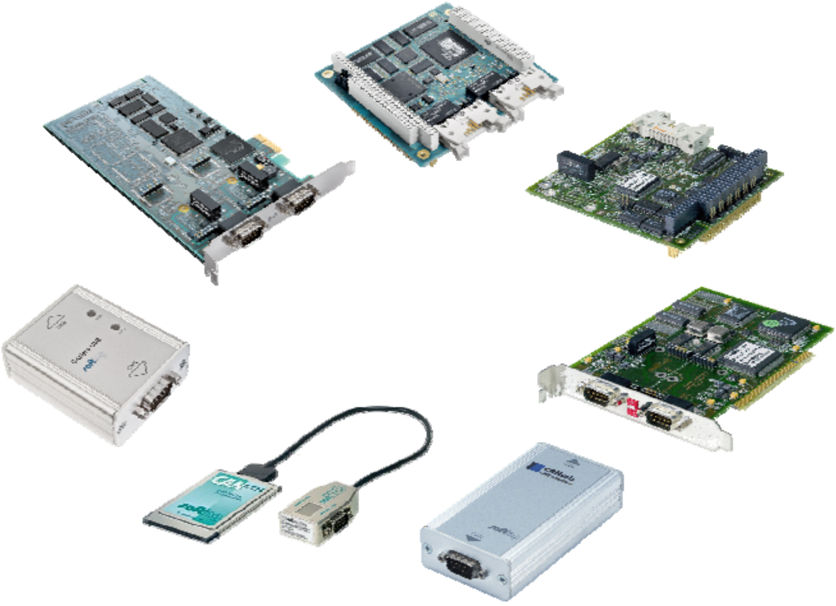
Softing Industrial Automation GmbH
Richard-Reitzner-Allee 6
D-85540 Haar
Tel.: (++49) 89/4 56 56-0
Fax.: (++49) 89/4 56 56-399
http://www.softing.com

Softing CAN Layer2 Manual

Software Description



Version 5.17
April 2012

Contents

CAN Layer 2 API - Software description

The Softing CAN Interface cards can be accessed from the user application in 2 ways.

First you can use the CAN Layer 2 API. This is a Windows DLL available for 32bit (canL2.dll) and 64bit (canL2_64.dll) applications. If you want to use the C language this is the right way for you.

If you prefer to use the Microsoft .Net Framework for building your application, you should use the .Net Softing CAN class library (CanL2dotNET.dll) instead.

1.1 Deployment

Because the libraries CANL2.dll and CANL2dotNet.dll contain the driver routines to access the hardware of the Softing CAN Interface cards, it is permitted to deliver these libraries with the customer application.

1.2 System requirements

- Microsoft Windows XP, Windows Vista, Windows 7 (32 and 64 bit), Windows Server 2003 R2 and Windows Server 2008
- Microsoft.NET Framework Version 2.0 (for .NET class library CANL2dotNet.dll)
- Softing CAN Interface (CAN-ACx-PCI, CAN-ACx-104, CANusb CANcard2, CAN-PROx-PC104+, CAN-PROx-PCIe or CANpro USB)

1.3 Driver concept

The API functions to program the interface for CAN access are supplied in a Windows DLL ‘canL2(_64).dll’. This library accesses the DPRAM on the Softing CAN Interface cards via the driver DLL ‘canchd(_64).dll’ and the hardware driver.

Softing provides alternatively a .Net class library which offers an interface to the Microsoft .Net framework.

The hardware driver is a WDM (Windows Driver Model) device driver.

Driver and driver DLL are placed in the system directory of the OS. We recommend to copy the API DLL ‘CanL2.dll’ and CANL2dotNET.dll to the local directory of the application to prevent access errors due to existence of API DLLs of different versions.

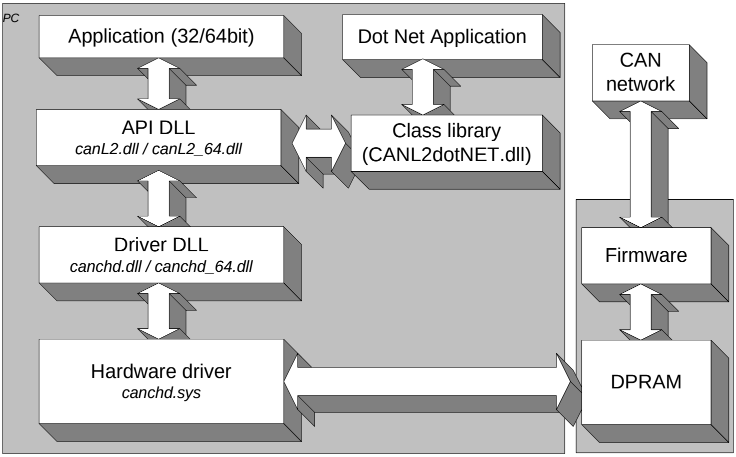


Fig. 1-1: Access structure of the API software

1.4 Operational modes of the interface

The Softing CAN Interface cards together with their driver library offer two alternative operating modes handling CAN messages: FIFO operation and CAN object buffer. Furthermore, the object buffer can be defined as static or dynamic.

1.4.1 FIFO mode

The communication between the CAN bus and the PC application through the dual ported RAM is processed sequentially using FIFOs (Fig. 1-2). The message that is entered first into the FIFO (First In First Out), is the next to be processed further. The FIFO size depends on the used hardware. (see Fig. 1.2 for details)

FIFO mode is chosen calling *CANL2_initialize_fifo_mode* (see Fig.1-5)

1.4.1.1 Transmission request

The ‘Transmit FIFO’ handles all transmit requests of the application entered by *CANL2_send_data*.

If the Transmit FIFO gets full new transmit requests are denied and the application is informed by the error return code.

1.4.1.2 Receive events and transmit acknowledges

Received messages, bus events and transmit acknowledges on successful transmission are transferred to the application through the ‘Receive FIFO’. They can be read out of the FIFO using *CANL2_read_ac*.

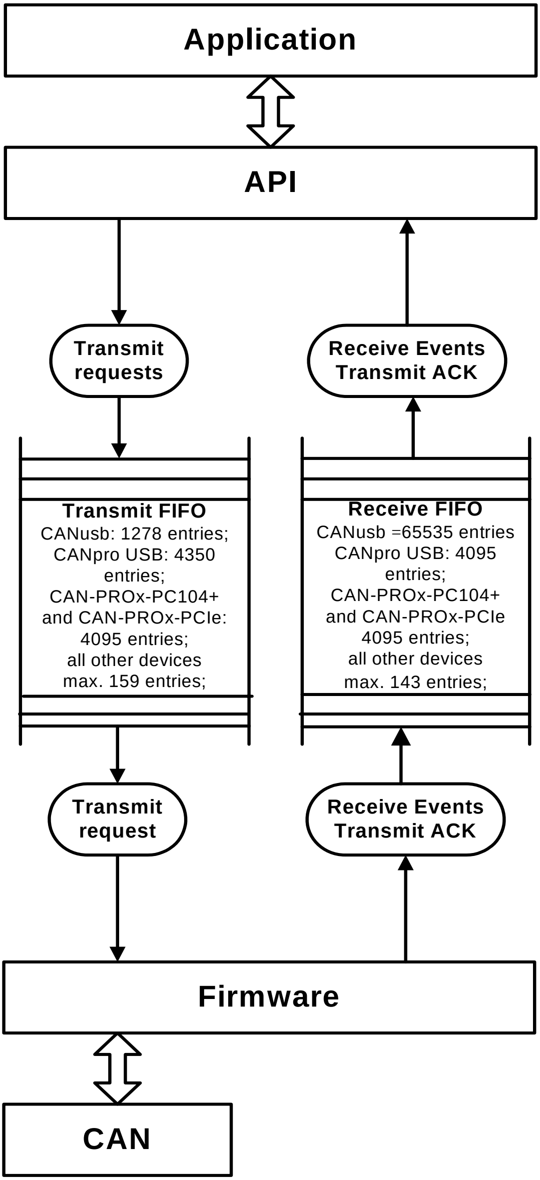

Application
API
Transmit requests
Receive Events Transmit ACK
Transmit FIFO
CANusb: 1278 entries;
CANpro USB: 4350 entries;
CAN-PROx-PC104+ and CAN-PROx-PCIe: 4095 entries;
all other devices max. 159 entries;
Receive FIFO
CANusb =65535 entries
CANpro USB: 4095 entries;
CAN-PROx-PC104+ and CAN-PROx-PCIe 4095 entries;
all other devices max. 143 entries;
Transmit request
Receive Events Transmit ACK
Firmware
CAN

Fig. 1-2: FIFO mode structure

1.4.2 Dynamic object buffer mode

The dynamic object buffer mode is chosen calling *CANL2_enable_dyn_obj_buffer* (see Fig. 1-6). In this operational mode the CAN messages and their data are stored in 2 object lists, i.e. transmission and reception list for each CAN channel (Fig. 1-3). Each list can bear a maximum of 200 objects.

The entries of the lists, i.e. CAN messages of interest, have to be defined by the application using *CANL2_define_object* in the initialization routine. An object includes identifier and data of a CAN message. The API handles the objects with their object number which is returned by *CANL2_define_object* to the application program.

It is possible at any time to read or write the data of a defined object. Thus, the application always has a consistent representation of a defined "CAN database".

The handling of transmission requests, received messages, transmit acknowledges and remote frames are individually switched on or off for each object by definition (*ReceiveIntEnable*, *AutoRemoteEnable*, *TransmitAckEnable*). The interface offers two main handling mechanisms for these interaction tasks, FIFO or polling. They can be configured using *CANL2_initialize_interface*.

1.4.2.1 Transmission requests

A transmit request is commanded by *CANL2_send_object* or *CANL2_write_object*.

If *TransmitReqFifoEnable* is set in *CANL2_initialize_inter-face*, the transmit request for an object is transferred to the CAN controller through a FIFO. Otherwise, the transmit object lists are polled for objects to be sent.

The FIFO has a maximum of 255 entries. An overrun of the FIFO is recognized and reported to the application.

Polling is processed from low to high object numbers.

1.4.2.2 Transmit acknowledges

On successful transmission of an object a corresponding acknowledge can inform the application using *CANL2_read_ac*.

The acknowledges can be switched on or off for either all objects (*TransmitAckEnableAll*) or for each transmit object by definition (*TransmitAckEnable*).

If the transmit acknowledge FIFO is configured (*TransmitAckFifoEnable*), the transmit acknowledges are transferred through a FIFO to the application. Otherwise, the transmit object lists are polled for acknowledged objects.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost transmit acknowledge messages are recognized, counted and reported to the application.

Polling is processed from low to high object numbers.

1.4.2.3 Receive events

Calling *CANL2_read_ac,* the application is informed about reception of objects and other bus events.

The report of a received object and generating an interrupt to the application can be switched on/off by definition (*ReceiveIntEnable*) for filter functionality. The data of the received object are entered into the receive object list in any case.

If a receive FIFO is configured (*ReceiveFifoEnable*), the received objects and status messages are transferred through a FIFO to the application. Otherwise, the receive object lists are polled for received objects.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost messages are recognized, counted and reported to the application.

Polling is processed from low to high object numbers.

1.4.2.4 Remote frames

If automatic transmission on reception of remote frames is configured by definition for an object (*AutoRemoteEnable*), the interface sends automatically a data frame with the same identifier. Otherwise, the remote frame is inserted into the object list and should be replied by the application.

If FIFO for auto remote transmission is configured (*TransmitRemoteFifoEnableAll*), the incoming remote frames are passed on for auto transmission through a FIFO. Otherwise, the remote request is stored in the transmit object lists, which are polled for transmission of data frames.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost remote transmit requests are recognized, counted and reported to the application.

Polling is processed from low to high object numbers.

NOTE:
The remote frame is only answered automatically after the first call of *CANL2_supply_object_data* or *CANL2_write_object* for the related object. This assures that no non-initialized data are transmitted. If a remote frame is received before the first call of *CANL2_supply_object_data* or *CANL2_write_object* an error is reported to the application.

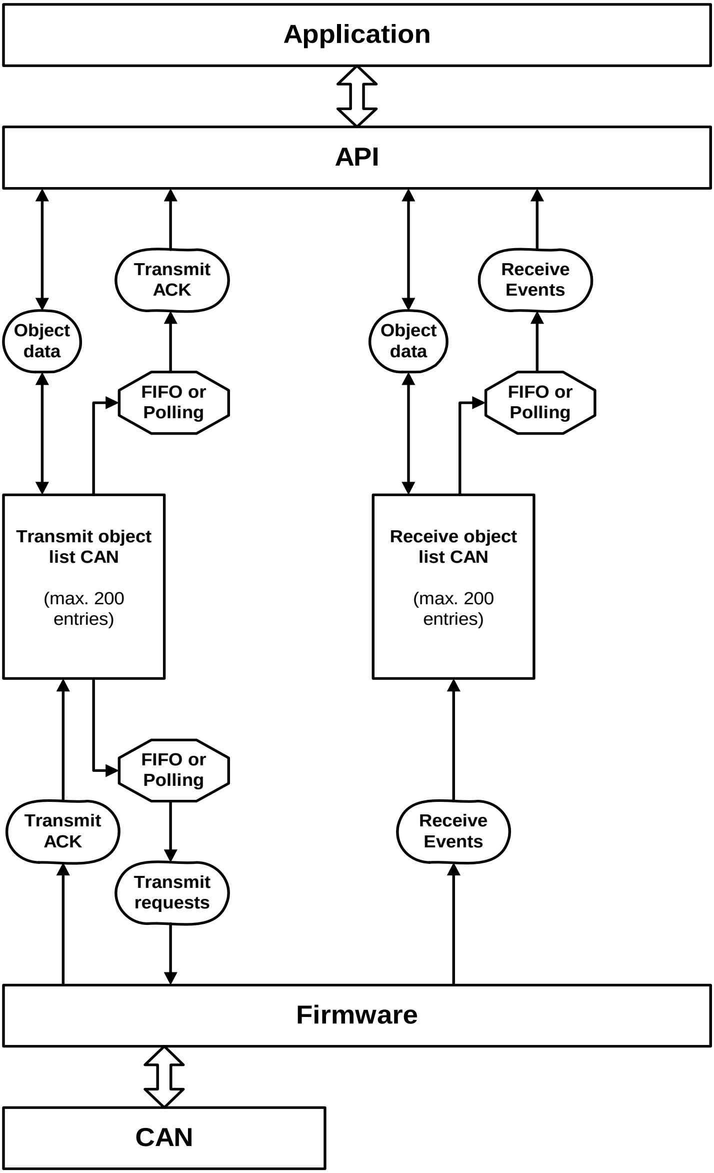


Fig. 1-3: Dynamic object buffer mode

1.4.3 Static object buffer mode (only for 11-bit identifiers)

The static object buffer mode is automatically chosen if none of the other operating modes is enabled (Fig. 1-7). In that mode the CAN messages and their data are stored in 2 object lists, one for transmission and one for reception (see Fig. 1-4).

In contrast to the dynamic object buffer, the object lists holds all 2048 standard CAN identifiers (11 bit format according to CAN 2.0A spec.). The objects of these lists can be optionally defined by the application using *CANL2_define_object*. Hence, an individual configuration of the handling for each object is possible.

It is possible to access the object data at any time. Thus, the application always has a consistent representation of the complete "CAN database" for all for 11 bit identifiers.

The handling of transmission requests, received messages, transmit acknowledges and remote frames can be configured individually by the application using *CANL2_initialize_interface*. The interface offers two main mechanisms for these interaction tasks, FIFO or polling.

1.4.3.1 Transmission request

A transmit request is commanded by *CANL2_send_object* or *CANL2_write_object*.

If the transmit FIFO is configured (*TransmitReqFifoEnable*), the transmit request for an object is transferred through a FIFO to the CAN controller. Otherwise, the transmit object list is polled for objects to be sent. This polling can be limited to those transmit objects defined using *CANL2_define_object*. Otherwise, all transmit objects are polled (*TransmitPollAll*).

The FIFO has a maximum of 255 entries. An overrun of the FIFO is recognized and reported to the application.

Polling is processed from low to high identifiers.

1.4.3.2 Transmit acknowledges

On successful transmission of an object a corresponding acknowledge can inform the application by using *CANL2_read_ac*.

The acknowledges can be switched on or off for either all objects (*TransmitAckEnableAll*) or for each transmit object by definition (*TransmitAckEnable*).

If the transmit acknowledge FIFO is configured (*TransmitAckFifoEnable*), the transmit acknowledges of an object are transferred through a FIFO to the application. Otherwise, the transmit object list is polled for acknowledged objects.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost transmit acknowledge messages are recognized, counted and reported to the application.

Polling is processed from low to high object numbers.

By calling *CANL2_read_ac* the application is informed about reception of objects and other bus events.

If *ReceiveEnableAll* is set, all data and remote frames are received by the interface. Otherwise, the user can define the objects to be received (*CANL2_define_object*).

Furthermore, the report of a received object to the application and generation of an interrupt can be switched on/off either globally (*ReceiveIntEnableAll*) or individually by definition (*ReceiveIntEnable*). The data of the received object are entered into the receive object list in any case.

If FIFO mode is configured (*ReceiveFifoEnable*), the received objects and status messages are transferred through a FIFO to the application. Otherwise, the receive object list is polled for received messages.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost messages are recognized, counted and reported to the application.

Polling is processed from low to high identifiers and can be limited to those receive objects defined using *CANL2_define_object*. Then the objects are polled in succession of their definition. Otherwise, all receive objects are polled (*ReceivePollAll*).

1.4.3.3 Remote frames

If automatic transmission on a reception of a remote frame is configured by definition for an object (*AutoRemoteEnable*) or globally (*AutoRemoteEnableAll*), the interface sends automatically a data frame with the same identifier. Otherwise, the remote request is stored in the transmit object list, which is polled for transmission of data frames.

