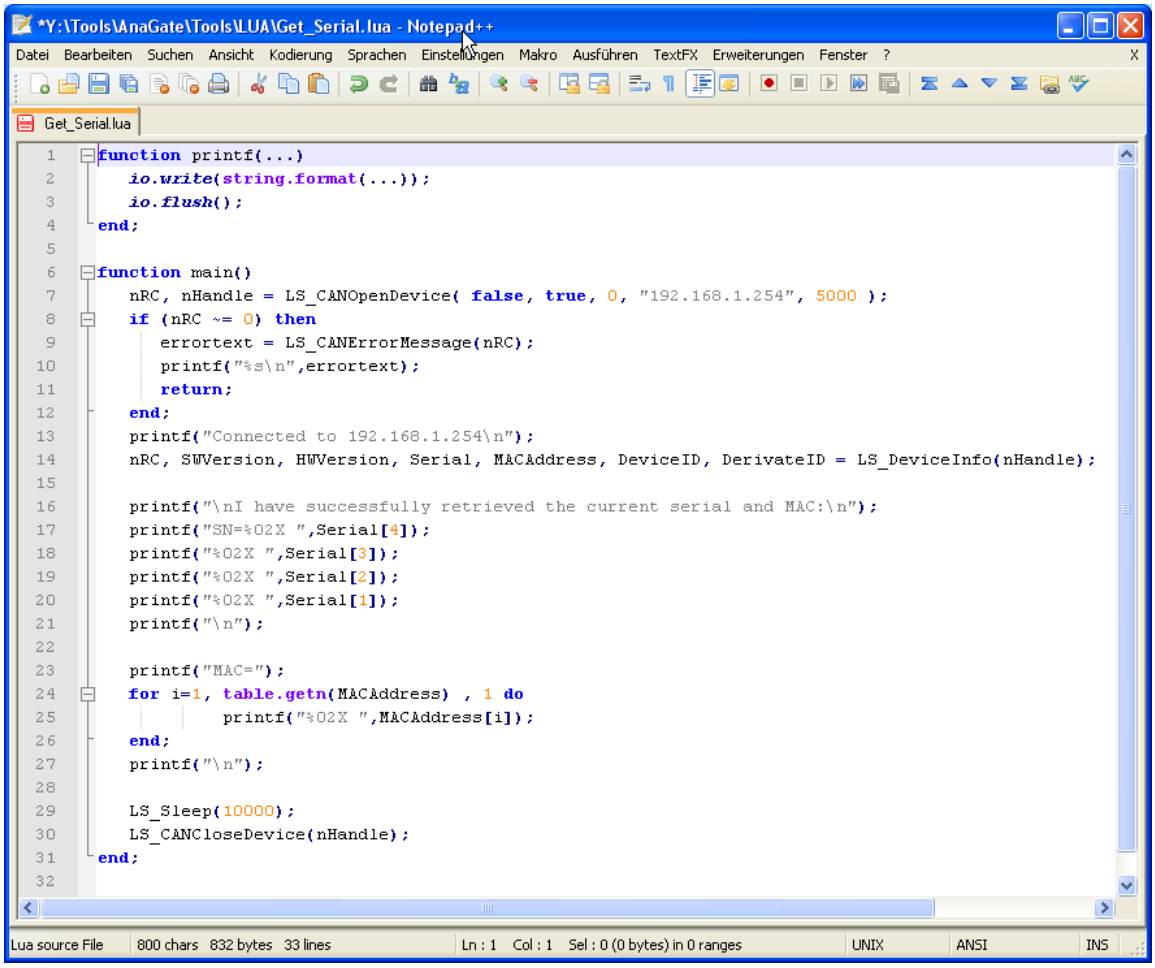


Figure 11.1. Edit Lua script in a text editor

When coding of a script is finished, it can be executed and tested on a personal computer as described below.

11.2. Running scripts on personal computer

To execute *Lua* script files on a personal computer, a current program version of the *Lua* interpreter must be available.

On the CD-ROM which is included in the scope of delivery a modified *Lua* interpreter can be found in the directory Lua. This interpreter consists of a single executable named LUA.exe which includes all functional extensions to operate the *AnaGate* hardware.

Tip

The lastest version of LUA.exe can be downloaded free of charge via the support pages of the product homepage [http://www.anagate.de/support/download.htm].

Except of the program executable `LUA.exe` no other program files are needed, so that there is only one single file to copy to the computer hard-disk (or file server, SUB stick, ...).

A script file is executed easily via the command line shell. Only the name of the script file has to specified to start it.

The following example shows how a script file named `Get_Serial.lua` is executed in the windows command prompt.

```
T:\Tools\LUA>LUA.exe Get_Serial.lua 1
Connected to 192.168.1.254  2
I have successfully retrieved the current serial and MAC:
SN=01 02 02 1D
MAC=00 50 C2 3C B2 1D
T:\Tools\LUA>
```

1. The file name of the script to execute has to be supplied as a parameter on start of the interpreter.
2. The serial number and MAC address of a device at IP address `192.168.1.254` are retrieved and written to the standard output.

11.3. Running scripts on *AnaGate* hardware

As already mentioned before, it is possible to execute self-created application scripts with an installed Lua script interpreter directly on the *AnaGate* hardware.

Via the HTTP interface of each device Lua script files can be downloaded to the device and executed locally. In the following you can see a screen-shot of the *Lua* configuration page of an *AnaGate CAN uno*.

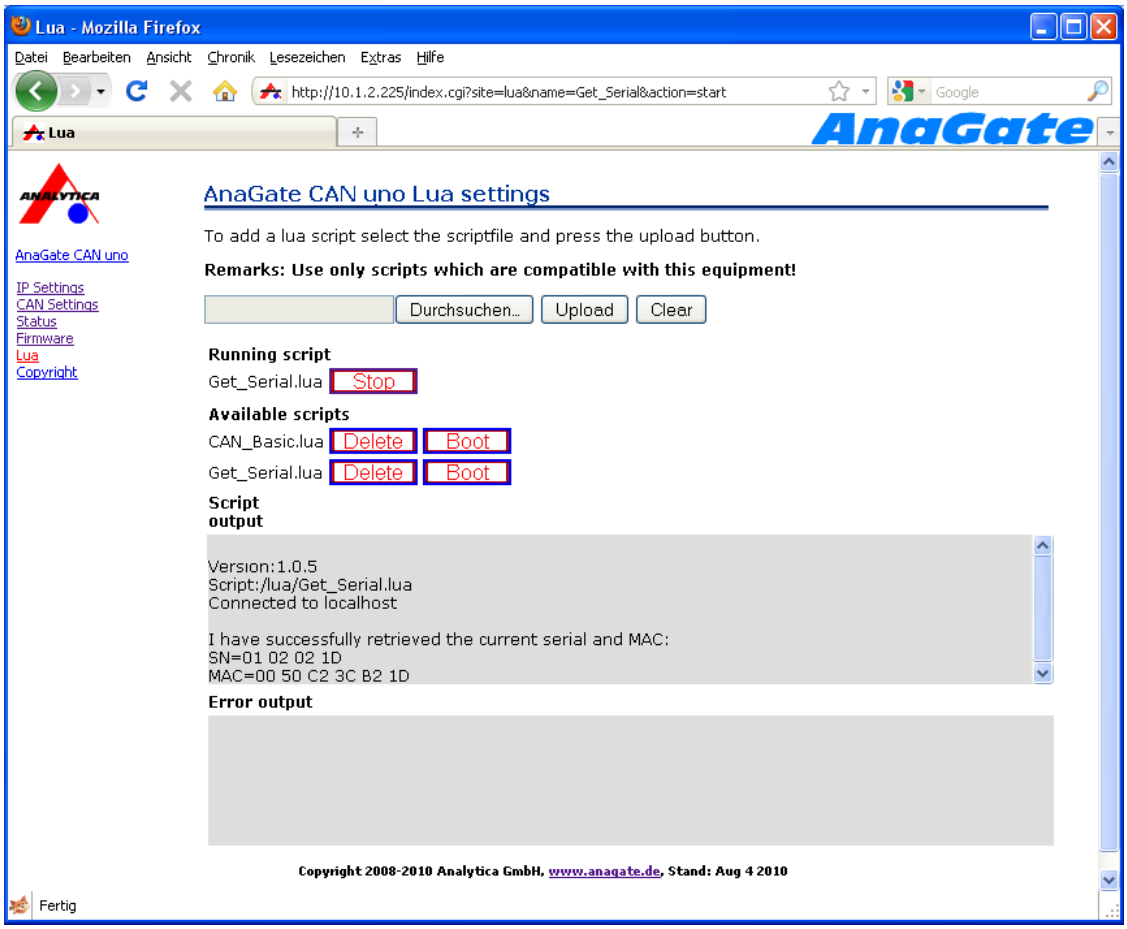


Figure 11.2. HTTP interface, Lua settings

Browse... Opens a file upload dialogue to select a Lua script file.

Upload Uploads the selected script file to the device.

Clear Clears the current script file selection.

Boot script — Script file executed on system start-up. Via the button **Delete** the boot script can be deactivated. Only one boot script is allowed.

Running script — Displays the currently executed script file. Via the button **Stop** the execution can be cancelled.

Available scripts — Displays all scripts which are currently available on the device.

To start the execution of a script click on the button **Start**. Via the button **Delete** a script can be deleted on the device and via **Boot** a script can be defined as boot script.

script output area — In this text area the standard output (stdout) of the currently executing script is displayed. Via the button **Clear** this text area can be cleared.

error output area — In this text area the standard error output (stderr) of the currently executing script is displayed. Via the button **Clear** this text area can be cleared.

Tip

The text areas for script and error output are not refreshed automatically. A manual page reload of the current page refreshes both text areas.

Chapter 12. Common function reference

LS_DeviceInfo

LS_DeviceInfo — Retrieves some global information from the AnaGate hardware.

Syntax

```
int RC, int SWVersion, int HWVersion, table(4) Serial, table(6)
MACAddress, int DeviceID, int SWDerivateID = LS_DeviceInfo(int Handle);
```

Parameter

hHandle Valid access handle returned by a call to `LS_CANOpenDevice`, `LS_I2COpenDevice` or `LS_SPIOpenDevice`.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

SWVersion — Firmware version. The version number consists of 3 numbers (major.minor.revision) which are stored in a 4-byte integer value.

HWVersion — Hardware version. The version number consists of 3 numbers (major.minor.revision) which are stored in a 4-byte integer value.

Serial — Serial number of the AnaGate hardware (4 byte).

MACAddress — MAC address of the AnaGate hardware (6 byte).

nDeviceID — Device specific identifier. Specifies the device type of the hardware.

- 1 = AnaGate I2C
- 2 = AnaGate CAN
- 3 = AnaGate SPI
- 8 = AnaGate Universal Programmer
- 9 = AnaGate Renesas

SWDerivateID — Indicates a customer-specific firmware version, if not `0x00`.

Description

Returns specific information about a device of the AnaGate product line.

See also

LS_CANOpenDevice, LS_SPIOpenDevice, LS_I2COpenDevice

LS_GetTime

LS_GetTime — Returns the current system time.

