WINCE-CAN-Interface

Software Documentation

NetDCUx + NetDCU-ADP/CAN2

Version 1.05 Date: 2006-05-04

F & S Elektronik Systeme GmbH
Untere Waldplätze 23
D-70569 Stuttgart
Tel.: 0711/6772240 Fax: 0711/6772243

Overview

NetDCU6, NetDCU8 and NetDCU9 offer one CAN port for direct integration of NetDCU into a CAN network. All other NetDCU's can use the extension board NDCU-ADP/CAN2 which is connected to NetDCU over FS-Bus.
This documentation describes the functionality of the CANINTF driver. Please check if the driver is installed and running. You can do this by opening the tool remote registry editor and search the entries in [HKLM\Drivers\Active]. If you don't' find the driver, you have to install it for each CAN interface.
With the three test programs CANCMD.EXE, CANREAD.EXE and CANWRITE.EXE you can easily check the functionality of the interface and the driver. The source of the programs is available.

Functions

Funktion	Beschreibung
CreateFile()	Opens the CAN interface and initialises the controller. Each CAN interface can be opened more than once.
CloseHandle()	Closes the previously opened CAN interface.
ReadFile()	Reads a received event in a formatted style from the CAN interface.
WriteFile()	Send a formatted string to the CAN interface.
DeviceIoControl()	Calls a special control function of the CAN driver.
SetCommTimeouts()	Changes Timeouts for read and write access.
GetCommTimeouts()	Returns current settings for read and write time out.
SetCommMask()	Set wait mask for WaitCommEvent().
GetCommMask()	Returns wait mask.
WaitCommEvent()	Wait for a event.

CreateFile()

This function opens the CAN device and returns a handle to it. It is possible to open a device more than one time.

```
HANDLE CreateFile(
LPCTSTR lpFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
DWORD dwCreationDispostion ,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile );
```

Parameters:

lpFileName

Name of the CAN device to open (CIDx:, x = 1,2, or 3)

dwDesiredAccess

Specifies the type of access to the object. An application can obtain read access, write access, read-write access, or device query access. This parameter can be any combination of the following values.

Value	Description
0	Specifies device query access to the object. An application can query device attributes without accessing the device.
GENERIC_READ	Specifies read access to the object. Data can be read from the file and the file pointer can be moved. Combine with GENERIC_WRITE for

	read-write access.
GENERIC_WRITE	Specifies write access to the object. Data can be written to the file and the file pointer can be moved. Combine with GENERIC_READ for read-write access.

dwShareMode
Ignored; set to 0.
lpSecurityAttributes
Ignored; set to NULL.
dwCreationDispostion
Should be set to OPEN_EXISTING.
dwFlagsAndAttributes
Ignored; set to 0.
hTemplateFile
Ignored; set to NULL.

Return Values
An open handle to the specified file indicates success. INVALID_HANDLE_VALUE indicates failure. To get extended error information, call **GetLastError**.

Remarks
Use the **CloseHandle** function to close an object handle returned by **CreateFile**. This function uses the registry values to initialize the Can controller. So if you set the registry value Baudrate to 250000, the CAN controller will be initialized with this value. It is not necessary to call DeviceIoControl() with parameter IOCTL_CAN_INIT after opening the CAN port.

ReadFile()

This function reads a event from the CAN device. If currently no event is in the input queue, the function uses the timeout settings for waiting.

```
BOOL ReadFile(
HANDLE hFile,
LPVOID lpBuffer,
DWORD nNumberOfBytesToRead,
LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped );
```

Parameters

hFile

Handle to the file CAN interface be read. The file handle must have been created with GENERIC_READ access to the file.

lpBuffer

Pointer to the buffer that receives the data read from the file.

nNumberOfBytesToRead

Number of bytes to be read from the CAN interface.

lpNumberOfBytesRead

Pointer to the number of bytes read. **ReadFile** sets this value to zero before doing any work or error checking.

lpOverlapped

Unsupported; set to NULL.

Return Values

The behaviour of ReadFile for the CAN device is different from that to other devices.
The CAN device can send and receive messages which differs in length between 1 and 8 bytes. Although each CAN

message has a so called identifier. The CAN identifier can be compared with the MAC address from Ethernet.
If now a message from the CAN bus arrives to our CAN device, a internal event is created and the data is stored in the input queue which is also organised in events. The result of this is, that ReadFile returns a event as a string. The format of this string is as follows:
<event_type>\t<event_message>
Values for event type are “received”, “transmitted”, “bus error”, “warning” and “device changed”. The Value for event_message depends from the type of event.

Nonzero indicates success. If the return value is nonzero and the number of bytes read is zero, the file pointer was beyond the current end of the file at the time of the read operation. Zero indicates failure. To get extended error information, call **GetLastError**.

Remarks
The behavior of **ReadFile** is governed by the current communication time-outs as set and retrieved using the **SetCommTimeouts** and **GetCommTimeouts** functions. Unpredictable results can occur if you fail to set the time-out values. For more information about communication time-outs, see **COMMTIMEOUTS**.

The sample CANREAD.EXE shows how to open a CAN device and how to read CAN messages from this device.

WriteFile()

This function writes data to a CAN device. **WriteFile** interprets the data as a formatted ASCII string and creates a CAN message form the given data.

BOOL WriteFile(
HANDLE *hFile*,
LPCVOID *lpBuffer*,
DWORD *nNumberOfBytesToWrite*,
LPDWORD *lpNumberOfBytesWritten*,
LPOVERLAPPED *lpOverlapped* **);**

Parameters

hFile
Handle to the CAN device to be written to. The file handle must have been created with GENERIC_WRITE access to the file.

lpBuffer
Pointer to the buffer containing the CAN message to be written to the CAN bus.

nNumberOfBytesToWrite
Length of the ASCII string that holds the message data.

lpNumberOfBytesWritten
Pointer to the number of bytes written by this function call. **WriteFile** sets this value to zero before doing any work or error checking.

lpOverlapped
Unsupported; set to NULL.