If the FIFO for auto remote transmission is configured (*TransmitRemoteFifoEnableAll*), the incoming remote frames are passed on for auto transmission through a FIFO. Otherwise, they are stored in the object list, which is polled for transmission of data frames.

The FIFO has a maximum of 255 entries. An overrun of the FIFO and the number of lost remote transmit requests are recognized, counted and reported to the application.

NOTE:
The remote frame is only answered automatically after the first call of *CANL2_supply_object_data* or *CANL2_write_object* for the related object. This assures that no non-initialized data are transmitted. If a remote frame is received before the first call of *CANL2_supply_object_data* or *CANL2_write_object* an error is reported to the application.

NOTE:
Please note that the objects defined first are also polled first, and in this way a higher priority and a lower polling time is maintained relative to the objects that follow. It is sensible to define objects in the sequence of their identifiers in order to make prioritization of objects with low identifiers the same as on the CAN bus. This is true for static as well as for dynamic object buffer mode.

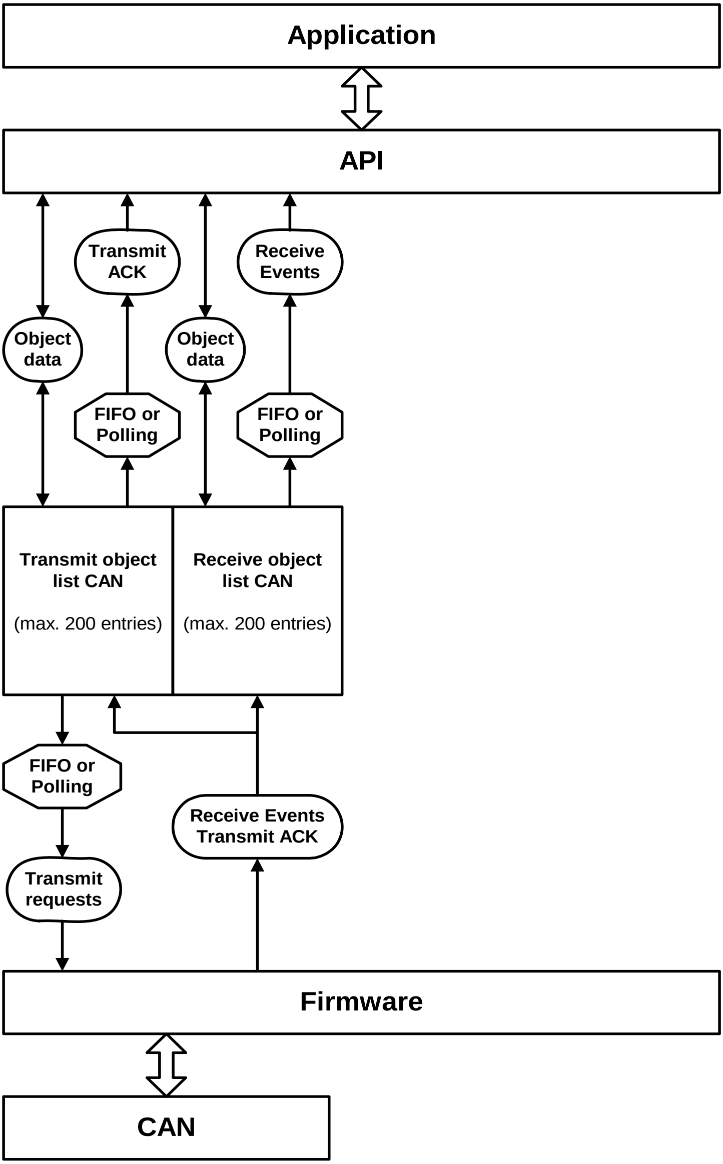


Fig. 1-4: Static object buffer mode

1.4.4 Comparison FIFO to object buffer mode

The advantage of object buffer compared to FIFO operation is that the **last** received data of an object are always available to the application in all cases. Even though, if older receptions still have not been processed, no new data are lost if an overrun of the object received message FIFO occurs.

When the transmit request FIFO is full the data can be buffered, and the application is freed of this task. Hence, the transmit request is denied but the data are buffered anyway.

It is possible at any time to read out or write in the receive and transmit objects. Thus, the application always has access to the provided CAN database.

An additional advantage of the object buffer is that data of objects are available to the application very quickly after they are received from the bus, even if the application still has not processed older messages. Accordingly, messages can be transmitted before lower priority messages, even if the lower priority messages were requested first. This is true if the object buffer is operated in polling mode.

FIFO operation offers the advantage that data of an object or an identifier are not overwritten by other received data of the same object until they are evaluated by the application (overrun). Therefore, when transmitting, a sequence of data of an object can be buffered and transmitted.

Furthermore, FIFO provides full access to all identifiers possible on CAN, even for extended identifier. No relation between identifier and defined object number has to be processed by the application.

1.5 Implementation

The Softing CAN Layer2 has to be used in a specific sequence of instructions for proper operation.

1.5.1 Board initialization

After program start each CAN channel to be used must be initialized by *INIL2_initialize_channel*.

1.5.2 FIFO mode

The function *CANL2_initialize_fifo_mode* is used to define all CAN specific parameters as well as functional options for the operation of the FIFO mode (e.g. generating acknowledgements for the confirmation of successful transmissions or activating the detection error frames).
The function finally places the CAN controller in operating status. From this point onwards transmit jobs can be issued and incoming data can be monitored.

To monitor the bus events *CANL2_read_ac* should be polled or has to be implemented in an interrupt thread.

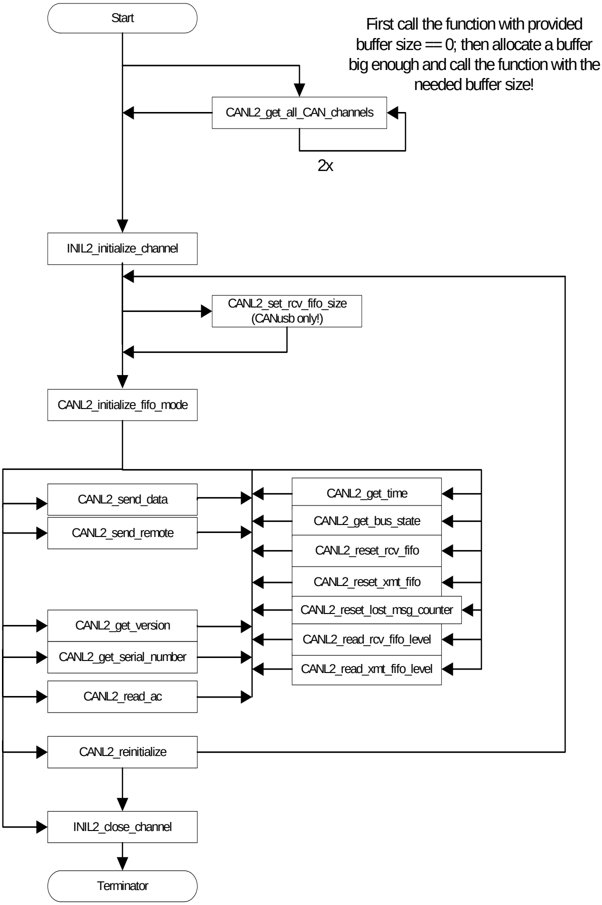

Start
First call the function with provided buffer size == 0; then allocate a buffer big enough and call the function with the needed buffer size!
CANL2_get_all_CAN_channels
2x
INIL2_initialize_channel
CANL2_set_rcv_fifo_size (CANusb only!)
CANL2_initialize_fifo_mode
CANL2_send_data
CANL2_send_remote
CANL2_get_version
CANL2_get_serial_number
CANL2_read_ac
CANL2_get_time
CANL2_get_bus_state
CANL2_reset_rcv_fifo
CANL2_reset_xmt_fifo
CANL2_reset_lost_msg_counter
CANL2_read_rcv_fifo_level
CANL2_read_xmt_fifo_level
CANL2_reinitialize
INIL2_close_channel
Terminator

Fig. 1-5: Flow chart programming FIFO mode

1.5.3 Object buffer mode

The CAN controller is placed into reset status using *CANL2_reset_chip*. Then, CAN specific parameters are initialized using *CANL2_initialize_chip* for the bit timing, *CANL2_set_acceptance* for filtering CAN messages and *CANL2_set_output_control* for the physical signal specification.

The operating modes of object buffer are enabled using *CANL2_initialize_interface*. Beforehand the object buffer can be switched to dynamic object buffer (Fig. 1-6) by calling *CANL2_enable_dyn_obj_buf*. Otherwise the static object buffer is chosen by default (Fig. 1-7).

Object specific settings can be made by calling *CANL2_define_object*. The definition is necessary in dynamic object buffer mode but optionally in static object buffer mode.

The function *CANL2_start_chip* ends the initialization and puts the CAN Interface in operating status. From this point onward transmit jobs can be issued and incoming data can be monitored.

To monitor the bus events *CANL2_read_ac* or *CANL2_read_rcv_data* are polled by the application.

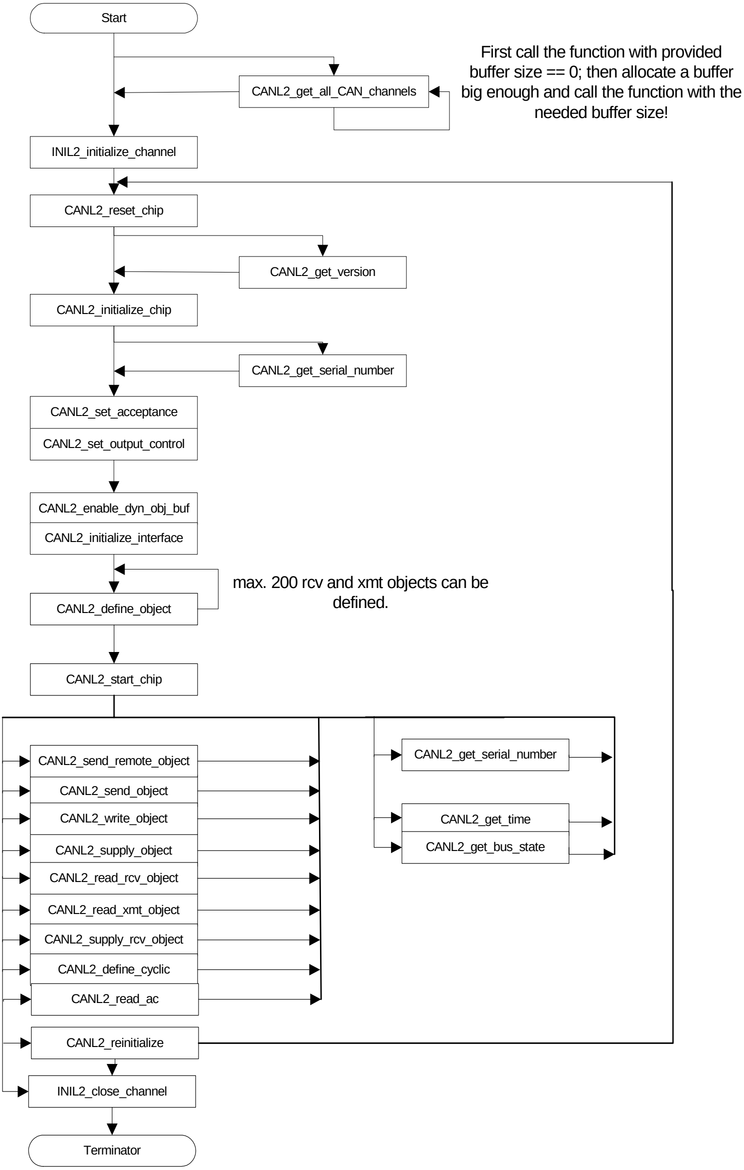


Fig. 1-6: Flow chart programming dynamic object buffer mode

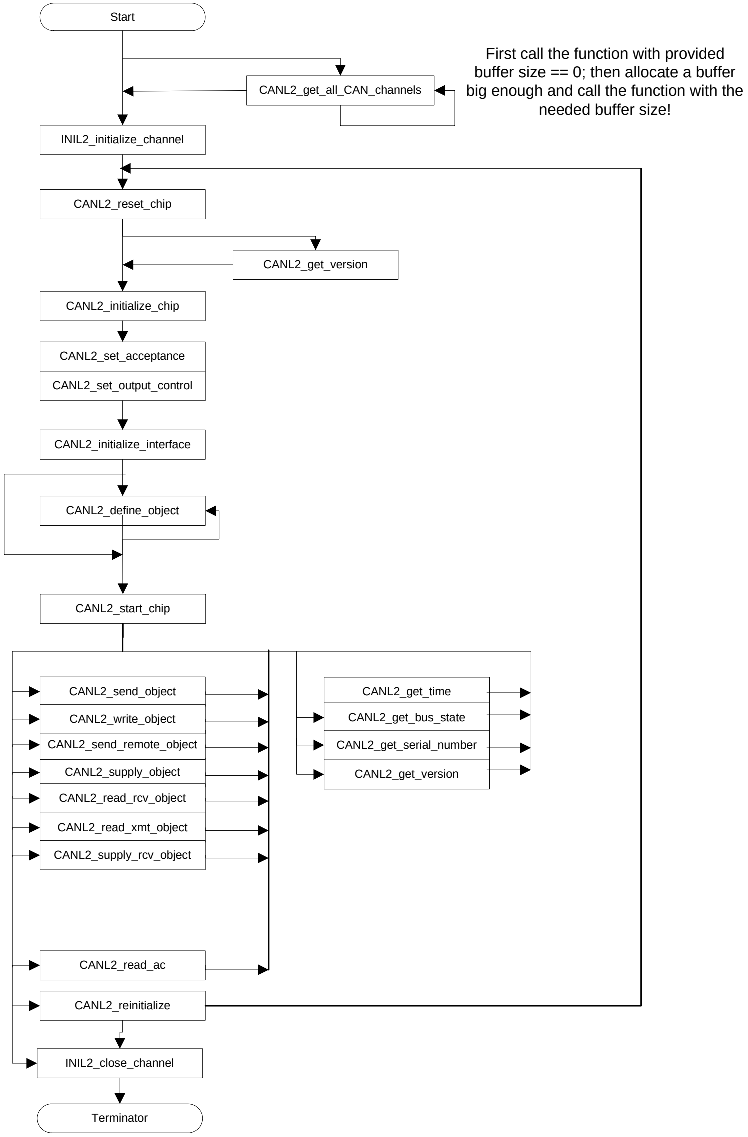


Fig. 1-7: Flow chart programming static object buffer mode

1.5.4 Reinitialization and termination

INIL2_reinitialize stops the current online operation of the specified CAN channel. Afterwards the operating mode and all corresponding parameters can be set anew. When the application program is to be ended *INIL2_close_channel* should be called. This function releases the system resources locked for the application by *INIL2_initialize_channel.*

Otherwise, the application may have problems to get the handle to the DPRAM a second time without system exit (e.g. applications with LabVIEW a.o.).

NOTE:
If *INIL2_close_channel* is not called at the end of operation the handle to the DPRAM may remain locked for the surrounding process. Thus, it can’t be accessed by a succeeding initialization without system exit (e.g. applications with LabVIEW).

This concerns all default exits of the application as well as program termination by errors which occur after successful call to *INIL2_initialize_channel*.

1.6 Description of the CAN Layer2 API

1.6.1 About the CAN Layer2 API

The CAN Layer2 **API** (**A**pplication **P**rogramming **I**nterface) is realized as a DLL for Windows.

Different operational modes of the interface can be configured: FIFO and object buffer mode. Thus, the programmer is enabled to adapt it to the communication task in the most suitable way.

The CAN Layer2 API is designed to be conform to the API's of Softings other CAN interfaces (PCMCIA, ISA, PC/104, PC/104plus, PCIexpress). It provides the following functionality:

- Initialization of CAN parameters, e.g. bit rate, output control a. o.
- Transmission and reception of data and remote frames
- Message filtering
- Acknowledgment on successful transmission (optional)
- Automatic response to remote frames (optional)
- Error state detection
- Bus state detection
- Interrupt support
- Cyclic transmission

This chapter describes the basic operational modes, functions and program sequences of the API.

1.6.2 Interrupt processing

1.6.2.1 Interrupt events

For many applications it is useful to be informed by interrupt about occurrence of CAN events. Otherwise, the Softing CAN Interface cards must be polled for new events which requires more PC processor time.

The firmware triggers a hardware interrupt to the PC on the following CAN events:

- Reception of data, remote and error frames
- Acknowledge on successful transmissions if enabled
- Change of bus state

1.6.2.2 Windows interrupt programming

If the CAN driver detects an interrupt it triggers a Windows event which can be evaluated by the application to control a process or thread. Thus, an application or thread can be created which is only processed in case of the interrupt.

As a prerequisite the interrupt event must be created by the application. The hardware driver must be supplied with the handle of this Windows event by API function *CANL2_set_interrupt_event*. Furthermore a thread must be created and started which gets into WAIT status until the interrupt event is triggered by the driver. Then, the necessary interrupt activities can be processed and the thread gets back into WAIT status.

Before termination of the Windows process the created resources should be released for proper operation.

The application of the Windows interrupt is exemplary implemented in the test program ‘can_test.exe’. The interrupt relevant functions are sampled in ‘Intexmpl.c’ in ‘Source’ directory of the installed software. This C source code provides macro functions for initialization and termination of the interrupt handling as well as an interrupt service thread which may be linked to a customer application.

1.6.3 INIL2_initialize_channel

```
int        INIL2_initialize_channel(
           CAN_HANDLE *ulChannelHandle,
            char *sChannelName)
```

Function Parameters:

Type/Name	Description
CAN_HANDLE *ulChannelHandle	OUT: handle of the channel; this handle is needed for all other API calls. The type “CAN_HANDLE” is defined in the header file “canl2.h” it is an unsigned long.
char *sChannelName	IN: Channel name of the channel, which should be used; must be set by the application;

INIL2_initialize_channel enables the memory access to the DPRAM of the Softing CAN Interface cards. Thus, it is necessarily called before any other API function.