Syntax

```
int RC, table(2) Time = LS_GetTime();
```

Parameter

This function does not have any parameters.

Return values

RC	Returns `0` if successful or an error value otherwise (Table A.6, "Return values for Lua scripting").
Time[1], Time[2]	*`Time[1]`* specifies the number of seconds elapsed since 01.01.1970. In *`Time[2]`* the fractions of a second is returned in milliseconds.

Description

Returns the system time as the number of elapsed seconds and milliseconds since midnight of January 1, 1970.

LS_Sleep

LS_Sleep — Suspends the execution until the time-out interval elapses.

Syntax

```
int RC = LS_Sleep(unsigned int Milliseconds);
```

Parameter

nMilliseconds — The time interval for which execution is to be suspended, in milliseconds.

Return value

RC — Returns `0` if successful or an error value otherwise (Table A.6, “Return values for Lua scripting”).

Description

Suspends the execution until the time-out interval in milliseconds has elapsed.

Chapter 13. CAN Reference

The CAN API can be used with all CAN gateway models of the AnaGate series. The programming interface is identical for all devices and generally uses the network protocol TCP or UDP.

The following devices can be addressed via the CAN API interface:

- AnaGate CAN
- AnaGate CAN uno
- AnaGate CAN duo
- AnaGate CAN quattro
- AnaGate CAN USB
- AnaGate CAN X2
- AnaGate CAN X4
- AnaGate CAN X8

Note

All CAN specific functionality of the AnaGate C-API is also available for Lua users. The Lua function extensions are documented below.

LS_CANOpenDevice

LS_CANOpenDevice — Opens a network connection (TCP) to an AnaGate CAN device.

Syntax

```
int RC, int Handle = LS_CANOpenDevice(bool SendDataConfirm, bool SendDataInd, uint8 CANPort, string IPAddress, int Timeout);
```

Parameter

SendDataConfirm — If set to true, all incoming and outgoing data requests are confirmed via the internal message protocol. Without confirmations a better transmission performance is reached.

SendDataInd — If set to false, all incoming telegrams are discarded.

CANPort — CAN port number. Allowed values are:

- `0` for port A (Modells AnaGate CAN uno, AnaGate CAN duo, AnaGate CAN quattro, AnaGate CAN USB and AnaGate CAN)
- `1` for port B (AnaGate CAN duo, AnaGate CAN quattro)
- `2` for port C (AnaGate CAN quattro)
- `3` for port D (AnaGate CAN quattro)

IPAddress — Network address of the AnaGate partner.

Timeout — Default timeout for accessing the AnaGate in milliseconds.

A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

Return value

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Handle — Access handle if successfully connected to the AnaGate device.

Description

Opens a TCP/IP connection to a CAN interface of an AnaGate CAN device. If the connection is established, CAN telegrams can be sent and received.

The connection should be closed with the function `CANCloseDevice` if not longer needed.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See the following example for the initial programming steps.

```
-- open: use no confirmations and receive incoming CAN data
local nRC, Handle = LS_CANOpenDevice(false, true, 0, "192.168.1.254", 5000)
if nRC == 0 then
   -- now do something
   LS_CANCloseDevice(Handle);
end
```

See also

LS_CANCloseDevice

LS_CANRestartDevice

LS_CANCloseDevice

LS_CANCloseDevice — Closes an open network connection to an AnaGate CAN device.

Syntax

```
int RC = LS_CANCloseDevice(int Handle);
```

Parameter

Handle Valid access handle.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Closes an open network connection to an AnaGate CAN device. The *`Handle`* parameter is a return value of a successful call to the function `LS_CANOpenDevice`.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See also

LS_CANOpenDevice

LS_CANRestartDevice

LS_CANRestartDevice — Restarts a AnaGate CAN device.

Syntax

```
int RC = LS_CANRestartDevice(string IPAddress, int Timeout);
```

Parameter

IPAddress — Network address of the AnaGate partner.

nTimeout — Default timeout for accessing the AnaGate in milliseconds. A timeout is reported if the AnaGate partner does not respond within the defined timeout period.

Return value

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Restarts the AnaGate CAN device at the specified network address. It implicitly disconnects all open network connections to all existing CAN interfaces. The Restart command is possible even if the maximum number of allowed connections is reached.

Important

It is recommended to use this command only in emergency cases if there is a need to connect even if the maximum number of concurrent connections is reached.

See also

LS_CANOpenDevice

LS_CANSetGlobals

LS_CANSetGlobals — Sets the global settings which are to be used on the CAN bus

Syntax

```
int RC = LS_CANSetGlobals(int Handle, uint32 Baudrate, uint8 OperatingMode, bool Termination, bool HighSpeedMode, bool TimeStampOn);
```

Parameter

Handle	Valid access handle.
Baudrate	The baud rate to be used. The following values are allowed: • 10.000 für 10kBit • 20.000 für 20kBit • 50.000 für 50kBit • 62.500 für 62,5kBit • 100.000 für 100kBit • 125.000 für 125kBit • 250.000 für 250kBit • 500.000 für 500kBit • 800.000 für 800kBit (not AnaGate CAN) • 1.000.000 für 1MBit
OperatingMode	The operating mode to be used. The following values are allowed. • 0 = default mode. • 1 = loop back mode: No telegrams are sent via CAN bus. Instead they are received as if they had been transmitted over CAN by a different CAN device. • 2 = listen mode: Device operates as a passive bus partner, meaning no telegrams are sent to the CAN bus (nor ACKs for incoming telegrams). • 3 = offline mode: No telegrams are sent or received on the CAN bus. Thus no error frames are generated on the bus if other connected CAN devices send telegrams with a different baud rate.
Termination	Use integrated CAN bus termination (`true`=yes, `false`=no). This setting is not supported by all AnaGate CAN models.

HighSpeedMode Use high speed mode (`true`=yes, `false`=no). This setting is not supported by all AnaGate CAN models.

The high speed mode was created for large baud rates with continuously high bus load. In this mode telegrams are not confirmed on the protocol layer and the software filters defined via `LS_CANSetFilter` are ignored.

TimeStampOn Use time stamp mode (`true`=yes, `false`=no). This setting is not supported by all AnaGate CAN models.

In activated time stamp mode an additional timestamp is sent with the CAN telegram. This timestamp indicates when the incoming message was received by the CAN controller or when the outgoing message was confirmed by the CAN controller.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Sets the global settings of the used CAN interface. These settings are effective for all concurrent connections to the CAN interface. The settings are not saved permanently on the device and are reset after every device restart.

Remarks

The settings of the integrated CAN bus termination, the high speed mode and the time stamp are not supported by the AnaGate CAN (hardware version 1.1.A). These settings are ignored by the device.

The offline mode is only supported by firmware versions 1.3.12 and higher. The AnaGate CAN doesn't support this mode.

See also

LS_CANGetGlobals

LS_CANGetGlobals

LS_CANGetGlobals — Returns the currently used global settings on the CAN bus.

Syntax

```
int RC, int Baudrate, uint8 OperatingMode, bool Termination, bool
HighSpeedMode, bool TimeStampOn = LS_CANGetGlobals(int Handle);
```

Parameter

Handle Valid access handle.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

nBaudrate — The baud rate currently used on the CAN bus.

OperatingMode — The operating mode to be used. The following values are returned.

- 0 = default mode.
- 1 = loop back mode: No telegrams are sent via CAN bus. Instead they are received as if they had been transmitted over CAN by a different CAN device.
- 2 = listen mode: Device operates as passive bus partner, meaning no telegrams are sent to the CAN bus (nor ACKs for incoming telegrams).
- 3 = offline mode: No telegrams are sent or received on the CAN bus. Thus no error frames are generated on the bus if other connected CAN devices send telegrams with a different baud rate.

Termination — Is the integrated CAN bus termination used? `true`=yes, `false`=no). This setting is not supported by all AnaGate CAn models.

HighSpeedMode — Is the high speed mode switched on? (`true`=yes, `false`=no). This setting is not supported by all AnaGate CAN models.

The high speed mode was created for large baud rates with continuously high bus load. In this mode telegrams are not confirmed on protocol layer and the software filters defined via `LS_CANSetFilter` are ignored.

Description

Returns the global settings of the used CAN interface. These settings are effective for all concurrent connections to the CAN interface.

Remarks

The settings of the integrated CAN bus termination, the high speed mode and the time stamp are not supported by the AnaGate CAN (hardware version 1.1.A). These settings are ignored by the device.

The offline mode is only supported by firmware versions 1.3.12 and higher. The AnaGate CAN doesn't support this mode.

See also

LS_CANSetGlobals

LS_CANWrite

LS_CANWrite — Sends a CAN telegram to the CAN bus via the AnaGate device.

Syntax

```
int RC = LS_CANWrite(int Handle, int32 CANID, table(uint8[DataLen])
Data, uint8 DataLen, uint8 Flags);
```

Parameter

Handle — Valid access handle.

CANID — CAN identifier of the sender. The parameter *Flags* defines whether the address is in extended format (29-bit) or standard format (11-bit).

Data — Data buffer with telegram data.

DataLen — Length of data buffer (max. 8 bytes).

Flags — The format flags are defined as follows.

- Bit 0: If set, the CAN identifier is in extended format (29 bit), otherwise not (11 bit).
- Bit 1: If set, the telegram is marked as remote frame.

Return value

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

This function sends a CAN telegram to the CAN bus via the AnaGate device similar to the `LS_CANWriteEx` function.

The `LS_CANWriteEx` function additionally returns a timestamp of the time at which the telegram was sent.

Note

With remote frames (RTR = remote transmission request) a destination node can request data from a source node. The data length is to be set to the number of requested bytes. On the CAN bus no data is sent; only the data size information is transmitted.

When using the `LS_CANWrite` or `LS_CANWriteEx` functions to send remote frames the data buffer and the buffer size equal to the number of requested bytes have to be set correctly.

See the following example for sending a data telegram to the connected CAN bus.

```
local tabData = {}
```

```
for i = 1, 8, 1 do
   table.insert(tabData, i)
end

local nFlags = 0x0 //  11bit address + standard (not remote frame)
local nCANId = 0x25 // send with CAN ID 0x25;

local nRC, hHandle = LS_CANOpenDevice(true, true, 0, "192.168.1.254", 5000)
if nRC == 0 then
   // send 8 bytes with CAN id 37
   nRC = LS_CANWrite(hHandle, nCANId, tabData, #tabData, nFlags)

   // send a remote frame to CAN id 37 (request 4 data bytes)
   nRC = LS_CANWrite( hHandle, nCANId, tabData, 4, 0x02 )

   LS_CANCloseDevice(hHandle)
end
```

Remarks

For devices of type AnaGate CAN (hardware version 1.1.A) the function `LS_CANWriteEx` is equal to `LS_CANWrite`. The return values *`nSeconds`* and *`pnMicroseconds`* will remain unchanged.

See also

LS_CANWriteEx

LS_CANWriteEx

LS_CANWriteEx — Sends a CAN telegram to the CAN bus via the AnaGate device.