Return Values

Nonzero indicates success. Zero indicates failure. To get extended error information, call **GetLastError**.

Remarks

DeviceIoControl()

This function sends a control code directly to the specified CAN device driver, causing the corresponding CAN device to perform the specified operation.

BOOL DeviceIoControl(
HANDLE *hDevice*,
DWORD *dwIoControlCode*,
LPVOID *lpInBuffer*,
DWORD *nInBufferSize*,
LPVOID *lpOutBuffer*,
DWORD *nOutBufferSize*,
LPDWORD *lpBytesReturned*,
LPOVERLAPPED *lpOverlapped* **);**

Parameters

hDevice

Handle to the CAN device that is to perform the operation. Call the **CreateFile** function to obtain a device handle.

dwIoControlCode

Specifies the control code for the operation. This value identifies the specific operation to be performed and the type of device on which the operation is to be performed.
See table “Control Codes” for a complete list.

lpInBuffer

Long pointer to a buffer that contains the data required to perform the operation.
This parameter can be NULL if the *dwIoControlCode* parameter specifies an operation that does not require input data.

nInBufferSize

Size, in bytes, of the buffer pointed to by *lpInBuffer*.

lpOutBuffer

Long pointer to a buffer that receives the operation's output data.
This parameter can be NULL if the *dwIoControlCode* parameter specifies an operation that does not produce output data.

nOutBufferSize

Size, in bytes, of the buffer pointed to by *lpOutBuffer*.

lpBytesReturned

Long pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by *lpOutBuffer*.
The *lpBytesReturned* parameter cannot be NULL. Even when an operation produces no output data, and *lpOutBuffer* can be NULL, the **DeviceIoControl** function makes use of the variable pointed to by *lpBytesReturned*. After such an operation, the value of the variable is without meaning.

lpOverlapped

Ignored; set to NULL.

Return Values

Nonzero indicates success. Zero indicates failure. To get extended error information, call **GetLastError**. If the control code is not implemented, GetLastError() returns ERROR_CALL_NOT_IMPLEMENTED.

Remarks

WaitCommEvent()

This function sends a control code directly to the specified CAN device driver, causing the corresponding CAN device to perform the specified operation.

BOOL WaitCommEvent(
HANDLE *hDevice*,
LPDWORD lpEvtMask);

Parameters

hDevice

Handle to the CAN device that is to perform the operation. Call the **CreateFile** function to obtain a device handle.

lpEvtMask

Long pointer to a 32-bit variable that receives a mask indicating the events that occurred. If an error occurs, the value is zero; otherwise, it is one or more of the following values:

CANBUS_EVENT_RECEIVED
CANBUS_EVENT_TRANSMITTED
CANBUS_EVENT_BUS_ERROR
CANBUS_EVENT_WARNING
CANBUS_EVENT_LEAVING_STANDBY
CANBUS_EVENT_ARBITRATION_LOST
CANBUS_EVENT_OVERRUN
CANBUS_EVENT_PASSIVE
CANBUS_EVENT_ENTERING_STANDBY
CANBUS_EVENT_DEVICE_CHANGED

lpOverlapped

Ignored; set to NULL.

Return Values
Nonzero indicates success. Zero indicates failure. To get extended error information, call **GetLastError**.
If the mask given by lpEvtmask is 0, WaitCommEvent() returns immediately and GetLastError() return error 87 (ERROR_INVALID_PARAMETER)

Remarks

The **WaitCommEvent** function monitors a set of events for a specified communications resource. To set and query the current event mask of a communications resource, use the **SetCommMask** and **GetCommMask** functions. When a communications event that is set by **SetCommMask** occurs, **WaitCommEvent** returns.

Control Codes

IOCTL_CAN_WRITE_ACCEPTANCE_FILTERIOCTL_CAN_ WRITE_ACCEPTANCE_FILTER

Writes the acceptance filter for the CAN receiver.

pBufIn = (*CAN_ACCEPRANCE_FILTER)
dwLenIn = sizeof(CAN_ACCEPTANCE_FILTER)

IOCTL_CAN_READ_ACCEPTANCE_FILTER

Read current acceptance filter.

pBufOut = (*CAN_ACCEPRANCE_FILTER)
dwLenOut = sizeof(CAN_ACCEPTANCE_FILTER)

IOCTL_CAN_SET_BAUDRATE

Set baud rate for Can bus.

pBufIn = (*unsigned long)
dwLenIn = sizeof(unsigned long)

IOCTL_CAN_GET_BAUDRATE

Read currently set baud rate.

pBufOut = (*unsigned long)
dwLenOut = sizeof(unsigned long)

IOCTL_CAN_INIT

Initialises the can controller.
No parameters needed. Check return code for success.

IOCTL_CAN_SET_CAN_MODE

Set the mode of the CAN chip.

pBufIn = (*unsigned long)
dwLenIn = sizeof(unsigned long)

Following values are powwible:
0: BasicCAN mode
1: Pelican mode, CAN 2.0B

This IOCTL changes the CAN mode of an open device and initializes the CAN controller. Because acceptance filter data is stored in different places for the two modes, acceptance filter data will be updated.

