	Professorship Computer Engineering **Automotive Software Engineering** Prof. Dr. Dr. h. c. Wolfram Hardt Dr. René Bergelt	TECHNISCHE UNIVERSITÄT CHEMNITZ
Practical Unit 2	Embedded Programming - CAN Bus	Winter Semester 2024/25

Contents

1. Networking in Vehicles

The first ECUs which were used to control the injection and ignition of the engine formed a single function unit. Consequently, they used simple point-to-point connections for their communication. Since then, the number of ECUs in modern vehicles has been ever increasing. This is the result of increasing requirements for security, comfort, economy as well as environment protection. In the beginning, complex functions were assigned to individual ECUs which worked independently of the other ECUs present in the vehicle. Subsequently, functions which cross the borders of domains and consequently ECUs were developed and implemented. This resulted in the situation at which we have arrived today where one ECU needs information (i.e. signals) from many other ECUs to work properly. Therefore, ECUs need to provide specific signals (e.g. sensor data) to other ECUs periodically. For example, the velocity of the car which is usually measured by an ECU near to the wheels is required by the following Advanced Driver Assistance Systems (ADAS) which are realized on separate ECUs at different locations in the car:

- Electronic Stability Control (ESC)
- Anti-lock Brake System (ABS)
- Adaptive Cruise Control (ACC)

Consequently, the speed information (i.e. speed signal) needs to be sent to all of these ECUs.
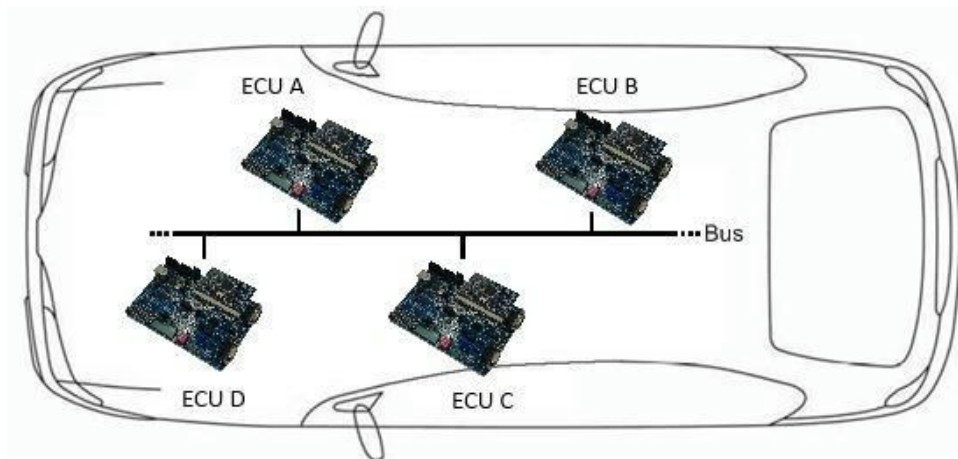
Figure 1: ECUS connected by a bus system

The realization of functions with increased communication needs like ACC is possible since the early 1990s, 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:

- Reduction of cost, weight and installation space
- Reuse of signals by multiple ECUs (e.g. data sent by one ECU can be received by multiple other ECUs)
- Improved reliability and maintainability
- Easier extensibility regarding the addition of new system components
- Simplification of the vehicle assembly

Figure 1 shows a general example for the interconnection of controllers and their communication over a so-called bus system in a car. A bus allows developers to send messages (containing for example a measured temperature) from one ECU to one or several others.

To stress the importance of the interconnection and the categorization of ECUs, the following example shall be used: The ACC (Adaptive Cruise Control) is an add-on for the cruise control. The system maintains a steady speed as set by the driver and automatically keeps the necessary security-distance. When approaching a slower vehicle the speed is adjusted to the speed of the preceding vehicle and thus the security distance is kept. If the distance increases the systems accelerates up to the defined speed. To realize this function several ECUs are necessary in the automobile. The ACC-ECU communicates with the engine, the ESP and the gearbox ECUs to adjust the driving speed. ESP is necessary to selectively decelerate individual wheels to avoid pulling of the vehicle to one side. This simple example shows the importance of the communication between the affected ECUs. Only by interaction of the different ECUs the functionality of the ACC can be displayed. It also shows that simple features need complex implementations.

