Content

1. Purpose

At first an overview of the purposes of this practical:

- Communication-Networks in automobiles
- basics of the CAN bus system
- implementation of the communication in the CAN bus system

2. Networking in vehicles

The first ECUs which were used to control the injection and ignition of the engine in a car did not work independently. These ECUs used simple point-to-point connections for communication. The number of ECUs in the automotive sector is increasing till today. This is the result of requirements for security, comfort, economy and also environment protection. At the beginning the functions were only assigned to individual ECUs. Particular ECUs worked mostly autonomously. After that, functions, which cross the borders of ECUs and domains, were developed and implemented. That means, the signals produced by one ECU must be sent to other ECUs periodically. And for a particular ECU, receiving specific signals is mandatory to work properly. For example the velocity of the car is used in following functions:

- Electronic Stability Program (ESP)
- anti-lock brakes (ABS)
- Adaptive Cruise Control (ACC)

Today most of the driving assistance systems, e.g. the Adaptive Cruise Control (ACC), which adjusts the speed to the traffic flow, are typical examples of such cross-system functions. An ACC controller computes results from measurements of the distance sensors, the relative speed and the angular

ASE Practical: Experiment 2

position of the car in front. With this data, the longitudinal dynamics can be controlled. This can be done by acceleration (engine torque) or deceleration (braking pressure) and is realized by signals to the engine ECU resp. ESP ECU. Thus, ACC can be regarded as powertrain and chassis-independent function.

The realization of functions with increased communication overhead like the ACC is possible since the early 90s of the 20th Century, when high performance bus systems such as CAN were invented. Compared to the traditional point to point communication with direct wiring they have the following advantages:

- Multiple use of signals
- Reduction of cost, weight and installation space
- Improved reliability, security and maintainability
- Easy connection of other system components
- Simplification of the vehicle assembly

	Class A	Class B	Class C	Class C+	Class D
Data rate	< 10 Kbit/s	< 125 Kbit/s	< 1 Mbit/s	< 10 MBit/s	> 10 Mbit/s
Application	Sensor-Actuator-Networking	Networking in the comfort area	Networking in the drive and chassis	Networking in the drive and chassis (X-By-Wire)	Networking in telematics and multimedia
Example	LIN	CAN	CAN	FlexRay, TT-CAN	MOST

Table 1: classification of bus systems

Besides the CAN-bus system, nowadays there are many other bus systems which can be implemented in parallel. They are used in different domains of the vehicle according to economical and technical requirements. The typical technical requirements are the data transmission rate, noise immunity, real-time capability or the maximum number of nodes which can be connected. According to these and further requirements the bus systems can be divided into different classes, which can be connected with each other via so-called gateways to construct a network. An example is shown in figure 1.
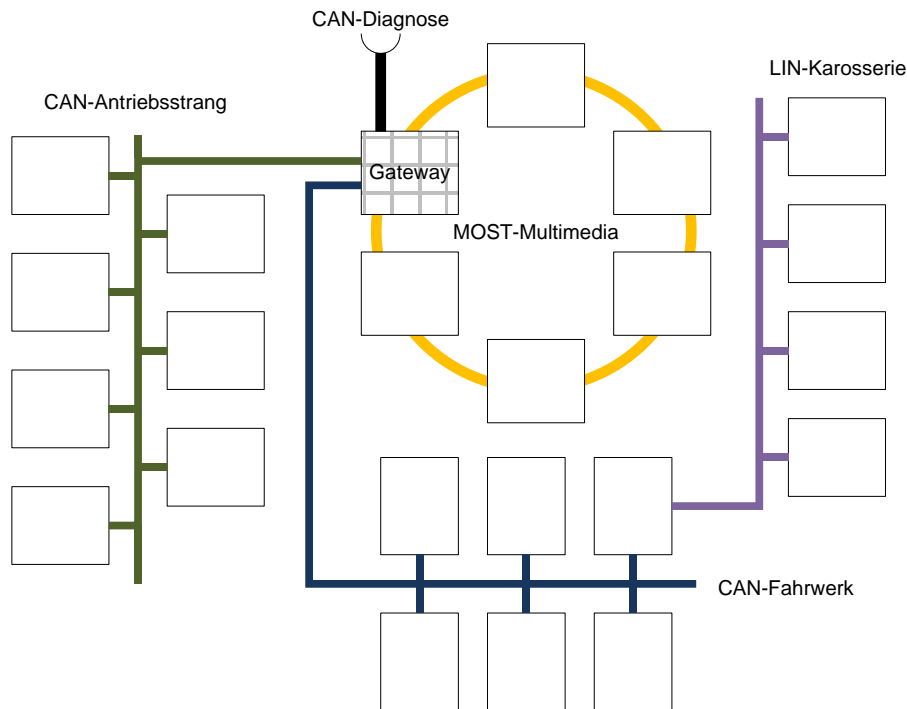
ASE Practical: Experiment 2