IOCTL_CAN_SET_COMMAND

Send special command of CAN chip.

pBufIn = (*unsigned long)dwCmd
dwLenIn = sizeof(unsigned long)

Following commands are possible:

- CANBUS_CMD_ENTER_STANDBY
  Enter standby of driver. Every event means “wake up”
- CANBUS_CMD_LEAVE_STANDBY
  Manually leave standby mode
- CANBUS_CMD_ABORT_TRANSMISSION
  Abort all transmissions. Empty output queue.
- CANBUS_CMD_CLEAR_OVERRUN
  Clear data overrun by setting SJA1000 CMR.3 to 1.
- CANBUS_CMD_SELF_RECEPTION_REQUEST
  Self Reception Request. Set SJA1000 CMR.4 to 1. (PeliCAN mode)
- CANBUS_CMD_LISTEN_ON
  Enable Listen Only Mode. SJA1000 MOD.1 = 1. (PeliCAN mode)

- CANBUS_CMD_LISTEN_OFF
  Disable Listen Only Mode. SJA1000 MOD.1 = 0. (PeliCAN mode)
- CANBUS_CMD_VIRTUALIZE_ON
  Enable virtualization of CAN commands.
- CANBUS_CMD_VIRTUALIZE_OFF
  Disable virtualization of CAN commands.

IOCTL_CAN_WRITE_TRANSMIT_DATA

Transmit data.

pBufIn = (*CAN_TRANSMIT_DATA)
dwLenIn = sizeof(CAN_TRANSMIT_DATA)

IOCTL_CAN_READ_EVENT_DATA

Read one event.

pBufOut = (*CAN_EVENT)
dwLenOut = sizeof(CAN_EVENT)

If the function fails, you must call GetLastError() to get more details. GetLastError() returns ERROR_NO_MORE_ITEMS, if no event is available.

IOCTL_CAN_READ_TIME

Read current CAN time.

pBufOut = (*CAN_TIME)
dwLenOut = sizeof(CAN_TIME)

In fact, CAN time is based on GetTickCount() which returns the amount of ticks (ms) since system boot.

IOCTL_CAN_SET_DEFAULT_FRAME_FORMAT

pBufIn = (*unsigned long)dwFormat
dwLenIn = sizeof(unsigned long)

Following values for dwFormat are possible:

0	11 bit if CAN 2.0A and 29 bit if CAN 2.0B
1	always 11 bit
2	always 29 bit

IOCTL_CAN_TEST_DEVICE

Check if CAN device is in reset mode or not..
No parameters needed. Check return code for success.
Return:

0	Not in reset mode
1	Reset mode active
-1	Device not active

IOCTL_CAN_READ_PROPERTIES

Read properties.

pBufOut = (*CAN_PROPERTIES)
dwLenOut = sizeof(CAN_PROPERTIES)

Fills the structure CAN_PROPERTIES and returns to the caller. For a description of CAN_PROPERTIES see page 18.

IOCTL_CAN_READ_REGISTER

Read value of SJA1000 register in operating mode.

pBufIn = (*BYTE)pchAddress
dwLenIn = sizeof(BYTE)

pBufOut = (*BYTE)pchValue
dwLenOut = sizeof(BYTE)

IOCTL_CAN_READ_REGISTER_RM

Read value of SJA1000 register in reset mode.

pBufIn = (*BYTE)pchAddress
dwLenIn = sizeof(BYTE)
pBufOut = (*BYTE)pchValue
dwLenOut = sizeof(BYTE)

IOCTL_CAN_WRITE_REGISTER

Write value to SJA1000 register in operating mode.

pBufIn = (*BYTE)pchAddress
dwLenIn = sizeof(BYTE)
pBufOut = (*BYTE)pchValue
dwLenOut = sizeof(BYTE)

IOCTL_CAN_WRITE_REGISTER_RM

Write value to SJA1000 register in reste mode.

pBufIn = (*BYTE)pchAddress
dwLenIn = sizeof(BYTE)
pBufOut = (*BYTE)pchValue
dwLenOut = sizeof(BYTE)

Structures

CAN_TRANSMIT

Type	Name	Meaning
int	fmt	Format of the identifier. See table “Transmit Formats”.
unsigned long	identifier	CAN Identifier
unsigned char	rtr	Remote Transmission Request bit (0=CAN message with data, 1=request data from receiver)
unsigned char	dlc	Data Length Code (= number of data bytes in "msg[]")
unsigned char	msg[8]	Data bytes.

Transmit Formats

Name	Comment
CANBUS_TRANS_FMT_DEFAULT	Send this data with default frame format.
CANBUS_TRANS_FMT_STD	Send this data in 11 bit format.
CANBUS_TRANS_FMT_EXT	Send this data in 29 bit format.

CAN_EVENT

Type	Name	Meaning
int	event	Type of the event.
CAN_TIME	time	
unsigned long	lost	
CAN_TRANSMIT_DATA	data	
int	arbitration	

CAN_ACCEPTANCE_FILTER

(See also chapter Acceptance Filter)

Type	Name	Meaning
unsigned long	code	Acceptance code
unsigned long	mask	Acceptance mask. Set to 0xFFFFFFFF to receive all messages.

CAN_PROPERTIES

Type	Name	Meaning
unsigned long	Version	Driver version.
TCHAR	device_name[]	Name of the device.
long	min	Smallest possible baudrate.
long	max	Largest possible baudrate.
int	nCommands	Number of possible commands.
int	commands[]	List with all possible commands.

Acceptance Filter

The stand-alone CAN controller SJA1000 is equipped with a versatile acceptance filter, which allows an automatic check of the identifier and data bytes. Using these effective filtering methods, messages or a group of messages not valid for a certain node can be prevented from being stored in the Receive Buffer. Thus it is possible to reduce the processing load of the host controller.
The filter is controlled by the acceptance code and mask registers according to the algorithms given in the data sheet [Data Sheet SJA1000, Phillips Semiconductors]. The received data is compared bitwise with the value contained in the Acceptance Code register. The Acceptance Mask Register defines the bit positions, which are relevant for the comparison (0 = relevant, 1 = not relevant). For accepting a message all relevant received bits have to match the respective bits in the Acceptance Code Register.