Besides the CAN bus system which you will work with in Unit 2, nowadays there are many other bus systems which usually are present in a car in parallel. Each bus system is used in different domains of the vehicle according to economical and technical requirements, e.g. engine, chassis, car body, multimedia and security. 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
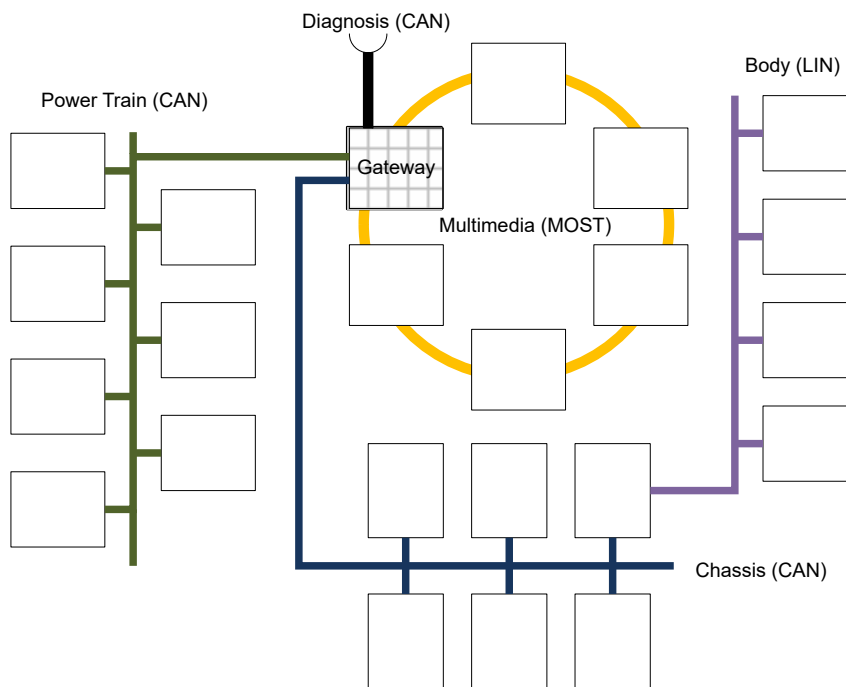
Figure 2: Example of a typical vehicular network topology

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 2.

2. CAN bus

With the introduction of the asynchronous CAN bus for which development was mainly driven by the cooperations Bosch and Intel, bus systems have been used in cars since 1990. Meanwhile CAN is also used as sensor actuator bus in the industry in the field of building or factory automation. The CAN bus is defined by the standard ISO 11898. However this standard only specifies the two lower layers of the ISO/OSI reference model. Due to this all functions above these two layers must be integrated in the application layer by application developers or are integrated in proprietary CAN hardware.

2.1. Physical Layer

The bit transfer layer describes the physical transformation of the logical signal to actual voltage values on the wire. During the development of CAN many tailored standards for the specification of this layer were created targeted at different application areas. In cars, the so called high-speed CAN and low-speed CAN are commonly used. Typically, high data rate with a short latency is required for motor management, engine control and car stability systems. For this kind of usage ISO 11898-2 defines the high-speed CAN with baud rates from 125 Kbit/s up to 1 Mbit/s. In contrary, the low-speed 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. Usually, increasing the data rate results in a lower maximum bus length so the system designers have to carefully plan placement and application requirements of the involved ECUs. Typically, an unshielded twisted-pair cable is used to build a CAN bus, where one wire is called CAN_H and the other one CAN_L. The bits will be transferred as differential signal where the actual signal value is calculated by subtracting the CAN_L level from CAN_H level in hardware. The signal value, which results from this process, represents the logic states. Like any digital transmission there are to valid logic states: "0" (dominant) and "1" (recessive). The advantage of the differential approach
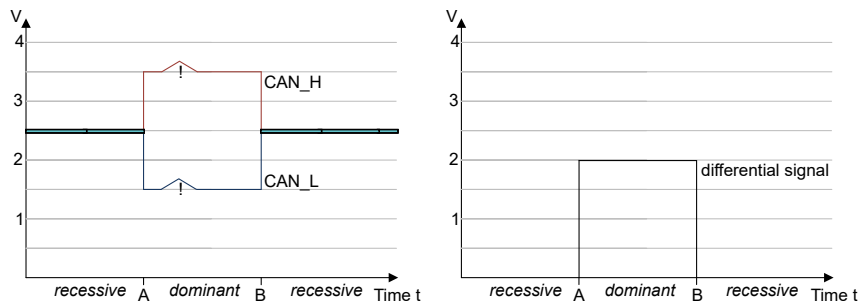
Figure 3: Voltage levels of the high-speed CAN and resulting differential signal

