ECAN527/527D/1000/CMK6486DX
CAN Interface Board

Version 4.0.x

DOS Driver API Manual

SWM-640020004
Rev. E

RTD Embedded Technologies, INC.
103 Innovation Blvd.
State College, PA 16803-0906

Phone: +1-814-234-8087
FAX: +1-814-234-5218

E-mail
sales@rtd.com
techsupport@rtd.com

web site
http://www.rtd.com

Revision History

08/15/2001	Release 1.0
01/27/2004	AllowBufferOverwrite function added. CloseHandle function description added. GetLastError function description added. Manual format updated to RTD USA standard format. Getting Started section added. LoadPortBitDir function added Read_Digital_IO function added Write_Digital_IO function added
08/31/2004	SetBitRate function added. SetSingleFilterStandard function added. SetSingleFilterExtended function added. SetDualFilterStandard function added. SetDualFilterExtended function added. MessageObjectSetup description added. Added Section on Message Filters. Added Section on Transmission and Reception of Messages
04/05/2010	Fixed bit rate enumeration. Changed to R6250 from R6150. Documented Ecan_ResetBoard function.
04/29/2010	Removed DevNum parameter from Ecan_CreateHandle function. Updated Ecan_ResetBoard function description.

ECAN 527/527D/1000 Driver for DOS
Published by:

RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College, PA 16803-0906

Copyright 2003 RTD Embedded Technologies, Inc.
All rights reserved
Printed in U.S.A.

The RTD Logo is a registered trademark of RTD Embedded Technologies. cpuModule and utilityModule are trademarks of RTD Embedded Technologies. PS/2, PC/XT, PC/AT and IBM are trademarks of International Business Machines Inc. MS-DOS, Windows, Windows 98, Windows NT, Windows 2000 and Windows XP are trademarks of Microsoft Corp. PC/104 is a registered trademark of PC/104 Consortium. All other trademarks appearing in this document are the property of their respective owners.

INSTALLATION 5
Files in this Distribution 5
Compiling and Using Samples 5
MESSAGE RECEPTION AND TRANSMISSION 6
CAN1S.EXE 6
CAN1R.EXE 7
MESSAGE FILTERING 9
Ecan527 Message Filtering 9
Ecan1000 Message Filtering 11
API REFERENCE 15
Driver Initialization Functions 16
Ecan_BoardReset 16
Ecan_CreateHandle 17
CloseHandle 18
Ecan_GetBoardName 19
Ecan_TestBoard 20
General Board Control Functions 21
Ecan_BusConfig 21
Ecan_SetupBoard 22
Ecan_GetBuffer 25
Ecan_GetInterrupts 26
Ecan_GetStatus 27
Ecan_LoadPortBitDir 29
Ecan_Read_Digital_IO 30
Ecan_SetBitRate 31
Ecan_SetBuffer 32
Ecan_SetLeds 33
Ecan_StartBoard 34
Ecan_StopBoard 35
Ecan_Write_Digital_IO 36
Message Manipulation Functions 37
Ecan_GetMessage 37
Ecan_GetQueuesCounts 39
Ecan_MessageObjectSetup 40
Ecan_SendCommand 42
Ecan_SendMessage 43
Ecan_SetDualFilterExtended 44
Ecan_SetDualFilterStandard 45
Ecan_SetFilter 46
Ecan_SetSingleFilterExtended 48
Ecan_SetSingleFilterStandard 49
Error Reporting Functions 50
GetLastError 50
LIMITED WARRANTY 51

INSTALLATION

Files in this Distribution

All the files in this distribution will operate on the Ecan527, Ecan527D, Ecan1000 and CMK6486DX boards.

When extracted, the Ecan527/1000 drivers contain 2 directories, DOS_LIB which contains all the driver code and SAMPLES which contains 3 samples:
CAN1R – A sample that demonstrates receiving ECAN messages.
CAN1S – A sample which transmits single messages with random data.
ECANTEST – A sample that can send and receive data simultaneously (for machines with 2 cards).

The SAMPLES Directory also contains an EXE directory which holds the executable files for the samples.

Compiling and Using Samples

Each sample directory contains the source code for the sample and a Borland 3.1 project file for the sample. Each project file contains the sample files and includes the driver files from the DOS_LIB directory.

When creating your own projects remember to include the DOS_LIB directory in your include path options, make sure the compiler is using the large memory model and set Register Variables to NONE.

The functions listed in this API reference are all defined in ECANFUNC.CPP and ECANFUNC.H. The structures for the API functions are defined in ESTRUCTS.H. All the other files in the DOS_LIB directory are part of the base driver code.

MESSAGE RECEPTION AND TRANSMISSION

At the heart of CAN communication is the ability to transmit and receive messages. In this package are two examples which demonstrate message sending and reception. CAN1S.exe is the sending example and CAN1R.exe is the reception example. Both samples are capable of using any supported bitrate.

CAN1S.EXE

The setup for sending messages looks like the following:

First a handle to the card is required. To do this we use the CreateHandle call. The call requires the device number, the card type as a bool, the base address of the card and the IRQ.

```
TransmitterHND = Ecan_CreateHandle(0,Transmitter_Type,BaseAddr,IRQ);
```

Next we select the BitRate that we want communication to use. We do it now before we start the card so that we don't have to stop the card and re-start it later. The BitRate value we send to it is one of the 'BitRates' enums.

```
Ecan_SetBitRate(TransmitterHND,BitRate);
```

Next we start the card. It is normally a good idea to check the return code from this function in case there was a problem with the base address or IRQ that we set in the CreateHandle call.

```
If (!Ecan_StartBoard(TransmitterHND))
 printf("Could not Start the board!\n");
```

In order to send a message we need to create an ECAN_MESSAGE_STRUCTURE.

```
ECAN_MESSAGE_STRUCTURE Msg;
```

We would like to send a standard message (one that has 11 bits in the ID).

```
Msg.Extended = false;
```

We would like to send the message on the default message channel

```
Msg.Channel = 0;
```

We want to send the full 8 bytes

```
Msg.DataLength = 8;
```

For this example we fill the data bytes with random data.

```
for (i = 0; i < 8;i++)
 Msg.Data[i] = rand();
```

Finally we transmit the message

```
Ecan_SendMessaage(TransmitterHND,&Msg);
```

For this simple example we used the default sending object to send the message. For the Ecan1000 card the default channel should always be used. However, in the Ecan527 card there are 14 different channels that can send messages. Each channel is called a 'message object'. To send a message using a different channel we must first make sure that channel is set to be a transmitting channel using the MessageObjectSetup function. This example sets message object 7 to be a transmission object.

```
ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE ObMsg;
ObMsg.Channel = 7;
```

We want message object 7 to be a transmit object.

```
ObMsg.State = MO_TRANSMIT;
```

We want it to transmit only standard (not extended) messages

```
ObMsg.Extended = false;
```

We want the message object to be enabled.

```
ObMsg.Valid = true;
```

Finally we call message object setup with the structure we filled out.

```
Ecan_MessageObjectSetup(TransmitterHnd,&ObMsg);
```

Now we can send the message on the object by changing the channel number to 7 in the ECAN_MESSAGE_STRUCTURE.