Acceptance Filtering in BasicCAN Mode

This mode is implemented in the SJA1000 as a plug-and-play replacement (hardware and software) for the PCA82C200. Thus the acceptance filtering corresponds to the possibilities, which were found in the PCA82C200 [Data Sheet PCx82C200, Philips Semiconductors, November 1992]. The filter is controlled by two 8-bit wide registers – Acceptance Code Register (ACR) and Acceptance Mask Register (AMR). The 8 most significant bits of the identifier of the CAN message are compared to the values contained in these registers, see also Example 1 below. Thus always groups of eight identifiers can be defined to be accepted for any node.

Example 1:
Messages with the following11- bit identifiers (ID10…0) are accepted (x don't care):

	MSB										
ACR	**0**	**1**	1	1	0	**0**	**1**	**0**			
AMR	0	0	**1**	**1**	**1**	0	0	0			
ID10…0	**0**	**1**	x	x	x	**0**	**1**	**0**	x	x	x

At the bit positions containing a "1" in the Acceptance Mask-register, any value is allowed in the composition of the identifier. The same is valid for the three least significant bits. Thus 64 different identifiers are accepted in this example. The other bit positions must be equal to the values in the Acceptance Code register.

Acceptance Filtering in PeliCAN Mode

The acceptance filtering has been expanded for the PeliCAN mode: Four 8-bit wide Acceptance Code registers (ACR0, ACR1, ACR2 and ACR3) and Acceptance Mask registers (AMR0, AMR1, AMR2 and AMR3) are available for a versatile filtering of messages. These registers can be used for controlling a single long filter or two shorter filters, as shown in the examples below. Which bits of the message are used for the acceptance filtering, depend on the received frame (Standard or Extended) and on the selected filter mode (single or dual filter). Table Summery of Acceptance Filter in Pelican mode (see below) gives more information about which bits of the message are compared with the Acceptance Code and Mask bits. As it is seen from the following examples and the table, it is possible to include the RTR bit and even data bytes in the acceptance filtering for Standard Frames. In any case for all message bits, which shall not be included in the acceptance filtering (e.g. if groups of messages are defined for acceptance), the Acceptance Mask Register must contain a “1” at the corresponding bit position.
If a message doesn’t contain data bytes (e.g. in a Remote Frame or if the Data Length Code is zero) but data bytes are included in the acceptance filtering, such messages are accepted, if the identifier up to the RTR bit is valid.

Example 2:
In assumption, that the same 64 Standard Frame messages as described in the example above have to be filtered in PeliCAN mode.
This can be done using one long filter (Single Filter Mode). The Acceptance Code Registers (ACRn) and Acceptance Mask Registers (AMRn) contain:

n	0	1(upper 4 bits)	2	3
ACRn	01XX X010	XXXX	XXXX XXXX	XXXX XXXX
AMRn	0011 1000	1111	1111 1111	1111 1111
Accepted mes-sages (ID.28 - ID.18, RTR)	01xx x010 xxxx			

("X" = irrelevant, "x" = don't care, only the upper 4 bits of ACR1 and AMR1 are used)

At the bit positions containing a "1" in the Acceptance Mask registers, any value is allowed in the composition of the identifier, for the Remote Transmission Request bit and for the bits of data byte 1 and 2.

Example 3:
Suppose the following 2 messages with a Standard Frame Identifier have to be accepted without any further decoding of the identifier bits. Data and Remote Frames have to be received correctly. Data bytes are not involved in the acceptance filtering.
Message 1: (ID.28) 1 011 1100 101 (ID.18)
Message 2: (ID.28) 1 111 0100 101 (ID.18)

Using the Single Filter Mode results in accepting four messages and not only the requested two:

n	0	1(upper 4 bits)	2	3
ACRn	1X11 X100	101x	XXXX XXXX	XXXX XXXX
AMRn	0100 1000	0001	1111 1111	1111 1111
Accepted messages (ID.28 - ID.18, RTR)	1011 0100 101x 1111 0100 101x (message 2) 1011 1100 101x (message 1) 1111 1100 101x			

("X" = irrelevant, "x" = don't care, only the upper 4 bits of ACR1 and AMR1 are used)
This result does not meet the request for receiving 2 messages without any further decoding.

Using the Dual Filter mode gives the correct result:

	Filter 1			Filter 2	
n	0	1	3 lower 4 bits	2	3 lower 4 bits
ACRn	1011 1100	101X XXXX	… XXXX	1111 0100	101X …
AMRn	0000 0000	0001 1111	… 1111	0000 0000	0001 …
Accepted mes-sages (ID.28 - ID.18, RTR)	1011 1100 101x message 1			1111 0100 101x message 2	

(“X” = irrelevant, “x” = don’t care)
Message 1 is accepted by Filter 1 and message 2 by Filter 2. As messages are accepted and stored into the Receive FIFO if they are accepted at least by one of the two filters, this solution meets the request.

Example 4:
In this example a group of messages with an Extended Frame Identifier are filtered using a long single acceptance filter.

n	0	1	2	3 (upper 6 bits)
ACRn	1011 0100	1011 000X	1100 XXXX	0011 0XXX
AMRn	0000 0000	0000 0001	0000 1111	0000 0111
Accepted mes-sages (ID.28 - ID.0, RTR)	1011 0100 1011 000x 1100 xxxx 0011 0x			

("X" = irrelevant, "x" = don't care, only the upper 6 bits of ACR3 and AMR3 are used)

Example 5:
There are systems, which use Standard Frames only and identify messages by the 11-bit identifier and the first two data bytes. Such a protocol is used, e.g., in the DeviceNet, where the first two data bytes define a message header and the fragmentation protocol, if messages contain more than 8 data bytes. For this system type the SJA1000 can filter two data bytes in single filter mode and one data byte in dual filter mode in addition to the 11-bit identifier and the RTR-bit.