is an increased robustness of the bus against interference, since normally interferences will influence in both wires at the same time. In this case the difference of the voltage levels will stay constant as illustrated in Figure 3.

2.2. Data Link Layer

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

- Addressing
- Definition of the messages
- Bus access
- Synchronization
- Failure management

Normally, the CAN bus will be set up as line structure, and no participant has a special role. 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 the participants will decide, if this message is relevant or not and should be received or not. Consequently, message identifiers usually define the content of messages (e.g. message ID 0x55 could mean that the message contains the speed signal).

In CAN bus systems there are four kinds of messages, which are also called telegrams. These are:

- **Data Frame**: data sent by one participant (actual application data)
- Remote Frame: data frame for remote transmission request (i.e. request data from specific participant in the network)
- Error Frame: indicates errors in the network
- Overload Frame: indicates that a bus participant is overloaded (i.e. too busy); usually not used in modern vehicles

In this practical unit we will only work with Data Frames which represent **normal** messages (i.e. data required by the application). Figure 4 shows the general structure of a data frame. CAN defines 2 different types of data frames: CAN 2.0A (standard frame) with an 11 bits message identifier and CAN2.0B (extended frame) which provides 29 bits for the identifier and 2 additional control bits. This means that when you want to send a message with an ID which needs more than 11 bits (i.e. a message id > 0x7FF) it **must** be sent as an extended frame. Both frame formats support a maximum of 8 bytes of user data (= actual information) in the data field.

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 4: Structure of a CAN message according to CAN2.0A (base frame format)

2.3. CAN controllers

Both physical layer and data link layer are usually handled by a special hardware component of the ECU: the CAN controller. For the ECUs used in the practical, the CAN controller is directly integrated in the microcontroller. Like all peripheral components it is connected to the actual processor through pins and shared registers. However, in contrast to the components used in practical unit 1 (LEDs, Switches, …) which where controlled through only 1 pin, the CAN controller provides a more complex interface. The CAN controller provides multiple so-called **message buffers** which can be configured either for receiving (RX) messages or transmitting (TX) messages. When a buffer is configured as RX buffer, the CAN controller will raise an interrupt when it received a message which can then be handled by the application (i.e. like the timer interrupt in Unit 1 of the practical). In addition, all buffers contain a status register where the application can check the state of the buffer or in case of TX buffers can instruct the CAN controller to transmit the message in the buffer. As the CAN controller is a separate hardware component, receiving and transmitting messages happens **in parallel** to the program execution on the microcontroller. For instance, in order to send a data frame, the microcontroller program needs to write a message identifier, data field (payload bytes) and payload length to a free TX message buffer. Once the controller has received the instruction to send the message in this buffer, it will generate a data telegram from this information (including calculating values for the control bits) which will be transformed to an input in form of a bit stream for the transceiver and subsequently it will influence the voltage value of the bus in order to "write" to it. The arbitration process is completely handled by the CAN controller and thus not visible for the application.

As a result of missing specifications 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 definition, coding and normalization of signals are left to the developers or the API of a bough CAN controller, respectively.

