FRIWO | SDK

SOFTWARE DEVELOPMENT KIT

End-to-end Development Environment Setup Solution

APPLICATION GUIDE – CAN COMMUNICATION

ABOUT

The **FRIWO SDK** enables the user to integrate own functionalities into a fully developed software environment for FRIWO products.

This document demonstrates how to adapt the received and sent **CAN messages** of the motor controller to the requirements of your own system. With the module, all identifiers used by the motor controller can be adapted to suit individual hardware, such as an individual vehicle display.

This guide gives a step-by-step overview of the implementation workflow from project creation to flashing and testing of the generated firmware on hardware. Therefore, it assumes the **FRIWO SDK** Tool Environment to be set up already. For a guidance of the basic setup please refer to our **Quickstart Guide**.

https://friwo.link/ag/quickstart-guide

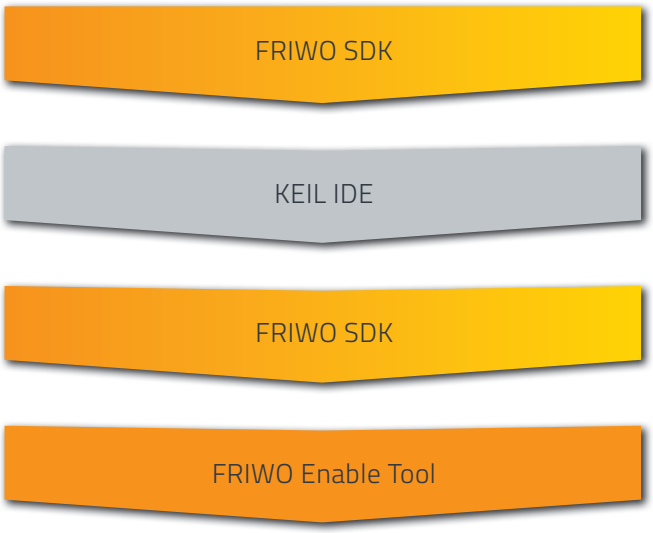
If you need a detailed description of the variable naming scheme, have a look at the **Software Manual**.

https://friwo.link/ag/manual

For a detailed description of the used module, please download our **Module Description**.

https://friwo.link/ag/module-description

THE BASIC WORKFLOW OF THE FRIWO SDK



We start, by creating a new project inside the **FRIWO SDK**. All necessary basic software gets pulled from the FRIWO Servers and made ready for usage.

In order to write your software, you need an IDE. We recommend **Keil IDE** or Visual Studio Professional.

After the software is written inside the IDE, we switch back to the **FRIWO SDK**. We start the compilation process and receive the customized firmware inside our workspace folder.

With the **FRIWO Enable Tool**, we can flash the customized firmware on the Motor Control Unit or Battery Management System.

CREATE A NEW PROJECT

First, a new project is created using the following steps.

- Open FRIWO SDK
- In main view press **Select Project**
- **Name** the project (i.e. CANApplicationGuideExample)
- Choose your **workspace folder** path (default: C:\Users\USERNAME\Documents\SDK_Workspace)
- Choose framework **MCU FRIWO Standard V1.1**
- Press **Create**

Next, we define the module to be customized.

- In main view press **Select Module**
- Select module **CAN** to be customized
- Confirm the dialog window
- Press **OK**

A new project folder is created in your workspace folder. The project's subfolder .\module_CAN contains the c-files CAN_custom.c and CAN_custom.h, as well as a variable description file CAN_variables.xml and the header-file canApi.h.

FIRST LOOK ON API HEADER FILE

Now the functionality of the CAN bus can be modified by adjusting the generated c-files of the CAN module. This is done by using an editor of your choice. The following procedure is explained using Keil µVision4 IDE.

For the first step we are having a look at the header-file *canApi.h*.