Using the Dual Filter mode, the following example shows effective filtering of messages in such a system:

	Filter 1			Filter 2	
n	0	1	3 lower 4 bits	2	3 lower 4 bits
ACRn	1110 1011	0010 1111	… 1001	1111 0100	XXX0 …
AMRn	0000 0000	0000 0000	…. 0000	0000 0000	1110 …
Accepted mes-sages	1110 1011 0010 ID+RTR 1111 first data byte … 1001			1111 0100 xxx ID 0 RTR	

(“X” = irrelevant, “x” = don’t care).

Filter 1 is used for filtering messages with

- the identifier “1 1 1 0 1 0 1 1 0 0 1”
- RTR = “0” i.e. Data Frames only and
- the data byte “1 1 1 1 1 0 0 1” (this means e.g. for the DeviceNet: all fragments for one message are filtered).

Filter 2 is used for filtering a group of 8 messages with

- the identifiers “1 1 1 1 0 1 0 0 0 0 0” through “1 1 1 1 0 1 0 0 1 1 1” and
- RTR = “0”, i.e. Data Frames only.

Table: Summery of Acceptance Filter in Pelican mode

Frame Type	**Single Filter mode**	**Dual Filter mode**
Standard	Message bits used for acceptance: - 11 bit identifier - RTR Bit - 1st data byte (8 bit) - 2nd data byte (8 bit) Acceptance Code & Mask registers used: - ACR0/upper 4 bits of ACR1/ACR2/ACR3 - AMR0/upper 4 bits of AMR1/AMR2/AMR3 (unused bits of the Acceptance Mask Register should be set to “1”)	<u>Filter 1</u> Message bits used for acceptance: - 11 bit identifier - RTR Bit - 1st data byte (8 bit) Acceptance Code & Mask registers used: - ACR0/ACR1/lower 4 bits of ACR3 - AMR0/AMR1/lower 4 bits of AMR3 <u>Filter 2</u> Message bits tested for acceptance: - 11 bit identifier - RTR Bit Acceptance Code & Mask registers used: - ACR2/upper 4 bits ofACR3 - AMR2/upper 4 bits of AMR3
Extended	Message bits used for acceptance: - 11 bit basic identifier	<u>Filter 1</u> Message bits used for acceptance: - 11 bit basic

	- 18 bit extended identi fier - RTR Bit Acceptance Code & Mask registers used: - ACR0/ACR1/ACR2/ upper 6 bits of ACR3 - AMR0/ AMR1/ AMR2/ upper 6 bits of AMR3 (unused bits of the Acceptance Mask Register should be set to “1”)	identifier - 5 most significant bits of extended identifier Acceptance Code & Mask registers used: - ACR0/ACR1 and AMR0/AMR1 <u>Filter 2</u> Message bits tested for acceptance: - 11 bit basic identifier - 5 most significant bits of extended identifier Acceptance Code & Mask registers used: - ACR2/ACR3 and AMR2/AMR3

