ADwin-CAN

Hardware Features

Register Assignment

Programming Examples

Version 1.0

August 97

Table of Contents

Installation

The ***ADwin-CAN*** board is connected to the processor module with connectors which are at the back of the board. The board operates in the stand-alone mode with the processor module or can be combined with other ***ADwin*** boards. In case further ***ADwin*** components are needed they have to be connected by using a backplane. The following diagram shows a possible combination of the board, seen from above. (***ADwin-CAN*** and an other ***ADwin*** module).

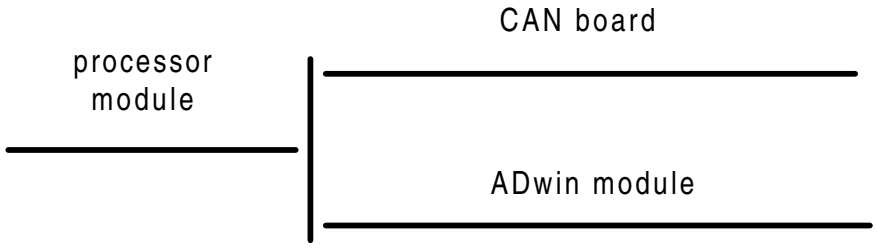

Figure 1: combination of a CAN board with an *ADwin* module

The ***ADwin*** module shown here can be any module of the ***ADwin*** series. If the length of the module exceeds the standard length the ***ADwin***-CAN board has to be expanded by an additional board. The complete expansion system encompasses up to three ***ADwin*** boards or one ***ADwin-light*** board. The ***ADwin-CAN*** board and every expansion board need an ISA slot.

The whole system is plugged in the PC. Now the board is located at the I/O area of the PC. The I/O address can be set with the white DIP switches. Please, note that the address has to be set on the left board (seen from the back of the PC). The switches on the other boards have to be set to "off". The following table shows the assignment between addresses and the switch positions.

	Switch no.							
	2	3	4	5	6	7	8	9
Base address:								
150H	off	off	on	off	on	off	on	off
190H	off	off	on	off	off	on	on	off
200H	off	off	off	off	off	off	off	on
300H	off	off	off	off	off	off	on	on

Table 1: Setting of the base addresses

Now the board can be booted and programmed.

The CAN bus connection is made by the 9 pin D-type male connector (below). The 9 pin D-type female connector (above) has no pin assignments. The following figure shows the pin assignment of the 9 pin D-type connector.

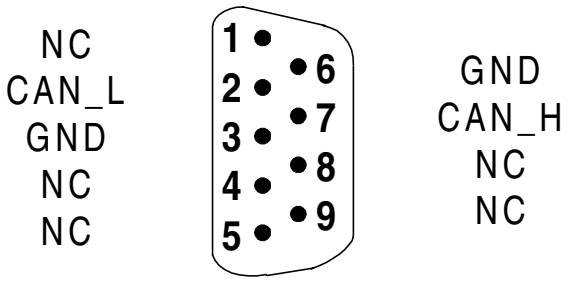

Figure 2: Pin assignment of the 9 pin D-type connector

Functions

The ***ADwin-CAN*** board provides the connection between the processor module of the ***ADwin*** board and the CAN bus. The board is equipped with the CAN controller AN82527. Therefore the operation of the ***ADwin***-CAN board is based on the functions of the CAN controller. There is direct access to the register of the controller from ***ADbasic***.

Operation of the CAN-Controller AN82527

The controller has 255 registers. These registers are used on the one hand for the configuration and status display of the CAN controller. Here the bus speed and interrupt handling etc. are set. Moreover the controller provides 15 objects for messages which are responsible for the communication. A message object can be configured to be able to send or receive messages. An exception is the message object number 15, which can only be used for receiving data. Contrary to the other objects the message object 15 has a buffer for further messages. The buffered message follows when the first incoming message has been read out.

If a message object for receiving messages is used, the corresponding bits for receiving messages have to be set and an identifier has to be specified. Afterwards all incoming messages with the identifier are stored in this message object.