- Open **Keil µVision4 IDE**
- Select **File->Open**
- Navigate to the project's workspace and there inside the subfolder .\module_CAN
- Select **canApi.h**
- Press **Open**

The header file shows all available functions and types to interact with the CAN peripheral and the data in- and out-ports of the motor controller firmware.

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* PUBLIC FUNCTION PROTOTYPES */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/* user-side CAN buffer access functions */

/* Utility Functions */

/**
 * @brief Initialize buffers and set the buffer type implementation
 * @param receiveBufferType: Buffer type of CAN receive buffer
 * @param transmitBufferType: Buffer type of CAN transmit buffer
 */
void canApi_SetupBuffer(buffer_BufferType receiveBufferType, buffer_BufferType
transmitBufferType);

/**
 * @brief Clear the content of the transmit buffer
 */
void canApi_ClearTransmitBuffer(void);

/**
 * @brief Clear the content of the receive buffer
 */
void canApi_ClearReceiveBuffer(void);

/**
 * @brief Puts a message into the buffer to send it using the CAN peripheral.
 * @param message: message to send
 * @return Status of the transmit buffer system
 */
canApi_StatusTypeDef canApi_SendMessage(const canApi_MessageTypedef *message);

/**
 * @brief Get a message from the CAN receive buffer system.
 * @param message: Target pointer to store the received message
 * @retval Status of the receive buffer system
 */
canApi_StatusTypeDef canApi_ReceiveMessage(canApi_MessageTypedef *message);
```

The most important functions are **canApi_UserInitCallback** and **canApi_UserPeriodicCallback** which must be implemented by the user. The firmware calls the init-callback at startup of the system. The user can initialize his own code here. He also must initalize the input- and output buffers for the CAN peripheral using the **canApi_SetupBuffer** function.

Buffer Type	Description
Ringbuffer	A simple FIFO ringbuffer implementation, strongly recommended for the receive buffer.
Prioritybuffer - Queue	A queue with individual message priorities. Highest priority in buffer is sent first. User can define priorities from 1 (lowest) to 255 (highest). If a message could not be sent during a cycle, its priority is raised by 1. This implementation can be used to prioritize control messages over information messages in the buffer (on the CAN bus itself, messages are prioritized by identifier). A priority value of 0 is reserved to mark an empty slot and should therefore not be assinged to a message.
Prioritybuffer – Replace	Functions the same as the queue, but replaces existing messages with a new one if the identifier is identical. This can be used for messages, which are outdated as soon as a new message with the same identifier is put into the buffer. Should not be used with higher protocols like UDS / ISO 15765-2.

In order to reduced the workload of the CAN firmware, the developer can configure hardware filter for the CAN peripheral. He has access to 26 individual filter banks which can hold up to 4 standard identifier each (or 2 extended identifier) in list mode. The list mode of a filter bank works like a whitelist for CAN identifier. The user can also configure a mask mode. In this case, a message is received if the received identifier matches the configured template at the bit positions defined by the mask.

Example for mask configuration:

- Mask = 0x1FFFFFFE and ID = 0x00000002 will catch IDs 2 and 3
- Mask = 0x1FFFFFF8 and ID = 0x00000000 will catch IDs from 0 to 7
- Mask = 0x00000001 and ID = 0x00000001 will catch all odd IDs.

All messages which are catched by a list filter or mask setup, can be evaluated in the software. All other messages are not received by the CAN peripheral. The use of suitable filters is strongly recommended, since a heavily loaded CAN bus can block the reception of important messages and significantly degrade the performance of the CAN module software and the main firmware.

The canApi.h offers a variety of different functions to adapt the CAN filter banks to the needs of the application. It is recommended to configure the filter banks inside the user implementation of the **canApi_UserInitCallBack** function.

FIRST LOOK ON CUSTOM USER FILE

The CAN module contains as default implementation a predefined CAN bus consisting of Friwo motor controller, Friwo battery and our standard display. For a detailed description of the structure of individual CAN messages, please refer to the corresponding CAN database files (.dbc files).

As the next step, open the CAN customer file.

- Select **File->Open**
- Navigate to the project's workspace and there inside the subfolder .\module_CAN
- Select **CAN_custom.c**
- Press **Open**

The file implements the mandatory user callbacks of the canApi header as well as a user defined system to manage message timeout and receive callbacks and the functions to send CAN frames. Each signal, which can be passed on to the firmware via a function on reception, also has a function for signaling a timeout. The user should set the timeouts of all signals of the API, which are not used in his implementation, to avoid that invalid values are used in the firmware.

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* PRIVATE TYPEDEF */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/** @brief define pointer to function for message timeout callback */
typedef void (*FptrOnTimeout)(void);

/** @brief define pointer to function for message receive callback */
typedef void (*FptrOnReceive)(const canApi_MessageTypedef *message);
/**
 * @brief Typedef to map received messages to timeout settings and callback functions.
 * This is part of the helper functions to manage message receival and timeout management.
 */
typedef struct
{
       uint32_t CanIdentifier; /**< @brief Identifier of the received message */
       uint8_t IDE; /**< @brief 0x00u = standard frame identifier, 0x01u = extended frame
identifier*/
       int16_t TimeoutCounter; /**< @brief Current timeout counter value, set to negative
value to disable timeout */
       int16_t TimeoutReloadValue; /**< @brief Reload counter value. Timeout counter is
reset to this value on message receive, set to negative value to disable timeout*/
       FptrOnTimeout TimeoutFunction; /**< @brief pointer to function which is called on
message timeout detection */
       FptrOnReceive ReceiveFunction; /**< @brief pointer to function which is called on
message receive */
}msgManagement_TypeDef;
```