APPENDIX: class CCAN

```
//
//
// F&S Elektronik Systeme GmbH
//
// Filename:     CAN.h
//
//


// Preprocessor directives and compiler switches
#ifndef __CAN_H__
#define __CAN_H__

// files at project folder
#include ".\..\inc\canbusio.h"

// Datatypes
typedef struct tagCANMSG
{
    DWORD      dwIdentifier;
    BYTE       byContent[8];
    BYTE       byDatalength;
}CANMSG;

// Class CCAN
class CCAN
{
// Memberfunctions
public:
    CCAN();
    ~CCAN();

    bool Init(TCHAR* pszName);
    bool Read( CANMSG* pcm );
    bool Write( CANMSG* pcm );
    DWORD SetCommMask( DWORD dwMask );
    DWORD Close( void );
private:
// Attributes
private:
    HANDLE m_hCAN;
    LONG m_l;
    CANMSG m_c;
    BYTE m_gCount;
protected:
};
```

```
//-----------------------------------------------------
//
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT
WARRANTY OF ANY KIND,
// EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED
TO THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
PARTICULAR PURPOSE.
//
// Copyright (c) 2002 F&S ELektronik Systeme GmbH
//
//  file:  canbusio.h
//  author:  H.Froelich
//  purpose: Interface to the CANINTF driver
//
//-----------------------------------------------------

#ifndef __CANBUSIO_H__
#define __CANBUSIO_H__

#define MAX_DEVICE_NAME_LENGTH (100)

#define CANBUS_FORMAT_CAN_2_0_A (0)
#define CANBUS_FORMAT_CAN_2_0_B (1)

// Chipset Flags
#define CANBUS_CFS_CAN_2_0_A (1<<0)   /* Chipset can used
for CAN 2.0 A */
#define CANBUS_CFS_CAN_2_0_B (1<<1)   /* Chipset can used
for CAN 2.0 B */
#define CANBUS_CFS_EXT_FRAME (1<<2)   /* Chipset support
extended frame format (only in CAN 2.0B mode) */
#define CANBUS_CFS_POLLING   (1<<3)     /* (not imple-
mented in canbus driver) Chipset driver supports only
polling (interrupts are default) */

// Events
enum t_canbus_events
{
    CANBUS_EVENT_RECEIVED=0x001,
    CANBUS_EVENT_TRANSMITTED=0x002,
    CANBUS_EVENT_BUS_ERROR=0x004,
    CANBUS_EVENT_WARNING=0x008,
    CANBUS_EVENT_LEAVING_STANDBY=0x010,
    CANBUS_EVENT_ARBITRATION_LOST=0x020,
    CANBUS_EVENT_OVERRUN=0x040,
    CANBUS_EVENT_PASSIVE=0x080,
    CANBUS_EVENT_ENTERING_STANDBY=0x100,
    CANBUS_EVENT_DEVICE_CHANGED=0x200
};
```

```
// Commands
enum t_canbus_commands
{
    CANBUS_CMD_ENTER_STANDBY=1,
    CANBUS_CMD_ABORT_TRANSMISSION,
    CANBUS_CMD_CLEAR_OVERRUN,
    CANBUS_CMD_LEAVE_STANDBY,
    CANBUS_CMD_SELF_RECEPTION_REQUEST,
    CANBUS_CMD_LISTEN_ON,
    CANBUS_CMD_LISTEN_OFF,
    CANBUS_CMD_VIRTUALIZE_ON,
    CANBUS_CMD_VIRTUALIZE_OFF,

    CANBUS_CMD_LAST_ENTRY /* this must be the last entry
in the list */
};

#define CANBUS_NUMBER_OF_CONSTANT_BAUDRATES (50)     /*
max. 50 entries for constant baudrates */

// The time is a 64-bit number...
struct canbus_time
{
    unsigned long low;
    unsigned long high;
};

// Acceptance Filter (Hardware filter in SJA1000)
struct canbus_acceptance_filter
{
    unsigned long code;
    unsigned long mask;
};

// Properties of the selected channel
struct canbus_properties
{
    unsigned long version;
    TCHAR device_name[MAX_DEVICE_NAME_LENGTH];
    long min;  // min baudrate
    long max;  // max baudrate
    int number_commands; // number of "commands[]" en-
tries
    int commands[CANBUS_CMD_LAST_ENTRY]; // supported
commands of the selected channel (depends on mode)
    int number_baudrates; // number of constant baudrates
in array "baudrates[]"
    unsigned long
baudrates[CANBUS_NUMBER_OF_CONSTANT_BAUDRATES];
    unsigned long chipset_flags; // see: CANBUS_CFS_...
```

```
    int number_registers; // for direct acces to the
chipset from an application
};

#define CANBUS_TRANS_FMT_DEFAULT (0) /* send this data
with default frame format */
#define CANBUS_TRANS_FMT_STD (1)     /* send this data in
11 bit format */
#define CANBUS_TRANS_FMT_EXT (2)     /* send this data in
29 bit format */
// Transmission structure
// With this structure a CAN message will sended
struct canbus_transmit_data
{
    int fmt; // see CANBUS_TRANS_FMT_...., ignore this in
canbus_event-structure (see below)
    unsigned long identifier;  // CAN Identifier
    unsigned char rtr;         // Remote Transmission
Request bit (0=CAN message with data, 1=request data from
receiver)
    unsigned char dlc;         // Data Length Code (=
number of data bytes in "msg[]")
    unsigned char msg[8];      // data bytes
};


struct canbus_event
{
    int event; // see: t_canbus_events
    struct canbus_time time;

    // used, if event=CANBUS_ISR_RECEIVED |
CANBUS_ISR_TRANSMITTED ---------------
    unsigned long lost;
    struct canbus_transmit_data data;

    // used, if event=CANBUS_ISR_ARBITRATION_LOST
    int arbitration;
};

// We'll need some defines
#include "WINIOCTL.h"

// New IOControlCode values

#define FILE_DEVICE_CAN              0x00008007

#define IOCTL_CAN_WRITE_ACCEPTANCE_FILTER \
    CTL_CODE(FILE_DEVICE_CAN, 0x801, METHOD_BUFFERED,
FILE_WRITE_ACCESS)
#define IOCTL_CAN_READ_ACCEPTANCE_FILTER \
```

```
    CTL_CODE(FILE_DEVICE_CAN, 0x802, METHOD_BUFFERED,
FILE_READ_ACCESS)
#define IOCTL_CAN_SET_BAUDRATE \
    CTL_CODE(FILE_DEVICE_CAN, 0x803, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_GET_BAUDRATE \
    CTL_CODE(FILE_DEVICE_CAN, 0x804, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_INIT \
    CTL_CODE(FILE_DEVICE_CAN, 0x805, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_SET_BAUDRATE_BY_CONSTANT \
    CTL_CODE(FILE_DEVICE_CAN, 0x806, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_GET_BAUDRATE_BY_CONSTANT \
    CTL_CODE(FILE_DEVICE_CAN, 0x807, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_SET_CAN_MODE \
    CTL_CODE(FILE_DEVICE_CAN, 0x808, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_SET_COMMAND \
    CTL_CODE(FILE_DEVICE_CAN, 0x809, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_WRITE_TRANSMIT_DATA \
    CTL_CODE(FILE_DEVICE_CAN, 0x80A, METHOD_BUFFERED,
FILE_WRITE_ACCESS)
#define IOCTL_CAN_READ_EVENT_DATA \
    CTL_CODE(FILE_DEVICE_CAN, 0x80B, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_READ_TIME \
    CTL_CODE(FILE_DEVICE_CAN, 0x80C, METHOD_BUFFERED,
FILE_READ_ACCESS)
#define IOCTL_CAN_SET_DEFAULT_FRAME_FORMAT \
    CTL_CODE(FILE_DEVICE_CAN, 0x80D, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_TEST_DEVICE \
    CTL_CODE(FILE_DEVICE_CAN, 0x80E, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CAN_READ_PROPERTIES \
    CTL_CODE(FILE_DEVICE_CAN, 0x80F, METHOD_BUFFERED,
FILE_ANY_ACCESS)

#endif /*__CANBUSIO_H__*/
```

```
//
//
// F&S Elektronik Systeme GmbH
//
// Filename:     CAN.cpp
//
//


/*- Brief Description --------------------------------*/
//  Handle communication operations on CAN interface
//

/*- files at project folder --------------------------*/
#include "StdAfx.h"
#include "CAN.h"

/*****************************************************/
//     Purpose: Construction
//-----------------------------------------------------
CCAN::CCAN()
{
  m_hCAN = INVALID_HANDLE_VALUE;

}
/*****************************************************/
//     Purpose: Destruction
//-----------------------------------------------------
CCAN::~CCAN()
{
  if( m_hCAN != INVALID_HANDLE_VALUE )
  {
    CloseHandle( m_hCAN );
  }
}
/*****************************************************/
//     Purpose:        Initial CAN port settings
//-----------------------------------------------------
bool CCAN::Init(TCHAR* pszName)
{
  m_hCAN = CreateFile(pszName,
GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0,
NULL );

  if( m_hCAN == INVALID_HANDLE_VALUE )
    return false;

  if(!DeviceIoControl( m_hCAN, IOCTL_CAN_INIT, NULL, 0,
NULL, 0, NULL, NULL ))
    return false;
```

```
  DWORD dwMode=CANBUS_FORMAT_CAN_2_0_A;

  if(!DeviceIoControl( m_hCAN,
IOCTL_CAN_SET_CAN_MODE,&dwMode, sizeof(DWORD), NULL, 0,
NULL, NULL ))
    return false;

  struct canbus_acceptance_filter filter1;

  filter1.mask = 0x3FF;
  filter1.code = 0x000;

  if( !DeviceIoControl( m_hCAN,
IOCTL_CAN_WRITE_ACCEPTANCE_FILTER, &filter1,
sizeof(struct canbus_acceptance_filter), NULL, 0, NULL,
NULL ) )
    return false;

  DWORD dwBaudrate=250000;//1000000;

  if(!DeviceIoControl( m_hCAN,
IOCTL_CAN_SET_BAUDRATE,&dwBaudrate, sizeof(DWORD), NULL,
0, NULL, NULL ))
    return false;

  // Empty buffer
  ::SetCommMask(m_hCAN,/*0*/CANBUS_EVENT_RECEIVED);
  return true;
}
```

```
/*****************************************************/
//     Purpose: Read one CAN message
//-----------------------------------------------------
bool CCAN::Read( CANMSG* pcm )
{
  DWORD dlc=0;
  DWORD dwEvent=0;

  struct canbus_event event;
  memset( &event, 0, sizeof( event ) );

  //
  while( m_hCAN != INVALID_HANDLE_VALUE )
  {
    DWORD dw;
    if( DeviceIoControl( m_hCAN,
IOCTL_CAN_READ_EVENT_DATA, NULL, NULL, &event,
sizeof(struct canbus_event), &dw, NULL ) )
    {
      // msg available

      if( event.event == CANBUS_EVENT_TRANSMITTED )
      {
        // check for lost message
        if( event.lost )
        {
          OutputMessage(_T("Lost Message: ID =
0x%x\r\n"), event.data.identifier );
        }
      }
      else if( event.event == CANBUS_EVENT_RECEIVED )
      {
        // new msg received. finish loop.
        pcm->dwIdentifier = event.data.identifier;
        pcm->byDatalength = event.data.dlc;
        memcpy( pcm->byContent, event.data.msg,
event.data.dlc>8?8:event.data.dlc );
#ifdef DEBUG
        OutputMessage(_T("CCAN: Receive msg -> ID %d,
DATA %hu %hu %hu %hu %hu %hu %hu %hu, DLC %hu \n\r"),pcm-
>dwIdentifier,pcm->byContent[0],pcm->byContent[1],pcm-
>byContent[2],pcm->byContent[3],pcm->byContent[4],pcm-
>byContent[5],pcm->byContent[6],pcm->byContent[7],pcm-
>byDatalength);
#endif
        return true;
      }
    }
    else
    {
```

```
#ifdef DEBUG
      OutputMessage(_T("-++ WaitCommEvent ++-\r\n"));
#endif
      // wait for new message
      ::SetCommMask( m_hCAN,CANBUS_EVENT_RECEIVED|
CANBUS_EVENT_BUS_ERROR| CANBUS_EVENT_WARNING);
      WaitCommEvent(m_hCAN,&dwEvent,NULL);

      if(!dwEvent)
      {
        CloseHandle(m_hCAN);
        m_hCAN = INVALID_HANDLE_VALUE;
        return false;
      }
    }
  }

  return false;
}
```

```
/*****************************************************/
//     Purpose: Write one CAN message
//-----------------------------------------------------
bool CCAN::Write( CANMSG* pcm )
{
  struct canbus_transmit_data transmit;

  transmit.identifier  = pcm->dwIdentifier;
  transmit.fmt         = CANBUS_TRANS_FMT_DEFAULT;
  transmit.rtr         = 0;
  transmit.dlc         = pcm->byDatalength;

  if( 8 != transmit.dlc )
  {
    ERRORMSG(1,(_T("CCAN::Write() transmit.dlc !=
8\r\n")));
  }

  if(transmit.dlc)
    memcpy(transmit.msg,pcm->byContent,transmit.dlc);

  DWORD dw;
  if( !DeviceIoControl( m_hCAN,
IOCTL_CAN_WRITE_TRANSMIT_DATA, &transmit, sizeof(struct
canbus_transmit_data), NULL, 0, &dw, NULL ))
  {
    OutputMessage(_T("IOCTL_CAN_WRITE_EVENT_DATA returned
0\r\n"));
    return false;
  }

#ifdef DEBUG
    OutputMessage(_T("CCAN: Send msg -> ID %d, DATA %hu
%hu %hu %hu %hu %hu %hu %hu, DLC %hu \n\r"),pcm-
>dwIdentifier,pcm->byContent[0],pcm->byContent[1],pcm-
>byContent[2],pcm->byContent[3],pcm->byContent[4],pcm-
>byContent[5],pcm->byContent[6],pcm->byContent[7],pcm-
>byDatalength);
#endif
  return true;
}
```

```
DWORD CCAN::SetCommMask( DWORD dwMask )
{
  return ::SetCommMask(m_hCAN,dwMask);
}

DWORD CCAN::Close( void )
{
  ::SetCommMask(m_hCAN,0);
  return 1;
}
```