CAN1R.EXE

The setup for receiving messages looks like the following:

First a handle to the card is required. To do this we use the CreateHandle call. The call requires the device number, the card type as a bool, the base address of the card and the IRQ.

```
ReceiverHND = Ecan_CreateHandle(0,Receiver_Type,BaseAddr,IRQ);
```

Next we select the BitRate that we want communication to use. We do it now before we start the card so that we don't have to stop the card and re-start it later. The BitRate value we send to it is one of the 'BitRates' enums.

```
Ecan_SetBitRate(ReceiverHND,BitRate);
```

For this example we also want a function to be called every time a message is received. We call our function 'ReceiverCallback'.

```
Ecan_SetupBoard(ReceiverHND,ReceiverCallback);
```

Next we start the card. It is normally a good idea to check the return code from this function in case there was a problem with the base address or IRQ that we set in the CreateHandle call.

```
If (!Ecan_StartBoard(ReceiverHND))
 printf("Could not Start the board!\n");
```

Next we want to make sure there are no masks set that might filter incoming messages

```
ECAN_FILTER_STRUCTURE Filter;
```

```
Filter.Clear();
Filter.SetAcceptCode(0,false);
Filter.SetAcceptMask(0,false,ReceiverType);
Ecan_SetFilter(ReceiverHND,&Filter);
```

Now we wait for a message to be received. When one is the ReceiverCallback function will be called. The first thing we do is get the number of messages waiting. We do this because it is possible that messages have come in so close together that we haven't been fast enough to service them individually.

```
Ecan_GetQueuesCounts(ReceiverHND,NULL,&RX_count);
```

Then, for each message in the queue we call GetInterrupts. This function tells us if there has been a receive interrupt. It also gets a message ready for us to read if there is one available.

```
Interrupt = Ecan_GetInterrupts(ReceiverHND,&QueueCount);
```

Then we check the value of Interrupt to see if there was a receive interrupt or if the queue filled up.

```
If (Interrupt == 0xFF)
 printf("Receive Queue is Full\n");
```

Next if the Interrupt was the receive bit then we get the message.

```
If (Interrupt & Int_RI_Bit)
 Ecan_GetMessage(ReceiverHND,&IncomingMsg);
```

If there were more messages waiting in the queue we go back and GetInterrupts again.

MESSAGE FILTERING

One of the most complex parts of CANbus setup is the implementation of message filters. Both the Ecan527 and the Ecan1000 cards have fully featured hardware filtering capabilities.

Ecan527 Message Filtering

The Ecan527 has 15 message objects. A message object can be thought of as a mechanism for receiving or transmitting messages. Fourteen of the objects can be set up for transmission or reception. The fifteenth message object can only receive messages. Within each message object you can set a certain bit pattern that will be compared against the Message ID of any message. If the bit patterns match then the message object will be received or sent. As well as this bit pattern within each message object, there is a provision for a Global Mask. The Mask determines which bits of the message ID must match the message object bit pattern before a successful transmission or reception of a message. Finally, there is a special mask for message object 15, which is used to determine which bits of the message object 15 ID must match the incoming ID for a successful reception to take place. In this case, the message object 15 mask is simply 'AND'ed with the Global mask. In other words, any 'don't care' bit in the Global mask will become a 'don't care' bit for message object 15. For the Ecan527, a 'don't care' bit is any bit in the mask that is set to zero.

In the package there is a sample called Ecan527.exe. This sample sets filters for message objects number 1 and number 15. Here we will go into detail about how the filtering is achieved.

After the Board is set up a Message_Object_Setup structure is created to set up message object number 1:

```
ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE ObMsg;
ObMsg.Channel = 1;
```

We want message object 1 to be a receive object.

```
ObMsg.State = MO_RECEIVE;
```

We want it to receive only standard (not extended) messages

```
ObMsg.Extended = false;
```

We want the message object to be enabled.

```
ObMsg.Valid = true;
```

Next, we have decided that we want the bit pattern in this message object to be all one's since there are 11 bits in a standard message, that equates to a hexadecimal value of 0x7FF;

```
ObMsg.SetID(0x7FF);
```

Finally we call message object setup with the structure we filled out.

```
Ecan_MessageObjectSetup(ReceiverHnd,&ObMsg);
```

After this, channels 2 through 14 are set to be transmit only. This is to prevent messages being received on them. Next, we set up message object 15.

```
ObMsg.Channel = 15;
```

```
ObMsg.State = MO_RECEIVE;
ObMsg.Extended = false;
ObMsg.Valid = true;
```

We want message object 15 to be able to receive messages with the ID of 0x7FE so we set the bit pattern in the message object to be 0x7FE.

```
ObMsg.SetID(0x7FE);
```

Then we call message object setup with this structure.

```
Ecan_MessageObjectSetup(ReceiverHND,&ObMsg);
```

Next we need to set up the Global Mask and the Message 15 special Mask to say that all the bits in the message objects must match those in the incoming message. The Global mask and the Message 15 mask are in the same structure so we must set them at the same time.

```
ECAN_FILTER_STRUCTURE Filter;
```

First, the global mask. We set it to 0x7FF to say that all 11 bits must match the ID bits in message objects 1-14. The first Boolean parameter is set to false because this is not an extended message. The second is true because this is an Ecan527 card.

```
Filter.SetAcceptMask(0x7FF,false,true)
```

Now we set the Message 15 Mask. We use 0x7FE for its mask because we don't care about the LSB. This message object can receive messages with all the bits set or just the first 10 set. We use the false parameter because this mask is not for extended messages.

```
Filter.SetMessage15Mask(0x7FE,false);
```

Finally we pass the structure to SetFilter.

```
Ecan_SetFilter(ReceiverHND,&Filter);
```

Now we have message object 1 set up to receive messages with ID's that have all 11 bits set (an ID of decimal 2047) and message object 15 set up to receive messages that have either 10 or 11 bits set (an ID of either 2046 or 2047).

When a message is received, its ID is compared to lowest message object first then each successive object until it reaches message object 15. Therefore, even though we have set up both object 1 and object 15 to receive message ID 2047 only message object 1 will receive it because it is tested first.

Remember to be careful with message filtering strategies as they can have a detrimental effect on the way the board operates. Please note the following situation as excerpted from Intel's *'82527 Serial Communications Controller Architectural Overview.'*

> "NOTE:
> Message objects programmed to transmit are also effected by the Global Masks (standard and extended). The 82527 uses the Global Mask registers to identify which of its message objects transmitted a message. If two 82527 transmit message objects have message IDs

that are non-distinct in all ``must-match" bit locations, a successful transmission of the higher numbered message object will not be recognized by the 82527. The lower numbered message object will be falsely identified as the transmit message object and its transmit request bit will be reset and its interrupt pending bit set. The actual transmit message object will re-transmit without end because its transmit request bit will not be reset.
This could result in a catastrophic condition since the higher numbered message object may dominate the CAN bus by resending its message without end.
To avoid this condition, applications should require all transmit message objects to use message IDs that are unique with respect to the “must-match" bits. If this is not possible, the application should disable lower numbered message objects with similar message IDs until the higher numbered message object has transmitted successfully.
Another configuration to avoid filtering issues is to dedicate messages 1-14 for transmit and use message 15 for receive. The message 15 mask will have no impact on messages 1-14.”