Syntax

```
int RC, int32 Seconds, int32 Microseconds = LS_CANWriteEx(int Handle,
int32 CANID, table(uint8[DataLen]) Data, uint8 DataLen, uint8 Flags);
```

Parameter

Handle	Valid access handle.
CANID	CAN identifier of the sender. The parameter *Flags* defines whether the address is in extended format (29-bit) or standard format (11-bit).
Data	Data buffer with telegram data.
DataLen	Length of data buffer (max. 8 bytes).
nFlags	The format flags are defined as follows.

- Bit 0: If set, the CAN identifier is in extended format (29 bit), otherwise not (11 bit).
- Bit 1: If set, the telegram is marked as remote frame.

Return value

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
Seconds	Timestamp of the confirmation of the CAN controller (seconds since 01.01.1970).
Microseconds	Micro seconds portion of the timestamp.

Description

This function sends a CAN telegram to the CAN bus via the AnaGate device similar to the `LS_CANWrite` function.

The `LS_CANWriteEx` function additionally returns a timestamp of the time at which the telegram was sent.

Note

With remote frames (RTR = remote transmission request) a destination node can request data from a source node. The data length is to be set to the number of requested bytes. On the CAN bus no data is sent; only the data size information is transmitted.

When using the `LS_CANWrite` or `LS_CANWriteEx` functions to send remote frames the data buffer and the buffer size equal to the number of requested bytes have to be set correctly.

See the following example for sending a data telegram to the connected CAN bus.

```
local tabData = {}
for i = 1, 8, 1 do
   table.insert(tabData, i)
end

local nFlags = 0x0 //  11bit address + standard (not remote frame)
local nCANId = 0x25 // send with CAN ID 0x25;

local nRC, hHandle = LS_CANOpenDevice(true, true, 0, "192.168.1.254", 5000);
if nRC == 0 then
  local nSeconds, nMicroSeconds
   // send 8 bytes with CAN id 37
   nRC, nSeconds, nMicroSeconds = LS_CANWriteEx(hHandle, nCANId, tabData, #tabData, nFlags)

   // send a remote frame to CAN id 37 (request 4 data bytes)
   nRC, nSeconds, nMicroSeconds = LS_CANWriteEx(hHandle, nCANId, tabData, 4, 0x02)

   LS_CANCloseDevice(hHandle)
end
```

Remarks

For devices of type AnaGate CAN (hardware version 1.1.A) the function `LS_CANWriteEx` is equal to `LS_CANWrite`. The return values *`nSeconds`* and *`nMicroseconds`* will remain unchanged.

See also

LS_CANWrite

LS_CANGetMessage

LS_CANGetMessage — Reads a CAN message from the internal buffer.

Syntax

```
int Available, int32 CANID, uint8 DataLen, table(uint8[Length]) Data,
uint8 Flags, int32 Seconds, int32 Microseconds = LS_CANGetMessage(int
Handle, int Timeout);
```

Parameter

Handle — Valid access handle.

Timeout — Maximum period of time in milliseconds to wait for the a new data telegram.

Return values

Available — Number of messages which are left in the internal message puffer after the `LS_CANGetMessage` call. If there is currently no message available `-10` (ERR_NO_DATA) is returned.

CANID — CAN identifier of the sender. Parameter *`nFlags`* defines whether the address is in extended format (29-bit) or standard format (11-bit).

DataLen — Length of data buffer.

Data — Data buffer with telegram data.

Flags — The format flags are defined as follows.

- Bit 0: If set the CAN identifier is in extended format (29 bit), otherwise not (11 bit).
- Bit 1: If set, the telegram is marked as remote frame.
- Bit 2: If set, the telegram has a valid timestamp.

Seconds — Timestamp of the confirmation of the CAN controller (seconds from 01.01.1970).

Microseconds — Micro seconds portion of the timestamp.

Description

This function reads a single CAN data telegram out of an internal message buffer. The message buffer is automatically filled with all incoming CAN data telegrams in a separate thread.

The parameter *`Timeout`* defines the maximum duration the function should wait for a new data telegram if there is currently no telegram in the internal buffer. If no new message is received within the time out, the return value *`Available`* is set to `-10` (ERR_NO_DATA).

See the following example which handles incoming CAN data telegrams.

```
local nRC, hHandle = LS_CANOpenDevice(true, true, 0, "192.168.1.254", 5000)
if nRC == 0 then
   -- set globals: 500Kbit, standard mode, termination on, no high speed, no timestamp
   nRC = LS_CANSetGlobals(hHandle, 500000, 0, true, false, false)
   local nCurMsg = 0

   repeat
      local nAvail, ID, Len, Data, Flags, Sec, Microsec = LS_CANGetMessage(hHandle, 100)
      if nAvail >= 0 then
         nCurMsg = nCurMsg + 1

         -- now do something with the incomming message data
         io.write(string.format("ID: %.8x", ID))  -- for example, write out CAN id
      else
         LS_Sleep(25)  -- wait 25 ms if no message available
      end
   until nCurMsg >= 100  -- read only 100 messages, then stop

   LS_CANCloseDevice(hHandle)
end
```

Remarks

For devices of type AnaGate CAN (hardware version 1.1.A) the return values *Seconds* and *Microseconds* are always set to zero.

See also

LS_CANWrite

LS_CANWriteEx

LS_CANSetFilter

LS_CANSetFilter — Sets the current filter settings for the connection.

Syntax

```
int RC = LS_CANSetFilter(int Handle, table(uint32[16]) Filter);
```

Parameter

Handle Valid access handle.

Filter Table of 8 filter entries (4 mask and 4 range filter entries). A filter entry contains of two 32-bit values. Unused mask filter entries must be initialized with 0 values. Unused range filter entries must be initialized with 0 for the start value and 0x1FFFFFFF for the end value.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

This function sets the current filter settings for the current connection. Filter can be used to suppress messages with specific CAN message IDs.

A mask filter consists of a mask value which defines the bits of the CAN identifier to examine and the appropriate filter value. If the CAN identifier matches in the indicated filter mask with the filter value the incoming CAN telegram is sent to the PC, otherwise not.

A range filter defines an address range with an appropriate start and end address. If the CAN identifier is not in the indicated filter range the incoming CAN telegram is not sent to the PC.

By default no filters are set so all CAN IDs are transmitted. If the parameter *bHighSpeedMode* of the `CANSetGlobals` function is set all filters are ignored to increase the data throughput.

See also

LS_CANGetFilter

LS_CANGetFilter

LS_CANGetFilter — Returns the current filter settings for the connection.

Syntax

```
int RC, table(uint32[16]) Filter = LS_CANGetFilter(int hHandle);
```

Parameter

Handle Valid access handle.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Filter Table of 8 filter entries (4 mask and 4 range filter entries). A filter entry consists of two 32-bit values. Unused mask filter entries are initialized with 0 values. Unused range filter entries are initialized with (0,0x1FFFFFFF) value pairs.

Description

This function retrieves the current filter settings for the current connection. Filters can be used to suppress messages with specific CAN message IDs.

See also

LS_CANSetFilter

LS_CANSetTime

LS_CANSetTime — Sets the current system time on the AnaGate device.

Syntax

```
int RC = LS_CANSetTime(int Handle, int32 Seconds, int32 Microseconds);
```

Parameter

Handle Valid access handle.

Seconds Time in seconds since 01.01.1970.

Microseconds Micro seconds.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The `LS_CANSetTime` function sets the system time on the AnaGate hardware.

If the time stamp mode is switched on via the `LS_CANSetGlobals` function, the AnaGate hardware adds a time stamp to each incoming CAN telegram and a time stamp to the confirmation of a telegram sent via the API (only if confirmations are switched on for data requests).

Remarks

The setting of the base time for the time stamp mode is not supported by the AnaGate CAN (hardware version 1.1.A). This setting is ignored by the device.

LS_CANErrorMessage

LS_CANErrorMessage — Returns a description of the given error code as a text string.

Syntax

```
string ErrorMsg = LS_CANErrorMessage(int RetCode);
```

Parameter

RetCode Error code for which the error description is to be determined.

Return value

ErrorMsg Textual description of the error code.

Description

Returns a textual description of the parsed error code (see Appendix A, *API return codes*).

See the following example in Lua scripting language.

```
local nRC = 0
local sErrorText = 'No Error'

//... call an API function here

sErrorText = LS_CANErrorMessage(nRC)
print(sErrorText)
```

LS_CANReadDigital

LS_CANReadDigital — Reads the current values of digital input and output registers of the AnaGate device.

Syntax

`int RC, uint32 InputBits, uint32 OutputBits = **LS_CANReadDigital**(int *Handle*);`

Parameter

Handle Valid access handle.

Return value

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
InputBits	Current value of the digital input register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.
OutputBits	Current value of the digital output register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The current values of the digital inputs and outputs can be retrieved with the `LS_CANReadDigital` function.

See the following example for setting and reading the digital IOs.

```
local nOutputs = 0x03

local nRC, hHandle = LS_CANOpenDevice(false, false, 0, "192.168.1.254", 5000)
if nRC == 0 then
   -- set the digital output register (PIN 0 and PIN 1 to HIGH value)
   nRC = LS_CANWriteDigital(hHandle, nOutputs)
   local nInputs
   -- read all input and output registers
   nRC, nInputs, nOutputs = LS_CANReadDigital(hHandle)

   LS_CANCloseDevice(hHandle)
end
```

See also

LS_CANWriteDigital

LS_CANWriteDigital

LS_CANWriteDigital — Writes a new value to the digital output register of the AnaGate device.

Syntax

```
int RC = LS_CANWriteDigital(int Handle, uint32 OutputBits);
```

Parameter

Handle	Valid access handle.
OutputBits	New register value. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The digital outputs can be written with the `LS_CANWriteDigital` function.

A simple example for reading/writing of the IOs can be found at the description of `LS_CANReadDigital`.

See also

LS_CANReadDigital

LS_CANReadAnalog

LS_CANReadAnalog — Reads the current values of analog inputs of the AnaGate device.

Syntax

```
int RC, uint32 PowerSupply, table(uint8[min(ReadCount,InputCount)])
AnalogInputs, uint16 ReadCount = LS_CANReadAnalog(int Handle, uint16
InputCount);
```

Parameter

Handle	Valid access handle.
InputCount	Number of analog inputs of the AnaGate device.

Return value

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
PowerSupply	Current supply voltage in millivolt.
AnalogInputs	Table with the current values of the analog inputs in millivolt.
ReadCount	Number of read analog inputs.

Description

The AnaGate CAN X series models have connectors for 4 analog inputs and 4 analog outputs at the top side.

The current values of the analog inputs and the current supply voltage can be retrieved with the `LS_CANReadAnalog` function.

See the following example for reading the analog inputs.

```
local nInputs = 4

local nRC, hHandle = LS_CANOpenDevice(false, false, 0, "192.168.1.254", 5000)
if nRC == 0 then
   local nPowerSupply, tAnalogInputs, nReadCount
   -- read input values
   nRC, nPowerSupply, tAnalogInputs, nReadCount = LS_CANReadAnalog(hHandle, nInputs)

   LS_CANCloseDevice(hHandle)
end
```

See also

LS_CANWriteAnalog

LS_CANWriteAnalog

LS_CANWriteAnalog — Writes new values to the analog outputs of the AnaGate device.