Figure 1: Example of a typical network topology.

3. CAN-Bus

With the introduction of the serial, asynchronous CAN bus, whose development was mainly driven by the cooperation of Bosch and Intel, bus systems were used in cars since 1990. Nowadays besides the CAN bus other, further developed bus systems are also implemented by all the car manufacturers. At the same time CAN is also used as sensor-actuator-bus in the industry (CAN in Automation).

The CAN bus is defined by the standard ISO 11898, in which only the two lower layers of the ISO/OSI reference model are specified. On one side it allows the implementation of the CAN bus to be more flexible, on the other hand it makes the usage of the communication easier and meanwhile supplies a more efficient communication. Because of this all functions upon these two layers must be integrated in the application layer individually.

3.1. Physical Layer

The bit transfer layer describes only the physically transformation of the signal on the lowest layer. During the development of CAN many tailored standards for the specification of this layer were created corresponding to the different application areas. In cars the so called Highspeed-CAN (CAN-C) and Lowspeed-CAN (CAN-B) are commonly used.

Typically high data rate with a short latency is requested for motor management, engine controlling and car stability systems. For this kind of usage ISO 11898-2 defines the Highspeed-CAN with baud rates from 125 Kbit/s up to 1 Mbit/s. In contrary the Lowspeed-CAN with a baud rate up to 125 Kbit/s already fulfills the requirement of ISO 11898-3 in the comfort and vehicle body area. No matter which standard is used, CAN is able to transfer dominant bit (binary "0") and recessive bit (binary "1") states. These states are NRZ-coded. NRZ means Non-Return-To-Zero.

Typically a simple two-wire cable is used, where one wire is called CAN_H and the other one CAN_L. Here the bits will be transferred as differential signals on each wire. A differential amplifier subtracts the CAN_L level from CAN_H level. The signal, which comes from this process, represents the logic states, which results in dominant and recessive state. The advantage of this solution is the robustness of the bus, because normally interferences will happen in both wires at the same time. In this case the difference of the voltage levels will stay constant.
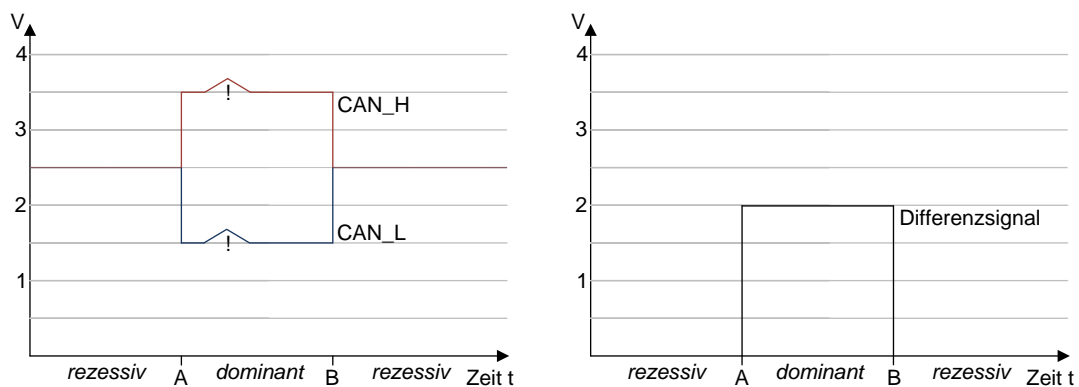
Figure 2: Voltage levels of the Highspeed CAN and resulting difference signal

For Lowspeed-CAN one wire is also sufficient because of the low data rate. Through this the reliability is increased. So Lowspeed-CAN with two wires has still a working wire, if a mechanical failure to one wire occurs. A very important prerequirement is that one corresponding transceiver can evaluate the voltage level of CAN_H and CAN_L respectively.