3. Practical Setup

In this unit you will control the CAN controller which is integrated in the practical ECU in order to receive and transmit CAN messages. The objective is to control elements present on a car's dashboard (such as speed gauge or indicator lights). Like in the automotive industry, software development usually does not happen against the actual hardware from the beginning since often the hardware is not yet available. Instead software simulations of components which will be separate hardware components in the actual car later are used. This also facilitates speeding up the prototyping phase. In this unit you will use the software **VirtualCockpit** as shown in Figure 5. You will test your implementation against this simulated, software-based dashboard on your development PC. For this, both your ECU and your practical PC which runs the VirtualCockpit are connected to the same CAN bus through a CAN cable and one of many USB CAN interfaces from the automotive industry as shown in Figure 6. While we use different USB CAN interfaces from automotive manufacturers (e.g. TinyCAN or Vector VN1610) the usage with regards to your application will not differ since the CAN bus is a standardized

Figure 5: The main window of the VirtualCockpit tool

communication protocol. Please see Figure 7 for a schematic of the hardware setup used in Unit 2.

(a) TinyCAN interface   (b) 2 Vector CAN interfaces

Figure 6: The USB CAN interfaces used in the practical

3.1. Controlling LEDs and Reading Sensors

To ease the controlling of LEDs, buttons etc. in this practical, the template project comes with pre-defined C macros. The macros **LED_X** (where X can be P, Rx, Tx, U1, U2, or U3) can be used to set the state of the corresponding LED instead of writing to the output register. Likewise, you can use the macros **SWX** (where X can be 1, 2, 3, 4) and **BTX** (where X can be 1, 2) to read the state of the switches and buttons. In addition, you can use the macros **POT** and **LDR** to read
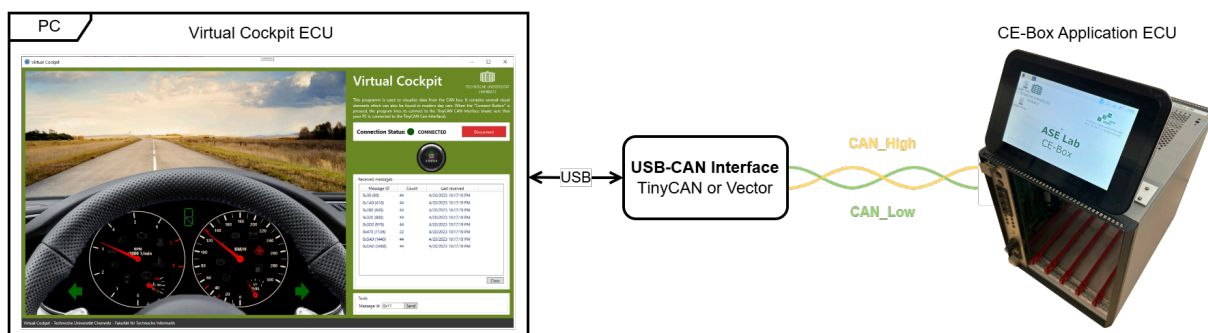

Figure 7: Schematic of the hardware setup for Unit 2

```
LED_P = 1; // turn on LED P
LED_U1 = 0; // turn off LED U1
LED_U2 = ! LED_U2; // toggle LED U2

if (SW1) {
   // Switch 1 is turned on
}

if (! BT1) {
   // Button 1 is not pressed
}

// read potentiometer value
int potentiometer_value = POT;
```

Listing 1: Usage examples for the template's helper macros

the current potentiometer and light sensor measurement, respectively. See Listing 1 for usage examples.

> The corresponding macros[1] are defined in the **source/siu.h** file.

---

[1]If you do not know what a C macro is or how it can be used, please check the documentation of GCC for example: https://gcc.gnu.org/onlinedocs/cpp/Macros.html

4. Tasks

Before you begin, start the program "VirtualCockpit" on your development PC in the lab if it has not been started yet. This tool allows you to see CAN messages which are present on the bus (i.e. have been sent by your ECU) and also allows you to send messages which can be received by your ECU if it is correctly configured. In addition, it provides the simulation of a car dashboard which you will use in Task 3.

4.1. Task 1 - Receiving messages

In this task, you will prepare the microcontroller pins and a message buffer to enable the reception of CAN messages.