This example uses the custom type **msgManagement_Typedef** to manage timeout- and receive callback functions for each message to receive. Each message can have its individual timeout value in milliseconds. Upon reaching the timeout value, a timeout callback function is executed once to set the timeouts for all signals used in the individual message. The receive callback sets the timeout variables back to their reload value and puts the received signals into the firmware by using the API function provided by the canApi header file. The reception and transmission of CAN messages is managed in the **canApi_UserPeriodicCallBack** function. The function is called every millisecond.

TRANSMIT A NEW PERIODIC MESSAGE

In this example we will add a new message to the system, which will be sent periodically. Let's assume that your vehicle has its own display, which gets its display data via CAN. We want to define a new message that we can use to display the current speed and mileage on the device.

In the first step, we declare the function to send a new message:

```
/* functions to send individual predefined messages */
static void MessageSend0x1BF(void); /* PE_Act_05 */
static void MessageSend0x1BD(void); /* MC_Temperature_01 */
static void MessageSend0x1BC(void); /* MC_Errorflags_01 */
static void MessageSend0x2B9(void); /* MC_State_01 */
static void MessageSend0x1BA(void); /* MC_Current_01 */
static void MessageSend0x601(void); /* MC_Prod_Data_01 */
static void MessageSend0x602(void); /* MC_Prod_Data_02 */
static void MessageSend0x603(void); /* MC_Prod_Data_03 */
static void MessageSend0x604(void); /* MC_Prod_Data_04 */
static void MessageSend0x1F0(void); /* MC_APP_01*/
static void MessageSend0x1F1(void); /* MC_APP_02*/
static void MessageSend0x1F2(void); /* MC_APP_03*/
static void MessageSend0x1F4(void); /* MC_APP_04*/
static void MessageSend0x90(void);  /* ICS_Info_01 */
static void MessageSend0x206(void); /* Odo */
static void MessageSend0x207(void); /* Display_01 */
static void MessageSend0x305(void); /* Display_02 */
static void MessageSend0x306(void); /* Display_03 */
static void MessageSend0x209(void); /* Error */
static void MessageSend0x1B5(void); /* Challenge for Immo Unlocking*/
static void MessageSend0x1B7(void); /* Unlock Code sent to GRID-BMS if needed by GRID */
static void MessageSend0x160(void); /* BMS Ctrl 01 */

static void MessageSendFictionalDisplay(void); /* new message for our example */
```