If the DPRAM access is denied the function returns an error code which corresponds to the error cause.

NOTE:
If *CANL2_initialize_channel* fails, other API functions should not be called since the non-initialized memory handle may cause an access violation.

Function Return Codes:

0	Initialization successful
-536215551	Internal Error
-536215550	General Error
-536215546	Illegal driver call
-536215542	Driver not loaded / not installed, or device is not plugged.
-536215541	Out of memory
-536215531	An error occurred while hooking the interrupt service routine
-536215523	Device not found
-536215522	Can not get a free address region for DPRAM from system
-536215521	Error while accessing hardware
-536215519	Can not access the DPRAM memory
-536215516	Interrupt does not work/Interrupt test failed!
-536215514	Device is already open
-536215512	An incompatible firmware is running on that device (CANalyzer/CANopen/DeviceNet firmware)
-536215511	Channel can not be accessed, because it is not open
-536215500	Error while calling a Windows function
-1002	Too many open channels.
-1003	Wrong DLL or driver version.
-1004	Error while loading the firmware. (This may be a DPRAM access error)
-1	Function not successful

Error codes which can only occur with the CANusb interface:

-602	Unable to open USB pipe
-603	Communication via USB pipe broken
-604	No valid lookup table entry found
-611	CANusb framework initialization failed
-1005	CANusb DLL (CANusbM.dll) not found.

1.6.4 CANL2_get_all_CAN_channels

```
int CANL2_get_all_CAN_channels (
                    unsigned long u32ProvidedBufferSize,
                    unsigned long *pu32NeededBufferSize,
                    unsigned long *pu32NumOfChannels,
                    CHDSNAPSHOT *pBuffer)
```

This function provides information about the CAN channels in your computer. It writes the information into the elements of the array “ pBuffer ”.

Function Parameters:

- **u32ProvidedBufferSize: (IN)**

This parameter must be initialized by the user with the size of pBuffer in Bytes.

- **pu32NeededBufferSize: (OUT)**

After the call of CANL2_get_all_CAN_channels the parameter holds the needed size of pBuffer. If u32ProvidedBufferSize < *pu32NeededBufferSize after the function call, a new pBuffer with an appropriate size should be allocated by the user and the function should be called again to get the information about the channels.

- **pu32NumOfChannels: (OUT)**

After the call of CANL2_get_all_CAN_channels the parameter holds the number of channels which the driver had found on your computer.

- **pBuffer: (OUT)**

Array of CHDSNAPSHOT; It holds the information about the CAN channels after the function call. (Channel names, type of firmware, which is loaded on a specific channel, whether the channel is opened by an other application or not, serial number of the Softing CAN Interface, physical channel number [1 or 2])

Elements of structure CHDSNAPSHOT:

- **unsigned long u32Serial: [4 Bytes]**

Serial number of the CAN Interface

- **unsigned long u32DeviceType: [4 Bytes]**

Type of Softing CAN Interface:

u32DeviceType	Softing CAN Interface
5	CANcard2
7	CAN-ACx-PCI
8	CAN-ACx-PCI/DN
9	CAN-ACx-104
10	CANusb
13	CAN-PROx-PCIe
14	CAN-PROx-PC104+
15	CANpro USB
261	EDICcard2

- **unsigned long u32PhysCh: [4 Bytes]**

Physical channel number; The physical channel number of the first channel on an CAN interface with 2 channels is 1, the number of the second channel on the same interface is 2.

- **unsigned long u32FwId: [4 Bytes]**

Type of the firmware which is loaded on the channel; layer 2 firmware has the firmware id 0x01.

- **BOOLEAN bIsOpen: [1 Byte]**

True if the channel is opened by a process, false otherwise.

- **ChannelName:** (String) **[80 Bytes]**

The name of the channel.

The size of the CHDSNAPSHOT structure is 97 Bytes. For every channel plugged in the Computer, 97 Bytes are needed.

There is an easy way to calculate the memory requirements. If CANL2_get_all_CAN_channels() is called with:

u32ProvidedBufferSize == 0 and pBuffer == NULL, then *pu32NeededBufferSize will contain the needed buffer size after the function call.

The buffer can be allocated now and the function can be called again with a sufficient buffer to get the required information.

Function Return Codes:

0:	Function successful
-1:	Function not successful

1.6.5 CANL2_set_rcv_fifo_size

```
int      CANL2_set_rcv_fifo_size(
   CAN_HANDLE can,
    int   FifoSize)
```

To accommodate to larger delays in processing the received messages the *Receive-FIFO* in FIFO mode is configurable in size for CANusb. *CANL2_set_rcv_fifo_size* must be called if other sizes than the default size of 255 entries is to be used.

NOTE:
This function is only applicable in FIFO mode. It is only supported by the Layer 2 API for CANusb.

Function Parameters:

- **can**
  Channel handle (see 1.6.3)

- **FifoSize**

The value defines the size of the *Receive-FIFO* as defined in the following table:

Value	No of entries
0	255 (default)
1	511
2	1023
3	2047
4	4095
5	8191
6	16383
7	32767
8	65535

Function Return Codes:

0: Function successful

-1: The FIFO size can not be changed, because the CAN controller is already online

-102: Parameter error

-1000: Invalid channel handle

1.6.6 CANL2_initialize_fifo_mode

```
int CANL2_initialize_fifo_mode(
                          CAN_HANDLE can,
                          L2CONFIG *pUserCfg)
```

This function provides a fast and easy way to initialize a CAN channel. After calling this function successfully, sending and receiving of CAN messages in FIFO mode is possible.

If you don’t know exactly the appropriate values of the elements of L2CONFIG you can configure the channel by using the SCIM (Softing CAN Interface Manager) in the Settings of your PC. In this case you must set the values in L2CONFIG which you want to retrieve from SCIM to GET_FROM_SCIM (-1).

Function Parameters:

Table 1.6-2: CANL2_initialize_fifo_mode

Name	Description	Range
can:	Channel handle (see 1.6.3)	-
pUserCfg:	Pointer to a CAN layer 2 configuration	-

Elements of structure L2CONFIG:

Element	Description	Range
double fBaudrate	not used, should be 0.0	0.0
long u32Prescaler	Prescaler (if set to –1 then the value will be retrieved from SCIM)	[1..32] or (-1)
long u32Tseg1	CAN Time segment 1	[1..16] or (-1)
long u32Tseg2	CAN Time segment 2	[1..8] or (-1)
long u32Sjw	Sync jump with	[1..4] or (-1)
long u32Sam	Number of samples	[0..1] or (-1)
long u32AccCodeStd	Acceptance code for 11 bit Identifier	Range of 11 bit Id or (-1)
long u32AccMaskStd	Acceptance mask for 11 bit Identifier	Range of 11 bit Id or (-1)
long u32AccCodeXtd	Acceptance code for 29 bit Identifier	Range of 29 bit Id or (-1)
long u32AccMaskXtd	Acceptance mask for 29 bit Identifier	Range of 29 bit Id or (-1)
long u32OutputCtrl	Output Control Register	[0..255] or (-1)

BOOL bEnableAck	Enable confirmation of successfully sent CAN message	TRUE or FALSE
BOOL bEnableErrorframe	Enable the detection of ERROR Frames on the CAN bus.	TRUE or FALSE
HANDLE hEvent	Handle of the Windows event, which should be set in case of a CAN event. If this value is set to –1 no interrupt mode is possible.	-

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong Parameter
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.7 CANL2_reset_chip

int CANL2_reset_chip(CAN_HANDLE can)

This function terminates a possible bus operation and places the CAN chip into reset status.

After reset the bit timing, acceptance register and output control register have to be defined before the CAN controller is started by the related API functions.

Function Parameters:

Table 1.6-2: CANL2_reset_chip

Name	Description	Range
can:	Channel handle (see 1.6.3)	-

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-108:	Wrong hardware; (CANcard2 or EDICcard2 with 25MHz instead of 24MHz)
-1000:	Invalid channel handle

1.6.8 CANL2_get_version

```
int CANL2_get_version(
        CAN_HANDLE  can,
        int         *sw_version,
        int         *fw_version,
        int         *hw_version,
        int         *license,
        int         *can_chip_type);
```

This function provides useful information the version numbers of hard-, soft- and firmware, license and CAN chip types of the CAN interface.

Function Parameters:

- **can:**

channel handle (see 1.6.3)

- **sw_version:**

Pointer to the entry of the version number of driver software.

The number is encoded as **sw_version* / 100 as the main version number; **sw_version* % 100 refers to the subordinate part of the number.

- **fw_version:**

Pointer to the entry of the version number of the firmware.

The number is encoded as **fw_version* / 100 as the main version number; **fw_version* % 100 refers to the subordinate part of the number.

- **hw_version:**

Pointer to the entry of the version number of the hardware.

The number is encoded as **hw_version* % 0x100H as the main version number; **hw_version* / 0x100H refers to the subordinate part of the number.

- **licence:**

Pointer entry of the license type of the CAN-AC2-PCI

01H:	Licensed for operation with interface software
02H:	Licensed for operation with CANalyzer software

- **can_chip_type:**

Pointer to entry containing the last three digits of the CAN chip type.

can_chip_type:	CAN	
5:	NEC72005	(CANcard)
161:	Infineon XC161 TwinCAN	(CAN-PROx-PC104+) (CAN-PROx-PCIe)
1000:	SJA1000	(CANcard-SJA) (CAN-AC2/PCI)
527:	Intel 82527	(CAN-AC2/527)
200:	Philips 82C200	(CAN-AC2)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-1000:	Invalid channel handle

1.6.9 CANL2_get_serial_number

```
CANL2_get_serial_number(
                    CAN_HANDLE can,
                    unsigned long *SerialNumber)
```

Function Parameters

Can: (IN) channel handle (see 1.6.3)

SerialNumber: (OUT) serialnumber of the CAN interface

This function returns the serial number of the Softing CAN Interface card in **SerialNumber*.

Function Return Codes:

0: Function successful

-1: Function not successful

-1000: Invalid channel handle

1.6.10 CANL2_initialize_chip

```
int     CANL2_initialize_chip(
                  CAN_HANDLE  can,
                  int         presc,
                  int         sjw,
                  int         tseg1,
                  int         tseg2,
                  int         sam)
```

Function Parameters:

Can: channel handle (see 1.6.3)

Table 1.6-2: Bit timing parameter

Name	Description	Range
presc:	CAN-Prescaler	[1..32]
sjw:	CAN-Synchronisation-Jump-Width	[1..4]
tseg1:	CAN-Time-Segment 1	[1..16]
tseg2:	CAN-Time-Segment 2	[1..8]
sam:	Number of samples	[0, 1]

The function defines the bit timing (baud rate) of the CAN chip. Parameters *presc*, *sjw*, *tseg1* and *tseg2* represent logical values that are used to describe the bit timing. These values are converted and written to the bus timing register 1 and 2 of the Philips SJA1000.

The **baud rate** is calculated by the following formula, whereby certain limit conditions must be maintained:

$$\text{Baud rate} = \frac{f_{crystal}}{2 * presc * (1 + tseg1 + tseg2)}$$

The crystal frequency $f_{crystal}$ is 16 MHz.

The limitations of the bit timing of the used CAN controllers lead to following **conditions**:

$$8 \leq (1 + tseg1 + tseg2) \leq 25$$

$$tseg1 + tseg2 \geq 2 * sjw$$

$$tseg2 \geq sjw$$

The prescaler divides the crystal frequency by *presc* to build the clock cycle time Δt.

The parameter *sam* defines how many **samples** are taken to detect the bit level.

$sam = 0 \rightarrow$ 1 sample (high speed buses)

$sam = 1 \rightarrow$ 3 samples (low/medium speed buses)

The sampling point is defined at the edge between time segment 1 and time segment 2. It is recommended to place the sampling point between 50% and 80% of the bit time. At high baud rates the communication is more stable if the sample is taken in the last quarter of the bit time.

The synchronization jump width is used to compensate the time shifts between the different CAN nodes in the network. It defines the maximum number *sync* of clock cycles by which the time segment 1 may be lengthened and time segment 2 shorted during resynchronization.

NOTE
A parameter value of -1 means that the API retrieves the used value from SCIM (Softing CAN Interface Manager).

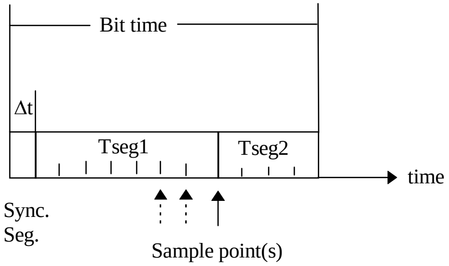


Fig. 1-8: Bit period

Table 1.6-3: Baud rate examples

baud rate	**presc**	**sjw**	**tseg1**	**tseg2**
1 Mbaud	1	1	4	3
800 kBaud	1	1	6	3
500 kBaud	1	1	8	7
250 kBaud	2	1	8	7
125 kBaud	4	1	8	7
100 kBaud	4	4	11	8
10 kBaud	32	4	16	8

Function Return Codes:

0:	Initialization successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.11 CANL2_set_output_control

```
int     CANL2_set_output_control   (
                    CAN_HANDLE  can,
                    int                 OutputControl)
```

Function Parameters:

- can	Channel handle (see 1.6.3)
- OutputControl:	Input/Output-Control-Register [0 to FF$_{Hex}$ or -1]

This function defines the setting of the Output Control Register (OCR) of the CAN chip. This is used to adapt the CAN chip to the physical bus interface being used.

If the CAN controller Philips SJA1000 is used with the CAN High Speed interface (default) the output control register must be set to a value of FB$_{Hex}$. If you like to adapt the interface to a different bus physic consult the SJA data sheet for the required OCR setting.

Setting the OCR=03H switches off the transmission lines Tx0 and Tx1 of the CAN controller. Thus, the Softing CAN Interface card can't send any data frame or any acknowledge bit on received messages. Thus, the interface can monitor the activities on the CAN network without influencing it.

The OCR specification of the Philips SJA1000 is described in Table 1.6-4 to 1.6-6.

The default values for CAN High Speed can also be chosen automatically by passing the default parameter -1 which assures compatibility with other using CAN High Speed Standard.

NOTE
A parameter value of (-1) means that the API retrieves the used value from SCIM (Softing CAN Interface Manager).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-1000:	Invalid channel handle

Output control specification of Philips SJA1000:

Table 1.6-4: Output control Philips SJA1000

Bit	Function
7	OCTP1
6	OCTN1
5	OCPOL1
4	OCTP0
3	OCTN0
2	OCPOL0
1	OCMODE1
0	OCMODE0

Table 1.6-5: Output control mode of Philips SJA1000

OCMODE1	OCMODE0	Function
1	0	Normal Mode (TX0 and TX1 CAN Output)
1	1	Normal mode (Tx0 CAN Output, TX1 Bus Clock)
0	0	not implemented
0	1	not implemented

The voltage levels at the CAN outputs TX0 and TX1 depend on both the output configuration, which is determined by OCTPx and OCTNx, and the output polarity, which is determined by OCPOLx. Table 1.6-6 shows output status as a function of these settings for Philips SJA00.

Table 1.6-6: Configuration of CAN output pins TX0 and TX1

Operating Mode	**OCTPx**	**OCTNx**	**OCPOLx**	**TXD**	**TPx**	**TNx**	**Level at TXx**
FLOAT	0	0	0	0	off	off	high resistance
	0	0	0	1	off	off	high resistance
	0	0	1	0	off	off	high resistance
	0	0	1	1	off	off	high resistance
PULL DOWN	0	1	0	0	off	on	logic "0"
	0	1	0	1	off	off	high resistance
	0	1	1	0	off	off	high resistance
	0	1	1	1	off	on	logic "0"
PULL UP	1	0	0	0	off	off	high resistance
	1	0	0	1	on	off	logic "1"
	1	0	1	0	on	off	logic "1"
	1	0	1	1	off	off	high resistance
PUSH PULL	1	1	0	0	off	on	logic "0"
	1	1	0	1	on	off	logic "1"
	1	1	1	0	on	off	logic "1"
	1	1	1	1	off	on	logic "0"

TXx:	Output pin x, x=0 for TX0, x=1 for TX1
TPx:	Transistor that switches from supply voltage to TXx
TNx:	Transistor that switches from TXx to ground
TXD:	Data to be transmitted, 0=dominant, 1=recessive

1.6.12 CANL2_set_acceptance

```
int CANL2_set_acceptance(
              CAN_HANDLE      can,
              unsigned int    AccCodeStd,
              unsigned int    AccMaskStd,
              unsigned long   AccCodeXtd,
              unsigned long   AccMaskXtd)
```

Function Parameters:

Table 1.6-7: Filter parameters

Name	**Description**	**Range**
can	Channel handle (see 1.6.3)	-
AccCodeStd:	Acceptance code for standard frames	[0 to $7FF_{Hex}$] or 0xFFFFFFFF
AccMaskStd:	Acceptance mask for standard frames	[0 to $7FF_{Hex}$] or 0xFFFFFFFF
AccCodeXtd:	Acceptance code for extended frames	[0 to $1FFFFFFF_{Hex}$] or 0xFFFFFFFF
AccMaskXtd:	Acceptance mask for extended frames	[0 to $1FFFFFFF_{Hex}$] or 0xFFFFFFFF

The function *CANL2_set_acceptance* initializes the acceptance filter of the CAN controller.

The acceptance filter defines which identifiers should be passed into the receive buffer of the CAN controller.

To receive an identifier all bits of the identifier that were initialized as 1 in the acceptance mask must match the corresponding bit in the acceptance code. A "0" in the acceptance mask register means "Don't care" for the identifier bit at this position.

The parameters are converted and written into the acceptance code and mask registers of the Philips SJA1000.

NOTE
A parameter value of (-1) means that the API retrieves the used value from SCIM (Softing CAN Interface Manager).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.13 CANL2_enable_dyn_obj_buf

int CANL2_enable_dyn_obj_buf(CAN_HANDLE can)

CANL2_enable_dyn_obj_buf configures the API to run in dynamic object buffer mode (see section 1.1.2). If this function is not used, then the Softing CAN Interface card operates with the static object buffer or in the FIFO mode (if *CANL2_enable_fifo* has been called).