4.1.1. Subtask 1: Configure the CAN controller
- Update the function **SIU_Init()** in the **source/siu.c** file to configure the pins for **reception**
- Determine the correct values to be given to the different fields of the "PCR" register using the "pin_muxing.pdf" known from unit 1
- The PCR register number for reception (RX) is 17
- Write a "1" to the "PA" field of the corresponding register
- Configure IBE (Input Buffer Enable) and OBE (Output Buffer Enable) correctly for **receiving**

4.1.2. Subtask 2: Handle received messages
- Once the CAN controller receives a message it will raise an interrupt, subsequently the function **can_receive** in **main.c** will be called
  - ‣ the function parameter **CANRxFrame crfp** contains the data of the received message
- Implement the following in this function:
  - ‣ toggle LED_Rx every time a message is received
  - ‣ toggle LED_U1 every time a message with **Message ID 0x11** is received
    - − for this, check **crfp.SID** of the received base frame
- Send test messages by using the Tools section of VirtualCockpit (see Figure 8)

4.1.3. Subtask 3: Filtering messages
- Configure the CAN controller's acceptance and mask register in the file **can.h** in such a way that it will only accept messages with IDs **0x10** and **0x11**
  - ‣ see the **can_filtering.pdf** file for hints how to determine the values for the two registers
  - ‣ If done correctly, the function **can_receive** will **only** be called if one of these two messages is received
- Send test messages by using the Tools section of VirtualCockpit (see Figure 8)

4.2. Task 2 - Sending messages

4.2.1. Subtask 1: Configure the CAN controller
- Update the function **SIU_Init()** in the **source/siu.c** file to configure the pins for **transmission**
- Determine the correct values to be given to the different fields of the "PCR" register using **PCR_configuration.pdf**.
- The PCR register number for transmission (TX) is 16
- Write a "1" to the "PA" field of the corresponding register

- Configure IBE (Input Buffer Enable) and OBE (Output Buffer Enable) correctly for **transmitting**

4.2.2. Subtask 2: Preparing a message buffer
- Configure a message buffer in the function **CAN_Init** in file **source/can.c** for sending an extended frame
    - ‣ select a buffer index between 8 - 14
    - ‣ use the information provided in the **can_buffer.pdf** file

4.2.3. Subtask 3: Sending a CAN message
- Update the function **can_send_engine_on** in **main.c** to send a message with **ID 0xDA0** using the buffer you configured in the previous subtask (replace X with the corresponding buffer index you chose)
    - ‣ Depending on the frame format to send (base or extended) write the message ID either to CAN_0.BUF[X].MSG_ID.B.STD_ID or CAN_0.BUF[X].MSG_ID.B.EXT_ID. The unused field should be set to 0.
    - ‣ Set the payload length of the message
        - – Set payload data using register "CAN_0.BUF[X].DATA.B[Y]" (where Y can go from 0 to 7 for the 8 payload bytes)
    - ‣ Write the correct status code into the CODE field of the message buffer (i.e. status register) to instruct the CAN controller to transmit the message (check **can_buffer.pdf**)
    - ‣ Indicate the transmission of a message by toggling LED_Tx
- Check that the "Engine Off" warning disappears in the Virtual Cockpit when the message is sent **every 100ms** and that the message is shown in the list of received messages
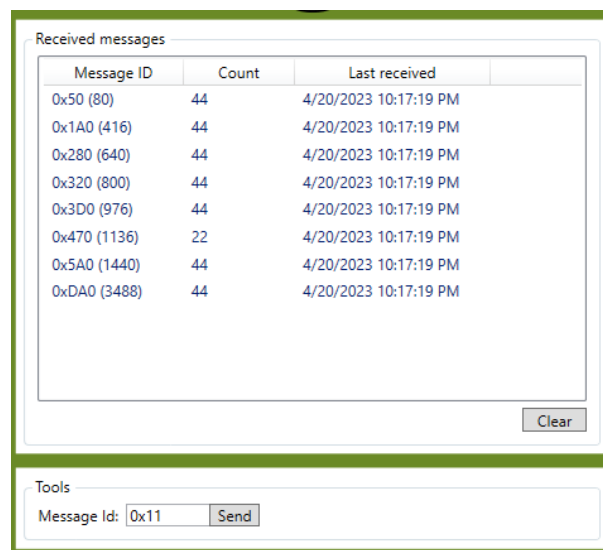