The implementation of the function is quite simple. We declare a message using the TypeDef of the API and fill it with the desired data. In this case, we use the **canApi_Get_INFO_ODO_Trip_Kilometers** and **canApi_Get_INFO_Vehicle_Speed** functions to get the values from our firmware. The message is sent using an extended identifier.

```
static void MessageSendFictionalDisplay(void)
{
        UInt32 temp_trip_m = 0;
        Int32  temp_speed;

        canApi_MessageTypedef message;
        message.DLC = 8; /* 8 bytes of data in the CAN frame */
        message.IDE = 1; /* use extended identifier */
        message.Identifier = 0x1FFFFF00; /* CAN identifier */
        message.Priority = 1;
        message.RTR = 0;

        temp_trip_m = canApi_Get_INFO_ODO_Trip_Kilometers();
        temp_speed = canApi_Get_INFO_Vehicle_Speed();

        message.Data[0] = (UInt8)(temp_trip_m); /* first four byte contain current milage */
        message.Data[1] = (UInt8)(temp_trip_m >> 8);
        message.Data[2] = (UInt8)(temp_trip_m >> 16);
        message.Data[3] = (UInt8)(temp_trip_m >> 24);
        message.Data[4] = (UInt8)(temp_speed); /* last four byte contain current speed */
        message.Data[5] = (UInt8)(temp_speed >> 8);
        message.Data[6] = (UInt8)(temp_speed >> 16);
        message.Data[7] = (UInt8)(temp_speed >> 24);

        canApi_SendMessage(&message);
}
```

In the last step, we add the call to our new function to the **canApi_UserPeriodicCallBack** function inside the desired timeslot to send it via CAN every 1000 milliseconds.

```
if (timeslot % 1000 == 0)
{
        /* send messages in 1000 millisecond interval */
        MessageSend0x1BD(); /* MC_Temperature_01 */
        MessageSend0x1F1(); /* MC_APP_02*/
        MessageSend0x1F2(); /* MC_APP_03*/
        MessageSend0x601(); /* MC_Prod_Data_01 */
        MessageSend0x602(); /* MC_Prod_Data_02 */
        MessageSend0x603(); /* MC_Prod_Data_03 */
        MessageSend0x604(); /* MC_Prod_Data_04 */

        MessageSendFictionalDisplay(); /* Send the data to our fictional display */
}
```

RECEIVE A NEW MESSAGE

With the module we can receive and evaluate customized messages. This can be used to adjust messages with existing signals, to make runtime data of other CAN nodes visible in the **Enable Tool** (for data logging and graphical plots) or to control other customized modules in the motor controller at runtime.

https://friwo.link/ag/enable-tool

In this example we will create a new message and associated callbacks to display data provided over the CAN bus in the Enable Tool. In the first step, we create a new public variable for this purpose.

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* PUBLIC VARIABLES */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

__attribute__((section(„EMERGE_NV_RAM_PAGE1“)))
MEDKit_Modul_Interfaces UInt32 CAN_C_Switch_KilometerToMiles = 0; /*
       Description: Select between Kilometer and Miles for CAN output;StateList;
       0 = Kilometer;1 = Miles; Limits: 0...1 */

__attribute__((section(„EMERGE_NV_RAM_PAGE1“)))
MEDKit_Modul_Interfaces UInt32 CAN_C_SwitchDataInfo_ID_207 = 0; /*
       Description: CAN-bus Display Data[-];StateList;0 = Boost; 1 = SetpointTorque;
2 = MaxTorque; 3 = MappingTorque; Limits: 0...3 */

__attribute__((section(„EMERGE_NV_RAM_PAGE1“)))
MEDKit_Modul_Interfaces UInt32 CAN_C_SwitchDataInfo_ID_306 = 0; /*
       Description: CAN-bus Display Data[-];StateList;0 = BateryVoltage;
       1 = RemainigDistance; Limits: 0...1 */