NOTE:
The static object buffer is usable for 11 and 29bit identifiers.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.14 CANL2_initialize_interface

```
int CANL2_initialize_interface(
        CAN_HANDLE      can,
        int             ReceiveFifoEnable,
        int             ReceivePollAll,
        int             ReceiveEnableAll,
        int             ReceiveIntEnableAll,
        int             AutoRemoteEnableAll,
        int             TransmitReqFifoEnable,
        int             TransmitPollAll,
        int             TransmitAckEnableAll,
        int             TransmitAckFifoEnable,
        int             TransmitRmtFifoEnable)
```

CANL2_initialize_interface configures properties and structure the object buffer (see sections 1.4.2 and 1.4.3). It may not be used in FIFO operation.

Function Parameters:

can	Channel handle (see 1.6.3)

- **ReceiveFifoEnable:**

Type of receive message handling from firmware to PC application.

1: Receive messages of data frames or remote frames are transferred to the PC through the receive message FIFO (see section 1.4.1).

0: The PC ascertains receive messages of data frames or remote frames by polling the objects in the receive object lists using the function *CANL2_read_ac* (see 1.4.2 and 1.5.4). Under certain conditions this can cause a longer running time of the function *CANL2_read_ac*, and can therefore result in lower throughput rates.

- **ReceivePollAll:**

This flag is only meaningful for *ReceiveFifoEnable* = 0 with static object buffer (should be 0 with dynamic object buffer).

1:	Polling of all receive objects when *CANL2_read_ac* is called (see section 1.4..3)
0:	Polling of only those receive objects which have been defined using *CANL2_define_object* (see Section 1.4.2 and 1.4.3)

- **ReceiveEnableAll:**

This flag is only meaningful with static object buffer (must be 0 with dynamic object buffer).

1:	All data frames and remote frames with standard identifiers are received. No receive objects need to be defined (However: *CANL2_define_object* can be used nevertheless, in order to activate receive objects for polling by the application under the conditions *ReceivePollAll* = 0 and *ReceiveFifoEnable* = 0)
0:	All receive objects that are passed to the PC must be defined beforehand using *CANL2_define_object*. Objects that are not defined using *CANL2_define_object* are not received by the (filter functionality).

- **ReceiveIntEnableAll:**

This flag is only meaningful while *ReceiveEnableAll* = 1 with static object buffer (should be 0 with dynamic object buffer).

1:	When receiving an arbitrary object (declared using *CANL2_define_object* or if *ReceiveEnableAll* = 1) the receive message is passed to the PC application. Additionally, an interrupt is generated to the PC. The application program can read the object using *CANL2_read_ac*.
0:	Receipt of an object is only reported to the PC (with interrupt) if the object has been declared in *CANL2_define_object* with *ReceiveIntEnable* = 1. Otherwise the data of the object will indeed be entered into object buffer (and they can be read using *CANL2_read_rcv_data*), but no information is generated for the application regarding receipt of the object (readable by *CANL2_read_ac*).

- **AutoRemoteEnableAll:**

This flag is only meaningful while *ReceiveEnableAll* = 1 with static object buffer (should be 0 with dynamic object buffer).

1:	When receiving an arbitrary remote frame the interface independently transmits a data frame with the same identifier (see 1.5.4).
0:	When receiving a remote frame the interface only transmits a data frame with the same identifier if the corresponding receive object has been declared in *CANL2_define_object* with *AutoRemoteEnable* = 1. Otherwise the remote frame is reported to the PC (calling *CANL2_read_ac* or *CANL2_read_rcv_data*). The PC must transmit an explicit response (data frame) then.

NOTE:
A data frame is transmitted after the first call of *CANL2_supply_object_data* or *CANL2_write_object* initialized the object data. A remote frame arriving before data initialization results in error report -6 in the function *CANL2_read_ac*.

- TransmitReqFifoEnable:

1:	Transmit jobs for data frames or remote frames are transferred to the CAN bus through the transmit job FIFO (see sections 1.4.2 and 1.4.3)
0:	Transmit jobs for data frames or remote frames are ascertained by the firmware by polling the objects in the transmit object lists (see sections 1.4.2 and 1.4.3).

- TransmitPollAll:

This flag is only meaningful for *TransmitReqFifoEnable* = 0 with static object buffer (should be 0 with dynamic object buffer).

1	Polling of all transmit objects (see section 1.4.3)
0:	Polling of only those transmit objects that have been defined using *CANL2_define_object* (see section 1.4.2 and 1.4.3)

- **TransmitAckEnableAll:**

1: The interface acknowledges (in conjunction with an interrupt to the PC) all data frames and remote frames after successful transmission on the bus. This acknowledgment can be read in *CANL2_read_ac* or *CANL2_read_xmt_data* (see 1.4.2 and 1.4.3).

0: All objects whose data frames and remote frames are to be acknowledged by the Interface after successful transmission, must have been declared with the parameter *TransmitAckEnable*=1 in *CANL2_define_object*. Transmission of all other objects is not reported to the application.

- **TransmitAckFifoEnableAll:**

1: Acknowledgements of transmitted data frames or remote frames are transferred to the application through the transmit-acknowledge-FIFO (????see sections 1.4.2 and 1.4.3).

0: Acknowledgements of transmitted data frames or remote frames are ascertained by polling of the objects in the interface (see sections 1.4.2 and 1.4.3). Under certain conditions this can cause a longer running time of the function *CANL2_read_ac* and thus lead to lower throughput rates of the interface.

- **TransmitRmtFifoEnableAll:**

This parameter selects the handling mechanism for objects with Auto Remote Control configured (*AutoRemoteEnable* is set).

1: Incoming remote frames are buffered in a FIFO and are passed on for transmission of data frames (see sections 1.4.2 and 1.4.3").

0: Incoming remote frames are stored in object lists, which are polled for transmission of data frames (see section 1.4.3").

Function Return Codes:

0: Function successful

-1: Function not successful

-102: Wrong parameter

-104: Timeout firmware communication

-1000: Invalid channel handle

1.6.15 CANL2_define_object

```
int CANL2_define_object(
        CAN_HANDLE    can,
        unsigned long Ident,
        int           *ObjectNumber,
        int           Type,
        int           ReceiveIntEnable,
        int           AutoRemoteEnable,
        int           TransmitAckEnable)
```

The function *CANL2_define_object* defines and configures the communication objects of the transmit and receive object lists in object buffer mode.

In dynamic object buffer mode all used objects have to be defined, while in static object buffer mode the function can be used optionally for individual configuration of the object handling.

In static object buffer mode the returned object number equals the identifier. But in dynamic object buffer mode it corresponds to the succession of definition in the related object list.

NOTE:
The API functions handle the objects by their object number. Hence, the user is recommended to setup a table of relations between identifier and object number in dynamic object buffer mode.

Function Parameters:

- can:

channel handle (see 1.6.3) .

- Ident:

Identifier

[0 to 7FF$_{Hex}$] for standard objects
[0 to 1FFFFFFF$_{Hex}$] for extended objects

- ObjectNumber:

In the mode dynamic object buffer the object number in the related object list is returned in this parameter. It is a handle for the online access to this object (*CANL2_send_object*, *CANL2_read_rcv_data*...).

The identifier itself will no longer be referenced. In the mode static object buffer the object number is equally to the identifier.

- Type:

Direction of transmission and type of identifier

0: Standard receive object: Data frames and remote frames with standard identifiers (11 bit) can be received.

1: Standard transmit object: Data frames and remote frames with standard identifiers (11 bit) can be transmitted.

2: Extended receive object: Data frames and remote frames with extended identifiers (29 bit) can be received.

3: Extended transmit object: Data frames and remote frames with extended identifiers (29 bit) can be transmitted.

- ReceiveIntEnable (only for receive objects):

1: When receiving an object with the identifier *Ident* the receive **message** is passed to the PC application. Additionally, an interrupt is generated to the PC. The application program can read the object using *CANL2_read_ac*.

0: After receipt of an object the object data are indeed entered into object buffer (and they can be read using *CANL2_read_rcv_data*), but no information is generated for the application regarding receipt of the object. No interrupt is generated to the PC.

- AutoRemoteEnable (only for receive objects):

1: When receiving a remote frame with the identifier *Ident* the firmware transmits a data frame with the same identifier independently form the PC (see sections 1.4.2 and 1.4.3).

0: When receiving a remote frame the remote frame is reported to the PC (can be read using *CANL2_read_ac* or *CANL2_read_rcv_data*). The PC must transmit an explicit response (data frame).

NOTE:
The remote frame is only answered automatically after the first call of *CANL2_supply_object_data* or *CANL2_write_object*. This assures that no non-initialized data are transmitted. A remote frame arriving before the first call of *CANL2_supply_object_data* or *CANL2_write_object* results in error report -115 in the function *CANL2_read_ac*. For the auto remote feature it is necessary to define a transmit object as well as a receive object with the same identifier.

- TransmitAckEnable (only for transmit objects):

1 A data frame or remote frame with the identifier *Ident* is acknowledged (in conjunction with an interrupt to the PC) after successful transmission. This acknowledgement can be read using *CANL2_read_ac* or *CANL2_read_xmt_data* (see 5.1.2 and 5.1.3).

0: A data frame or remote frame with the identifier *Ident* is not acknowledged to the application after successful transmission on the bus.

NOTE:
Please note that the objects defined first are also polled first, and in this way a higher priority and a lower polling time is maintained relative to the objects that follow. It is sensible to define objects in the sequence of their identifiers in order to make prioritization of objects with low identifiers the same as on the CAN bus. This is true for static as well as for dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-109	Dyn. Obj. buffer mode not enabled
-1000:	Invalid channel handle

1.6.16 CANL2_start_chip

int CANL2_start_chip(CAN_HANDLE can)

The function *CANL2_start_chip* puts the CAN controller into operational mode. From now on transmit jobs can be issued and reception of messages is monitored.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.17 CANL2_define_cyclic

```
int CANL2_define_cyclic(
        CAN_HANDLE   can,
        int          ObjectNumber,
        unsigned int Rate,
        unsigned int Cycles)
```

The function *CANL2_define_cyclic* defines cyclic transmission of a communication object for the CAN channel which was previously defined by *CANL2_define_object*.

The cyclic transmission is started and stopped by the value of *Rate.* The settings (transmission start/stop) are put into operation by the first call of *CANL2_send_object* or *CANL2_write_object* for the object after the definition call.

Alternatively, the cyclic transmission is stopped automatically if the defined number of cycles *Cycles* is reached.

NOTE:
If defined and started a cyclic object has to be stopped before any succeeding redefinition. Redefinition of the cycle rate while running the transmission results in faulty transmission.

The transmitted data contents are defined by *CANL2_supply_object* or *CANL2_write_object*. They can be modified during cyclic transmission as well.

NOTE:
This function can only be used in dynamic object buffer mode.

Function Parameters:

- can:

  channel handle (see 1.6.3).

- ObjectNumber:

  Object reference returned by *CANL2_define_object*.

- Cycles [0..65535]:

0:	Unlimited cyclic repetition
1..65535:	Number of cyclic repetitions

- Rate [0..65535]:

0:	Disable cyclic transmission (stop)
1..65535:	Transmission rate in ms

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.18 CANL2_send_remote_object

```
int CANL2_send_remote_object(
        CAN_HANDLE  can,
        int         ObjectNumber,
        int         DataLength)
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	Object number
- DataLength:	Number of data bytes

This function initiates transmission of a remote frame for a transmit object specified by the object number. The remote frame has a data length 0; however, the data length code is physically transmitted with the data length code *DataLength*.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-110	Last request still pending
-116	Transmit request FIFO overrun
-1000:	Invalid channel handle

1.6.19 CANL2_supply_object_data

```
int CANL2_supply_object_data(
        CAN_HANDLE  can,
        int         ObjectNumber,
        int         DataLength,
        byte        *pData)
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	Object number
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be transmitted

This function enters current data into the object buffer of the transmit object specified by *ObjectNumber*.

The data are not transmitted directly onto the bus, but rather they are prepared for pickup by a remote frame (Auto Remote) or a later transmit job (later: *CANL2_send_object*).

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-115	Object is not defined
-1000:	Invalid channel handle

1.6.20 CANL2_supply_rcv_object_data

```
int CANL2_supply_rcv_object_data(
        CAN_HANDLE  can,
        int         ObjectNumber,
        int         DataLength,
        byte        *pData),
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be written in the object buffer

This function enters new data into the object buffer of the specified receive object.

This function can be used for initialization of receive objects in order to get reasonable values even before the first reception of a respective data frame took place.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it’s equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-115	Object is not defined
-1000:	Invalid channel handle

1.6.21 CANL2_send_object

```
int CANL2_send_object(
        CAN_HANDLE  can,
        int         ObjectNumber,
        int         DataLength),
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes to be transmitted

This function transmits a data frame for the transmit object specified by *ObjectNumber*. The data frame has a length of *DataLength* bytes. The data transmitted are the last entered into transmit object buffer using *CANL2_supply_object_data* or *CANL2_write_object*.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO to be further processed. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see 1.4.2 and 1.4.3). *CANL2_send_object* transmits a data frame on CAN channel 1, *CANL2_send_object2* transmits a data frame on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-116	Transmit request FIFO overrun
-1000:	Invalid channel handle

1.6.22 CANL2_write_object

```
int CANL2_write_object(
          CAN_HANDLE  can,
          int         ObjectNumber,
          int         DataLength,
          byte        *pData),
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be transmitted

This function performs an update of the data in the object buffer of the transmit object specified by *ObjectNumber*. Then a data frame is transmitted with *DataLength* bytes.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO to be further processed. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by CANL2_define_object. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-115	Object is not defined
-116	Transmit request FIFO overrun
-1000:	Invalid channel handle

1.6.23 CANL2_read_rcv_data

```
int CANL2_read_rcv_data(
        CAN_HANDLE     can,
        int            ObjectNumber,
        byte           *pRCV_Data,
        unsigned long  *Time)
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	Object number
- pRCV_Data:	Pointer to the address field of data being received
- Time:	Pointer to a time stamp parameter

This function copies the data of the receive object specified by *ObjectNumber* to the address *pRCV_Data*. The data are read, even if no new data were received. 8 data bytes are always copied to *pRCV_Data*, independent of the length of the received data frame.

If data in the object buffer are overwritten before they were read by the application or a remote request is not read quickly enough an overrun is signaled to the application by the function return code (overrun in object buffer).

If a remote frame was received the user is informed by a specific return code.

Time returns the instant of the last received data with a resolution of 1 microsecond (time stamp is reset in *CANL2_start_chip*).

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	No new data received
1:	Data frame received
2:	Remote frame received
-104:	Timeout firmware communication
-111:	Receive data frame overrun
-112:	Receive remote frame overrun
-113:	Object is undefined
-1000:	Invalid channel handle

1.6.24 CANL2_read_xmt_data

```
int CANL2_read_xmt_data(
        CAN_HANDLE  can,
        int         ObjectNumber,
        int         *pDataLength,
        byte        *pXMT_Data),
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- ObjectNumber:	ObjectNumber
- pDataLength:	Pointer to entry of number of transmitted data bytes
- pXMT_Data:	Pointer to the address field of data to be transmitted

This function reads the data and the initialized data length of the transmit object specified by *ObjectNumber*. Further, it checks whether a frame has been transmitted for this object.

If no transmission acknowledgments are returned by the object the function return code 1 indicates that the last transmit job was acknowledged by another CAN node. The return code -1 means that the last transmission acknowledgment has not been read by the application yet.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	No message was transmitted
1:	Message was transmitted
-104:	Timeout firmware communication
-114:	Transmit acknowledge overrun
-1000:	Invalid channel handle

1.6.25 CANL2_send_data

```
int CANL2_send_data(
        CAN_HANDLE     can,
        unsigned long  Ident,
        int            Xtd,
        int            DataLength,
        byte           *pData)
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- Ident:	Identifier
- Xtd:	Identifier length 0: Standard Identifier 1: Extended Identifier
- DataLength:	Number of data bytes to be transmitted
- pData:	Pointer to the address field of the data

This function transmits a data frame with the passed parameters on the CAN channel.

The transmit request is processed through the transmit FIFO. If the FIFO is full the application is informed by the return value.

NOTE:
The function *CANL2_send_data* can only be used in FIFO mode, not in object buffer mode. The function *CANL2_send_data2* can only be used in FIFO mode or static object buffer mode, not in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.26 CANL2_send_remote

```
int CANL2_send_remote(
        CAN_HANDLE     can,
        unsigned long  Ident,
        int            Xtd,
        int            DataLength)
```

Function Parameters:

- can:	Channel handle (see 1.6.3)
- Ident:	Identifier
- Xtd:	Identifier length 0: Standard Identifier 1: Extended Identifier
- DataLength:	Number of data bytes requested remote

This function transmits a remote frame with the Identifier *Ident* on the CAN channel. The remote frame has data length 0; however, the data length specified by the parameter *DataLength* is transmitted in the DLC field of the remote frame.

The transmit request is processed through the transmit FIFO. If the FIFO is full the application is informed by the return value.

NOTE:
The function *CANL2_send_remote* can only be used in FIFO mode, not in object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.27 CANL2_read_ac

```
int CANL2_read_ac(

        CAN_HANDLE  can,
        PARAM_STRUCT *ac_param)
```

By calling this function the application is informed about data transmission and reception as well as about various error conditions and bus events.

Several different CAN events can be distinguished by evaluation of the function return code (see Table 1.6-8). Certain information and parameters of interest are transferred in the elements of the parameter structure *PARAM_STRUCT*.

Parameters:

- **Can:**

channel handle (see 1.6.3)

Elements of structure PARAM_STRUCT:

NOTE:
RC1 through RC12 in brackets specify the function return codes of *CANL2_read_ac* for which the described parameter is valid. The application should not evaluate the parameter if it comes with a different function return code than stated below.