Ecan1000 Message Filtering

For the Ecan1000 there are four different functions that apply to the four different types of filtering that the board can perform. These are:

Ecan_SetSingleFilterStandard

Ecan_SetSingleFilterExtended

Ecan_SetDualFilterStandard

Ecan_SetDualFilterExtended

The functions that end in ‘standard’ are for standard message frames (those that have an 11 bit ID). The functions that end in ‘extended’ are for extended messages (messages with a 29 bit ID). Each of these functions is used in the example program called Ecan1000.exe. Simply uncomment the section that uses the filtering method you are interested in and re-compile the program to test the filter. Each filter type is described in detail in Philips’s document ‘*SJA1000 Stand-alone CAN controller DATA SHEET*’. Some of that document will be repeated here for clarity.

All of the filtering methods have two things in common. They all deal with one or more Acceptance Code Registers (ACR) and one or more Acceptance Mask Registers (AMR). An Acceptance Code Register is a bit pattern that an incoming message is compared to and must match before it will be accepted. An Acceptance Mask Register can be used to set certain bits to ‘don’t care’ before the comparison process takes place. For the Ecan1000 a ‘don’t care’ bit is any bit set to a one in a mask.

The first filter type ‘SetSingleFilterStandard’ allows an incoming message’s 11 Bit ID to be compared, its RTR bit to be compared, and the first 2 bytes of the data to be compared. If all comparisons succeed then the message is accepted. The function takes a desired 11 Bit ACR and AMR for the ID comparison. Then a 1Bit RTR ACR and AMR. Finally a 16Bit AMR and ACR for comparison against the first 2 data bytes of the message. In the example the code looks like this:

First we make the ACR for the message ID equal to 0xFF;

```
Unsigned int ID_ACR = 0xFF;
```

Then we set the Acceptance mask register to say that we want all the bits of the ACR to be compared against the incoming ID. Note that a 1 bit means don't care and a zero bit means 'must match'.

```
Unsigned int ID_AMR = 0x0;
```

We don't care about the RTR bit so we set the Acceptance Mask register to a one.

```
Unsigned int RTR_AMR = 0x1;
```

Since we told the RTR AMR that we didn't care about the bit, it doesn't matter what we set the acceptance code register to.

```
Unsigned int RTR_ACR = 0x0;
```

We don't want to filter using the first two data bytes of the message so we set the AMR for the data bytes to 'Don't care' all bits.

```
Unsigned int DATA_AMR = 0xFFFF;
```

Since we set the data AMR bits to 'don't care' it doesn't matter what we set the ACR to.

```
Unsigned int DATA_ACR = 0x0;
```

Then we call the 'SetSingleFilterStandard' function to set up the filter.

```
Ecan_SetSingleFilterStandard(ReceiverHND, ID_ACR, ID_AMR, RTR_ACR,
RTR_AMR, DATA_ACR, DATA_AMR);
```

Now we have set up a filter that will accept all messages with an ID of 0xFF and we don't care what the RTR bit is set to or what the first two data bytes are.

The next Filter function 'SetSingleFilterExtended' sets up a filter for extended messages (those with a message ID of 29 bits). The filter allows all 29 bits of the ID to be compared as well as the RTR bit. If the compares succeed the message is accepted. The function takes a desired 29 bit ACR ID, a 29 bit AMR ID and a 1 bit RTR ACR and AMR. In the example, the code looks like this:

First we make the ACR for the Message ID equal to 0xFFFFF.

```
Unsigned long ID_ACR = 0xFFFFF;
```

Next we want to signal that we want all bits to match, so we set the AMR to 'must match'

```
Unsigned long ID_AMR = 0x0;
```

We don't care about the RTR bit so we set it to 'don't care'

```
Unsigned int RTR_AMR = 0x1;
```

Since we set the RTR Mask to be 'don't care' it doesn't matter what we set the code to.

```
Unsigned int RTR_ACR = 0x0;
```

Now we call the 'SetSingleFilterExtended' function to set up the filter.

```
Ecan_SetSingleFilterExtended(ReceiverHND, ID_ACR, ID_AMR, RTR_ACR,
RTR_AMR);
```

Now we will accept any extended message that has an ID of 0xFFFFF regardless of the state of the RTR bit.

The next filter function 'SetDualFilterStandard' sets up two filters for a standard message. In a two filter configuration the message must pass only one of the filters comparison tests in order to be accepted. In other words, the message will only be rejected if it fails both filter comparisons. The comparisons check all 11 bits of the message ID, the RTR bit and the first filter also checks the first 8 bits of data in the message. The example code looks like this:

First we will make the Acceptance Code Register for Filter 1 be 0x9.

```
Unsigned int ID_ACR1 = 0x9;
```

Now we want to make the Acceptance Mask Register say all bits 'must match'.

```
Unsigned int ID_AMR1 = 0x0;
```

Next we set up filter 2. Make the Acceptance Code Register for Filter 2 be 0x10.

```
Unsigned int ID_ACR2 = 0x10;
```

Now we make the AMR say that all bits 'must match'.

```
Unsigned int ID_AMR2 = 0x0;
```

We don't care about the state of the RTR bit for either filter so set the AMR for the RTR bit to 'don't care'.

```
Unsigned int RTR_AMR1 = 0x1;

Unsigned int RTR_AMR2 = 0x1;
```

Since we set the AMR's to 'don't care' it doesn't matter what we set the RTR's ACR to.

```
Unsigned int RTR_ACR1 = 0x0;

Unsigned int RTR_ACR2 = 0x0;
```

We also want to filter based on the first byte of data in the messages. We will accept messages with 0xF as the first data byte.

```
Unsigned int Data_ACR = 0xF;
```

Set the Data Acceptance Mask register to say that all bits 'must match'

```
Unsigned int Data_AMR = 0x0;
```

Now we send the filter information to 'SetDualFilterStandard'.

```
Ecan_SetDualFilterStandard(ReceiverHND, ID_ACR1, ID_AMR1,  ID_ACR2,
ID_AMR2, RTR_ACR1, RTR_AMR1, RTR_ACR2, RTR_AMR2, Data_ACR, Data_AMR);
```

Now that the filters are set up we will receive all messages that have an ID of 0x9 AND the first data byte is 0xF OR if the message has a message ID of 0x10 it will pass the second filter test and be accepted.