APPENDIX: CANWRITE.EXE

```
// Windows Header Files:
#include <windows.h>
#include ".\..\inc\canbusio.h"


int _tmain(int argc, TCHAR *argv[])
{
  FILE* pfCAN;
  int t;
  TCHAR devname[50];
  BOOL bSetBaudrate = FALSE;
  DWORD dwBaudrate;

  printf("CANWRITE Version 001\r\n");

  _tcscpy( devname,_T("CID2:") );

  for(t=1;t<argc;t++)
  {
    if( !_tcscmp(argv[t],_T("-?")) )
    {
      printf("canwrite\n"\
         "  -h or -? or ? show this help\n"\
         "  -b <baudrate>       : set baudrate\n"\
         "  -d <CID1:CID2:> : selects CAN channel\n");
      return 0;
    }
    else if(!_tcscmp(argv[t],_T("-d")) && (argc > (t+1)))
    {
      _tcscpy( devname, argv[t+1] );
    }
    else if(!_tcscmp(argv[t],_T("-b")) && (argc > (t+1)))
    {
      // Baudrate setzen
      bSetBaudrate = TRUE;
      dwBaudrate = _ttol(argv[t+1]);
    }
  }

  pfCAN = _wfopen(devname,_T("w+t"));
  if( pfCAN )
  {
    HANDLE hCAN;
    hCAN = CreateFile( devname,
              GENERIC_READ | GENERIC_WRITE,
              0, NULL, OPEN_EXISTING,
              0, NULL );

    if( hCAN == INVALID_HANDLE_VALUE )
```

```
    {
      DWORD dwLastError = GetLastError();
      wprintf( _T("%s can't open device (%d)\r\n"),
               devname, dwLastError );
    }
    else
    {
      if( bSetBaudrate )
      {
        DeviceIoControl( hCAN, IOCTL_CAN_SET_BAUDRATE,
         &dwBaudrate, sizeof(DWORD),
         NULL, 0, NULL, NULL );
      }
      // Baudrate lesen
      DeviceIoControl( hCAN, IOCTL_CAN_GET_BAUDRATE,
        NULL, 0, &dwBaudrate, sizeof(DWORD),
        NULL, NULL );
      CloseHandle( hCAN );
    }

    wprintf(_T("Start sending 1000 messages at %s,
             Baudrate=%d Hz\r\n"),
             devname, dwBaudrate );

    for(t=1;t<1000;t++)
    {
      fprintf(pfCAN,"%x 0 1 12\n",t);
      fflush(pfCAN);
      Sleep(20);
    }
    fclose(pfCAN);
  }

  return 0;
}
```