Syntax

```
int RC = LS_CANWriteAnalog(int Handle, table(uint32[OutputCount]) AnalogOutputs, uint16 OutputCount);
```

Parameter

Handle	Valid access handle.
AnalogOutputs	Array of new analog output values in millivolt.
OutputCount	Number of AnalogOutputs values.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The AnaGate CAN X series models have connectors for 4 analog inputs and 4 analog outputs at the top side.

The analog outputs can be written with the `LS_CANWriteAnalog` function. The upper output voltage is limited by the supply voltage of the AnaGate device. The current value of the supply voltage can be read with the `LS_CANReadAnalog` function.

See the following example for writing the analog outputs.

```
local aOutputs = {0, 12000, 24000, 0}

local nRC, hHandle = LS_CANOpenDevice(false, false, 0, "192.168.1.254", 5000)
if nRC == 0 then
   -- write output values
   nRC = LS_CANWriteAnalog(hHandle, aOutputs, #aOutputs)

   LS_CANCloseDevice(hHandle)
end
```

See also

LS_CANReadAnalog

Chapter 14. SPI Reference

The Serial Peripheral Interface (SPI) is a synchroneous data link standard named by Motorola which operates in full duplex mode. The SPI gateway models of the AnaGate series provide access to a SPI bus via standard networking.

With the SPI API these SPI gateways can be controlled easily. The programming interface is identical for all devices and generally uses the network protocol TCP.

The following devices can be addressed via the SPI API interface:

- AnaGate SPI
- AnaGate Universal Programmer

Note

All SPI specific functionality of the AnaGate C-API is also available für Lua users. The Lua function extensions are documented below.

LS_SPIOpenDevice

LS_SPIOpenDevice — Opens a network connection to an AnaGate SPI device.

Syntax

```
int RC, int Handle = LS_SPIOpenDevice(string IPAddress, int Timeout);
```

Parameter

IPAddress — Network address of the AnaGate partner.

Timeout — Default timeout for accessing the AnaGate in milliseconds.

A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Handle — Access handle if successfully connected to the AnaGate device.

Description

Opens a TCP/IP connection to an AnaGate SPI (resp. AnaGate Universal Programmer). After the connection is established, access to the SPI bus is possible.

Note

The AnaGate SPI (resp. the SPI interface of an AnaGate Universal Programmer) does not allow more than one concurrent network connection. As long as a connection is active, all new connections are refused.

See the following example for the initial programming steps.

```
local nRC, nHandle = LS_SPIOpenDevice("192.168.1.254", 5000)
if nRC ~= 0 then
   print(LS_SPIErrorMessage(nRC))
   os.exit()
end

-- now do something

LS_SPICloseDevice(nHandle)
```

See also

LS_SPICloseDevice

LS_SPICloseDevice

LS_SPICloseDevice — Closes an open network connection to an AnaGate SPI device.

Syntax

```
int RC = LS_SPICloseDevice(int Handle);
```

Parameter

Handle Valid access handle.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Closes an open network connection to an AnaGate SPI device. The *hHandle* parameter is a return value of a successful call to the function `LS_SPIOpenDevice`.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See also

LS_SPIOpenDevice

LS_SPISetGlobals

LS_SPISetGlobals — Sets the global settings which are to be used on the AnaGate SPI.

Syntax

```
int RC = LS_SPISetGlobals(int Handle, uint32 Baudrate, uint8 SigLevel,
uint8 AuxVoltage, uint8 ClockMode);
```

Parameter

Handle — Valid access handle.

Baudrate — The baud rate to be used. The values can be set individually, e.g.

- 500,000 for 500 kBit
- 1,000,000 for 1 MBit
- 5,000,000 for 5 MBit

Note

The required baud rate can be different from the value actually used because of internal hardware restrictions (frequency of the oscillator). If it is not possible to adjust the baud rate exactly to the parsed value, the nearest smaller possible value is used instead.

SigLevel — The voltage level for SPI signals to be used. The following values are allowed:

- 0 = Outputs in High Impedance Modus (Standard mode).
- 1 = +5.0 V for the signals.
- 2 = +3.3 V for the signals.
- 3 = +2.5 V for the signals.

AuxVoltage — The voltage level of the support voltage to be used. The following values are allowed:

- 0 = support voltage is +3.3 V.
- 1 = support voltage is 2.5 V.

ClockMode — The phase and polarity of the clock signal. The following values are allowed:

- 0 = CPHA=0 and CPOL=0.
- 1 = CPHA=0 and CPOL=1.

- 2 = CPHA=1 and CPOL=0.
- 3 = CPHA=1 and CPOL=1.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Sets the global settings of SPI interface of the AnaGate SPI or the AnaGate Universal Programmer. These settings are not saved permanently on the device and are reset after every device restart.

See also

LS_SPIGetGlobals

LS_SPIGetGlobals

LS_SPIGetGlobals — Returns the currently used global settings of the AnaGate SPI.

Syntax

```
int RC, uint32 Baudrate, uint8 SigLevel, uint8 AuxVoltage, uint8
ClockMode = LS_SPIGetGlobals(int Handle);
```

Parameter

Handle Valid access handle.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Baudrate — The baud rate currently used on the SPI bus in bits per second.

SigLevel — The voltage level currently used by the AnaGate SPI. The following values are possible:

- 0 = Outputs in High Impedance Modus (Standard mode).
- 1 = +5.0 V for the signals.
- 2 = +3.3 V for the signals.
- 3 = +2.5 V for the signals.

AuxVoltage — The voltage level of the support voltage currently used by the AnaGate SPI. The following values are possible:

- 0 = support voltage is +3.3 V.
- 1 = support voltage is 2.5 V.

ClockMode — The phase and polarity of the clock signal currently used by the AnaGate SPI. The following values are possible:

- 0 = CPHA=0 and CPOL=0.
- 1 = CPHA=0 and CPOL=1.
- 2 = CPHA=1 and CPOL=0.
- 3 = CPHA=1 and CPOL=1.

Description

Returns the currently used global settings of the SPI interface of the AnaGate SPI or the AnaGate Universal Programmer.

See also

LS_SPISetGlobals

LS_SPIDataReq

LS_SPIDataReq — Writes and reads data to/from the SPI bus.

Syntax

```
int RC, table(uint8[ReadLen]) ReadData = LS_SPIDataReq(int Handle,
table(uint8[WriteLen]) WriteData, uint16 WriteLen, uint16 ReadLen);
```

Parameter

Handle	Valid access handle.
WriteData	Buffer with the data that is to be sent to the SPI partner.
WriteLen	Length of the data buffer *WriteData* (byte count).
ReadLen	Number of bytes to read.

Return value

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
ReadData	Table with data received from the SPI partner.

Description

Sends data to the SPI bus and receives data from the SPI bus.

On the SPI bus data is transferred on two seperate data lines full duplex (SDO and SDI). The `LS_SPIDataReq` has to split a single data transfer into two steps because of the spacial separation to the SPI bus. First the write data buffer is put into a TCP data telegram and sent to the AnaGate SPI. The AnaGate SPI performs the real data transfer on the SPI bus and sends a confirmation back including the data received from the bus.

Important

It is impossible to detect that no device is present at the SPI bus. So, if no device is attached the requested amount of bytes is returned anyway - in this case the read buffer is filled with `0`.

See the following example for sending a command to the connected SPI bus.

```
local tabWrite = {}
for i = 1, 10, 1 do
   table.insert(tabWrite, i)
end

local nRC, hHandle = SPIOpenDevice("192.168.1.254", 5000)
if nRC == 0 then
   local tabRead
```



```
   // send 1 byte and receive 1 byte
   nRC, tabRead = LS_SPIDataReq(hHandle, tabWrite, 1, 1)
   // send 1 byte and receive 5 byte
   nRC, tabRead = LS_SPIDataReq(hHandle, tabWrite, 1, 5)
   // send 2 byte and receive 1 byte
   nRC, tabRead = LS_SPIDataReq(hHandle, tabWrite, 2, 1)

   LS_SPICloseDevice(hHandle)
end
```


LS_SPIErrorMessage

LS_SPIErrorMessage — Returns a description of the given error code as a text string.

Syntax

```
string ErrorMsg = LS_SPIErrorMessage(int ErrorCode);
```

Parameter

ErrorCode Error code for which the error description is to be determined.

Return value

ErrorMsg Textual description of the error code.

Description

Returns a textual description of the parsed error code (see Appendix A, *API return codes*).

See the following example in Lua scripting language.

```
local nRC = 0
local sErrorText = 'No Error'

//... call an API function here

sErrorText = LS_SPIErrorMessage(nRC)
print(sErrorText)
```


LS_SPIReadDigital

LS_SPIReadDigital — Reads the current values of digital input and output registers of the AnaGate device.

Syntax

`int RC, uint32 InputBits, uint32 OutputBits = **LS_SPIReadDigital**(int *Handle*);`

Parameter

Handle Valid access handle.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

InputBits Current value of the digital input register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.

OutputBits Current value of the digital output register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The current values of the digital inputs and outputs can be retrieved with the `LS_SPIReadDigital` function.

See the following example for setting and reading the digital IOs.

```
local nOutputs = 0x03

local nRC, hHandle = LS_SPIOpenDevice(400000, "192.168.1.254", 5000)
if nRC == 0 then
   // set the digital output register (PIN 0 and PIN 1 to HIGH value)
   nRC = LS_SPIWriteDigital(hHandle, nOutputs)
   local nInputs
   // read all input and output registers
   nRC, nInputs, nOutputs = LS_SPIReadDigital(hHandle)

   LS_SPICloseDevice(hHandle)
end
```

See also

LS_SPIWriteDigital

LS_SPIWriteDigital

LS_SPIWriteDigital — Writes a new value to the digital output register of the AnaGate device.

Syntax

```
int RC = LS_SPIWriteDigital(int Handle, uint32 OutputBits);
```

Parameter

Handle	Valid access handle.
OutputBits	New register value. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The digital outputs can be written with the `LS_SPIWriteDigital` function.

A simple example for reading/writing of the IOs can be found at the description of `LS_SPIReadDigital`.

See also

LS_SPIReadDigital

Chapter 15. I2C Reference

Philips Semiconductors (now NXP Semiconductors) has developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter-IC or I2C-bus. Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL). Serial, 8-bit oriented, bidirectional data transfers can be performed at up to 100 kbit/s in Standard mode, up to 400 kbit/s in Fast mode, up to 1 Mbit/s in Fast mode Plus (Fm+) or up to 3.4 Mbit/s in High speed mode. [NXP-I2C].

The I2C gateway models of the AnaGate series provides access to a I2C bus via a standard networking. With the I2C API these I2C gateways can be easily controlled. The programming interface is identical for all devices and used the network protocol TCP in general.

Following devices can be addressed via the I2C API interface:

- AnaGate I2C
- AnaGate Universal Programmer

LS_I2COpenDevice

LS_I2COpenDevice — Opens a network connection to an AnaGate I2C or an AnaGate Universal Programmer).

Syntax

```
int RC, int Handle = LS_I2COpenDevice(uint32 Baudrate, string IPAddress,
int Timeout);
```

Parameter

Baudrate — Baud rate to be used for the I2C bus. The value can be set individually, like

- 100000 for 100 kBit (standard mode)
- 400000 for 400 kBit (fast mode)

Note

Values above 400 kBit are ignored by the AnaGate SPI.