The last function ‘SetDualFilterExtended’ allows a dual filter to be set up for extended messages (those messages with an ID length of 29 bits.). These filters allow bits 13 – 28 (the most significant 16 bits) of the message ID to be compared against each filter. The example code looks like the following:

Make the Acceptance Code for Filter 1 0xFFFF. (All 16 bits set).

```
Unsigned int ID_ACR1 = 0xFFFF;
```

Make the Acceptance Mask Register for filter 1 say ‘all bits must match’.

```
Unsigned int ID_AMR1 = 0x0;
```

Make the Acceptance Code Register for filter 2 0xFFFE. (15 bits set).

```
Unsigned int ID_ACR2 = 0xFFFE;
```

Make the Acceptance Mask Register say ‘All bits must match’.

```
Unsigned int ID_AMR2 = 0x0;
```

Now set up the filter using the ‘SetDualFilterExtended’ function

```
Ecan_SetDualFilterExtended(ReceiverHND, ID_ACR1, ID_AMR1, ID_ACR2,
ID_AMR2);
```

Now when an extended message that has the most significant 16 bits set (1FFFExxx) is found it will be accepted by filter 1. When a message with the most significant 15 bits set (1FFFCxxx) is found it will be accepted by filter 2.

API REFERENCE

Driver Initialization Functions

Ecan_BoardReset

Syntax
void Ecan_BoardReset(bool Ecan1000, ULONG bAddress);

Description
This routine is used to reset the ECAN527(D) or ECAN1000. Depending on the hardware version of the ECAN527D, this function will reset one controller, or both controllers. Contact technical support for more information. The safest way to reset the controllers is to call this function for each board address, and before any other board setup.

This function should be called before Ecan_CreateHandle.

Parameters

bool	Ecan1000	ECAN527 or ECAN1000
ULONG	bAddress	Board Address

Return Value
N/A

Example Function Call
Ecan_BoardReset(BoardType, BoardAddress);

Ecan_CreateHandle

Syntax

HANDLE Ecan_CreateHandle(bool Ecan1000=false, ULONG bAddress=0, USHORT bIrq=0);

Description

This routine is used to open Ecan1000 or Ecan527 board. This function must be called before any API function call. At the end of the application program, the board must be closed with the CloseHandle function.

Parameters

Ecan1000:	Open Ecan1000 board – if true, Ecan527 board – if False.
bAddress:	The board's Base Address.
bIrq:	The board's Interrupt Channel. If bIrq=0, interrupts will not be used.

Return Value

Device handle	(pointer to the driver object). Board is opened successfully.
NULL	There was an error while closing the board.

To get extended error information, call GetLastError.

Example function call

```
…
HANDLE hDevice = Ecan_CreateHandle(0, (true), 0x300, 7);        //
Open the first of the Ecan1000 boards.
…
Ecan_StartBoard(hDevice);
…
working with the board
…
Ecan_StopBoard(hDevice);
CloseHandle(hDevice);    // Close the driver after work.
```

CloseHandle

Syntax
void CloseHandle(HANDLE hDevice);

Description
This routine releases the ECAN board and clears up memory. This function must be called once for each call to Ecan_CreateHandle.

Parameters
None

Return Value
None

Example Function Call
```
…
HANDLE hDevice = Ecan_CreateHandle(0, (true), 0x300, 7);        //
Open the first of the Ecan1000 boards.
…
Ecan_StartBoard(hDevice);
…
working with the board
…
Ecan_StopBoard(hDevice);
CloseHandle(hDevice);    // Close the driver after work.
```

Ecan_GetBoardName

Syntax
ULONG Ecan_GetBoardName(HANDLE hDevice);

Description
This routine is used to determine the board name (Ecan1000 or Ecan527).

Parameters
hDevice: Device handle.

Return Value
1000 For the Ecan1000 board.
527 For the Ecan527 board.

Example function call
```
…
if(Ecan_GetBoardName(hDevice )==527)
        printf(“Ecan527 board”);
else
        printf(“Ecan1000 board”);
…
```

Ecan_TestBoard

Syntax
bool Ecan_TestBoard(HANDLE hDevice);

Description
This routine is used to hardware test a board.
This routine turns the board to the RESET MODE.

Parameters
hDevice: Device handle.

Return Value
TRUE Hardware test passed.
FALSE Hardware test Failed.

Example function call

```
If(!Ecan_TestBoard(hDevice ))
{
        printf("Board test failed, check the hardware settings.");
        CloseHandle(hDevice);
}
```

General Board Control Functions

Ecan_BusConfig

Syntax
bool Ecan_BusConfig(HANDLE hDevice, UCHAR BusTiming0, UCHAR BusTiming1, UCHAR ClockOut=0, UCHAR BusConfig=0xff);

Description
This routine sets CAN Bit Timing and Bus configuration. Ecan_StopBoard and Ecan_StartBoard should be called after this function has been used. If you want to use standard Bit Timings it is easier to alter the BitRate by calling the ECAN_SetBitRate function.

Parameters

hDevice:	Device handle.
BusTiming0	Value for the BUS TIMING REGISTER 0 (BIT TIMING REGISTER 0)
BusTiming1	Value for the BUS TIMING REGISTER 1 (BIT TIMING REGISTER 1)
ClockOut	Value is used to set frequency divisor at the external CLKOUT pin relatively to the frequency of the external oscillator (XTAL). Divisor may have value of 1, 2, 4, 6, 8, 10, 12 or 14. If ClockOut =0, then divisor will not be changed.
BusConfig	Value for the Output Control Register (Bus Configuration Register) If BusConfig=0xff, then register will be reset to the default value.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call
```
…
Ecan_BusConfig( hDevice, 0, 0x14, 0, 0xff);    // 1Mbit/s
Ecan_StartBoard(hDevice);
…
```

Ecan_SetupBoard

Syntax

bool Ecan_SetupBoard(HANDLE hDevice, cbECAN_Isr hEvent, bool ReceiveIntEn=true, bool ErrorIntEn=false, bool TransmitIntEn=false, bool BusErrorIntEn=false, bool DataOverrunIntEn=false, bool ArbitrationLostIntEn=false, bool ErrorPassiveIntEn=false, bool WakeUpIntEn=false, ULONG RXSize = 0x0f, ULONG TXSize = 0x0f);

Description

This routine is used to setup the board.
Connect user's interrupt service function.
Select interrupt sources of which interrupts the application will be notified.
Interrupts that are not selected will be handled inside the driver only. By default, the application gets Receive Interrupt only.
This routine turns the board to the RESET MODE.