- **unsigned long Ident:**

Identifier (FIFO mode) or object number (object buffer mode) of the data or remote frame which was received or successfully transmitted.

(RC1, RC2, RC3, RC8, RC9, RC10, RC11, RC12)

- **int DataLength:**

Number of received (RC1, RC9) or transmitted (RC3, RC10) data bytes.

The *DataLength* of the received frame is only valid in FIFO mode and should not be used in object buffer mode. In object buffer mode the data length of the CAN messages should be predefined by the project.

- **int RecOverrun_flag:**

The last received data of object *Ident* were not read by the PC and were overwritten by the new data (RC1, RC2, RC9, RC12). Only valid in object buffer mode.

- **int RCV_fifo_lost_msg:**

Number of lost messages in receive FIFO (RC1, RC2, RC8, RC9, RC11, RC12). Only valid in FIFO mode.

- **byte RCV_data[8]:**

Data bytes of the received data frame (RC1, RC9) and in FIFO Mode (RC1, RC9, RC3, RC10).

- **int AckOverrunFlag:**

This flag is set if an unread transmit acknowledge for a transmit object is overwritten by a new one (RC3, RC10). Only valid in object buffer mode.

- **int XMT_ack_fifo_lost_acks**:

Number of lost acknowledges messages in transmit-acknowledge-FIFO in object buffer mode due to FIFO overrun(RC3, RC10).

Only valid in mode object buffer configured with *TransmitAckFifoEnable*=1.

- **int XMT_rmt_fifo_lost_remotes:**

Number of lost jobs in remote transmit FIFO (RC4). Only valid in object buffer mode initialized with *TransmitRmtFifoEnable*=1.

- **int Bus_state:**

Returns the new CAN bus status if a status change occurred (RC5).

0:	error active
1:	error passive
2:	bus off

- **int Error_state:**

Not used with:
CANcard2, CAN-ACx-PCI, CAN-ACx-104, CANusb.

With CAN-PROx-PCIe, CAN-PROx-104+:

	MSB			LSB
Bits	31..24	23..16	15..9	7..0
Value	Receive Error Counter	Transmit Error Counter	not used	only valid with Lowspeed Module: ERROR signal was active since the last call to CANL2_read_ac , if this value is unequal to 0.

- **int can:**

channel handle (see 1.6.3)

(RC1, RC2, RC3, RC4, RC5, RC7, RC8, RC9, RC10, RC11, RC12,RC15)

- **unsigned long Time:**

Time stamp of signaled events with a resolution of 1µs. The timer is reset in *CANL2_start_chip*. (RC1, RC2, RC9, RC12, RC3, RC5, RC8, RC10, RC11, RC15)

Table 1.6-8: Function return codes of CANL2_read_ac

FRC	**Explanation**
0:	No new event
1:	Standard data frame received
2:	Standard remote frame received
3:	Transmission of a standard data frame is confirmed
4:	Overrun of the remote transmit FIFO. Only with object buffer and auto remote feature.
5:	Change of bus status
6:	not implemented
7:	Not used
8:	Transmission of a standard remote frame is confirmed.
9:	Extended data frame received
10:	Transmission of an extended data frame is confirmed
11:	Transmission of an extended remote frame is confirmed
12:	Extended remote frame received
13, 14	Not valid. Only useful with CANcard API
15:	Error frame detected
-1:	Function not successful
-104:	Timeout firmware communication
- 115:	access to an abject denied, because the object has not been initialized with data using CANL2_supply_object()
-1000:	Invalid channel handle

1.6.28 CANL2_reinitialize

int CANL2_reinitialize(CAN_HANDLE can);

CANL2_reinitialize stops the current online operation of the specified CAN channel. Afterwards the operating mode and all corresponding parameters can be set anew (see Fig. 1-5, 1-6, 1-7).

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.29 CANL2_get_time

int CANL2_get_time(CAN_HANDLE can, unsigned long *time);

Function Parameters:

- can:	Channel handle (see 1.6.3).
- time:	Time (32bit) in µs

CANL2_get_time returns the 32bit time from the onboard timer of the Softing CAN Interface card in the parameter *time*. The unit of *time* is µs.

The timer is reset by *CANL2_reset_chip*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.30 CANL2_get_bus_state

int CANL2_get_bus_state(CAN_HANDLE can);

Function Parameters:

- can:	Channel handle (see 1.6.3)

CANL2_get_bus_state returns the current bus status of the CAN controller.

If the CAN controller is in bus off state it must be reset and started again to enable further access to the bus.

Function Return Codes:

0:	Error active
1:	Error passive
2:	Bus off
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.31 CANL2_reset_lost_msg_counter

int CANL2_reset_lost_msg_counter(CAN_HANDLE can);

CANL2_reset_lost_msg_counter resets the counter for the receive messages which were lost while the receive FIFO remained full in FIFO mode.

The lost message counter is supplied in the parameter structure of *CANL2_read_ac*.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0: Function successful

-1: Function not successful

-104: Timeout firmware communication

-1000: Invalid channel handle

1.6.32 CANL2_read_rcv_fifo_level

int CANL2_read_rcv_fifo_level(CAN_HANDLE can);

CANL2_read_rcv_fifo_level returns the number of events in the receive FIFO waiting to be read by *CANL2_read_ac*.

The FIFO level can be reset to 0 by *CANL2_reset_rcv_fifo* which clears the FIFO.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0 ... n:	Messages in receive FIFO
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.33 CANL2_reset_rcv_fifo

int CANL2_reset_rcv_fifo(CAN_HANDLE can);

CANL2_reset_rcv_fifo resets the receive fifo in FIFO mode.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.34 CANL2_read_xmt_fifo_level

int CANL2_read_xmt_fifo_level(CAN_HANDLE can);

CANL2_read_xmt_fifo_level returns the number of transmit jobs in the transmit FIFO waiting to be transmitted by the interface.

A pending transmission request which is already entered into the transmit buffer of the CAN controller is not counted.

The FIFO level can be reset to 0 by *CANL2_reset_xmt_fifo* which clears the FIFO.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0 ... n:	Jobs in transmit FIFO
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.35 CANL2_reset_xmt_fifo

int CANL2_reset_xmt_fifo(CAN_HANDLE can);

CANL2_reset_xmt_fifo resets the transmit FIFO in FIFO mode.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.36 CANL2_set_interrupt_event

```
CANL2_set_interrupt_event(
          CAN_HANDLE  can,
          HANDLE      InterruptEvent)
```

This function gives a HANDLE (pointer) of a Windows event to the driver which is set if an interrupt is signaled to the PC by the Softing CAN Interface card.

The event must be created beforehand by the application with *CreateEvent* which is a function of the Windows API and returns the required HANDLE. This Windows event can be used to control the processing of a Windows process or thread.

For more detailed information about the interrupt handling refer to Chapter 1.6.2.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-1000:	Invalid channel handle

1.6.37 CANL2_init_signals

```
int CANL2_init_signals(
       CAN_HANDLE              can,
       unsigned long           ulChannelDirectionMask,
       unsigned long           ulChannelOutputDefaults)
```

NOTE
This function is for use with the Softing CAN lowspeed module only! If no lowspeed module is installed, then this function is not needed.

1.6.37.1 CAN Lowspeed module overview

Via CAN API, all status and control signals from and to the module plug-in stations are made available for the user’s convenience. In this manner, the user has all features of the CAN Lowspeed Transceivers under his control; he can freely determine the change of operating modes between CAN Highspeed and CAN Lowspeed and read in the identifier of the module, if necessary.

After a RESET of the CAN Interface card, the operating mode CAN Highspeed is set by default in the CAN Lowspeed module. To permit access to the status and control signals and switch over to the CAN Lowspeed mode, the possibility of access to the signals of the module plug-in stations must first be initialized. The read-in of the status signals and switching of the control signals then takes place via read and write functions.

The table 1.6.36 gives a survey of the configurations to be selected during initialization of the read and write functions and of the signal bit assignment to the connection pins of the module plug-in stations and to the signals arriving there.

During reset, the signals are initialized in such a way that CAN highspeed is selected and the two LS Transceivers are in the sleep mode

The function initializes the direction of the signal and defines it for further use.

1.6.37.2 How to use CANL2_init_signals

The parameter can is the CAN handle. (see 1.6.3)

For operation of the CAN LS modules, the bits of the parameter ”ulChannelDirectionMask” must be selected according to table 1.6.36. The following applies to every bit position:

1 defines the signal direction as ”output”

0 defines the signal direction as ”input”.

NOTE
Ports exclusively for input cannot be defined as outputs. Unassigned bit positions can be defined as desired. ”Input” and ”output” are defined from the position of the microprocessor C165 or XC161.

Operation of the CAN Lowspeed Module:

The parameter "ulChannelOutputDefaults" indicates the default status at the output for the bit positions for which the signal direction ”output” has been defined via ”ulChannelDirectionMask". (For the settings, see table 1.6.36;)

This table supplies information on the reset status = inactive transceivers; the default status may deviate.)

1 defines the default level "high" for an ”output”.

0 defines the default level "low" for an ”output”.

The values of the bit positions for which the signal direction ”input” has been determined, are irrelevant.

This function must be called before CANL2_write_signals() and CANL2_read_signals(), to setup the port pins of the microcontroller on the CAN Interface board.

Function Return Codes:

0:	Function successful
-1:	Error: signals have already been initialized
-2:	An exclusive input port has been defined as output.
-104:	Timeout firmware communication
-1000:	Invalid channel handle

NOTE
Execution of the initialization function CANL2_init_signals() is permitted only once after loading of the firmware into the Softing CAN Interface. Reinitialization of the status and control signals is possible only

The following table shows the *Signal bit assignment to control and status signals*

Bit pos.	**Direction of the signal**	**Reset default status**	**Description of the signal**
0	**Out->**	**H**	LS Transceiver signal: EN (signal inverted, resetlevel = L)
1	**Out->**	**H**	LS Transceiver signal:/STB (signal inverted, reset level=L)
2	**Out->**	**H**	HS/LS switchover (H = Highspeed)
3	**Out->**	**H**	LS Transceiversignal: /WAKE
4	**In <-**	**H**	LS Transceiver signal: NERR
5	**In <-**	**H**	Module identifier bit 0=1
6	**In <-**	**L**	Module identifier bit 1=0
7	**In <-**	**L**	Module identifier bit 2=0

Table 1.6.36

Example:

Initialisation of the lowspeed module:

```
CANL2_init_signals(can, 0x0000000F, 0x00000004);
```

Switching to CAN lowspeed:

```
CANL2_write_signals(can, 0x00000000, 0x00000004);
```

Switching back to CAN highspeed:

```
CANL2_write_signals(can, 0x00000004, 0x00000004);
```

1.6.38 CANL2_read_signals

```
CANPC_read_signals(
        CAN_HANDLE          can,
        unsigned long *ulChannelRead)
```

NOTE
This function is for use with the Softing CAN lowspeed module only! If no lowspeed module is installed, then this function is not needed.

The function reads in the current signal statuses. The parameter ”ulChannelRead” is coded according to table **1.6.36**.

The following applies to every bit position:

Operation of the CAN Lowspeed Module

1 means that the signal level is ”high”.

0 means that the signal level is ”low”.

If a signal has been defined as output, the output is read back –if possible- or the value set and buffered last is returned.

Unassigned bit positions are evaluated at ”0”.

The function is useful to detect, whether a lowspeed piggyback is installed on the hardware. If a lowspeed piggyback is plugged, then the module identifier bit “bit0” is 1, and the module identifier bits “bit1” and “bit2” are 0. (see table 1.6.36)

The parameter can is the CAN handle. (see 1.6.3)

Function Return Codes:

0:	Function successful
-1:	Error: signals have not yet been initialized
-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.6.39 CANL2_write_signals

```
int CANPC_write_signals(
    CAN_HANDLE       can,
    unsigned long    ulChannelWrite,
    unsigned long    ulCareMask)
```

NOTE
This function is for use with the Softing CAN lowspeed module only! If no lowspeed module is installed, then this function is not needed.

The function sets output signals according to the parameter definition.

The parameter can is the CAN handle. (see 1.6.3)

The parameters ”ulChannelWrite” and ”ulCareMask” are coded according to table 1.6.36.

Signals set at ”0” in the ”ulCareMask” are ignored. Only those signals are written which are set at ”1” in the ”ulCareMask” parameter. For these signals, the parameter ”ulChannelWrite” is evaluated and the following applies:

1 means that the signal level is set at ”high”.

0 means that the signal level is set at ”low”.

A write access to an unassigned bit position is ignored.

Function Return Codes:

0:	Function successful
-1:	Error: signals have not yet been initialized, this must be done by using CANL2_init_signals()
-2:	Error: write access to an input signal
-104:	Timeout firmware communication
-1000:	Invalid channel handle

Example: Lowspeed/Highspeed switchover:

Switching to CAN lowspeed:

```
CANL2_write_signals(can, 0x00000000, 0x00000004);
```

Switching back to CAN highspeed:

```
CANL2_write_signals(can, 0x00000004, 0x00000004);
```

1.6.40 INIL2_close_channel

int INIL2_close_channel(CAN_HANDLE can)

This function releases and unlocks the system resources which were allocated by *INIL2_initialize_channel*.

The function call should be applied at any possible application exit after successful call to *INIL2_initialize_channel*. Otherwise, the application may have problems to get the handle to the DPRAM a second time without system exit (e.g. applications with LabVIEW a.o.).

Parameters:

can	Channel handle (see 1.6.3)

Function Return Codes:

0:	Function successful
-1000:	Invalid channel handle

1.7 Description of the Softing CAN class library

1.7.1 Interrupt processing

1.7.1.1 Interrupt events

For many applications it is useful to be informed by interrupt about occurrence of CAN events. Otherwise, the Softing CAN Interface cards must be polled for new events which requires more PC processor time.

The firmware triggers a hardware interrupt to the PC on the following CAN events:

- Reception of data, remote and error frames
- Acknowledge on successful transmissions if enabled
- Change of bus state

1.7.1.2 .Net interrupt programming

The class library provides an event which can be used for triggering a delegate function.

The name of the event is: CANL2Channel::CANEvent

The delegate function must have the following format:

Visual Basic:

```
Public Sub OnCanEvent(ByVal param As PARAM_CLASS,
                      ByVal type As Integer)
```

C#:

```
static public void
OnCanEvent(PARAM_CLASS param, int type)
```

Members of PARAM_CLASS:

NOTE:
RC1 through RC12 in brackets specify the event type code for which the described parameter is valid. The application should not evaluate the parameter if it comes with a different event type code than stated below.

- **unsigned long Ident:**

Identifier (FIFO mode) or object number (object buffer mode) of the data or remote frame which was received or successfully transmitted.

(RC1, RC2, RC3, RC8, RC9, RC10, RC11, RC12)

- **int DataLength:**

Number of received (RC1, RC9) or transmitted (RC3, RC10) data bytes.

The *DataLength* of the received frame is only valid in FIFO mode and should not be used in object buffer mode. In object buffer mode the data length of the CAN messages should be predefined by the project.

- **int RecOverrun_flag:**

The last received data of object *Ident* were not read by the PC and were overwritten by the new data (RC1, RC2, RC9, RC12). Only valid in object buffer mode.

- **int RCV_fifo_lost_msg:**

Number of lost messages in receive FIFO (RC1, RC2, RC8, RC9, RC11, RC12). Only valid in FIFO mode.

- **byte RCV_data[8]:**

Data bytes of the received data frame (RC1, RC9).

- **int AckOverrunFlag:**

This flag is set if an unread transmit acknowledge for a transmit object is overwritten by a new one (RC3, RC10). Only valid in object buffer mode.

- **int XMT_ack_fifo_lost_acks**:

Number of lost acknowledges messages in transmit-acknowledge-FIFO in object buffer mode due to FIFO overrun(RC3, RC10).

Only valid in mode object buffer configured with *TransmitAckFifoEnable*=1.

- **int XMT_rmt_fifo_lost_remotes:**

Number of lost jobs in remote transmit FIFO (RC4). Only valid in object buffer mode initialized with *TransmitRmtFifoEnable*=1.

- **int Bus_state:**

Returns the new CAN bus status if a status change occurred (RC5).

0:	error active
1:	error passive
2:	bus off

- **int Error_state:**

Not used. Only for conformity to CANcard and CAN-AC2 (ISA) API.

- **int can:**

Number of CAN channel where the event occurred which is defined by the function return code.

(RC1, RC2, RC3, RC4, RC5, RC7, RC8, RC9, RC10, RC11, RC12,RC15)

- **unsigned long Time:**

Time stamp of signaled events with a resolution of 1µs. The timer is reset in *CANL2_start_chip*. (RC1, RC2, RC9, RC12, RC3, RC5, RC8, RC10, RC11, RC15)

Table 1.7-8: event types (possible values of the type parameter)

event	**Explanation**
0:	No new event
1:	Standard data frame received
2:	Standard remote frame received
3:	Transmission of a standard data frame is confirmed
4:	Overrun of the remote transmit FIFO. Only with object buffer and auto remote feature.
5:	Change of bus status
6:	not implemented
7:	Not used
8:	Transmission of a standard remote frame is confirmed.
9:	Extended data frame received
10:	Transmission of an extended data frame is confirmed
11:	Transmission of an extended remote frame is confirmed
12:	Extended remote frame received
13, 14	Only useful with CANcard
15:	Error frame detected
-1:	Function not successful
-3:	Error accessing DPRAM
-4:	Timeout firmware communication
-6:	access to an abject denied, because the object has not been initialized with data using CANL2_supply_object()
-99:	Board not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

To install a delegate function you must only add the following line to your initialization routine (if your instance is called “channel” and your delegate function is called “OnCanEvent”)

Visual Basic style:

```
AddHandler channel.CANEvent, AddressOf OnCanEvent
```

C# style:

```
channel.CANEvent +=
new CANL2Channel.CANEventDelegate(OnCanEvent);
```

1.7.2 CANL2Channel::INIL2_initialize_channel

Visual Basic style:

Public Function
INIL2_initialize_channel(channel_name As String) As Integer

C# style:

int INIL2_initialize_channel(string channel_name)

Function Parameters:

Type/Name	Description
String channel_name	IN: Name of the channel, which should be used; must be set by the application;

INIL2_initialize_channel enables the memory access to the DPRAM of the Softing CAN Interface cards. Thus, it is necessarily called before any other API function.