3.2. Data Link Layer

In the data link layer, which is defined by standard ISO 11898-1, following functions are described:

- addressing
- creation of the message
- access to the bus
- synchronization
- failure management

Normally the CAN bus will be set up as line structure, and there is no prominent participant. So single-points-of-failure are avoided and the whole bus system can be extended easily. The messages will be broadcasted to all the participants. These messages have no addresses of source or destination. Instead they will carry a message identifier. With this and according to the acceptance filter the participants will decide, if this message is relevant or not and should be received or not.

Basically in CAN bus systems there are four kinds of messages, which are known as telegrams. They are:

- Data Frame
    - data produced by one participant

- Remote Frame
  - data frame for remote transmission request
  - request data from specific participant in the network
- Error Frame
  - shows errors in the network
- Overload Frame
  - forced delay between remote and data frame

This practical will involve the important data frame exclusively. It is responsible for typical data exchange. Figure 3 shows the general structure of the frame. The data frames distinguish themselves with two different styles: CAN2.0A with an 11 bits message identifier and CAN2.0B with an extended 29 bits identifier. The latter has two additional control bits.

SOF[1] (1 Bit)	Message Identifier (11)	Control Bits (7)	Data Field (0 - 64)	CRC[2] (16)	ACK[3] (2)	EOF[4] (7)

[1] Start of Frame
[2] Cyclic Redundancy Check
[3] Acknowledgement
[4] End of Frame

Figure 3: structure of a CAN message according to CAN2.0A

The control bits specify if the message is a remote frame, which format is used (CAN2.0A or CAN2.0B) and how long the real payload is. The maximal payload is 8 bytes. After the data field follows a CRC-checksum, which is used to check the data for errors. After EOF bits there must be at least three bits, which are called inter-frame-space, before the next frame can be transmitted.

The CAN bus uses the CSMA/CS principle, which means, all the participants who want to send a message listen to the bus and wait until the bus is free. In the case all the participants try to send messages at the same time the collision will be avoided and solved. This is done in a so called arbitration phase: here the message identifier and the two states of the bus, recessive and dominant, are used. At the beginning the bus will be in recessive state (binary "1"). If competing accesses from at least two participants of the bus happen, the telegram will be sent by both of them without knowing of a collision. The bus will be set to dominant state with the SOF-bit (binary "0"). With sending the corresponding message identifiers and listening to the bus, it will be decided which participant gets exclusive bus access. All participants will send their message identifier descent and bit by bit. Dominant level bits always overwrite recessive level bits. Therefore, if participant A sent a dominant "0" and another participant B a recessive "1", the bus will pick the dominant level. B will hear it at the same time and notice that his message was not accepted, then cancel his sending process and wait until next arbitration phase. According to this principle always only one participant can get exclusive bus access, if the message identifiers are clearly defined. The participant who can send messages is always the one with the smallest message identifier. Therefore the message identifier also sets the priority of a message.

The CAN bus, just like mentioned before, is an asynchronous bus. There is no clock signal on the bus. For the correct execution of the above described access method CSMA/CA, the synchronization of the participants is necessary. Before every access the bus is in a so called idle state, which is the recessive state. Then the bus will be put in the dominant state by the SOF-Bit. All the participants will synchronize in this falling edge. In this way in the process of signal sending all the participants will

ASE Practical: Experiment 2

work synchronously. This kind of synchronization will also be done during the sending process on every falling edge after the synchronization. In order to avoid long series of same bit values and the limitation of synchronization caused by it, bit stuffing is executed. If between SOF and the end of CRC-checksum five same bit values appear one after another, then every time at the end of this series one converse bit will be sent. In this way maximal 19 bits can be added to the data frame. After receiving those bits will be deleted.

No matter if sender or receiver detects a fault on the bus, the corresponding fault telegram will be broadcasted. Thus sender side can detect two bus faults. Bit fault, where the sent bits are not in the right correlation with read bus state or ACK fault, which means, the receiving of data telegrams is not acknowledged.