Parameters

HDevice:	Device handle.
hEvent:	Pointer to the user's interrupt service function.
ReceiveIntEn:	TRUE – turn on Receive Interrupt notification
ErrorIntEn:	TRUE – turn on Error Interrupt notification
TransmitIntEn:	TRUE – turn on Transmit Interrupt notification
BusErrorIntEn:	TRUE – turn on Bus Error Interrupt notification
DataOverrunIntEn:	TRUE – turn on Data Overrun Interrupt notification
ArbitrationLostIntEn:	TRUE – turn on Arbitration Lost Interrupt notification
ErrorPassiveIntEn:	TRUE – turn on Error Passive Interrupt notification
WakeUpIntEn:	TRUE – turn on Wake-Up Interrupt notification
RXSize:	Number of messages allowed in Receive buffer.
TXSize:	Number of messages allowed in Transmit buffer.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

```
void far ReceiverCallBack()
{
        // Process the receiving interrupt, get an incoming message frame.
}
…
void main()
{
HANDLE hDevice = Ecan_CreateHandle(0, (true), 0x300, 7); // Open the
first of the Ecan1000 boards.
Ecan_SetupBoard (hDevice, ReceiverCallBack);
Ecan_StartBoard();
…
working with the board
…
Ecan_StopBoard();
CloseHandle(hDevice);     // Close the driver after work.
}
```

Ecan_GetBuffer

Syntax
ULONG Ecan_GetBuffer(HANDLE hDevice, USHORT StartAddress, USHORT Count, void* Buffer, ULONG BuffSize);

Description
This routine allow direct read from the CAN controller's internal RAM (80-byte for Ecan1000 boards and 256-byte for Ecan527 boards). Through this function and Ecan_SetBuffer you can take access to all specific possibilities of the board.

Parameters

hDevice:	Device handle.
StartAddress:	Offset from the start of the CAN controller's internal RAM
Count:	The number of bytes to be transferred from the board.
Buffer:	Pointer to the user-supplied buffer that is to receive the data read from the controller.
BuffSize:	Buffer size.

Return Value
The number of bytes transferred to the buffer.

Example function call
```
UCHAR ModeReg;
Ecan_GetBuffer(hDevice, 0, 1, &ModeReg, sizeof(ModeReg)); // reads Mode Register of Ecan1000
printf("Mode Register = %d",  ModeReg);
```

Ecan_GetInterrupts

Syntax
UCHAR Ecan_GetInterrupts(HANDLE hDevice, ULONG *QueueSize=NULL, bool DontQueueUse=false);

Description
This routine is used to determine the interrupt source.

Parameters
hDevice:	Device handle.
QueueSize:	Pointer to a variable that receives the length of the receive queue. If DontQueueUse is TRUE, QueueSize receives 0.
DontQueueUse:	Normally an interrupt data receives from the driver's Interrupt queue. But if this parameter is TRUE an interrupt data receives directly from the hardware.

Return Value
The Interrupt Register value in format of the Ecan1000 board.

Remarks:
Interrupt register values:

Int_RI_Bit	0x01	receive interrupt bit
Int_TI_Bit	0x02	transmit interrupt bit
Int_EI_Bit	0x04	error warning interrupt bit
Int_DOI_Bit	0x08	data overrun interrupt bit
Int_WUI_Bit	0x10	wake-up interrupt bit
Int_EPI_Bit	0x20	error passive interrupt bit
Int_ALI_Bit	0x40	arbitration lost interrupt bit
Int_BEI_Bit	0x80	bus error interrupt bit

Example function call
```
// Process all interrupts from the queue
UCHAR Interrupt;
while((Interrupt = Ecan_GetInterrupts(hDevice))!=0) // While not all accumulated
   //  interrupts are processed
{
// Process the interrupt (get received message for example).
}
```

Ecan_GetStatus

Syntax
bool Ecan_GetStatus(HANDLE hDevice,
PECAN_STATUS_STRUCTURE msg);

Description
This routine is used to receive the CAN controller status information.

Parameters

hDevice:	Device handle.
msg:	Pointer to a variable that receives the length of the transmit queue.
RX_Count:	Pointer to a variable that receives the length of the receive queue.
ClearRX:	TRUE – clears the receive queue.
ClearTX:	TRUE – clears the transmit queue.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks:
ECAN_STATUS_STRUCTURE declaration:

```
typedef struct _ECAN_STATUS_STRUCTURE // Header of command
{
    bool BusOf;          // BusOf status
    bool Warning;        // Erorr Warning
    bool WakeUp;         // not use in Ecan1000
    bool TXOK;           // in Ecan1000 - Transmission Complete
    bool RXOK;           // Receive Message Successfully (not use in
                         Ecan1000)
    bool TS;             // Transmit Status
    bool RS;             // Receive Status
    bool TBS;            // Transmit Buffer Status
    bool DOS;            // Data Overrun Status
    bool RBS;            // Receive Buffer Status
    UCHAR    Arbitration;    // Arbitration lost capture register value
    UCHAR    ErrorCode;    // Last Error Code register value
    bool DontQueueUse;       // Don't use software queues in driver.
    UCHAR    Reserved[3]; // For alignments on 4 bytes
```

```
        _ECAN_STATUS_STRUCTURE(){memset( this, 0, sizeof(*this) );}
} ECAN_STATUS_STRUCTURE, *PECAN_STATUS_STRUCTURE;
```

Example function call

```
ECAN_STATUS_STRUCTURE Status;
if(Ecan_GetStatus( HANDLE hDevice, &Status))
{
if(Status.BusOf)     printf("Bus-of state");
}
```

Ecan_LoadPortBitDir

Syntax
bool Ecan_LoadPortBitDir(HANDLE hDevice,
PECAN_PORTBITDIR_STRUCTURE msg);

Description
This function sets the direction of the pins on the DIO port (Ecan527 Only)

Parameters
hDevice: Device handle.
PECAN_PORTBITDIR_STRUCTURE

Bit7	True = Output, False = Input
Bit6	True = Output, False = Input
Bit5	True = Output, False = Input
Bit4	True = Output, False = Input
Bit3	True = Output, False = Input
Bit2	True = Output, False = Input
Bit1	True = Output, False = Input
Bit0	True = Output, False = Input

Return Value
True Success.
False Failure.
To get extended error information, call GetLastError.

Example Function Call
```
ECAN_PORTBITDIR_STRUCTURE PortDirMsg;
PortDirMsg.Bit7 = false;
PortDirMsg.Bit6 = false;
PortDirMsg.Bit5 = false;
PortDirMsg.Bit4 = false;
PortDirMsg.Bit3 = false;
PortDirMsg.Bit2 = false;
PortDirMsg.Bit1 = false;
PortDirMsg.Bit0 = false;
Ecan_LoadPortBitDir(hDevice,&PortDirMsg);
```

Ecan_Read_Digital_IO

Syntax
bool Ecan_Read_Digital_IO(HANDLE hDevice,
PECAN_READWRITEDIGITALIO_STRUCTURE msg);

Description
This function reads a value from the digital IO port. Note: To get any value back from the port at least one pin must be set for input. See 'Ecan_LoadPortBitDir'.

Parameters
hDevice Device handle
PECAN_READWRITEDIGITAL_IO
Value Value read from the port.

Return Value
True Success.
False Failure.
To get extended error information, call GetLastError.