IPAddress — Network address of the AnaGate partner.

Timeout — Default timeout for accessing the AnaGate in milliseconds.

A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Handle — Access handle if successfully connected to the AnaGate device.

Description

Opens a TCP/IP connection to an AnaGate I2C (resp. AnaGate Universal Programmer). After the connection is established, access to the I2C bus is possible.

Note

The AnaGate I2C (resp. the I2C interface of an AnaGate Universal Programmer) does not allow more than one concurrent network connection. During an established network connection all new connections are refused.

See the following example for the initial programming steps.

```
local nRC, nHandle = LS_I2COpenDevice(1000000, "192.168.1.254", 5000)
if nRC ~= 0 then
   print(LS_I2CErrorMessage(nRC))
   os.exit()
end

-- now do something

LS_I2CCloseDevice(nHandle)
```

See also

LS_I2CCloseDevice

LS_I2COpenDeviceEx

LS_I2COpenDeviceEx — Opens a network connection to an AnaGate I2C or an AnaGate Universal Programmer). In addition to the LS_I2COpenDevice function the port must be specified.

Syntax

```
int RC, int Handle = LS_I2COpenDeviceEx(uint32 Baudrate, uint8 DevicePort, string IPAddress, int Timeout);
```

Parameter

Baudrate — Baud rate to be used for the I2C bus. The value can be set individually, like

- 100000 for 100 kBit (standard mode)
- 400000 for 400 kBit (fast mode)

Note

Values above 400 kBit are ignored by the AnaGate SPI.

DevicePort — Number of the I2C bus (begins with 0)

IPAddress — Network address of the AnaGate partner.

Timeout — Default timeout for accessing the AnaGate in milliseconds.

A timeout is reported if the AnaGate partner does not respond within the defined timeout period. This global timeout value is valid on the current network connection for all commands and functions which do not offer a specific timeout value.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Handle — Access handle if successfully connected to the AnaGate device.

Description

Opens a TCP/IP connection to an AnaGate I2C (resp. AnaGate Universal Programmer). After the connection is established, access to the I2C bus is possible.

Note

The AnaGate I2C (resp. the I2C interface of an AnaGate Universal Programmer) does not allow more than one concurrent network connection. During an established network connection all new connections are refused.

See the following example for the initial programming steps.

```
local nRC, nHandle = LS_I2COpenDeviceEx(1000000, 0, "192.168.1.254", 5000)
if nRC ~= 0 then
   print(LS_I2CErrorMessage(nRC))
   os.exit()
end

-- now do something

LS_I2CCloseDevice(nHandle)
```

See also

LS_I2CCloseDevice

LS_I2CCloseDevice

LS_I2CCloseDevice — Closes an open network connection to an AnaGate I2C device.

Syntax

```
int RC = LS_I2CCloseDevice(int Handle);
```

Parameter

Handle Valid access handle.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Closes an open network connection to an AnaGate I2C device. The *`Handle`* parameter is a return value of a successful call to the function `LS_I2COpenDevice`.

Important

It is recommended to close the connection because all internally allocated system resources are freed again and the connected AnaGate device is signalled that the active connection is not longer in use and can be used again for new connect requests.

See also

LS_I2COpenDevice

LS_I2CReset

LS_I2CReset — Resets the I2C Controller in an AnaGate I2C device.

Syntax

```
int RC = LS_I2CReset(int Handle);
```

Parameter

Handle Valid access handle.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

Resets the I2C Controller in an AnaGate I2C device.

LS_I2CRead

LS_I2CRead — Reads data from an I2C partner.

Syntax

```
int RC, table(uint8[BufferLen]) Data = LS_I2CRead(int Handle, uint16 SlaveAddress, uint16 BufferLen);
```

Parameter

Handle	Valid access handle.
SlaveAddress	Slave address of the I2C partner. The slave address can represent a so-called 7-bit or 10-bit address.(see Appendix B, *I2C slave address formats*).
BufferLen	Number of bytes to read.

Return values

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
Data	Byte buffer in which the data received from the I2C partner is stored.

Description

Reads data from an I2C partner. The user must ensure that the address of the I2C partner is set up correctly.

The R/W bit of the slave address does not have to be explicitly set by the user.

See also

LS_I2CWrite

LS_I2CWrite

LS_I2CWrite — Writes data to an I2C partner.

Syntax

```
int RC, uint16 ErrorByte = LS_I2CWrite(int Handle, uint16 nSlaveAddress,
table(uint8[BufferLen]) Buffer, uint16 BufferLen);
```

Parameter

Handle	Valid access handle.
SlaveAddress	Slave address of the I2C partner. The slave address can represent a so-called 7-bit or 10-bit address. (see Appendix B, *I2C slave address formats*).
Buffer	Byte buffer with the data that is to be sent to the I2C partner.
BufferLen	Size of bytes to be read.

Return value

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
ErrorByte	Number of byte in data buffer which raised the error if the function failed.

Description

Writes data to an I2C partner. The user must ensure that the data buffer and the address of the I2C partner are set up correctly.

The R/W bit of the slave address does not have to be explicitly set by the user.

See also

LS_I2CRead

LS_I2CSequence

LS_I2CSequence — This command is used to write a sequence of write and read commands to an I2C device.

Syntax

```
int RC, uint16 NumberOfBytesRead, uint16 ByteNumberLastError,
table(uint8[NumberOfBytesToRead]) ReadBuffer = LS_I2CSequence(int
Handle, table(uint8[NumberOfBytesToWrite]) WriteBuffer, uint16
NumberOfBytesToWrite, uint16 NumberOfBytesToRead);
```

Parameter

Handle — Valid access handle.

WriteBuffer — Byte buffer, containing the commands which are to be sent to the AnaGate I2C. The single commands are stored sequentially in this byte buffer.

A read command is defined as follows:

Structure of read command for LS_I2CSequence

Read command	Description
2 bytes (LSB,MSB)	Slave address in 7 or 10 bit format. The R/W bit must be set explicitly to 1.
2 bytes (LSB,MSB)	Bit 0-14: Number of data bytes to be read from the I2C device. The successfully read data bytes are stored in the *ReadBuffer* buffer. Bit 15: If this bit is set then the stop bit at the end of the read command is omitted.

A write command is defined as follows:

Structure write command for LS_I2CSequence

Write command	Description
2 bytes (LSB,MSB)	slave address in 7 or 10 bit format, the R/W bit must be set to explicitly to 1.
2 bytes (LSB,MSB)	Bit 0-14: Number of data bytes to be written to the I2C device. Bit 15: If this bit is set then the stop bit at the end of the write command is omitted.

Write command	Description
N bytes	data bytes.

NumberOfBytesToWrite byte size of the data to write

NumberOfBytesToRead Expected combined byte size of the read data.

Return values

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

NumberOfBytesRead Number of bytes that were actually read from the I2C partner.

ByteNumberLastError Position of the byte in the *`WriteBuffer`* that caused an error.

ReadBuffer Byte buffer which contains the data received from the I2C partner. The received data from separate commands is stored in the buffer sequentially (first the data of the first command, then the data of the second command, ...).

Description

The user must ensure that the data buffer and the address of the I2C partner are set up correctly.

LS_I2CReadDigital

LS_I2CReadDigital — Reads the current values of digital input and output registers of the AnaGate device.

Syntax

`int RC, uint32 InputBits, uint32 OutputBits = **LS_I2CReadDigital**(int *Handle*);`

Parameter

Handle Valid access handle.

Return value

RC	Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).
InputBits	Current value of the digital input register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.
OutputBits	Current value of the digital output register. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use and are set to 0.

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The current values of the digital inputs and outputs can be retrieved with the `LS_I2CReadDigital` function.

See the following example for setting and reading the digital IOs.

```
local nOutputs = 0x03

local nRC, hHandle = LS_I2COpenDevice(400000, "192.168.1.254", 5000)
if nRC == 0 then
   // set the digital output register (PIN 0 and PIN 1 to HIGH value)
   nRC = LS_I2CWriteDigital(hHandle, nOutputs)
   local nInputs
   // read all input and output registers
   nRC, nInputs, nOutputs = LS_I2CReadDigital(hHandle)

   LS_I2CCloseDevice(hHandle)
end
```

See also

LS_I2CWriteDigital

LS_I2CWriteDigital

LS_I2CWriteDigital — Writes a new value to the digital output register of the AnaGate device.

Syntax

```
int RC = LS_I2CWriteDigital(int Handle, uint32 OutputBits);
```

Parameter

Handle	Valid access handle.
OutputBits	New register value. Currently only bits 0 to 3 are used; the remaining bits are reserved for future use.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

All models of the AnaGate series (except the model AnaGate CAN uno in DIN rail case) have connectors for 4 digital inputs and 4 digital outputs at the rear panel. The AnaGate CAN uno in DIN rail case has connectors for 2 digital inputs and 2 digital outputs at the top side instead.

The digital outputs can be written with the `LS_I2CWriteDigital` function.

A simple example for reading/writing of the IOs can be found at the description of `LS_I2CReadDigital`.

See also

LS_I2CReadDigital

LS_I2CErrorMessage

LS_I2CErrorMessage — Returns a description of the given error code as a text string.

Syntax

```
string ErrorMsg = LS_I2CErrorMessage(int ErrorCode);
```

Parameter

ErrorCode Error code for which the error description is to be determined.

Return value

ErrorMsg Textual description of the error code.

Description

Returns a textual description of the parsed error code (see Appendix A, *API return codes*).

See the following example in Lua scripting language.

```
local nRC = 0
local sErrorText = 'No Error'

//... call a API function here

sErrorText = LS_I2CErrorMessage(nRC)
print(sErrorText)
```


LS_I2CReadEEProm

LS_I2CReadEEProm — Reads data from an EEPROM on the I2C bus.

Syntax

```
int RC, table(uint8[DataLen]) Data = LS_I2CReadEEProm(int Handle, uint16 SubAddress, uint32 Offset, uint16 OffsetFormat, uint16 DataLen);
```

Parameter

Handle — Valid access handle.

SubAddress — Sub address of the EEPROM to communicate with. The valid values for *SubAddress* are governed by the setting used in the parameter *OffsetFormat* (bits 8-10). If the EEPROM type needs to use bits of the *Chip Enable Address* to address the internal memory, only the remaining bits can be used to select the device itself.

- No bit is used for addressing: 0 to 7
- 1 bit is used for addressing: 0 to 3
- 2 bits are used for addressing: 0 to 1
- 3 bits are used for addressing: 0

Offset — Data offset on the EEPROM from which the transferred data is to be read.

OffsetFormat — Defines how a memory address of EEPROM has to be specified when accessing the device.

Bits 0-7 indicate the number of bits which are used in the address byte (word) of the I2C command for addressing the device memory.

Bits 8-10 indicate how much and which bits of the *Chip Enable Bits* are used for addressing the device memory (see Table C.1, " Usage of the Chip Enable Bits of I2C EEPROMs " for allowed values).