Message Identifier	Order of Bus-Access in case of concurrent access
`0x42 0 0 0	0 1 0 0	0 `**`0`**` 1 0`	2
`0x30 0 0 0	0 `**`0`**` 1 1	0 0 0 0`	1
`0x44 0 0 0	0 1 0 0	0 1 0 0`	3

Table 2 Arbitration phase with 3 competing participants

3.3.   Application

All with CAN bus connected participants use three important components for communication: CPU, CAN controller and CAN transceiver (figure 4). The Transceiver implements the physical layer. It transforms the bit stream of the CAN controller to corresponding voltage levels and vice versa. The transceiver of the demo-board accords to the ISO 19898 and is used for Highspeed CAN with transfer rates up to 1 Mbits/s.
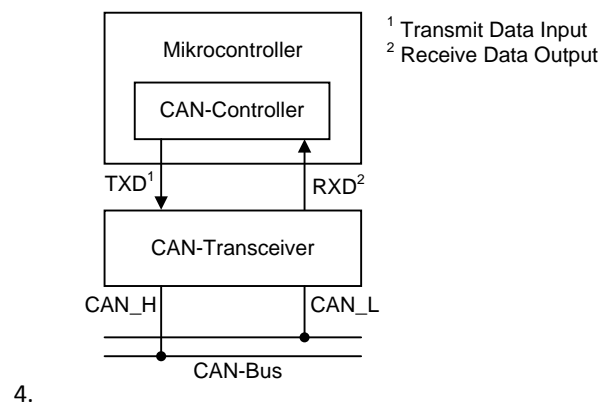

Figure 4: CAN system with integrated CAN-controller

The CAN controller is responsible for the implementation of the data link layer. It is composed of different components, sending and receiving cache, filter components for acceptance filtering and a management unit. In the case of the demo-boards the CAN controller is cost efficient integrated in the microcontroller. The software which is executed on the microcontroller initializes the sending of data telegrams. At the beginning the message identifier, the data field, and the data length code will be in the sending cache of the CAN controller. The CAN controller generates a correspondent data telegram from this triple, which will be transformed to an input in form of a bit stream for the transceiver. The receiving process is quite the same. After fault- and acceptance-test the received bit

stream will be transformed to the above mentioned triple. The triple in the receiver cache can be read by the application software.

As a result of missing guidelines for the layers above the data link layer, developers can choose a message and signal implementation. This means that the allocation of message identifiers and the defining, coding and normalization of signals are left to the developers.

4. Tasks

For this practical, easy to use CAN functions will be available.

Timer function:

Command / Register	Description
conf_timer0(int mode, int value)	mode = 1, timer on mode = 0, timer off value = time in milliseconds.
PIT.CH[0].TFLG.B.TIF	Writing a 1 to this register clears the timer interrupt flag

Easy names can now be used for LED output registers. The command **"LEDX"** (X=0, 1, 2, 3, 4, 5, 6, 7) can be used instead of writing the name of the output register. Hence **"LED0=0;"** will glow the LED D0 on the board.

1) Prepare the timer function and indicate the working of timer function through a blinking LED. Note: The interrupt flag has not been cleared. It has to be cleared after processing interrupt. Please refer to the "Interrupt_vector_table.pdf" to find the correct IRQ number for "PIT timer channel 0". The Interrupt method is in software mode. Please find the **"Ext_Isr()"** function at the end of **"api.c"** and implement a function call (**"timer0_intr()"** is the function to be called) in the corresponding case (the number in each case represents the corresponding IRQ number from the vector table). Hence when an interrupt occurs, the lines of code in the corresponding case are executed.