If the DPRAM access is denied the function returns an error code which corresponds to the error cause.

NOTE:
If *CANL2_initialize_channel* fails, other API functions should not be called since the non-initialized memory handle may cause an access violation.

Function Return Codes:

0	Initialization successful
-536215551	Internal Error
-536215550	General Error
-536215546	Illegal driver call
-536215542	Driver not loaded / not installed, or device is not plugged.
-536215541	Out of memory
-536215531	An error occurred while hooking the interrupt service routine
-536215523	Device not found
-536215522	Can not get a free address region for DPRAM from system
-536215521	Error while accessing hardware
-536215519	Can not access the DPRAM memory
-536215516	Interrupt does not work/Interrupt test failed!
-536215514	Device is already open
-536215512	An incompatible firmware is running on that device (CANalyzer/CANopen/DeviceNet firmware)
-536215511	Channel can not be accessed, because it is not open
-536215500	Error while calling a Windows function
-1002	Too many open channels.
-1003	Wrong DLL or driver version.
-1004	Error while loading the firmware. (This may be a DPRAM access error)
-1	Function not successful

Error codes which can only occur with the CANusb interface:

-602	Unable to open USB pipe
-603	Communication via USB pipe broken
-604	No valid lookup table entry found
-611	CANusb framework initialization failed
-1005	CANusb DLL (CANusbM.dll) not found.

1.7.3 CANL2Channel::CANL2_get_all_CAN_channels

Visual Basic style:

```
Public Function
CANL2_get_all_CAN_channels(
        ByRef pu32NumOfChannels As UInteger,
        ByRef snapshot As CHDSNAPSHOT) As Integer
```

C# style:

```
int CANL2_get_all_CAN_channels (
        ref UInt32 pu32NumOfChannels,
        ref CHDSNAPSHOT[] snapshot)
```

This function provides information about the CAN channels in your computer. It writes the information into the elements of the array “snapshot”.

Function Parameters:

- **pu32NumOfChannels: (INOUT)**

This parameter must be initialized by the user with the number of elements in the array “ CHDSNAPSHOT[]”.

After the call of CANL2_get_all_CAN_channels the parameter holds the number of CAN channels in your Computer.

- **snapshot: (OUT)**

Array of the class CHDSNAPSHOT; It holds the information about the CAN channels after the function call. (Channel names, type of firmware, which is loaded on a specific channel, whether the channel is opened by an other application or not, serial number of the Softing CAN Interface, physical channel number [1 or 2])

Members of CHDSNAPSHOT:

- **u32Serial:**

Serial number of the CAN Interface (32 Bit unsigned integer)

- **u32DeviceType:**

Type of Softing CAN Interface: (32 Bit unsigned integer)

u32DeviceType	Softing CAN Interface
5	CANcard2
7	CAN-Acx-PCI
8	CAN-Acx-PCI/DN
9	CAN-Acx-104
10	CANusb
13	CAN-PROx-PCIe
14	CAN-PROx-PC104+
15	CANpro USB
261	EDICcard2

- **u32PhysCh:** (32 Bit unsigned integer)

Physical channel number; The physical channel number of the first channel on an CAN interface with 2 channels is 1, the number of the second channel on the same interface is 2.

- **u32FwId:** (32 Bit unsigned integer)

Type of the firmware which is loaded on the channel; layer 2 firmware has the firmware id 0x01.

- **bIsOpen:** (Bool)

True if the channel is opened by a process, false otherwise.

- **ChannelName:** (String)) **[80 Bytes]**

The name of the channel.

Function Return Codes:

0:	Function successful
-1:	Function not successful

1.7.4 CANL2Channel::CANL2_set_rcv_fifo_size

Visual Basic style:

Public Function
CANL2_set_rcv_fifo_size (FifoSize As Integer) As Integer

C# style:

int CANL2_set_rcv_fifo_size (int FifoSize)

To accommodate to larger delays in processing the received messages the *Receive-FIFO* in FIFO mode is configurable in size for CANusb. *CANL2_set_rcv_fifo_size* must be called if other sizes than the default size of 255 entries is to be used.

NOTE:
This function is only applicable in FIFO mode. It is only supported by the Layer 2 API for CANusb.

Function Parameters:

- FifoSize:

The value defines the size of the *Receive-FIFO* as defined in the following table:

Value	No of entries
0	255 (default)
1	511
2	1023
3	2047
4	4095
5	8191
6	16383
7	32767
8	65535

Function Return Codes:

0: Function successful

-1: The FIFO size can not be changed, because the CAN controller is already online

-102: Parameter error

-1000: Invalid channel handle

1.7.5 CANL2Channel::CANL2_initialize_fifo_mode

Visual Basic style:

Public Function
CANL2_initialize_fifo_mode (
ByVal pUserCfg As L2CONFIG) As Integer

C# style:

int CANL2_initialize_fifo_mode(L2CONFIG pUserCfg)

This function provides a fast and easy way to initialize a CAN channel. After calling this function successfully sending and receiving CAN messages in FIFO mode is possible.

If you don’t know exactly the appropriate values of the members of L2CONFIG you can configure the channel by using the SCIM (Softing CAN Interface Manager) in the Settings of your PC. If you do not change the data members of the class L2CONFIG the values from the SCIM configuration will be used.

Function Parameters:

Table 1.7-2: CANL2_initialize_fifo_mode

Name	**Description**	**Range**
pUserCfg:	Instance of the class L2CONFIG	-

Elements of class L2CONFIG:

Element	Description	Range
Int32 s32Prescaler	Prescaler	[1..32] or (-1)
Int32 s32Tseg1	CAN Time segment 1	[1..16] or (-1)
Int32 s32Tseg2	CAN Time segment 2	[1..8] or (-1)
Int32 s32Sjw	Sync jump with	[1..4] or (-1)
Int32 s32Sam	Number of samples	[0..1] or (-1)
Int32 s32AccCodeStd	Acceptance code for 11 bit Identifier	Range of 11 bit Id or (-1)
Int32 s32AccMaskStd	Acceptance mask for 11 bit Identifier	Range of 11 bit Id or (-1)
Int32 s32AccCodeXtd	Acceptance code for 29 bit Identifier	Range of 29 bit Id or (-1)
Int32 s32AccMaskXtd	Acceptance mask for 29 bit Identifier	Range of 29 bit Id or (-1)
Int32 s32OutputCtrl	Output Control Register	[0..255] or (-1)
bEnableAck	Enable confirmation of successfully sent CAN message	TRUE or FALSE

bEnableErrorframe	Enable the detection of ERROR Frames on the bus.	TRUE or FALSE

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong Parameter
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.6 CANL2Channel::CANL2_reset_chip

Visual Basic style:

Public Function CANL2_ reset_chip() As Integer

C# style:

int CANL2_ reset_chip (void)

This function terminates a possible bus operation and places the CAN chip into reset status.

After reset the bit timing, acceptance register and output control register have to be defined before the CAN controller is started by the related API functions.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-108:	Wrong hardware; (CANcard2 or EDICcard2 with 25MHz instead of 24MHz)
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.7 CANL2Channel::CANL2_get_version

Visual Basic style:

```
Public Function
CANL2_get_version (
                ByRef  sw_version As Integer,
                ByRef  fw_version As Integer,
                ByRef  hw_version As Integer,
                ByRef  license As Integer,
                ByRef  can_chip_type As Integer) As Integer
```

C# style:

```
int CANL2_get_version(
       ref int           sw_version,
        ref int          fw_version,
        ref int          hw_version,
        ref int          license,
        ref int          can_chip_type);
```

This function provides useful information about the version numbers of hard-, soft- and firmware, license and CAN chip type of the CAN Interface.

Function Parameters:

- **sw_version:**

Pointer to the entry of the version number of driver software.

The number is encoded as *sw_version* / 100 as the main version number; *sw_version* % 100 refers to the subordinate part of the number.

- **fw_version:**

Pointer to the entry of the version number of the firmware.

The number is encoded as *fw_version* / 100 as the main version number; *fw_version* % 100 refers to the subordinate part of the number.

- **hw_version:**

Pointer to the entry of the version number of the hardware.

The number is encoded as *hw_version* % 0x100H as the main version number; *hw_version* / 0x100H refers to the subordinate part of the number.

- **licence:**

Pointer entry of the license type of the CAN-AC2-PCI

01H:	Licensed for operation with interface software
02H:	Licensed for operation with CANalyzer software

- **can_chip_type:**

Pointer to entry containing the last four digits of the CAN chip type.

can_chip_type:	CAN	
5:	NEC72005	(CANcard)
161:	Infineon XC161 TwinCAN controller	(CAN-PROx-PC104+) (CAN-PROx-PCIe)
1000:	SJA1000	(CANcard-SJA) (CAN-AC2/PCI)
527:	Intel 82527	(CAN-AC2/527)
200:	Philips 82C200	(CAN-AC2)

Function Return Codes:

0:	Function successful
-1:	Function not successful
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.8 CANL2Channel::CANL2_get_serial_number

Visual Basic style:

```
Public Function
CANL2_get_serial_number (
        ByRef SerialNumber As Integer) As Integer
```

C# style:

```
Int CANL2_get_serial_number(ref UInt32 SerialNumber)
```

This function returns the serial number of the Softing CAN Interface card in *SerialNumber*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.9 CANL2Channel::CANL2_initialize_chip

Visual Basic style:

```
Public Function
CANL2_initialize_chip (
                ByVal   presc As Integer,
                ByVal   sjw As Integer,
                ByVal   tseg1 As Integer,
                ByVal   tseg2 As Integer,
                ByVal   sam As Integer) As Integer
```

C# style:

```
int     CANL2_initialize_chip(
                int             presc,
                int             sjw,
                int             tseg1,
                int             tseg2,
                int             sam)
```

Function Parameters:

Table 1.7-2: Bit timing parameter

Name	Description	Range
presc:	CAN-Prescaler	[1..32] or -1
sjw:	CAN-Synchronisation-Jump-Width	[1..4] or (-1)
tseg1:	CAN-Time-Segment 1	[1..16] or (-1)
tseg2:	CAN-Time-Segment 2	[1..8] or (-1)
sam:	Number of samples	[0, 1] or (-1)

The function defines the bit timing (baud rate) of the CAN chip. Parameters *presc*, *sjw*, *tseg1* and *tseg2* represent logical values that are used to describe the bit timing. These values are converted and written to the bus timing register 1 and 2 of the Philips SJA1000.

The **baud rate** is calculated by the following formula, whereby certain limit conditions must be maintained:

$$\text{Baud rate} = \frac{f_{crystal}}{2 * presc * (1 + tseg1 + tseg2)}$$

The crystal frequency $f_{crystal}$ is 16 MHz.

The limitations of the bit timing of the used CAN controllers lead to following **conditions**:

$$8 \leq (1+ tseg1 + tseg2) \leq 25$$

$$tseg1 + tseg2 \geq 2 * sjw$$

$$tseg2 \geq sjw$$

The prescaler divides the crystal frequency by *presc* to build the clock cycle time Δt.

The parameter *sam* defines how many **samples** are taken to detect the bit level.

$sam = 0 \rightarrow$ 1 sample (high speed buses)

$sam = 1 \rightarrow$ 3 samples (low/medium speed buses)

The sampling point is defined at the edge between time segment 1 and time segment 2. It is recommended to place the sampling point between 50% and 80% of the bit time. At high baud rates the communication is more stable if the sample is taken in the last quarter of the bit time.

The synchronization jump width is used to compensate the time shifts between the different CAN nodes in the network. It defines the maximum number *sync* of clock cycles by which the time segment 1 may be lengthened and time segment 2 shorted during resynchronization.

NOTE
A parameter value of (-1) means that the API retrieves the used value from SCIM (Softing CAN Interface Manager).

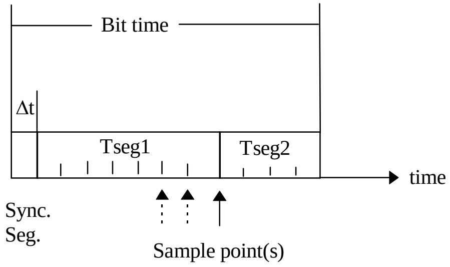


Fig. 1-8: Bit period

Table 1.7-3: Baud rate examples

baud rate	**presc**	**sjw**	**tseg1**	**tseg2**
1 Mbaud	1	1	4	3
800 kBaud	1	1	6	3
500 kBaud	1	1	8	7
250 kBaud	2	1	8	7
125 kBaud	4	1	8	7
100 kBaud	4	4	11	8
10 kBaud	32	4	16	8

Function Return Codes:

0:	Initialization successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.10 CANL2Channel::CANL2_set_output_control

Visual Basic style:

```
Public Function
CANL2_set_output_control (
                 ByVal OutputControl As Integer) As Integer
```

C# style:

```
int     CANL2_set_output_control   (
                 int                  OutputControl)
```

Function Parameters:

- OutputControl:	Input/Output-Control-Register [0 to $\mathrm{FF}_{\mathrm{Hex}}$ or -1]

This function defines the setting of the Output Control Register (OCR) of the CAN chip. This is used to adapt the CAN chip to the physical bus interface being used.

If the CAN controller Philips SJA1000 is used with the CAN High Speed interface (default) the output control register must be set to a value of $\mathrm{FB}_{\mathrm{Hex}}$. If you like to adapt the interface to a different bus physic consult the SJA data sheet for the required OCR setting.

Setting the OCR=03H switches off the transmission lines Tx0 and Tx1 of the CAN controller. Thus, the Softing CAN Interface card can’t send any data frame or any acknowledge bit on received messages. Thus, the interface can monitor the activities on the CAN network without influencing it.

The OCR specification of the Philips SJA1000 is described in Table 1.7-4 to 1.7-7.

The default values for CAN High Speed can also be chosen automatically by passing the default parameter -1 which assures compatibility with other using CAN High Speed Standard.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

Output control specification of Philips SJA1000:

Table 1.7-4: Output control Philips SJA1000

Bit	**Function**
7	OCTP1
6	OCTN1
5	OCPOL1
4	OCTP0
3	OCTN0
2	OCPOL0
1	OCMODE1
0	OCMODE0

Table 1.7-5: Output control mode of Philips SJA1000

OCMODE1	**OCMODE0**	**Function**
1	0	Normal Mode (TX0 and TX1 CAN Output)
1	1	Normal mode (Tx0 CAN Output, TX1 Bus Clock)
0	0	not implemented
0	1	not implemented

The voltage levels at the CAN outputs TX0 and TX1 depend on both the output configuration, which is determined by OCTPx and OCTNx, and the output polarity, which is determined by OCPOLx. Table 1.7-7 shows output status as a function of these settings for Philips SJA00.

Table 1.7-6: Configuration of CAN output pins TX0 and TX1

Operating Mode	**OCTPx**	**OCTNx**	**OCPOLx**	**TXD**	**TPx**	**TNx**	**Level at TXx**
FLOAT	0	0	0	0	off	off	high resistance
	0	0	0	1	off	off	high resistance
	0	0	1	0	off	off	high resistance
	0	0	1	1	off	off	high resistance
PULL DOWN	0	1	0	0	off	on	logic "0"
	0	1	0	1	off	off	high resistance
	0	1	1	0	off	off	high resistance
	0	1	1	1	off	on	logic "0"
PULL UP	1	0	0	0	off	off	high resistance
	1	0	0	1	on	off	logic "1"
	1	0	1	0	on	off	logic "1"
	1	0	1	1	off	off	high resistance
PUSH PULL	1	1	0	0	off	on	logic "0"
	1	1	0	1	on	off	logic "1"
	1	1	1	0	on	off	logic "1"
	1	1	1	1	off	on	logic "0"

TXx:	Output pin x, x=0 for TX0, x=1 for TX1
TPx:	Transistor that switches from supply voltage to TXx
TNx:	Transistor that switches from TXx to ground
TXD:	Data to be transmitted, 0=dominant, 1=recessive

1.7.11 CANL2Channel::CANL2_set_acceptance

Visual Basic style:

```
Public Function
CANL2_set_acceptance (
        ByVal AccCodeStd As UInteger,
        ByVal AccMaskStd As UInteger,
        ByVal AccCodeXtd As UInteger,
        ByVal AccMaskXtd As UInteger) As Integer
```

C# style:

```
int CANL2_set_acceptance(
        UInt32  AccCodeStd,
        UInt32  AccMaskStd,
        UInt32  AccCodeXtd,
        UInt32  AccMaskXtd)
```

Function Parameters:

Table 1.7-7: Filter parameters

Name	Description	Range
AccCodeStd:	Acceptance code for standard frames	[0 to $7FF_{Hex}$] or 0xFFFFFFFF
AccMaskStd:	Acceptance mask for standard frames	[0 to $7FF_{Hex}$] or 0xFFFFFFFF
AccCodeXtd:	Acceptance code for extended frames	[0 to $1FFFFFFF_{Hex}$] or 0xFFFFFFFF
AccMaskXtd:	Acceptance mask for extended frames	[0 to $1FFFFFFF_{Hex}$] or 0xFFFFFFFF

The function *CANL2_set_acceptance* initializes the acceptance filter of the CAN controller.

The acceptance filter defines which identifiers should be passed into the receive buffer of the CAN controller.

To receive an identifier all bits of the identifier that were initialized as 1 in the acceptance mask must match the corresponding bit in the acceptance code. A "0" in the acceptance mask register means "Don't care" for the identifier bit at this position.

The parameters are converted and written into the acceptance code and mask registers of the Philips SJA1000.

NOTE
A parameter value of 0xFFFFFFFF means that the API retrieves the used value from SCIM (Softing CAN Interface Manager).