It is also possible to use a message object to receive messages with different identifiers. This is done by using the mask (CAN registers 6 and 7). The mask's contents is linked bit by bit with the identifier of the incoming message. If the bit is "1" in the mask, the identifier bit of the incoming message has to be identical to the identifier bit of the message object so that the data of the incoming message can be stored to the message object. If the bit is "0" in the mask, then it is irrelevant for the message if the identifier bits are identical or not. The following table gives an example for this transfer procedure. (x= incoming message will be accepted; 0 = message will not be accepted).
The two least significant bits of the mask are to be 0, all other bits are to be 1.

ID message object/ ID message	1	2	3	4
1	x	x	x	0
2	x	x	x	0
3	x	x	x	0
4	0	0	0	x

Table 2 : message transfer

For the message object 15 there is an own mask which is independent of the other mask (CAN register 12 - 15).

If a message is sent, a message object has to be configured so that it can be sent that means those bits are set which enable the sending of messages. The identifier of the message, the number of data bytes as well as the data themselves have to be transferred. Now the message is ready to be sent and as soon as access to the bus is possible it will be sent.

Accessing the CAN controller

Reading and writing into the register of the CAN controller is done by an ASIC. Both events are carried out in two steps. First the address is presented, then the data. A time of min. 500 ns has to pass during the processes of reading the data and setting of the address. The time is necessary for the CAN controller to transfer the data safely to the data lines.
The communication is essentially easier with these functions, which are included in the inherent library. The graphic shown below illustrates the data transfer:

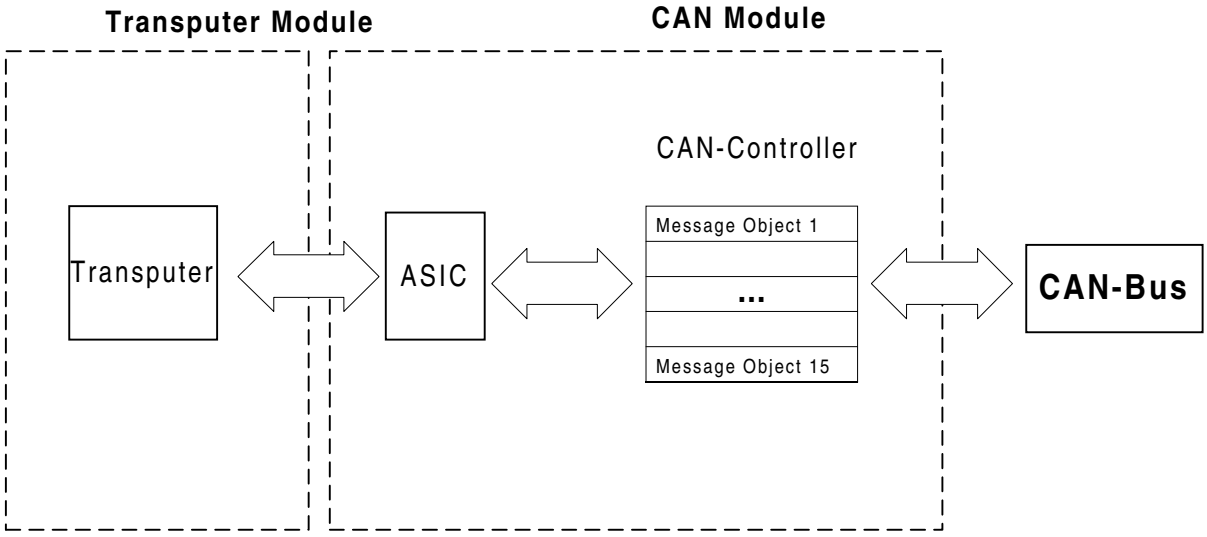

Figure 3 : Data transfer via *ADwin-CAN* board

Hardware Features of the ADwin-CAN Board

The CAN bus frequency depends on the fact how the controller is clocked and configured. The external frequency of the CAN controller is 16 MHz.

The interrupt output of the CAN controller is connected with the event input of the processor. This makes it possible to react to incoming messages immediately without any impact on the processor by polling. In order to apply this function the interrupt of the CAN controller has to be released (control register).

The CAN controller operates in mode 1.
The CAN bus is optically isolated from the PC.

Programming the *ADwin*-CAN board

The ***ADwin-CAN*** board can easily be programmed with the real-time development tool ***ADbasic.*** For programming the ***ADwin-CAN*** board there is a library which contains all functions for setting and reading registers, for initializing message objects , for sending and receiving data sets, and for initializing the CAN controller. All these functions are included in the Library „can.inc“. This library has to be implemented in the ***ADbasic*** program which can be carried out by the command

#include can.inc

This command has to be put at the beginning of the program. In addition, the directory has to be entered in the development environment where the include-files can be found. To enter the directory is made in the menue "Options/Directory".