Note

The maximum addressable size of an EEPROM is derived from the sum of all the bits. For example a M24C08 uses 8 bits of the address byte and an extra bit in the slave address. The total 9 bits can address up to 512 bytes.

DataLen — Length of the data buffer.

Return values

RC — Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Data Table with data read from EEPROM.

Description

The `LS_I2CReadEEProm` function reads data from an I2C EEPROM.

Of course all access to the memory of an EEPROM is done by standard I2C read or write commands. So when reading from the memory only the matching slave address, the memory offset address and the data have to be sent to the I2C bus.

`LS_I2CReadEEProm` translates the given memory address on the chip by means of the sub address and the addressing mode of the present EEPROM type. The slave address of the EEPROM is determined automatically and not mandatory for the function call.

A programming example which clears a **ST24C1024** can be found at the description of LS_I2WriteEEPROM.

See also

LS_I2CWriteEEProm

Appendix C, *Programming I2C EEPROM*

LS_I2CWriteEEProm

LS_I2CWriteEEProm — Writes data to an I2C EEPROM.

Syntax

```
int RC = LS_I2CWriteEEProm(int Handle, uint16 SubAddress, uint32 Offset,
table(uint8[DataLen]) Data, uint16 DataLen, uint16 nOffsetFormat);
```

Parameter

Handle	Valid access handle.
SubAddress	Sub address of the EEPROM to communicate with. The valid values for *SubAddress* are governed by the setting used in the parameter *OffsetFormat* (bits 8-10). If the EEPROM type needs to use bits of the *Chip Enable Address* to address the internal memory, only the remaining bits can be used to select the device itself. • No bit is used for addressing: 0 to 7 • 1 bit is used for addressing: 0 to 3 • 2 bits are used for addressing: 0 to 1 • 3 bits are used for addressing: 0
Offset	Data offset on the EEPROM to which the transferred data is to be written.
Data	byte table with the data that is to be written.
DataLen	Length of the data buffer.
OffsetFormat	Defines how a memory address of EEPROM has to be specified when accessing the device. Bits 0-7 indicate the number of bits which are used in the address byte (word) of the I2C command for addressing the device memory. Bits 8-10 indicate how much and which bits of the *Chip Enable Bits* are used for addressing the device memory (see Table C.1, " Usage of the Chip Enable Bits of I2C EEPROMs " for allowed values).

Note

The maximum addressable size of an EEPROM is derived from the sum of all the bits. For example a M24C08 uses 8 bits of the address byte and an extra bit in the slave address. The total 9 bits can address up to 512 bytes.

Return value

RC Returns `0` if successful or an error value otherwise (Appendix A, *API return codes*).

Description

The `LS_I2CWriteEEProm` function writes data to an I2C EEPROM.

Of course all access to the memory of an EEPROM is done by standard I2C read or write commands. So when writing to the memory only the matching slave address, the memory offset address and the data have to be sent to the I2C bus.

`LS_I2CWriteEEProm` translates the given memory address on the chip by means of the sub address and the addressing mode of the present EEPROM type. The slave address of the EEPROM is determined automatically and not mandatory for the function call.

Tip

It is important to note that an EEPROM is divided into memory pages and that a single write command can only program data within a page. Users of `LS_I2CWriteEEProm` must ensure to do not write across page limits. The page size depends on the EEEPOM type.

See the following example for writing data to a **ST24C1024**.

```
local tabData = {}
for i = 1, 256, 1 do
   table.insert(tabData, 0x0)
end

local nSubAddress = 0 1
local nOffsetFormat = 0x10+0x0F 2

local RC, hHandle = LS_I2COpenDevice(400000, "192.168.1.254", 5000)
if RC == 0 then
   for page = 0, 512-1, 1 do
      RC = LS_I2CWriteEEProm(hHandle, nSubAddress, i*256, tabData, #tabData, nOffSetFormat) 3
   end
   LS_I2CCloseDevice(hHandle)
end
```

1. It is possible to address 4 individual ST24C1024 on a single I2C bus. By selection of sub address `0` the control pins E2 and E1 have to be LOW.
2. 17 address bits are used to address the 128 kB of a ST24C1024. 16 bits are set via the address bytes of the write command: `16=0x0F`. The address bit A16 is set via the E0 bit of the *Chip Enable Address*, therefore addressing mode `1` (E2-E1-A0) must be set: `0x10`.
3. The page size of a ST24C1024 is 256 byte, each page is programmed full within the for-loop.

See also

LS_I2CReadEEProm

Appendix C, *Programming I2C EEPROM*

Chapter 16. Lua programming examples

16.1. Examples for devices with CAN interface

```
-- Filter: alle CAN-Identifier akzeptieren
local aFilter = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                  0x00, 0x1FFFFFFF, 0x00, 0x1FFFFFFF,
                  0x00, 0x1FFFFFFF, 0x00, 0x1FFFFFFF }

local aSendData = { 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xF8 }
--*********************************************
function main()
  local nRC, nHandle, nHandle2, oTime
  -- Open
  nRC, nHandle = LS_CANOpenDevice(false, true, 0, "10.1.2.160", 5000)
  if nRC ~= 0 then
    print(LS_CANErrorMessage(nRC))
    os.exit()
  end

  nRC, nHandle2 = LS_CANOpenDevice(false, true, 0, "10.1.2.161", 5000)
  if nRC ~= 0 then
    print(LS_CANErrorMessage(nRC))
    os.exit()
  end

  -- Filter setzen
  nRC = LS_CANSetFilter(nHandle,  aFilter)
  nRC = LS_CANSetFilter(nHandle2, aFilter)

  -- aktuelle Zeit auf den AnaGate Device setzen
  nRC, oTime = LS_GetTime()
  nRC = LS_CANSetTime(nHandle,  oTime[1], oTime[2])
  nRC = LS_CANSetTime(nHandle2, oTime[1], oTime[2])

  -- Globals setzen
  nRC = LS_CANSetGlobals(nHandle,  500000, 0, true, false, false)
  nRC = LS_CANSetGlobals(nHandle2, 500000, 0, true, false, false)

  -- Endlosschleife
  repeat
    -- 1 Datenpakete auf dem 1. AnaGate CAN Device versenden
    nRC = LS_CANWrite(nHandle, 1, aSendData, #aSendData, 0)

    LS_Sleep(20) -- 20Millisekunden warten

    -- Datenpaket auf 2. AnaGate CAN Device empfangen
    local nAvail, ID, Len, Data, Flags, Sec, Microsec = LS_CANGetMessage(nHandle2, 10)
    while nAvail>=0 do
      nAvail, ID, Len, Data, Flags, Sec, Microsec = LS_CANGetMessage(nHandle2, 10)
    end
  until false

  -- Verbindungen beenden
  LS_CANCloseDevice(nHandle)
  LS_CANCloseDevice(nHandle2)
end
```

Example 16.1.

16.2. Examples for devices with SPI interface

```
--************************************************
local function printf(...)
  io.write(string.format(...))
  io.flush()
end
--************************************************
function main()
  -- Verbindung zu AnaGate SPI-Device herstellen
  local nRC, nHandle = LS_SPIOpenDevice("10.1.2.162", 5000)
  if nRC ~= 0 then
    local errortext = LS_SPIErrorMessage(nRC)
    print(errortext)
    os.exit()
  end

  -- Setzen der globalen Einstellungen
  nRC = LS_SPISetGlobals(nHandle, 100000, 2, 0, 0)

  -- OP-Codes des SPI-Partners mit Daten
  local OPWriteEnab = {0x06}
  local OPStatusReg = {0x05, 0x00}
  local OPRead      = {0x03, 0x00, 0x00, 0x00}
  local Value
  -- WriteEnable-Flag des SPI-Partners setzen
  nRC, Value = LS_SPIDataReq(nHandle, OPWriteEnab, #OPWriteEnab, 1)

  -- Statusregister des SPI-Partners abfragen
  nRC, Value = LS_SPIDataReq(nHandle, OPStatusReg, #OPStatusReg, 2)
  for i, v in ipairs(Value) do
    printf("Data Status: %02X\n", v)
  end

  -- Lesen von 20 Bytes ab Adresse 0x00
  nRC, Value = LS_SPIDataReq(nHandle, OPRead, #OPRead, 20)
  for i, v in ipairs(Value) do
    printf("Data Status: %02X\n", v)
  end

  -- Alle digitalen Ausgaenge zuruecksetzen
  LS_SPIWriteDigital(nHandle, 0)

  -- Verbindung zu AnaGate SPI-Device beenden
  LS_SPICloseDevice(nHandle)
end
```

Example 16.2.

16.3. Examples for devices with I2C interface

-
-

```
--************************************************
```



```
local function printf(...)
  io.write(string.format(...))
  io.flush()
end
--*************************************************
function main()
  local aSendData = {}
  for i = 1, 128, 1 do
    table.insert(aSendData, i-1)
  end

  local nRC, nHandle = LS_I2COpenDevice(1000000, "10.1.2.162", 5000)
  if nRC ~= 0 then
    print(LS_I2CErrorMessage(nRC))
    os.exit()
  end

  --Read EEPROM
  local CountBytes = 1024
  for Address = 0, CountBytes*64, CountBytes do
    local Value
    nRC, Value = LS_I2CReadEEProm(nHandle, 1, Address, 16, CountBytes)
    for i, v in ipairs(Value) do
      printf("%02X ", v)
      if i % 16 == 0 then
        printf("\n")
      end
    end
  end

  --Write EEPROM
  CountBytes = #aSendData
  for Address = 0, CountBytes * 10, CountBytes do
    nRC = LS_I2CWriteEEProm(nHandle, 1, Address, aSendData, CountBytes, 16)
  end

  LS_I2CWriteDigital(nHandle, 0)
  LS_I2CCloseDevice(nHandle)
end
```

Example 16.3.

-
-

```
--*************************************************
local function printf(...)
  io.write(string.format(...))
  io.flush()
end
--*************************************************
function main()
  local aSendData = {}
  for i = 1, 128, 1 do
    table.insert(aSendData, i-1)
  end

  local nRC, nHandle = LS_I2COpenDevice(1000000, "10.1.2.162", 5000)
```



```
  if nRC ~= 0 then
    print(LS_I2CErrorMessage(nRC))
    os.exit()
  end

  --Write
  local aData = {0x00, 0x05} -- ab Adresse 5 lesen
  local Value
  nRC, Value = LS_I2CWrite(nHandle, 0xa2, aData, #aData)

  --Read
  nRC, Value = LS_I2CRead(nHandle, 0xa2, 1024)
  for i, v in ipairs(Value) do
    printf("%02X ", v)
    if i % 16 == 0 then
      printf("\n")
    end
  end
  printf("\n")

  LS_I2CWriteDigital(nHandle, 0)
  LS_I2CCloseDevice(nHandle)
  end
```

Example 16.4.