Example Function Call
```
ECAN_READWRITEDIGITALIO_STRUCTURE DIOValMsg;
Ecan_Read_Digital_IO(ReceiverHND,&DIOValMsg);
printf("Value Read %d  \n\r",DIOValMsg.Value);
```

Ecan_SetBitRate

Syntax
bool Ecan_SetBitRate(HANDLE hDevice,BitRates BitRate)

Description
This function alters the communication bitrate. The board MUST be stopped and re-started for the change to take effect.

Parameters

hDevice		Device handle
BitRate		One of the BitRates enums:
	R1000000	1MB/s
	R800000	
	R500000	
	R400000	
	R250000	
	R200000	
	R160000	
	R125000	
	R100000	
	R80000	
	R62500	
	R50000	
	R40000	
	R31250	
	R25000	
	R20000	
	R16000	
	R15625	
	R12500	
	R10000	
	R8000	
	R7813	(actually 7812.5)
	R6250	
	R5000	

Return Value

True	Success.
False	Failure.

To get extended error information, call GetLastError.

Example Function Call
Ecan_SetBitRate(ReceiverHND,R500000); // Set bitrate to 500K

Ecan_SetBuffer

Syntax
ULONG Ecan_SetBuffer(HANDLE hDevice, USHORT StartAddress, USHORT Count, void* Buffer, ULONG BuffSize);

Description
This routine allow direct write to the CAN controller's internal RAM (80-byte for Ecan1000 boards and 256-byte for Ecan527 boards). Through this function and Ecan_SetBuffer you can take access to all specific possibilities of the board.

Parameters
hDevice:	Device handle.
StartAddress:	Offset from start of the CAN controller's internal RAM
Count:	The number of bytes to be transferred from the buffer.
pBuffer:	A pointer to the user-supplied buffer that contains the data to be written to the controller.
BuffSize:	Buffer size.

Return Value
The number of bytes transferred to the controller.

Example function call
```
// Write 0 to the Mode Register of the SJA1000 controller (Ecan1000 board)
UCHAR ModeReg=0;
Ecan_SetBuffer(hDevice, 0, 1, &ModeReg, sizeof(ModeReg));
printf("Mode Register = %d", ModeReg);
```

Ecan_SetLeds

Syntax
bool Ecan_SetLeds(HANDLE hDevice, bool RedLed, bool GreenLed);

Description
This routine controls Leds on the Ecan527 - class board. Note that pins 1 and 2 of the Digital IO are connected to the Led's. Turning the Led's on will output data on those pins. Also, the pins must be set to output (default) for the Led's to light (see Ecan_LoadPortBitDir);

Parameters
hDevice:	Device handle.
RedLed:	TRUE – enable red led.
	FALSE– disable red led.
RedGreen:	TRUE – enable green led.
	FALSE– disable green led.

Return Value
TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call
Ecan_SetLeds(hDevice, false, true); // the Red led is off and the Green led is on.

Ecan_StartBoard

Syntax
bool Ecan_StartBoard(HANDLE hDevice);

Description
This routine switches board to OPERATING MODE.

Parameters
hDevice: Device handle.

Return Value
TRUE If there is no error.
FALSE If there is an error.
To get extended error information, call GetLastError.

Example function call

```
…
HANDLE hDevice = Ecan_CreateHandle(0, (true), 0x300, 7);        //
Open the first of the Ecan1000 boards.
…
Ecan_StartBoard();
…
working with the board
…
Ecan_StopBoard();
CloseHandle(hDevice);    // Close the driver after work.
```

Ecan_StopBoard

Syntax
bool Ecan_StartBoard(HANDLE hDevice);

Description
This routine turns the board to the RESET MODE.

Parameters
hDevice: Device handle.

Return Value
TRUE If there is no error.
FALSE If there is an error.
To get extended error information, call GetLastError.

Example function call

```
…
HANDLE hDevice = Ecan_CreateHandle(0, (true), 0x300, 7);
        //Open the first of the Ecan1000 boards.
…
Ecan_StartBoard();
…
working with the board
…
Ecan_StopBoard();
CloseHandle(hDevice);    // Close the driver after work.
```

Ecan_Write_Digital_IO

Syntax

bool Ecan_Write_Digital_IO(HANDLE hDevice,
PECAN_READWRITEDIGITALIO_STRUCTURE msg);

Description

This routine writes a value out to the DIO port. Note: For any value to appear on the port at least one of the pins must be set to output using 'Ecan_LoadPortBitDir'.

Parameters

hDevice Device handle
PECAN_READWRITEDIGITAL_IO
Value Value to write to the port

Return Value

True Success.
False Failure.
To get extended error information, call GetLastError.

Example Function Call

```
ECAN_READWRITEDIGITALIO_STRUCTURE DIOValMsg;
DIOValMsg.Value = VALUEOUT;
        Ecan_Write_Digital_IO(ReceiverHND,&DIOValMsg);
```

Message Manipulation Functions

Ecan_GetMessage

Syntax
bool Ecan_GetMessage(HANDLE hDevice,
PECAN_MESSAGE_STRUCTURE msg);

Description
This routine is used to retrieve a received message.

Parameters
hDevice: Device handle.
msg: A pointer to the ECAN_MESSAGE_STRUCTURE.

Return Value
TRUE If there is no error.
FALSE If there is an error.
To get extended error information, call GetLastError.

Remarks
ECAN_MESSAGE_STRUCTURE declaration:

```
typedef struct _ECAN_MESSAGE_STRUCTURE // Header of command
{
    UCHAR    Channel;    // Not used in the Ecan1000.
                         //In Ecan527, Channel 0 - Default
                         Transmitting or Default Receiving
                         Message Object
                         //1 - 15 Message Object by number.
    bool  Extended;      // If true - Extended frame format
    bool  Remote;        // if true - remote frame (Attention For
                         Ecan527! this value = inverse direction
                         bit)
    UCHAR    ID_0;       // Identifier 0 (Arbitration 0 - in Ecan527
                         documentation)
    UCHAR    ID_1;       // Identifier 1 (Arbitration 1 - in Ecan527
                         documentation)
    UCHAR    ID_2;       // Identifier 2 (Arbitration 2 - in Ecan527
                         documentation)
    UCHAR    ID_3;       // Identifier 3 (Arbitration 3 - in Ecan527
                         documentation)
    UCHAR    DataLength; // Data Length Code (DLC)
```

```
        UCHAR       Data[8];        // Data Bytes
        bool    NextMsg;            // if true - In the queue available a next
                                    message
        bool    DontQueueUse;       // Don't use driver's software queues
        UCHAR       Reserved[2]; // For alignments on 4 bytes
        _ECAN_MESSAGE_STRUCTURE(){ Clear(); };
        void Clear(){ memset( this, 0, sizeof(*this) ); }
        // Identifiers services functions
        ULONG GetID()
        {
                ULONG ret = ID_0; ret = (ret<<8)|ID_1;
                if( Extended )ret = (((ret<<8)|ID_2)<<8)|ID_3;
                return ret;
        };
        void SetID(ULONG ID)
        {
                if( Extended ){ ID_3 = (UCHAR)ID; ID = ID>>8; ID_2 =
(UCHAR)ID; ID = ID>>8; };
                ID_1 = (UCHAR)ID; ID = ID>>8; ID_0 = (UCHAR)ID;
        };

} ECAN_MESSAGE_STRUCTURE,
*PECAN_MESSAGE_STRUCTURE;
```

Example function call

```
ECAN_MESSAGE_STRUCTURE Msg;//
ECAN_MESSAGE_STRUCTURE with default settings
if (!Ecan_GetMessage(hDrvice, &Msg ) )
{
        printf("Can not receive a message!");
}
```

Ecan_GetQueuesCounts

Syntax
bool Ecan_GetQueuesCounts(HANDLE hDevice, ULONG *TX_Count, ULONG *RX_Count=NULL, bool ClearRX=false, bool ClearTX=false)

Description
This routine is used to receive the messages queues length and/or to clear these queues.