The functions for the CAN board are explained below.

SET_REG

Syntax: *SET_REG(REGISTER NUMBER,VALUE)*

REGISTER NUMBER: register number of the CAN controller
VALUE: data byte to be written into the register

Describes a register of the CAN controller with a value.

Application example:

```
#INCLUDE CAN.INC

INIT:

init_can()
set_reg(0,1)              'Sets the control register to the value 1

EVENT:
...
```

Note:

The register number which has to be indicated in this command corresponds to the register number of the CAN controller. That means that the control register is called with the address 0, the status register with address 1 etc. up to the last register of the CAN controller which will be called with address 255.

GET_REG

Syntax: *GET_REG(REGISTER NUMBER)*

REGISTER NUMBER: Register number of the CAN controller

Reads the register of the CAN controller and transfers the value to a variable.

Application example:

```
#INCLUDE CAN.INC

INIT:

init_can()
par_1 = get_reg(1)        'Reads the status register and transfers the value 'to
                          parameter 1.

EVENT:
...
```

Note:

The register number which has to be indicated in this command corresponds to the register number of the CAN controller. That means that the control register is called with the address 0, the status register with address 1 etc. up to the last register of the CAN controller which will be called with address 255.

INIT_CAN

Syntax: *INIT_CAN()*

Initializes the CAN controller board and the CAN controller.
The command has no parameters for transferring and returning data.
This command is necessary before using the CAN controller, because not only the CAN controller of the board will be initialized but also addresses are set in ***ADbasic***, which are necessary for the communication with the CAN controller, too.
The command executes the following actions on the CAN controller.

- reset
- all filters are set to "Don't care"
- clock-out register will be set to 0 (external frequency will not be divided)
- bus configuration register is set to 0
- the data transfer rate for the CAN bus is set to 125 Kbit/s
- all message objects are locked

Application Example:

```
#INCLUDE CAN.INC

INIT:
init_can()               'initializing the CAN controller

EVENT:
...
```

Note:

This command has to be executed at the start of the program.

EN_RECEIVE

Syntax: *EN_RECEIVE(object number, identifier)*

object number: number of the message object in the CAN controller
identifier: identifier of the messages, which are to be received in this message object.

Enables a message object to receive messages and sets the identifier which is allocated to the message object.

Application Example:

```
#INCLUDE CAN.INC

INIT:

init_can()
en_receive(1,200)        'Initializes the message object 1 to receive CAN
                         'messages with the identifier 200

EVENT:
...
```

Note:

Only the command "en_receive" enables a message object to receive messages from the CAN bus. If a message object is called with the functions "EN_TRANSMIT“ and "EN_RECEIVE“, only the last call which was carried out is decisive.

EN_TRANSMIT

Syntax: *EN_TRANSMIT(object number, identifier)*

Object number: Number of the message object in the CAN controller
Identifier: Identifier which the messages sent from this message object should have

Enables a message object to transmit messages and sets the identifier which is allocated to the message object.

Application Example:

```
#INCLUDE CAN.INC

INIT:

init_can()
en_transmit(6,40)        'Initializes the message object 6 to transmit CAN
                         'messages with the identifier 40

EVENT:
...
```

Note:

Only the command "en_transmit" enables a message object to transmit messages from the CAN bus. If a message object is called with the functions "EN_TRANSMIT“ and "EN_RECEIVE“, only the last call which was carried out is decisive.

TRANSMIT

Syntax: *TRANSMIT(MSG_num)*

MSG_num: Number of the message object in the CAN controller

As soon as the message object is allowed to get access, a data telegram is sent via CAN-bus. For this message the number of data bytes as well as the data bytes themselves are taken from the field "can_msg". This field has nine elements. The first eight elements encompass the data bytes one to eight. The ninth element defines the number of data bytes. A value between one and eight has to be entered here. The values in the field "can_msg" have to be entered before executing the "transmit" command.