Figure 8: The area of VirtualCockpit to check received messages (top) and to sent messages to the bus / ECU (bottom)

4.3. Task 3 - Controlling the Virtual Cockpit

In the previous tasks you set up the connection between VirtualCockpit and your ECU. In this task the virtual dashboard portrayed by the Virtual Cockpit tool and your ECU will interact as follows:

Initially, the Virtual Cockpit is in **off** mode. If you click on the "Start Engine" button it will sent a single CAN message with ID 0x11 which is a wake-up signal for your ECU which then starts it operation.

Once your ECU "woke up" it should start sending different messages which control the display elements of the virtual dashboard (speed gauge, indicators etc.)

Please find the complete specification for the ECU here:
- The ECU starts in idle mode and **waits** for the "Start Engine" CAN message (ID 0x11) to be received
- If the "Start Engine" message was received the ECU should turn on LED_P and start sending the following messages:
  ‣ **EngineOn** message
  ‣ **Speed** message:
    – The current potentiometer value should be mapped to the speed
      • Turning the potentiometer all to the left (measurement value = 0) results in a speed of 0 km/h
      • Turning the potentiometer all to the right (measurement value = 4095) results in the max. speed of 240 km/h
      • the values in-between should be interpolated accordingly (e.g. turning halfway results in 50% max. speed -> 120 km/h)
  ‣ **Lights** message:
    – based on the state of Switch 1 and Switch 2, the LEDs and indicators should blink as defined by Table 1
    – Choose an appropriate blinking interval for the indicators.
  ‣ **RPM** message:
    – The RPM value should start with 8000 and decrease by 250 every second until 0
    – When the RPM reaches 0 then it should be increased by 250 every second until 8000
    – This procedure should be continuous
- If the CAN message "Stop Engine" (ID 0x10) is received turn off the LED_P and switch back to idle mode (i.e. sending **no** messages)

Please see Table 2 for the description of the corresponding CAN messages your ECU will have to send. Remember that there are two different data frame formats, standard and extended format with different message id sizes. Choose the right frame format depending on the Message ID. Every message needs to be **sent every 100 milliseconds**.

While you can reuse a single message buffer for sending the messages, it is usually more efficient to use one message buffer per message. For this, select and configure a **separate**

Table 1: Rules for the indicator function

Input SW1	Input SW2	Function	Onboard Output	Can message
0	0	Indicators off	LEDs off	see Table 2
0	1	Left indicator on	LED_U1 blinks	see Table 2
1	0	Right indicator on	LED_U3 blinks	see Table 2
1	1	Hazard warning lights	LED_U1 and LED_U3 blink	see Table 2

buffer for each message you need to send. For sending the actual message you can use the prepared **can_send_*** functions in **main.c**. Just remember to actually call them (e.g. by adding them to **can_send**).

> Virtual Cockpit's "Start Engine" button sends a message with ID 0x11, if the message 0xDA0 is **not present** on the bus (e.g. the car is off). If the car is on, a message with ID 0x10 is sent instead (-> "Stop Engine")

> If you want to re-use a single buffer instead of configuring multiple buffers, make sure to wait until the CAN Controller signals than the buffer is idle in the field before updating the buffer it.

Table 2: Definition of CAN messages which need to be sent by your ECU

Message ID	Description	Encoded signals / remarks
0xDA0	Engine On	Data length = 0 *no payload*
0x5A0	Vehicle speed	Data length = 1 1st byte : *speed* value in km/h
0x280	Engine RPM	Data length = 2 As the RPM value is too large to fit in a single byte you need to **split it in two bytes** and provide them in the correct order (little endian): 1st byte: Lower byte of *rpm* value 2nd byte: Higher byte of *rpm* value
0x470	Lights	Data length = 1 1st byte: 0x00 = No indicator on 0x01 = Left indicator on 0x02 = Right indicator on 0x03 = Hazard indicator on