Parameters

hDevice:	Device handle.
TX_Count:	Pointer to a variable that receives the length of the transmit queue.
RX_Count:	Pointer to a variable that receives the length of the receive queue.
ClearRX:	TRUE – clears the receive queue.
ClearTX:	TRUE – clears the transmit queue.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

```
// Wait until the transmit queue will freed
ULONG TX_Count = 0x80;
while ( TX_Count >= 1 )
{
Ecan_GetQueuesCounts( hTransmitter, &TX_Count);
};
```

Ecan_MessageObjectSetup

Syntax
bool Ecan_MessageObjectSetup(HANDLE hDevice,
PECAN_MESSAGE_OBJECT_SETUP_STRUCTURE msg);

Description
This routine allows the application to setup message objects

Parameters
hDevice: Device handle.
msg: Handle to the
_ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE

Return Value
TRUE If there is no error.
FALSE If there is an error.
To get extended error information, call GetLastError.

Remarks:
_ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE declaration

```
typedef struct _ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE // Header of command
{
        UCHAR Channel;// Number of Message Object
                                // if Channel==0, then using default Transmit or Receive Message Object
        MESSAGE_OBJECT_STATE State; // Program object for one of three tasks
                                // MO_TRANSMIT-transmit
                                // MO_REMOTE_TRANSMIT - transmit after receiving remote frame with same ID
                                // MO_RECEIVE - receive

        bool    Valid;          // If true  - enable Message Object after setup
        bool    Extended;       // If true - use Extended frame format
        bool    RXIE;           // Enable Receive Message Interrupt for this object
        bool    TXIE;           // Enable Transmit Message Interrupt for this object
        bool    DontQueueUse;// if FALSE for MO_TRANSMIT, then changes Default Transmitting Object to this
        bool    MakeDefault;
        UCHAR ID_0;             // Identifier 0 (Arbitration 0 - in Ecan527 documentation)
        UCHAR ID_1;             // Identifier 1 (Arbitration 1 - in Ecan527 documentation)
        UCHAR ID_2;             // Identifier 2 (Arbitration 2 - in Ecan527 documentation)
        UCHAR ID_3;             // Identifier 3 (Arbitration 3 - in Ecan527 documentation)
        UCHAR   Reserved[3];
        _ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE()
        { Clear(); }
        void Clear();
        // Identifiers services functions
        ULONG GetID()
        void SetID(ULONG ID)
        } ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE, *PECAN_MESSAGE_OBJECT_SETUP_STRUCTURE;
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_MESSAGE_OBJECT_SETUP,
msg, sizeof(*msg), NULL, 0, &ret_code, NULL) ;
```

Ecan_SendCommand

Syntax
bool Ecan_SendCommand(HANDLE hDevice, bool TR=false, bool RRB=false, bool AT=false, bool CDO=false, bool SRR=false);

Description
A command bit initiates an action within the transfer layer of the SJA1000 (Ecan1000 board).

Parameters
hDevice:	Device handle.
TR:	Transmission Request.
RRB:	Release Receive Buffer.
AT:	Abort Transmission.
CDO:	Clear Data Overrun.
SRR:	Self Reception Request.

Return Value
TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call
Ecan_SendCommand(hDevice, false, false, true); // Abort current Transmission

Ecan_SendMessage

Syntax
bool Ecan_SendMessage(HANDLE hDevice,
PECAN_MESSAGE_STRUCTURE msg);

Description
This routine is used to transmit message.

Parameters
hDevice: Device handle.
msg: A pointer to the ECAN_MESSAGE_STRUCTURE (see Ecan_GetMessage for details).

Return Value
TRUE If there is no error.
FALSE If there is an error.
To get extended error information, call GetLastError.

Example function call
```
ECAN_MESSAGE_STRUCTURE Msg;
Msg.Extended=true;      // Extended frame format
Msg.Channel=0;          // by default
Msg.DataLength=2;       // 2 bytes to transmit
Msg.Data[0]=0xa5;       // Data 0
Msg.Data[1]=0x5a;       // Data 1
Msg.SetID(0x01);        // Message Identifier

// Send message through the transmit queue (and the default transmit
object in Ecan527).
Ecan_SendMessage( hDevice, &Msg );
…
```

Ecan_SetDualFilterExtended

Syntax

bool Ecan_SetDualFilterExtended(HANDLE hDevice,unsigned int ID_ACR1,unsigned int ID_AMR1,unsigned int ID_ACR2,unsigned int ID_AMR2)

Description

This function sets up dual filters for extended messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR1	16 Bit Acceptance Code for the message ID for Filter 1
ID_AMR1	16 Bit Acceptance Mask for the message ID for Filter 1
ID_ACR2	16 Bit Acceptance Code for the message ID for Filter 2
ID_AMR2	16 Bit Acceptance Mask for the message ID for Filter 2

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_SetDualFilterStandard

Syntax

bool Ecan_SetDualFilterStandard(HANDLE hDevice,unsigned int ID_ACR1,unsigned int ID_AMR1,unsigned int ID_ACR2,unsigned int ID_AMR2, unsigned int RTR_ACR1,unsigned int RTR_AMR1,unsigned int RTR_ACR2,unsigned int RTR_AMR2, unsigned int Data_ACR,unsigned int Data_AMR)

Description

This function sets up dual filters for standard messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR1	11 Bit Acceptance Code for the message ID for Filter 1
ID_AMR1	11 Bit Acceptance Mask for the message ID for Filter 1
ID_ACR2	11 Bit Acceptance Code for the message ID for Filter 2
ID_AMR2	11 Bit Acceptance Mask for the message ID for Filter 2
RTR_ACR1	1 Bit Acceptance Code for the RTR Bit for Filter 1
RTR_AMR1	1 Bit Acceptance Mask for the RTR Bit for Filter 1
RTR_ACR2	1 Bit Acceptance Code for the RTR Bit for Filter 2
RTR_AMR2	1 Bit Acceptance Mask for the RTR Bit for Filter 2
Data_ACR	8 Bit Acceptance Code for the first data byte for Filter 1
Data_AMR	8 Bit Acceptance Mask for the first data byte for Filter 1

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_SetFilter

Syntax
bool Ecan_SetFilter(HANDLE hDevice, PECAN_FILTER_STRUCTURE msg);

Description
This routine sets message filters and masks configurations.