Application Example:

```
#INCLUDE CAN.INC

INIT:

Init_can()
en_transmit(6,40)        'Initializes the message object 6 to transmit CAN
                         'messages with the identifier 40
can_msg[1] = 100         'The values to be sent later are transferred to 'the
                         'field
can_msg[2] = 200         'the message contains two data bytes with the 'values
                         100 and 200
can_msg[9] = 2

EVENT:

transmit(6)              'sends a message via message object 6 with the
                         'identifier 40
```

Note:

This command is only executable when the corresponding message object has been configured before with "en_transmit" to be able to send.

READ_MSG

Syntax: *READ_MSG(MSG_number)*

MSG_num: Number of the message object in the CAN controller

Reads out a received message from the message object. At the same time it will be checked if a new message has arrived for this message object since the last call. If not, no registers will be read out and the return value is 0. If a new message is written into the message object the data are read out and transferred to the specified field "can_msg". In this case the return value corresponds to the identifier of the message received. The bits indicating that a message has been received are reset so that a new message can be received.

Application Example:

```
#INCLUDE CAN.INC

INIT:

init_can()
en_transmit(1,200)        'Initializes the message object 1 to read the CAN
                          'messages with the identifier 200

EVENT:
par_9 = read_msg(1)       'Reads out the received data from the message
                          'object 1 and transfers the identifier to 'parameter 9.
                          'The data bytes are in the field 'can_msg.
```

Note:

The command "en_receive" enables the corresponding message object to be able to receive data.

Example Programs

The following program details the initializing of the CAN controller in the "Init" section and the cyclic read out as well as sending messages in the "Event" section:

```
REM program initializes the CAN controller,
REM configures one message object as sender,
REM and one as receiver. The program exchanges
REM every 10 ms data between CAN controller
REM and transputer

#include can.inc


dim result as integer

init:
  init_can                    'initializing the CAN controller

 REM set filter
  set_reg(6,255)              'Global Mask Register Standard
  set_reg(7,255)

  set_reg(12,255)             'Message 15 Mask Register
  set_reg(13,255)
  set_reg(14,255)
  set_reg(15,255)

  set_reg(0,65)               ‘write authorization for CPU
                              ‘to the configuration register
  set_reg(63,0)               'bus transfer rate
  set_reg(79,20)              'out of 1 Mbit/s
  set_reg(0,0)                'write authorization is returned

  en_receive(2,385)           'message object 2 is configured
                              'to be able to read (Identifier 385)

  en_transmit(3,1)            'message object 3 is configured
                              'to be able to write (Identifier 1)


event:

REM read 1 dataset and write 1 dataset

  result = read_msg(2)        'read data
                              'If there are new data they will be
                              'written to the field can_msg


  can_msg[1]=1                'data, to be sent
  can_msg[2]=2                'are written to the
  can_msg[3]=3                'field can_msg. This field
  can_msg[4]=4                'contains the data to be sent
  can_msg[5]=5                'later.
  can_msg[6]=6
  can_msg[7]=7
  can_msg[8]=8
  can_msg[9]=8                '8 data bytes

  transmit(3)                 'send message
```

The following example program details the initializing of the CAN controller and the interrupt-controlled read-out of new messages:

```
REM program initializes the CAN controller,
REM and configures a message object as receiver.
REM The program is interrupt-controlled
REM and reads the messages
REM as soon as a new message arrives.

#include can.inc

dim result,status,object as integer

init:
  init_can                    'initializing the CAN controller

 REM set filter
  set_reg(6,255)              'Global Mask Register Standard
  set_reg(7,255)

  set_reg(12,255)             'Message 15 Mask Register
  set_reg(13,255)
  set_reg(14,255)
  set_reg(15,255)

  en_receive(1,385)           'message object 1 is configured to be able to
                              'read
                              'only messages with the identifier 385
                              'are stored.

  set_reg(0,65)               'write authorization for CPU
                              'to the configuration register
  set_reg(63,0)               'bus transfer rate
  set_reg(79,20)              'out of 1 Mbit/s
  set_reg(0,0)                'write authorization is returned

  status = get_reg(1)         ' read status
  set_reg(0,2)                ' interrupt enable; on the CAN controller


event:

object = get_reg(5fh)         'read interrupt register

if (objekt = 2) then          'Get the number of the
  object = 15                 'message object
else                          'where the new message has arrived
  object = object - 2
endif

result = read_msg(object)     'read new data
                              'the data are in the field can_msg
```