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.12 CANL2Channel:: CANL2_enable_dyn_obj_buf

CANL2Channel::CANL2_enable_dyn_obj_buf

Visual Basic style:

Public Function CANL2_enable_dyn_obj_buf () As Integer

C# style:

int CANL2_enable_dyn_obj_buf(void)

CANL2_enable_dyn_obj_buf configures the API to run in dynamic object buffer mode (see section 1.4.2). If this function is not used, then the Softing CAN Interface card operates with the static object buffer or in the FIFO mode (if *CANL2_enable_fifo* has been called).

NOTE:
The static object buffer is usable for 29bit identifiers.

Parameters:

none	

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.13 CANL2Channel::CANL2_initialize_interface

Visual Basic style:

```
Public Function
CANL2_initialize_interface (
        ByVal ReceiveFifoEnable As Integer,
        ByVal ReceivePollAll As Integer,
        ByVal ReceiveEnableAll As Integer,
        ByVal ReceiveIntEnableAll As Integer,
        ByVal AutoRemoteEnableAll As Integer,
        ByVal TransmitReqFifoEnable As Integer,
        ByVal TransmitPollAll As Integer,
        ByVal TransmitAckEnableAll As Integer,
        ByVal TransmitAckFifoEnable As Integer,
        ByVal TransmitRmtFifoEnable As Integer) As Integer
```

C# style:

```
int CANL2_initialize_interface(

        int             ReceiveFifoEnable,
        int             ReceivePollAll,
        int             ReceiveEnableAll,
        int             ReceiveIntEnableAll,
        int             AutoRemoteEnableAll,
        int             TransmitReqFifoEnable,
        int             TransmitPollAll,
        int             TransmitAckEnableAll,
        int             TransmitAckFifoEnable,
        int             TransmitRmtFifoEnable)
```

CANL2_initialize_interface configures properties and structure the object buffer (see sections 1.4.2 and 1.4.3). It may not be used in FIFO operation.

- **ReceiveFifoEnable:**

Type of receive message handling from firmware to PC application.

1: Receive messages of data frames or remote frames are transferred to the PC through the receive message FIFO (see section 1.4.1).

0: The PC ascertains receive messages of data frames or remote frames by polling the objects in the receive object lists using the function *CANL2_read_ac* (see 5.1.2 and 5.1.3). Under certain conditions this can cause a longer running time of the function *CANL2_read_ac*, and can therefore result in lower throughput rates.

- **ReceivePollAll:**

This flag is only meaningful for *ReceiveFifoEnable* = 0 with static object buffer (should be 0 with dynamic object buffer).

1: Polling of all receive objects when *CANL2_read_ac* is called (see section 1.4.3)

0: Polling of only those receive objects which have been defined using *CANL2_define_object* (see Section 1.4.2 and 1.4.3)

- **ReceiveEnableAll:**

This flag is only meaningful with static object buffer (must be 0 with dynamic object buffer).

1: All data frames and remote frames with standard identifiers are received. No receive objects need to be defined (However: *CANL2_define_object* can be used nevertheless, in order to activate receive objects for polling by the application under the conditions *ReceivePollAll* = 0 and *ReceiveFifoEnable* = 0)

0: All receive objects that are passed to the PC must be defined beforehand using *CANL2_define_object*. Objects that are not defined using *CANL2_define_object* are not received by the (filter functionality).

- ReceiveIntEnableAll:

This flag is only meaningful while *ReceiveEnableAll* = 1 with static object buffer (should be 0 with dynamic object buffer).

1: When receiving an arbitrary object (declared using *CANL2_define_object* or if *ReceiveEnableAll* = 1) the receive message is passed to the PC application. Additionally, an interrupt is generated to the PC. The application program can read the object using *CANL2_read_ac*.

0: Receipt of an object is only reported to the PC (with interrupt) if the object has been declared in *CANL2_define_object* with *ReceiveIntEnable* = 1. Otherwise the data of the object will indeed be entered into object buffer (and they can be read using *CANL2_read_rcv_data*), but no information is generated for the application regarding receipt of the object (readable by *CANL2_read_ac*).

- AutoRemoteEnableAll:

This flag is only meaningful while *ReceiveEnableAll* = 1 with static object buffer (should be 0 with dynamic object buffer).

1: When receiving an arbitrary remote frame the interface independently transmits a data frame with the same identifier (see 5.1.3).

0: When receiving a remote frame the interface only transmits a data frame with the same identifier if the corresponding receive object has been declared in *CANL2_define_object* with *AutoRemoteEnable* = 1. Otherwise the remote frame is reported to the PC (calling *CANL2_read_ac* or *CANL2_read_rcv_data*). The PC must transmit an explicit response (data frame) then.

NOTE:
A data frame is transmitted after the first call of *CANL2_supply_object_data* or *CANL2_write_object* initialized the object data. A remote frame arriving before data initialization results in error report -6 in the function *CANL2_read_ac*.

- **TransmitReqFifoEnable:**

1:	Transmit jobs for data frames or remote frames are transferred to the CAN bus through the transmit job FIFO (see sections 1.4.2 and 1.4.3)
0:	Transmit jobs for data frames or remote frames are ascertained by the firmware by polling the objects in the transmit object lists (see sections 1.4.2 and 1.4.3).

- **TransmitPollAll:**

This flag is only meaningful for *TransmitReqFifoEnable* = 0 with static object buffer (should be 0 with dynamic object buffer).

1	Polling of all transmit objects (see section 1.4.3)
0:	Polling of only those transmit objects that have been defined using *CANL2_define_object* (see section 1.4.2 and 1.4.3)

- **TransmitAckEnableAll:**

1: The interface acknowledges (in conjunction with an interrupt to the PC) all data frames and remote frames after successful transmission on the bus. This acknowledgment can be read in *CANL2_read_ac* or *CANL2_read_xmt_data* (see section 1.4.2 and 1.4.3).

0: All objects whose data frames and remote frames are to be acknowledged by the Interface after successful transmission, must have been declared with the parameter *TransmitAckEnable*=1 in *CANL2_define_object*. Transmission of all other objects is not reported to the application.

- **TransmitAckFifoEnableAll:**

1: Acknowledgements of transmitted data frames or remote frames are transferred to the application through the transmit-acknowledge-FIFO (see sections 1.4.2 and 1.4.3).

0: Acknowledgements of transmitted data frames or remote frames are ascertained by polling of the objects in the interface (see sections 1.4.2 and 1.4.3). Under certain conditions this can cause a longer running time of the function *CANL2_read_ac* and thus lead to lower throughput rates of the interface.

- **TransmitRmtFifoEnableAll:**

This parameter selects the handling mechanism for objects with Auto Remote Control configured (*AutoRemoteEnable* is set).

1:	Incoming remote frames are buffered in a FIFO and are passed on for transmission of data frames (see sections 1.4.2 and 1.4.3").
0:	Incoming remote frames are stored in object lists, which are polled for transmission of data frames (see section 1.4.3").

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.14 CANL2Channel::CANL2_define_object

Visual Basic style:

```
Public Function
CANL2_define_object (
        ByVal Ident As UInteger,
        ByRef ObjectNumber As Integer,
        ByVal Type As Integer,
        ByVal ReceiveIntEnable As Integer,
        ByVal AutoRemoteEnable As Integer,
        ByVal TransmitAckEnable As Integer) As Integer
```

C# style:

```
int CANL2_define_object(

        unsigned long   Ident,
        ref int         ObjectNumber,
        int             Type,
        int             ReceiveIntEnable,
        int             AutoRemoteEnable,
        int             TransmitAckEnable)
```

The function *CANL2_define_object* defines and configures the communication objects of the transmit and receive object lists in object buffer mode.

In dynamic object buffer mode all used objects have to be defined, while in static object buffer mode the function can be used optionally for individual configuration of the object handling.

In static object buffer mode the returned object number equals the identifier. But in dynamic object buffer mode it corresponds to the succession of definition in the related object list.

NOTE:
The API functions handle the objects by their object number. Hence, the user is recommended to setup a table of relations between identifier and object number in dynamic object buffer mode.

Function Parameters:

- Ident:

Identifier

[0 to $7FF_{Hex}$] for standard objects
[0 to $1FFFFFFF_{Hex}$] for extended objects

- ObjectNumber:

In the mode dynamic object buffer the object number in the related object list is returned in this parameter. It is a handle for the online access to this object (*CANL2_send_object*, *CANL2_read_rcv_data*...).

The identifier itself will no longer be referenced. In the mode static object buffer the object number is equally to the identifier.

- Type:

Direction of transmission and type of identifier

0: Standard receive object: Data frames and remote frames with standard identifiers (11 bit) can be received.

1: Standard transmit object: Data frames and remote frames with standard identifiers (11 bit) can be transmitted.

2: Extended receive object: Data frames and remote frames with extended identifiers (29 bit) can be received.

3: Extended transmit object: Data frames and remote frames with extended identifiers (29 bit) can be transmitted.

- ReceiveIntEnable (only for receive objects):

1: When receiving an object with the identifier *Ident* the receive **message** is passed to the PC application. Additionally, an interrupt is generated to the PC. The application program can read the object using *CANL2_read_ac*.

0: After receipt of an object the object data are indeed entered into object buffer (and they can be read using *CANL2_read_rcv_data*), but no information is generated for the application regarding receipt of the object. No interrupt is generated to the PC.

- AutoRemoteEnable (only for receive objects):

1: When receiving a remote frame with the identifier *Ident* the firmware transmits a data frame with the same identifier independently form the PC (see sections 1.4.2 and 1.4.3).

0: When receiving a remote frame the remote frame is reported to the PC (can be read using *CANL2_read_ac* or *CANL2_read_rcv_data*). The PC must transmit an explicit response (data frame).

NOTE:
The remote frame is only answered automatically after the first call of *CANL2_supply_object_data* or *CANL2_write_object*. This assures that no non-initialized data are transmitted. A remote frame arriving before the first call of *CANL2_supply_object_data* or *CANL2_write_object* results in error report -6 in the function *CANL2_read_ac*. For the auto remote feature it is necessary to define a transmit object as well as a receive object with the same identifier.

- TransmitAckEnable (only for transmit objects):

1 A data frame or remote frame with the identifier *Ident* is acknowledged (in conjunction with an interrupt to the PC) after successful transmission. This acknowledgement can be read using *CANL2_read_ac* or *CANL2_read_xmt_data* (see 5.1.2 and 5.1.3).

0: A data frame or remote frame with the identifier *Ident* is not acknowledged to the application after successful transmission on the bus.

NOTE:
Please note that the objects defined first are also polled first, and in this way a higher priority and a lower polling time is maintained relative to the objects that follow. It is sensible to define objects in the sequence of their identifiers in order to make prioritization of objects with low identifiers the same as on the CAN bus. This is true for static as well as for dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-102:	Wrong parameter
-104:	Timeout firmware communication
-109	Dyn. Obj. buffer mode not enabled
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.15 CANL2Channel::CANL2_start_chip

Visual Basic style:

Public Function CANL2_start_chip () As Integer

C# style:

int CANL2_start_chip(void)

The function *CANL2_start_chip* puts the CAN controller into operational mode. From now on transmit jobs can be issued and reception of messages is monitored.

Parameters:

none	

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.16 CANL2Channel::CANL2_define_cyclic

Visual Basic style:

```
Public Function
CANL2_define_cyclic (
        ByVal ObjectNumber As Integer,
        ByVal Rate As UInteger,
        ByVal Cycles As UInteger) As Integer
```

C# style:

```
int CANL2_define_cyclic(

        int             ObjectNumber,
        unsigned int    Rate,
        unsigned int    Cycles)
```

The function *CANL2_define_cyclic* defines cyclic transmission of a communication object for the CAN channel which was previously defined by *CANL2_define_object*.

The cyclic transmission is started and stopped by the value of *Rate.* The settings (transmission start/stop) are put into operation by the first call of *CANL2_send_object* or *CANL2_write_object* for the object after the definition call.

Alternatively, the cyclic transmission is stopped automatically if the defined number of cycles *Cycles* is reached.

NOTE:
If defined and started a cyclic object has to be stopped before any succeeding redefinition. Redefinition of the cycle rate while running the transmission results in faulty transmission.

The transmitted data contents are defined by *CANL2_supply_object* or *CANL2_write_object*. They can be modified during cyclic transmission as well.

NOTE:
This function can only be used in dynamic object buffer mode.

Function Parameters:

- ObjectNumber:

Object reference returned by *CANL2_define_object*.

- Cycles [0..65535]:

0:	Unlimited cyclic repetition
1..65535:	Number of cyclic repetitions

- Rate [0..65535]:

0:	Disable cyclic transmission (stop)
1..65535:	Transmission rate in ms

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.17 CANL2Channel::CANL2_send_remote_object

Visual Basic style:

```
Public Function
CANL2_send_remote_object (
        ByVal ObjectNumber As Integer,
        ByVal DataLength As Integer) As Integer
```

C# style:

```
int CANL2_send_remote_object(
        int     ObjectNumber,
        int     DataLength)
```

Function Parameters:

- ObjectNumber:	Object number
- DataLength:	Number of data bytes

This function initiates transmission of a remote frame for a transmit object specified by the object number. The remote frame has a data length 0; however, the data length code is physically transmitted with the data length code *DataLength*.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it’s equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-110	Last request still pending
-116	Transmit request FIFO overrun
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.18 CANL2Channel::CANL2_supply_object_data

Visual Basic style:

```
Public Function
CANL2_supply_object_data (
        ByVal ObjectNumber As Integer,
        ByVal DataLength As Integer,
        ByVal pData() As Byte) As Integer
```

C# style:

```
int CANL2_supply_object_data(

       int                 ObjectNumber,
        int                DataLength,
        byte[]             pData)
```

Function Parameters:

- ObjectNumber:	Object number
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be transmitted

This function enters current data into the object buffer of the transmit object specified by *ObjectNumber*.

The data are not transmitted directly onto the bus, but rather they are prepared for pickup by a remote frame (Auto Remote) or a later transmit job (later: *CANL2_send_object*).

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-115	Object is not defined
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.19 CANL2Channel::CANL2_supply_rcv_object_data

Visual Basic style:

```
Public Function
CANL2_supply_rcv_object_data (
        ByVal ObjectNumber As Integer,
        ByVal DataLength As Integer,
        ByVal pData() As Byte) As Integer
```

C# style:

```
int CANL2_supply_rcv_object_data(
        int             ObjectNumber,
        int             DataLength,
        byte[]          pData),
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be written in the object buffer

This function enters new data into the object buffer of the specified receive object.

This function can be used for initialization of receive objects in order to get reasonable values even before the first reception of a respective data frame took place.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it’s equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-115	Object is not defined
-1000:	Invalid channel handle

1.7.20 CANL2Channel::CANL2_send_object

Visual Basic style:

```
Public Function
CANL2_send_object (
        ByVal ObjectNumber As Integer,
        ByVal DataLength As Integer) As Integer
```

C# style:

```
int CANL2_send_object(

        int                 ObjectNumber,
        int                 DataLength),
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes to be transmitted

This function transmits a data frame for the transmit object specified by *ObjectNumber*. The data frame has a length of *DataLength* bytes. The data transmitted are the last entered into transmit object buffer using *CANL2_supply_object_data* or *CANL2_write_object*.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO to be further processed. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it's equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3). *CANL2_send_object* transmits a data frame on CAN channel 1, *CANL2_send_object2* transmits a data frame on CAN channel 2.