Parameters
hDevice: Device handle.
msg: Pointer to the _ECAN_FILTER_STRUCTURE class object, containing information about filters and interrupt masks to be set.

Return Value
TRUE If there is no error.
FALSE If there is an error.
To get extended error information, call GetLastError.

Remarks:
ECAN_STATUS_STRUCTURE declaration:

```
typedef struct _ECAN_FILTER_STRUCTURE // Header of command
{
        UCHAR AC0;    // Acceptance Code Register 0 (For Ecan527 - Not used)
        UCHAR AC1;    // Acceptance Code Register 1 (For Ecan527 - Not used)
        UCHAR AC2;    // Acceptance Code Register 2 (For Ecan527 - Not used)
        UCHAR AC3;    // Acceptance Code Register 3 (For Ecan527 - Not used)
        UCHAR AM0;    // Acceptance Mask Register 0 (For Ecan527 - Global Mask Standard (Address-06H))
        UCHAR AM1;    // Acceptance Mask Register 1 (For Ecan527 - Global Mask Standard (Address-07H))
        UCHAR AM2;    // Acceptance Mask Register 2 (For Ecan527 - Not used)
        UCHAR AM3;    // Acceptance Mask Register 3 (For Ecan527 - Not used)
        // Not used in the Ecan1000
        UCHAR ME0;    // Global Mask Extended (Address-08H)
        UCHAR ME1;    // Global Mask Extended (Address-09H)
        UCHAR ME2;    // Global Mask Extended (Address-0AH)
        UCHAR ME3;    // Global Mask Extended (Address-0BH)
        UCHAR MM0;    // Message 15 mask (Address 0CH)
        UCHAR MM1;    // Message 15 mask (Address 0DH)
        UCHAR MM2;    // Message 15 mask (Address 0EH)
        UCHAR MM3;    // Message 15 mask (Address 0FH)

        bool      DualFilter; // For Ecan1000 only
        bool      DontQueueUse;
        UCHAR Reserved[2];      // For alignments on 4 bytes
```

```
    _ECAN_FILTER_STRUCTURE(){ Clear(); };
    void Clear()

    void SetAcceptCode(ULONG ID, bool LongFormat=true)

    ULONG GetAcceptCode(bool LongFormat=true)

    void SetAcceptMask(ULONG ID, bool LongFormat=true, bool Ecan527=false)

    ULONG GetAcceptMask(bool LongFormat=true, bool Ecan527=false)

    // For Ecan 527
    void SetExtended(ULONG ID)

    ULONG GetExtended()

    void SetMessage15Mask(ULONG ID)

    ULONG GetMessage15Mask()

}ECAN_FILTER_STRUCTURE, *PECAN_FILTER_STRUCTURE;
```

Example function call

```
ECAN_FILTER_STRUCTURE Filter;
…
// Fill the filter structure.
…
bool Ecan_SetFilter(hDevice, Filter);
```

(Also see the section on Message Filtering in this manual)

Ecan_SetSingleFilterExtended

Syntax

bool Ecan_SetSingleFilterExtended(HANDLE hDevice,unsigned long ID_ACR,unsigned long ID_AMR,unsigned int RTR_ACR,unsigned int RTR_AMR)

Description

This function sets up a single filter for extended messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR	29 Bit Acceptance Code for the message ID
ID_AMR	29 Bit Acceptance Mask for the message ID
RTR_ACR	1 Bit Acceptance Code for the RTR Bit
RTR_AMR	1 Bit Acceptance Mask for the RTR Bit

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_SetSingleFilterStandard

Syntax

bool Ecan_SetSingleFilterStandard(HANDLE hDevice,unsigned int ID_ACR,unsigned int ID_AMR,unsigned int RTR_ACR,unsigned int RTR_AMR,unsigned int Data_ACR,unsigned int Data_AMR)

Description

This function sets up a single filter for standard messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR	11 Bit Acceptance Code for the message ID
ID_AMR	11 Bit Acceptance Mask for the message ID
RTR_ACR	1 Bit Acceptance Code for the RTR Bit
RTR_AMR	1 Bit Acceptance Mask for the RTR Bit
DATA_ACR	16Bit Acceptance Code for the first 2 data bytes
DATA_AMR	16Bit Acceptance Mask for the first 2 data bytes

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Error Reporting Functions

GetLastError

Syntax
ULONG GetLastError();

Description
This routine returns the last error code after a function returns failure.

Parameters
None

Return Value
The return value of this function can be one of the following

STATUS_SUCCESS	0x00000000L
STATUS_UNSUCCESSFUL	0xC0000001L
STATUS_INVALID_PARAMETER	0xC000000DL
STATUS_RESOURCE_TYPE_NOT_FOUND	0xC000008AL
STATUS_INSUFFICIENT_RESOURCES	0xC000009AL
STATUS_ADAPTER_HARDWARE_ERROR	0xC00000C2L
STATUS_DEVICE_BUSY	0x80000011L
STATUS_BUS_RESET	0x8000001DL
STATUS_NO_MEMORY	0xC0000017L
STATUS_BUFFER_TOO_SMALL	0xC0000023L
STATUS_DATA_OVERRUN	0xC000003CL

LIMITED WARRANTY

RTD Embedded Technologies, Inc. warrants the hardware and software products it manufactures and produces to be free from defects in materials and workmanship for one year following the date of shipment from RTD Embedded Technologies, INC. This warranty is limited to the original purchaser of product and is not transferable.

During the one year warranty period, RTD Embedded Technologies will repair or replace, at its option, any defective products or parts at no additional charge, provided that the product is returned, shipping prepaid, to RTD Embedded Technologies. All replaced parts and products become the property of RTD Embedded Technologies. Before returning any product for repair, customers are required to contact the factory for an RMA number.

THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY PRODUCTS WHICH HAVE BEEN DAMAGED AS A RESULT OF ACCIDENT, MISUSE, ABUSE (such as: use of incorrect input voltages, improper or insufficient ventilation, failure to follow the operating instructions that are provided by RTD Embedded Technologies, "acts of God" or other contingencies beyond the control of RTD Embedded Technologies), OR AS A RESULT OF SERVICE OR MODIFICATION BY ANYONE OTHER THAN RTD Embedded Technologies. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES ARE EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND RTD Embedded Technologies EXPRESSLY DISCLAIMS ALL WARRANTIES NOT STATED HEREIN. ALL IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES FOR MECHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED TO THE DURATION OF THIS WARRANTY. IN THE EVENT THE PRODUCT IS NOT FREE FROM DEFECTS AS WARRANTED ABOVE, THE PURCHASER'S SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. UNDER NO CIRCUMSTANCES WILL RTD Embedded Technologies BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, AND SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College PA 16803-0906
USA
Our website: www.rtd.com