•

```
--************************************************
local function printf(...)
  io.write(string.format(...))
  io.flush()
end
--************************************************
function main()
  local aSendData = {}
  for i = 1, 128, 1 do
    table.insert(aSendData, i-1)
  end

  local nRC, nHandle = LS_I2COpenDevice(1000000, "10.1.2.162", 5000)
  if nRC ~= 0 then
    print(LS_I2CErrorMessage(nRC))
    os.exit()
  end

  --Sequence
  local aData = {0xa2, 0x00,  --SLA
                 0x02, 0x00,  --Laenge Schreibkommando
                 0x00, 0x00,  --Daten Schreibkommando
                 0xa3, 0x00,  --SLA 1. Lesekommando
                 0x30, 0x00,  --Laenge 1. Lesekommando
                 0xa3, 0x00,  --SLA 2. Lesekommando
                 0x20, 0x00}  --Laenge 2. Lesekommando
  local CountRead, LastError, Value
  nRC, CountRead, LastError, Value = LS_I2CSequence(nHandle, aData, #aData, 0x0050)
  printf("CountRead:%02X  LastError:%02X\n", CountRead, LastError)
  for i, v in ipairs(Value) do
    printf("%02X ", v)
  end
  printf("\n")
```



```
  LS_I2CWriteDigital(nHandle, 0)
  LS_I2CCloseDevice(nHandle)
end
```

Example 16.5.

Appendix A. API return codes

The following is a list of the return values of the API functions. These values are defined in the header file `AnaGateErrors.h`.

Value	Name	Description
0	ERR_NONE	No errors.
0x000001	ERR_OPEN_MAX_CONN	Open failed, maximum number of connections reached.
0x0000FF	ERR_OP_CMD_FAILED	Command failed with unknown failure.
0x020000	ERR_TCPIP_SOCKET	Socket error occurred in TCP/IP layer.
0x030000	ERR_TCPIP_NOTCONNECTED	Connection to TCP/IP partner can't be established or is disconnected.
0x040000	ERR_TCPIP_TIMEOUT	No answer was received from TCP/IP partner within the defined timeout.
0x050000	ERR_TCPIP_CALLNOTALLOWED	Command is not allowed at this time.
0x060000	ERR_TCPIP_NOT_INITIALIZED	TCP/IP-Stack can't be initialized.
0x0A0000	ERR_INVALID_CRC	AnaGate TCP/IP telegram has incorrect checksum (CRC).
0x0B0000	ERR_INVALID_CONF	AnaGate TCP/IP telegram wasn't confirmed by partner.
0x0C0000	ERR_INVALID_CONF_DATA	AnaGate TCP/IP telegram confirmation is invalid.
0x900000	ERR_INVALID_DEVICE_HANDLE	Invalid device handle.
0x910000	ERR_INVALID_DEVICE_TYPE	Function can't be executed on this device handle as it is assigned to a different device type of AnaGate series.

Table A.1. Common return values for all devices of AnaGate series

Value	Name	Description
0x000120	ERR_I2C_NACK	I2C NACK
0x000121	ERR_I2C_TIMEOUT	I2C Timeout
0x000125	ERR_I2C_POLL_TIMEOUT	Acknowledge polling timeout after write access

Table A.2. Return values for AnaGate I2C

A textual description of the return value can be retrieved with the function `I2CErrorMessage()`.

Value	Name	Description
0x000220	ERR_CAN_NACK	CAN-NACK
0x000221	ERR_CAN_TX_ERROR	CAN Transmit Error
0x000222	ERR_CAN_TX_BUF_OVERLOW	CAN buffer overflow
0x000223	ERR_CAN_TX_MLOA	CAN Lost Arbitration
0x000224	ERR_CAN_NO_VALID_BAUDRATE	CAN Setting no valid Baudrate

Table A.3. Return values for AnaGate CAN

A textual description of the return value can be retrieved with the function `CANErrorMessage()`.

Value	Name	Description
0x000320	ERR_SPI_INCONSISTENT_TELEGRAM	SPI inconsistent telegram data
0x000321	ERR_SPI_SEND_ERROR	SPI error while sending
0x000330	ERR_SPI_SEQ_UNKNOWN_COMMAND	SPI sequence unknown command op code
0x000331	ERR_SPI_SEQ_TIMEOUT	SPI sequence timeout

Table A.4. Return values for AnaGate SPI

A textual description of the return value can be retrieved with the function `SPIErrorMessage()`.

Value	Name	Description
0x000920	ERR_RENESAS_TIMEOUT	Renesas timeout
0x000921	ERR_RENESAS_INVALID_ID	Renesas Invalid ID
0x000922	ERR_RENESAS_FLASH_ERASE_FAILED	Renesas failed erase the flash
0x000923	ERR_RENESAS_PAGE_PROG_FAILED	Renesas failed prog the page

Table A.5. Return values for AnaGate Renesas

A textual description of the return value can be retrieved with the function `RenesasErrorMessage()`.

Value	Name	Description
-1	ERR_SYNTAX	Syntax error
-2	ERR_RANGE	Value out of valid range.
-3	ERR_NOT_A_NUMBER	Parameter is not of type *number*.
-4	ERR_NOT_A_STRING	Parameter is not of type *string*.
-5	ERR_NOT_A_BOOL	Parameter is not of type *boolean*.
-6	ERR_NOT_A_TABLE	Parameter is not of type *table*.
-10	ERR_NO_DATA	No data available.

Table A.6. Return values for Lua scripting

Appendix B. I2C slave address formats

A standard I2C address is the first byte sent by the I2C master, whereas only the first seven bit form the address; the last bit (R/W bit) defines the direction in which the data is sent. I2C has a 7-bit address space and can address 112 slaves on a single bus (16 of the 128 addresses are reserved for special purposes).

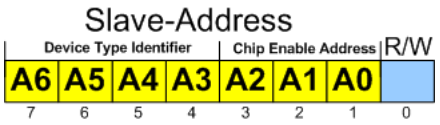


Figure B.1. Definition of an I2C slave address in 7 bit format

Each I2C capable IC has a fixed bus address. The 4 upper bits of the bus address are called *Device Type Identifier* and define the chip type. The lower three bits, called sub address or *Chip Enable Address*, are usually defined by the corresponding wired control pins. So in total up to 8 similar ICs can be used on a single I2C bus.

Due to a lack of address space a 10 bit addressing mode was introduced later. This new mode is downwards compatible to the 7 bit standard mode by using 4 of the 16 reserved addresses. Both addressing modes can be used simultaneously, which allows the usage of 1136 slaves on a single bus.

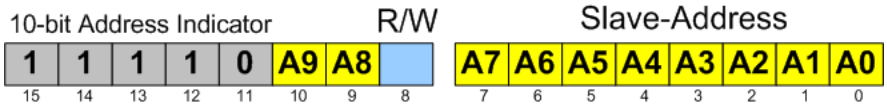


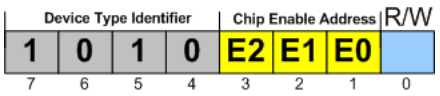
Figure B.2. Definition of a I2C slave address in 10 bit format

Note

Devices of type *AnaGate I2C* and *AnaGate Universal Programmer* generally support both addressing modes. The API functions `I2CRead` and `I2CWrite` address the slaves via a two byte parameter.

Addressing of serial EEPROM

The device type identifier of a serial EEPROM is defined as `0xA`. This results in the following schematic structure of an address (the chip enable bits are often named E0, E1 and E2 in literature):



	Device Type Identifier				Chip Enable [1] [2]			R/W	EEPROM Memory
	b7	b6	b5	b4	b3	b2	b1	b0	
M24C01	1	0	1	0	E2	E1	E0	R/W	128 byte
M24C02	1	0	1	0	E2	E1	E0	R/W	256 byte

	Device Type Identifier				Chip Enable [1] [2]			R/W	EEPROM Memory
	b7	b6	b5	b4	b3	b2	b1	b0	
M24C04	1	0	1	0	E2	E1	A8	R/W	512 byte
M24C08	1	0	1	0	E2	A9	A8	R/W	1024 byte
M24C16	1	0	1	0	A10	A9	A8	R/W	2048 byte
M24C64	1	0	1	0	E2	E1	E0	R/W	8192 byte

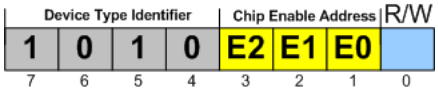
[1] E0, E1 and E2 are compared against the respective external pins on the memory device.
[2] A10, A9 and A8 represent the most significant bits of the address.

Table B.1. I2C EEPROM addressing examples

Appendix C. Programming I2C EEPROM

The *AnaGate I2C* and the *AnaGate Universal Programmer* are very well suited for programming serial I2C EEPROM. To support this special requirement two different API functions are made available: `I2CReadEEProm` and `I2CWriteEEProm`.

Like all other I2C-capable devices EEPROMs are addressable on the I2C bus via a unique slave address (see also Appendix B, *I2C slave address formats*). The so-called *Device Type Identifier* for this type of devices is `0xA`. In principle 8 similar devices can be connected and addressed via the *Chip Enable Bits* E0, E1 und E0.



A data transmission is started with a **Start** signal by the master, followed by the slave address. The slave address is confirmed by the slave with an **ACK**. Depending on the R/W bit data is written (data to slave) or read (data from slave). The last byte of a read access has to be confirmed with a **NAK** by the master to signal the slave the end of the read transmission. The data transmission is always terminated by a **Stop** signal from the master.

When using EEPROMs the memory address is transmitted after transmission of the slave address to advice the slave which memory address is to be written or read. Depending on the used EEPROM type the memory address is sent as a single byte (8 bit) or as two bytes (16 bit, MSB first).

To expand the address space from 8 or 16 bit, some EEPROM types use the *Chip Enable Bits* E0, E1, E2 as additional address bits. Which bits are used in individual cases is defined by the chip producer. In the following all possible combinations of the bits usage are listed:

Mode [1]	**Usage**	**Description**
0x0	E2-E1-E0	Bits are only used to select the chip.
0x1	E2-E1-A0	Bit E0 is used to expand the address space. It is used for address bit A8 (resp. A16).
0x2	E2-A1-A0	E0 and E1 are used to expand the address space. E0 is used for address bit A8 (resp. A16) and E1 is used for A9 (resp. A17).
0x3	A2-A1-A0	E0, E1 and E2 are used to expand the address space. E0 is used for address bit A8 (resp. A16), E1 for A9 (resp. A17) and E2 for A10 (resp. A18).
0x5	A0-E1-E0	Bit E2 is used to expand the address space. It is used for address bit A8 (resp. A16).

Mode [1]	Usage	Description
0x6	A1-A0-E0	E2 and E1 are used to expand the address space. E1 is used for address bit A8 (resp. A16) and E2 is used for A9 (resp. A17).

[1] Set this mode flag in bits 8-10 of parameter *nOffsetFormat* in the API functions `I2CReadEEProm` and `I2CWriteEEProm`.

Table C.1. Usage of the Chip Enable Bits of I2C EEPROMs

Appendix D. FAQ - Frequent asked questions

Here is a list of frequently asked questions concerning installation and usage of the *AnaGate* product.

D.1. Common questions

Q: No network connection (1)

A: Please check the physical connection to the device first. In general the *AnaGate* has to be connected directly to a personal computer or to an active network component (hub, switch). If the AnaGate device is connected to a personal computer a cross-wired network cable must be used to connect the device, otherwise the included network cable is to be used.

The physical interconnection is OK if the yellow link LED turns on when LAN cable is plugged in. The yellow light stays on until the connection breaks down. On some hardware models the link LED flickers synchronously to the green activity LED if there is traffic on the network line.