2) This task concerns with preparation of controller pins and message buffers for the enabling transmission of a CAN message.
   1. Please utilize the function **"conf_can()"** to configure the pins. Please determine correct values to be given to the different fields of PCR register using "PCR_configuration.pdf". The PCR register number for transmission is "16" and for reception is "17". A "1" shall be written to the **"PA"** field of the PCR register.
   2. Please use the document "buffer.pdf" (notations used in document, MB = message buffer, TX = transmission) to configure the sending buffer. Use buffer "4" with the standard format (11 bits) for sending messages.
   Note: successful transmission of message causes an interrupt for the corresponding message buffer. Hence the corresponding flag must be cleared. Interrupt flag for message buffer interrupts can be cleared by writing a 1 to **"CAN_0.IFRL.B.BUFXXI"** where XX = corresponding message buffer. The IRQ number can be found in the interrupt vector table under the name "**FLEXCAN_BUF_XX_YY**" where XX and YY represent the range of buffer numbers for the corresponding IRQ)
   3. A CAN message should be sent every 200 ms. It should contain a speed value between 0 and 300 km/h. To achieve this, two data bytes must be used, as a resolution of 9 bits is needed for the speed. The speed value should not exceed 300 km/h!
   Every time a message is sent, the speed should be incremented by 5 km/h. When a speed of 300km/h is reached, the speed should be decremented by 5 km/h to 0 km/h.

The message ID should be the last number of the IP address of the computer you are working on. For example, when the computer has the IP 100.101.102.103, the message ID should be the ID 103.

To verify the results, use the program "Virtual Cockpit". Please refer to its manual, to understand its functionality. If problems occur, use the program "Can-View" to check the messages sent by your ECU.

The speed message has to be in the following format:

ID	See text above	
Data Byte 0	Bit 0	LSB of speed value
	Bit 1	Bit 1 of speed value
	Bit 2	Bit 2 of speed value
	Bit 3	Bit 3 of speed value
	Bit 4	Bit 4 of speed value
	Bit 5	Bit 5 of speed value
	Bit 6	Bit 6 of speed value
	Bit 7	Bit 7 of speed value
Data Byte 1	Bit 0	MSB of speed value
	Bit 1	-
	Bit 2	-
	Bit 3	-
	Bit 4	-
	Bit 5	-
	Bit 6	-
	Bit 7	-

3) Buffer "0" has already been initialized for receiving. Note that every successful reception causes an interrupt. Hence configuration the interrupt function call *"can_receive()"* should be done also the corresponding interrupt flag should be cleared .

The program "Virtual Cockpit" has a start button. When this button is pressed, the program connects to the "TinyCAN" CAN interface and sends a message with ID 1 to the bus. Implement a program which turns on an LED, when this message is received. In addition, when the board is restarted, it should remain in an idle state, until it receives this message. In this state, the program should only wait for this message. After its reception, all LEDs should be switched on for 1 second. Moreover it should start sending the message from task 2.3.

4) As described in Section 3.3, the CAN controller has an acceptance filter. Try to use the acceptance filter so the ISR only is called when a message with a filter defined ID arrives.

Please use the mask register *"CAN_0.RXIMR[0].R"* in **"conf_can()"** to configure masking.

For checking against acceptance the buffer should be initialized with acceptance values after **"init()"**. The values in the mask register are checked against the value in the *"STD_ID"* field in the corresponding register.

Please configure the mask register and the acceptance IDs in the buffer such that only the messages with ID = 0x8Z (where Z = 1, 3, 5, 7, 9, B, D, F) are accepted and generate an interrupt. The table below shows a bit map for the mask register *"CAN_0.RXIMR[0].R"* for

ASE Practical: Experiment 2

buffer 0. Bits marked with "C" should be changed for masking. The bits which are not marked with a "C" should be used with the same binary value as provided in table.

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Value	0	0	0	C	C	C	C	C	C	C	C	C	C	C	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

5. Additional Task

The virtual cockpit contains several status lights for turn indicator lights, high- and low-beam, engine, etc. In the table below you can see the CAN-Messages which should be sent to set the status of the corresponding functionality of the car. Try to send different messages to the bus and enable/disable the different status lights. Use the buttons "BT5" and "BT6" to turn on/off the turn indicators. The switches "SW1", "SW3" and "SW4" should be used for the headlights (Parking Light, Low-/High-Beam).

Can-Message for the blinker/headlight status:

Message Field	Values	Functionality
ID (decimal)	2	
Data Byte 1	1 – on 0 - off	left blinker
Data Byte 2	1 – on 0 - off	right blinker
Data Byte 3	1 – on 0 - off	hazard light
Data Byte 4	1 – on 0 - off	parking light
Data Byte 5	1 – on 0 - off	low beam
Data Byte 6	1 – on 0 - off	high beam
Data Byte 7	-	-
Data Byte 8	-	-