APPENDIX : CANREAD.EXE

```
// Windows Header Files:
#include <windows.h>
#include ".\..\inc\canbusio.h"


int _tmain(int argc, TCHAR *argv[])
{
  FILE* pfCAN;
  int t;
  TCHAR devname[50];
  char buffer[100];
  BOOL bSetBaudrate = FALSE;
  DWORD dwBaudrate;

  printf("CANREAD Version 001\r\n");

  _tcscpy( devname,_T("CID1:") );

  for(t=1;t<argc;t++)
  {
    if( !_tcscmp(argv[t],_T("-?")) )
    {
      printf("canread\n"\
      "  -h or -? or ? show this help\n"\
      "  -b <baudrate>   : set baudrate\n"\
      "  -d <CID1:CID2:> : selects CAN channel\n");
      return 0;
    }
    else if(!_tcscmp(argv[t],_T("-d")) && (argc > (t+1)))
    {
      _tcscpy( devname, argv[t+1] );
    }
    else if(!_tcscmp(argv[t],_T("-b")) && (argc > (t+1)))
    {
      // Baudrate setzen
      bSetBaudrate = TRUE;
      dwBaudrate = _ttol(argv[t+1]);
    }
  }

  pfCAN = _wfopen(devname,_T("r"));
  if( pfCAN )
  {
    HANDLE hCAN;
    hCAN = CreateFile( devname,
          GENERIC_READ | GENERIC_WRITE, 0,  NULL,
          OPEN_EXISTING, 0,  NULL );

    if( hCAN == INVALID_HANDLE_VALUE )
```

```
    {
      DWORD dwLastError = GetLastError();
      wprintf( _T("%s can't open device (%d)\r\n"),
              devname, dwLastError );
    }
    else
    {
      if( bSetBaudrate )
      {
        DeviceIoControl( hCAN, IOCTL_CAN_SET_BAUDRATE,
       &dwBaudrate, sizeof(DWORD), NULL, 0, NULL, NULL );
      }
      // Baudrate lesen
      DeviceIoControl( hCAN, IOCTL_CAN_GET_BAUDRATE,
       NULL, 0, &dwBaudrate, sizeof(DWORD), NULL, NULL );
      CloseHandle( hCAN );
    }

    wprintf( _T("Start receiving messages at %s,
             Baudrate=%d Hz\r\n"), devname, dwBaudrate );

    while(fgets(buffer,sizeof(buffer),pfCAN))
    {
      printf(buffer);
    }
    fclose( pfCAN );
  }

  return 0;
}
```

Index