If the link LED is always off then please check the wiring between the *AnaGate* and the hub, switch or the personal computer.

Q: No network connection (2)

A: If the link LED indicates a proper Ethernet connection (see previous FAQ) but you still can't connect to the *AnaGate* then please try the following:

1. Check if the *AnaGate* can be reached via ping. To do so in Windows, open a command prompt and enter the command **ping a.b.c.d**, where `a.b.c.d` is the device IP address.

2. In case the *AnaGate* is unreachable via ping, reset the device to factory settings. Set the IP address of your PC to `192.168.1.253` and the subnet mask to `255.255.255.0`. Check if the *AnaGate* can be reached via **ping 192.168.1.254**.

3. If the device can be reached via ping then the next step is to try if you can open a TCP connection to port 5001. Open a Windows command prompt and enter **telnet a.b.c.d 5001**, where `a.b.c.d` is the device IP address. If this command fails check if a firewall runs on your PC or if there is a packet filter in the network between your PC and the *AnaGate*.

Q: No network connection after changing the network address

A: After changing the network address of the AnaGate device via web interface the device is not longer reachable. The used internet browser displays only an empty web page, additional error messages are not available.

Please check if your anti-virus software has blocked the new network address. After changing the network address you are redirected to the new network address in the browser. Such activity is suspicious for some anti-virus software so they block the new web page, sometimes even without notification of the user.

Q: Connection problems using multiple devices

A: If multiple devices with identical IP addresses are used in a local area network at the same time the connections to the devices are not stable. Because of this behaviour it is necessary to use different IP addresses.

This problem can also occur if devices with identical IP addresses are used not concurrently but within short intervals. For example this can arise if some new devices which have the default IP address `192.168.1.254` are configured from a single PC.

The **Address Resolution Protocol (ARP)** is used in IPv4 networks to determine the MAC address of a given IP address. The necessary information is cached in the *ARP table*. If there is a wrong entry in the ARP table or even an entry which is not up-to-date it is not possible to communicate with the corresponding host.

An entry in the ARP table is deleted if it is not used any more after a short period time. The time interval used depends on the operating system. On a current Linux distribution an unused entry is discarded after about 5 minutes. The ARP cache can be displayed and manipulated with the **arp** on Windows and Linux.

```
C:\>arp -a

Schnittstelle: 10.1.2.50 --- 0x2
  Internetadresse       Physikal. Adresse     Typ
  192.168.1.254         00-50-c2-3c-b0-df     dynamisch
```

The command **arp -d** can be used to empty the *ARP Cache*.

Note

Possibly the *ARP cache* of the PC has to be deleted if the IP address of a device is changed.

Q: Using a firewall

A: When working with a firewall a TCP port has to be opened for communication with the AnaGate device:

Device	Port number
AnaGate I2C	5000

Device	Port number
AnaGate I2C X7	5100, 5200, 5300, 5400, 5500, 5600, 5700
AnaGate CAN	5001
AnaGate CAN USB	5001
AnaGate CAN uno	5001
AnaGate CAN duo	5001, 5101
AnaGate CAN quattro	5001, 5101, 5201, 5301
AnaGate CAN X1	5001
AnaGate CAN X2, AnaGate CAN-FD X2	5001, 5101
AnaGate CAN X4, AnaGate CAN-FD X4	5001, 5101, 5201, 5301
AnaGate CAN X8	5001, 5101, 5201, 5301, 5401, 5501, 5601, 5701
AnaGate SPI	5002
AnaGate Renesas	5008
AnaGate Universal Programmer UP/UPP	5000, 5002, 3333, 4444, 20, 21
AnaGate Universal Programmer UPR	5000, 5002, 5008, 3333, 4444, 20, 21
AnaGate Universal Programmer UP 2.0	5000, 5002, 3333, 4444, 20, 21

Table D.1. Using AnaGate hardware with firewall

D.2. Questions concerning AnaGate CAN

Q: What is the value of the termination resistor when the termination option of the device is activated?

A: The termination resistor of the *AnaGate* is driven by a FET transistor. The resistor itself has 110 Ohm while the internal resistance of the FET is 10 Ohm if the FET is activated. So the resulting resistance is 120 Ohm, as required by the CAN bus.

Q: Does Analytica offer a CAN gateway which does not have a galvanically isolated CAN interface?

A: Any device that is actively connected to a CAN bus should be galvanically isolated. Especially when using USB-operated devices (like the *AnaGate USB*) it is essential to have a galvanically isolated device because the device is power supplied by the PC.

Q: How to directly interconnect two CAN ports!

A: If you want to interconnect two *AnaGate CAN* just via a direkt link CAN cable, you have to switch on the internal termination on both *AnaGate CAN* devices. A CAN bus network must have a termination on each side.

Note

it may work with lower baud rates without termination, but it is recommended to use a termination.

Q: Receiving a NAK when sending a CAN telegram.

A: If no CAN partner is connected to the *AnaGate CAN* (aka the CAN network), it is not possible to send CAN telegrams. The *AnaGate CAN* gets a NAK from the CAN controller. These NAK errors are sent to the AnaGate client via a data confirmation telegram.

Warning

If data confirmations are switched off no errors are sent to the client. The option *confirmations for data requests* can be set via the `CANSetGlobals` function. In High Speed Mode data confirmations are always disabled.

D.3. Questions concerning the SPI interface

Q: No SPI communication

A: If SPI communication with your SPI device fails, please proceed as follows:

1. Check that the SPI device and the SPI interface of the *AnaGate UP 2.0* are connected to a power supply.
2. Check that no other devices/µC are active on the SPI bus.
3. Ensure that no other electrical components can interfere with communication on the SPI bus between the AnaGate Universal Programmer and the SPI device.

D.4. Questions concerning the I2C interface

Q: No I2C communication

A: If I2C communication with your I2C device fails, please proceed as follows:

1. Check that the I2C device and the I2C interface of the *AnaGate UP 2.0* are connected to a power supply.
2. Check that no other devices/µC are active on the I2C bus.
3. Ensure that the SDA and SCL circuits are provided with an adequate pull-up resistance (e.g. 4.7 kOhm) to the voltage supply (3.3 V resp. 5 V).
4. Ensure that no other electrical components can interfere with communication on the I2C bus between the AnaGate I2C and the I2C device.
5. Ensure that the chip-enable address of the I2C device and the software are identical.

Q: What is the correct order to connect the GND, SCL and SDA when using an external power supply?

A: To avoid potential damage to the *AnaGate UP 2.0* the GND pin MUST be connected to the application board first. Only then can the SCL and SDA pins be allowed to make contact with the application board.

D.5. Questions concerning the JTAG interface

Q: No JTAG communication

A: If JTAG communication with your JTAG device fails, please proceed as follows:

1. Check that the JTAG device and the JTAG interface of the *AnaGate UP 2.0* are connected to a power supply.

2. Ensure that the TDO pin of the last JTAG device is connected to the TDO pin of the *AnaGate UP 2.0*.

D.6. Questions concerning the Renesas interface

Q: No Renesas communication

A: If the communication with your Renesas device fails, please proceed as follows:

1. Check that the Renesas device and the Renesas interface of the *AnaGate UP 2.0* are connected to a power supply.

2. Ensure that the Reset and Mode pins are connected with an adequate pull-up (resp. pull-down) resistance to Vcc (resp. GND).

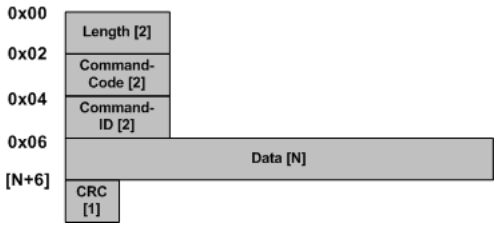
Appendix E. FAQ - Programming API

Here is a list of frequently asked questions concerning the programming API and the communication protocol.

E.1. Questions concerning the communication protocol

Q: The calculation of the check sum (CRC) does not work.

A: The following figure illustrates the basic layout of an *AnaGate* telegram:



The checksum is defined as a byte calculated by XOR from all the existing bytes in an AnaGate telegram, excluding the length bytes and the CRC byte.

The following C code function computes a valid CRC of an already created command telegram.

```
unsigned char CalcCRC( char * pBuffer, int nBufferLength )
{
   int i;
   unsigned char nCRC = pBuffer[2]; // skip the length bytes

   // XOR all bytes in the message except the length information and the last byte
   for( i = 3; i < nBufferLength - 1; ++i )
   {
      nCRC ^= pBuffer[i];
   }
   return nCRC;
}
```

When using the function `CalcCRC` the parameter *`pBuffer`* must point to the data buffer which contains the already created complete data telegram. The length parameter *`nBufferLength`* depends on the created command type and can be computed as shown below:

```
buffer length = sizeof( command length ) + sizeof( command code )
                + sizeof( command id ) + sizeof( CRC ) + sizeof(data)
              = 7 + sizeof(data)
```

Appendix F. Technical support

The AnaGate hardware series, software tools and all existing programming interfaces are developed and supported by Analytica GmbH. Technical support can be requested as follows:

Internet

The AnaGate web site [http://www.anagate.de/en/index.html] of Analytica GmbH contains information and software downloads for AnaGate Library users:

- Product updates featuring bug fixes or new features are available here free of charge.

E-Mail

If you require technical assistance over the Internet please send an e-mail to

<support@anagate.de>

To help us provide you with the best possible support please keep the following information and details at hand when you contact our support team.

- Version number of the used programming tool or AnaGate library
- AnaGate hardware series model and firmware version
- Name and version of the operating system you are using

Bibliography

Books

[LuaRef2006-EN] Roberto Ierusalimschy, Luiz Henrique Figueiredo, and Waldemar Celes. Copyright © 2006 R. Ierusalimschy, L. H. de Figueiredo, W. Celes. ISBN 85-903798-3-3. Lua.org. *Lua 5.1 Reference Manual*.

[LuaProg2006-EN] Roberto Ierusalimschy. Copyright © 2006 Roberto Ierusalimschy, Rio de Janeiro. ISBN 85-903798-2-5. Lua.org. *Programming in Lua (second edition)*.

[LuaProg2013-EN] Roberto Ierusalimschy. Copyright © 2013 Roberto Ierusalimschy, Rio de Janeiro. ISBN 85-903798-5-X. Lua.org. *Programming in Lua, Third Edition*.

Other publications

[NXP-I2C] NXP Semiconductors. Copyright © 2007 NXP Semiconductors. *UM10204*. I2C-bus specification and user manual. Rev. 03. 19.06.2007.

[TCP-2010] Analytica GmbH. Copyright © 2010 Analytica GmbH. *Manual TCP-IP communication* . Version 1.2.6. 15.05.2008.

[Prog-2013] Analytica GmbH. Copyright © 2013 Analytica GmbH. *AnaGate API 2* . Programmer's Manual . Version 2.0. 15.05.2013.

[CiA-DS301] Copyright © 2002 CAN in Automation (CiA) e. V.. CAN in Automation (CiA) e.V.. 13.02.2002. *Cia 301, CANopen Application Layer and Communication Profile*.