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-116	Transmit request FIFO overrun
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.21 CANL2Channel::CANL2_write_object

Visual Basic style:

```
Public Function
CANL2_write_object (
        ByVal ObjectNumber As Integer,
        ByVal DataLength As Integer,
        ByVal pData() As Byte) As Integer
```

C# style:

```
int CANL2_write_object(

        int             ObjectNumber,
        int             DataLength,
        byte[]          pData),
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- DataLength:	Number of data bytes
- pData:	Pointer to the address field of data to be transmitted

This function performs an update of the data in the object buffer of the transmit object specified by *ObjectNumber*. Then a data frame is transmitted with *DataLength* bytes.

If *TransmitFifoEnable* is set the transmit job is entered into the transmit FIFO to be further processed. Otherwise the transmit request is registered in the transmit object list to be polled by the firmware.

ObjectNumber is the reference to the object returned by CANL2_define_object. In static object buffer mode it’s equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	Function successful
-104:	Timeout firmware communication
-110	Last request still pending
-115	Object is not defined
-116	Transmit request FIFO overrun
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.22 CANL2Channel::CANL2_read_rcv_data

Visual Basic style:

```
Public Function
CANL2_read_rcv_data (
        ByVal ObjectNumber As Integer,
        ByRef pRCV_Data() As Byte,
        ByRef Time As UInteger) As Integer
```

C# style:

```
int CANL2_read_rcv_data(
        int             ObjectNumber,
        ref byte[]      pRCV_Data,
        ref UInt32      Time)
```

Function Parameters:

- ObjectNumber:	Object number
- pRCV_Data:	Pointer to the address field of data being received
- Time:	Pointer to a time stamp parameter

This function copies the data of the receive object specified by *ObjectNumber* to the address *pRCV_Data*. The data are read, even if no new data were received. 8 data bytes are always copied to *pRCV_Data*, independent of the length of the received data frame.

If data in the object buffer are overwritten before they were read by the application or a remote request is not read quickly enough an overrun is signaled to the application by the function return code (overrun in object buffer).

If a remote frame was received the user is informed by a specific return code.

Time returns the instant of the last received data with a resolution of 1 microsecond (time stamp is reset in *CANL2_start_chip*).

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it’s equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	No new data received
1:	Data frame received
2:	Remote frame received
-104:	Timeout firmware communication
-111:	Receive data frame overrun
-112:	Receive remote frame overrun
-113:	Object is undefined
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.23 CANL2Channel::CANL2_read_xmt_data

Visual Basic style:

```
Public Function
CANL2_read_xmt_data (
        ByVal ObjectNumber As Integer,
        ByRef pDataLength As Integer,
        ByVal pXMT_Data () As Byte) As Integer
```

C# style:

```
int CANL2_read_xmt_data(
        int             ObjectNumber,
        ref int         pDataLength,
        byte[]          pXMT_Data),
```

Function Parameters:

- ObjectNumber:	ObjectNumber
- pDataLength:	Pointer to entry of number of transmitted data bytes
- pXMT_Data:	Pointer to the address field of data to be transmitted

This function reads the data and the initialized data length of the transmit object specified by *ObjectNumber*. Further, it checks whether a frame has been transmitted for this object.

If no transmission acknowledgments are returned by the object the function return code 1 indicates that the last transmit job was acknowledged by another CAN node. The return code -1 means that the last transmission acknowledgment has not been read by the application yet.

ObjectNumber is the reference to the object returned by *CANL2_define_object*. In static object buffer mode it’s equal to the CAN identifier, while in dynamic object buffer mode it depends on the succession of definition (see sections 1.4.2 and 1.4.3).

NOTE:
This function can only be used in object buffer mode, not in FIFO mode.

Function Return Codes:

0:	No message was transmitted
1:	Message was transmitted
-104:	Timeout firmware communication
-114:	Transmit acknowledge overrun
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.24 CANL2Channel::CANL2_send_data

Visual Basic style:

```
Public Function
CANL2_send_data (
        ByVal Ident As UInteger,
        ByVal Xtd As Integer,
        ByVal DataLength As Integer,
        ByVal pData () As Byte) As Integer
```

C# style:

```
int CANL2_send_data(

        unsigned long   Ident,
        int             Xtd,
        int             DataLength,
        byte[]          pData)
```

Function Parameters:

- Ident:	Identifier
- Xtd:	Identifier length 0: Standard Identifier 1: Extended Identifier
- DataLength:	Number of data bytes to be transmitted
- pData:	Pointer to the address field of the data

This function transmits a data frame with the passed parameters on the CAN channel.

The transmit request is processed through the transmit FIFO. If the FIFO is full the application is informed by the return value.

NOTE:
The function *CANL2_send_data* can only be used in FIFO mode, not in object buffer mode. The function *CANL2_send_data2* can only be used in FIFO mode or static object buffer mode, not in dynamic object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.25 CANL2Channel::CANL2_send_remote

Visual Basic style:

```
Public Function
CANL2_send_remote (
          ByVal Ident As UInteger,
          ByVal Xtd As Integer,
          ByVal DataLength As Integer) As Integer
```

C# style:

```
int CANL2_send_remote(

          unsigned long   Ident,
          int             Xtd,
          int             DataLength)
```

Function Parameters:

- Ident:	Identifier
- Xtd:	Identifier length 0: Standard Identifier 1: Extended Identifier
- DataLength:	Number of data bytes requested remote

This function transmits a remote frame with the Identifier *Ident* on the CAN channel. The remote frame has data length 0; however, the data length specified by the parameter *DataLength* is transmitted in the DLC field of the remote frame.

The transmit request is processed through the transmit FIFO. If the FIFO is full the application is informed by the return value.

NOTE:
The function *CANL2_send_remote* can only be used in FIFO mode, not in object buffer mode.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.26 CANL2Channel::CANL2_read_ac

Visual Basic style:

```
Public Function
CANL2_read_ac
  (ByVal param  As PARAM_CLASS) As Integer
```

C# style:

```
int CANL2_read_ac(PARAM_CLASS param)
```

By calling this function the application is informed about data transmission and reception as well as about various error conditions and bus events.

Several different CAN events can be distinguished by evaluation of the function return code (see Table 1.7-8). Certain information and parameters of interest are transferred in the elements of the parameter class PARAM_CLASS.

Members of PARAM_CLASS**:**

NOTE:
RC1 through RC12 in brackets specify the function return codes of *CANL2_read_ac* for which the described parameter is valid. The application should not evaluate the parameter if it comes with a different function return code than stated below.

- **unsigned long Ident:**

Identifier (FIFO mode) or object number (object buffer mode) of the data or remote frame which was received or successfully transmitted.

(RC1, RC2, RC3, RC8, RC9, RC10, RC11, RC12)

- **int DataLength:**

Number of received (RC1, RC9) or transmitted (RC3, RC10) data bytes.

The *DataLength* of the received frame is only valid in FIFO mode and should not be used in object buffer mode. In object buffer mode the data length of the CAN messages should be predefined by the project.

- **int RecOverrun_flag:**

The last received data of object *Ident* were not read by the PC and were overwritten by the new data (RC1, RC2, RC9, RC12). Only valid in object buffer mode.

- **int RCV_fifo_lost_msg:**

Number of lost messages in receive FIFO (RC1, RC2, RC8, RC9, RC11, RC12). Only valid in FIFO mode.

- **byte RCV_data[8]:**

Data bytes of the received data frame (RC1, RC9).

- **int AckOverrunFlag:**

This flag is set if an unread transmit acknowledge for a transmit object is overwritten by a new one (RC3, RC10). Only valid in object buffer mode.

- **int XMT_ack_fifo_lost_acks**:

Number of lost acknowledges messages in transmit-acknowledge-FIFO in object buffer mode due to FIFO overrun(RC3, RC10).

Only valid in mode object buffer configured with *TransmitAckFifoEnable*=1.

- **int XMT_rmt_fifo_lost_remotes:**

Number of lost jobs in remote transmit FIFO (RC4). Only valid in object buffer mode initialized with *TransmitRmtFifoEnable*=1.

- **int Bus_state:**

Returns the new CAN bus status if a status change occurred (RC5).

0:	error active
1:	error passive
2:	bus off

- **int Error_state:**

Not used. Only for conformity to CANcard and CAN-AC2 (ISA) API.

- **int can:**

Number of CAN channel where the event occurred which is defined by the function return code.

(RC1, RC2, RC3, RC4, RC5, RC7, RC8, RC9, RC10, RC11, RC12,RC15)

- **unsigned long Time:**

Time stamp of signaled events with a resolution of 1µs. The timer is reset in *CANL2_start_chip*. (RC1, RC2, RC9, RC12, RC3, RC5, RC8, RC10, RC11, RC15)

Table 1.7-8: Function return codes of CANL2_read_ac

FRC	Explanation
0:	No new event
1:	Standard data frame received
2:	Standard remote frame received
3:	Transmission of a standard data frame is confirmed
4:	Overrun of the remote transmit FIFO. Only with object buffer and auto remote feature.
5:	Change of bus status
6:	not implemented
7:	Not used
8:	Transmission of a standard remote frame is confirmed.
9:	Extended data frame received
10:	Transmission of an extended data frame is confirmed
11:	Transmission of an extended remote frame is confirmed
12:	Extended remote frame received
13, 14	Not valid. Only useful with CANcard API
15:	Error frame detected
-1:	Function not successful
-104:	Timeout firmware communication
-115:	access to an abject denied, because the object has not been initialized with data using CANL2_supply_object()
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.27 CANL2Channel::CANL2_reinitialize

Visual Basic style:

Public Function CANL2_reinitialize () As Integer

C# style:

int CANL2_reinitialize(void);

CANL2_reinitialize stops the current online operation of the specified CAN channel. Afterwards the operating mode and all corresponding parameters can be set anew (see Fig. 1-5, 1-6, 1-7).

Parameters:

none	

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.28 CANL2Channel::CANL2_get_time

Visual Basic style:

```
Public Function
CANL2_get_time (ByRef time As UInteger) As Integer
```

C# style:

```
int CANL2_get_time(ref UInt32 time);
```

Function Parameters:

- time:	Time (32bit) in µs

CANL2_get_time returns the 32bit time from the onboard timer of the Softing CAN Interface card in the parameter *time*. The unit of *time* is µs.

The timer is reset by *CANL2_reset_chip*.

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.29 CANL2Channel::CANL2_get_bus_state

Visual Basic style:

Public Function CANL2_get_bus_state () As Integer

C# style:

int CANL2_get_bus_state(void);

Function Parameters:

void

CANL2_get_bus_state returns the current bus status of the CAN controller.

If the CAN controller is in bus off state it must be reset and started again to enable further access to the bus.

Function Return Codes:

0:	Error active
1:	Error passive
2:	Bus off
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.30 CANL2Channel::CANL2_reset_lost_msg_counter

Visual Basic style:

Public Function CANL2_reset_lost_msg_counter () As Integer

C# style:

int CANL2_reset_lost_msg_counter(void);

CANL2_reset_lost_msg_counter resets the counter for the receive messages which were lost while the receive FIFO remained full in FIFO mode.

The lost message counter is supplied in the PARAM_CLASS class of *CANL2_read_ac*.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

none	

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.31 CANL2Channel::CANL2_read_rcv_fifo_level

Visual Basic style:

Public Function CANL2_read_rcv_fifo_level () As Integer

C# style:

int CANL2_read_rcv_fifo_level(void);

CANL2_read_rcv_fifo_level returns the number of events in the receive FIFO waiting to be read by *CANL2_read_ac*.

The FIFO level can be reset to 0 by *CANL2_reset_rcv_fifo* which clears the FIFO.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

none	

Function Return Codes:

0 ... 154:	Messages in receive FIFO
-1:	Function not successful
-4:	Timeout firmware communication
-99:	Board not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.32 CANL2Channel::CANL2_reset_rcv_fifo

Visual Basic style:

Public Function CANL2_reset_rcv_fifo () As Integer

C# style:

int CANL2_reset_rcv_fifo(void);

CANL2_reset_rcv_fifo resets the receive fifo in FIFO mode.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

none	

Function Return Codes:

0:	Function successful
-1:	Function not successful
-3:	Error accessing DPRAM
-4:	Timeout firmware communication
-99:	Board not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.33 CANL2Channel::CANL2_read_xmt_fifo_level

Visual Basic style:

Public Function CANL2_read_xmt_fifo_level () As Integer

C# style:

int CANL2_read_xmt_fifo_level(void);

CANL2_read_xmt_fifo_level returns the number of transmit jobs in the transmit FIFO waiting to be transmitted by the interface.

A pending transmission request which is already entered into the transmit buffer of the CAN controller is not counted.

The FIFO level can be reset to 0 by *CANL2_reset_xmt_fifo* which clears the FIFO.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

none	

Function Return Codes:

0 ... n:	Messages in receive FIFO
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.34 CANL2Channel::CANL2_reset_xmt_fifo

Visual Basic style:

Public Function CANL2_reset_xmt_fifo () As Integer

C# style:

int CANL2_reset_xmt_fifo(void);

CANL2_reset_xmt_fifo resets the transmit FIFO in FIFO mode.

NOTE
This function is not useful in dynamic object buffer mode.

Parameters:

none	

Function Return Codes:

0:	Function successful
-1:	Function not successful
-104:	Timeout firmware communication
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

1.7.35 CANL2Channel::CANL2_init_signals

Visual Basic style:

```
Public Function CANL2_init_signals(
     ByVal ulChannelDirectionMask As UInteger,
     ByVal ulChannelOutputDefaults As UInteger) As Integer
```

C# style:

```
int CANL2_init_signals(
    UInt32 ulChannelDirectionMask,
    UInt32  ulChannelOutputDefaults);
```

NOTE
This function is for use with the Softing CAN lowspeed module only! If no lowspeed module is installed, then this function is not needed.

1.7.35.1 CAN Lowspeed module overview

Via CAN API, all status and control signals from and to the module plug-in stations are made available for the user's convenience. In this manner, the user has all features of the CAN Lowspeed Transceivers under his control; he can freely determine the change of operating modes between CAN Highspeed and CAN Lowspeed and read in the identifier of the module, if necessary.

After a RESET of the CAN Interface card, the operating mode CAN highspeed is set by default in the CAN Lowspeed module. To permit access to the status and control signals and switch over to the CAN Lowspeed mode, the possibility of access to the signals of the module plug-in stations must first be initialized. The read-in of the status signals and switching of the control signals then takes place via read and write functions.

The table 1.6.36 gives a survey of the configurations to be selected during initialization of the read and write functions and of the signal bit assignment to the connection pins of the module plug-in stations and to the signals arriving there.

During reset, the signals are initialized in such a way that CAN highspeed is selected and the two LS Transceivers are in the sleep mode

The function initializes the direction of the signal and defines it for further use.

1.7.35.2 How to use CANL2_init_signals

The function initializes the direction of the signal and defines it for further use.

For operation of the CAN LS modules, the bits of the parameter ”ulChannelDirectionMask” must be selected according to table 1.6.36. The following applies to every bit position:

1 defines the signal direction as ”output”

0 defines the signal direction as ”input”.

NOTE
Ports exclusively for input cannot be defined as outputs. Unassigned bit positions can be defined as desired. ”Input” and ”output” are defined from the position of the microprocessor C165 or XC161.

Operation of the CAN Lowspeed Module:

The parameter "ulChannelOutputDefaults" indicates the default status at the output for the bit positions for which the signal direction ”output” has been defined via ”ulChannelDirectionMask". (For the settings, see table 1.6.36;)

This table supplies information on the reset status = inactive transceivers; the default status may deviate.)

1 defines the default level "high" for an ”output”.

0 defines the default level "low" for an ”output”.

The values of the bit positions for which the signal direction ”input” has been determined, are irrelevant.

This function must be called before CANL2_write_signals() and CANL2_read_signals(), to setup the port pins of the microcontroller on the CAN Interface board.

Function Return Codes:

0:	Function successful
-1:	Error: signals have already been initialized
-2:	An exclusive input port has been defined as output.
-104:	Timeout firmware communication
-1000:	Invalid channel handle

NOTE

Execution of the initialization function CANL2_init_signals() is permitted only once after loading of the firmware into the Softing CAN Interface. Reinitialization of the status and control signals is possible only

The following table shows the *Signal bit assignment to control and status signals*

Bit pos.	**Direction of the signal**	**Reset default status**	**Description of the signal**
0	**Out->**	**H**	LS Transceiver signal: EN (signal inverted, resetlevel = L)
1	**Out->**	**H**	LS Transceiver signal:/STB (signal inverted, reset level=L)
2	**Out->**	**H**	HS/LS switchover (H = Highspeed)
3	**Out->**	**H**	LS Transceiversignal: /WAKE
4	**In <-**	**H**	LS Transceiver signal: NERR
5	**In <-**	**H**	Module identifier bit 0=1
6	**In <-**	**L**	Module identifier bit 1=0
7	**In <-**	**L**	Module identifier bit 2=0

Table 1.6.36

Example:

Initialisation of the mowspeed module:

CANL2_init_signals(can, 0x0000000F, 0x00000004);
Switching to CAN lowspeed:
CANL2_write_signals(can, 0x00000000, 0x00000004);
Switching back to CAN highspeed:
CANL2_write_signals(can, 0x00000004, 0x00000004);

1.7.36 CANL2Channel::CANL2_read_signals

Visual Basic style:

```
Public Function CANL2_read_signals(
    ByRef ulChannelRead As UInteger) As Integer
```

C# style:

```
int CANL2_readt_signals(ref UInt32  ulChannelRead)
```

NOTE
This function is for use with the Softing CAN lowspeed module only! If no lowspeed module is installed, then this function is not needed.

The function reads in the current signal statuses. The parameter ”ulChannelRead” is coded according to table **1.6.36**.

The following applies to every bit position:

Operation of the CAN Lowspeed Module

1 means that the signal level is ”high”.

0 means that the signal level is ”low”.

If a signal has been defined as output, the output is read back –if possible- or the value set and buffered last is returned.

Unassigned bit positions are evaluated at ”0”.

The function is useful to detect, whether a lowspeed piggyback is installed on the hardware. If a lowspeed piggyback is plugged, then the module identifier bit “bit0” is 1, and the module identifier bits “bit1” and “bit2” are 0. (see table 1.6.36)

Function Return Codes:

0:	Function successful
-1:	Error: signals have not yet been initialized

-104:	Timeout firmware communication
-1000:	Invalid channel handle

1.7.37 CANL2Channel::CANL2_write_signals

Visual Basic style:

```
Public Function CANL2_write_signals(
     ByVal ulChannelWrite As UInteger,
     ByVal ulCareMask As UInteger) As Integer
```

C# style:

```
int CANL2_write_signals(
    UInt32 ulChannelWrite,
    UInt32  ulCareMask);
```

NOTE
This function is for use with the Softing CAN lowspeed module only! If no lowspeed module is installed, then this function is not needed.

The function sets output signals according to the parameter definition.

The parameters ”ulChannelWrite” and ”ulCareMask” are coded according to table 1.6.36.

Signals set at ”0” in the ”ulCareMask” are ignored. Only those signals are written which are set at ”1” in the ”ulCareMask” parameter. For these signals, the parameter ”ulChannelWrite” is evaluated and the following applies:

1 means that the signal level is set at ”high”.

0 means that the signal level is set at ”low”.

A write access to an unassigned bit position is ignored.

Function Return Codes:

0:	Function successful
-1:	Error: signals have not yet been initialized, this must be done by using CANL2_init_signals()
-2:	Error: write access to an input signal
-104:	Timeout firmware communication
-1000:	Invalid channel handle

Example: Lowspeed/Highspeed switchover:

Switching to CAN lowspeed:

```
CANL2_write_signals(can, 0x00000000, 0x00000004);
```

Switching back to CAN highspeed:

```
CANL2_write_signals(can, 0x00000004, 0x00000004);
```

1.7.38 CANL2Channel::INIL2_close_channel

Visual Basic style:

Public Function INIL2_close_channel () As Integer

C# style:

int INIL2_close_channel(void)

This function releases and unlocks the system resources which were allocated by *INIL2_initialize_channel*.

The function call should be applied at any possible application exit after successful call to *INIL2_initialize_channel*. Otherwise, the application may have problems to get the handle to the DPRAM a second time without system exit (e.g. applications with LabVIEW a.o.).

Parameters:

none	

Function Return Codes:

0:	Function successful
-1000:	Channel not initialized: INIL2_initialize_channel() was not yet called or a INIL2_close_channel() was done

Index