__attribute__((section(„EMERGE_DISP_RAM“)))
MEDKit_Modul_Interfaces UInt32 CAN_M_ReceivedTestData = 0; /*
       Description: CAN-bus Test data, received value via CAN in demo code */
```

We also declare the two callback functions for the receive event and timeout event of the new message:

```
/* callback functions for received messages and their timeouts */
static void MessageTimeout0x111(void);
static void MessageReceive0x111(const canApi_MessageTypedef *message);
static void MessageTimeout0x1B6(void);
static void MessageReceive0x1B6(const canApi_MessageTypedef *message);
static void MessageTimeout0x171(void);
static void MessageReceive0x171(const canApi_MessageTypedef *message);
static void MessageTimeout0x172(void);
static void MessageReceive0x172(const canApi_MessageTypedef *message);
static void MessageTimeout0x176(void);
static void MessageReceive0x176(const canApi_MessageTypedef *message);
static void MessageTimeout0x178(void);
static void MessageReceive0x178(const canApi_MessageTypedef *message);
static void MessageTimeout0x310(void);
static void MessageReceive0x310(const canApi_MessageTypedef *message);
static void MessageTimeout0x521(void);
static void MessageReceive0x521(const canApi_MessageTypedef *message);
static void MessageTimeout0x50C(void);
static void MessageReceive0x50C(const canApi_MessageTypedef *message);

static void MessageTimeoutDemo(void);
static void MessageReceiveDemo(const canApi_MessageTypedef *message);
```

Next, we register the callbacks and timeouts by creating a new entry in the **msgManagement_array.** We set the receive CAN identifier to 0x600 standard identifier with a maximum timeout value of 500 milliseconds.

```
/**
 * @brief array of commands with their corresponing execution functions
 * All received messages and their timeouts and callbacks must be defined here.
 */
static msgManagement_TypeDef msgManagment_array[] =
{
	/*{Identifier, IDE, TimeoutCounter, TimeoutReloadValue, TimeoutCallback,
ReceiveCallback}*/
	{0x111, 0, 200, 200,  MessageTimeout0x111, MessageReceive0x111},
	/* Message EXT_Torque_Control_01 */
	{0x1B6, 0, 200, 200,  MessageTimeout0x1B6, MessageReceive0x1B6},
	/* Message EXT_Immo_Control_01 */
	{0x171, 0, 200, 200,  MessageTimeout0x171, MessageReceive0x171},
	/* Message BMS_Info_01 */
	{0x172, 0, 2500, 2500,  MessageTimeout0x172, MessageReceive0x172},
	/* Message BMS_Info_02 */
	{0x176, 0, 2500, 2500,  MessageTimeout0x176, MessageReceive0x176},
	/* Message BMS_Info_06 */
	{0x178, 0, 2500, 2500,  MessageTimeout0x178, MessageReceive0x178},
	/* Message BMS_Info_08 */
	{0x310, 0, 200, 200,  MessageTimeout0x310, MessageReceive0x310},
	/* Message Dyno_Act_01 */
	{0x521, 0, 200, 200,  MessageTimeout0x521, MessageReceive0x521},
	/* Message ISA_Scale_F1_Current_Sensor */
	{0x50C, 0, 200, 200,  MessageTimeout0x50C, MessageReceive0x50C},
	/* Message CAN Display Reset Message */
	{0x600, 0, 500, 500, MessageTimeoutDemo, MessageReceiveDemo},
	/* Demo message for display in EnableTool */
};
```

Now we need to implement our callback functions. To keep this guide simple, we just assign the received value to our test variable. If the length of the CAN frame is wrong or a timeout occurs, the test variable is set to 0. The data from the CAN frame is read as an unsigned 32-bit value in little-endian format.

```
static void MessageTimeoutDemo(void)
{
	CAN_M_ReceivedTestData = 0;
}
static void MessageReceiveDemo(const canApi_MessageTypedef *message)
{
	uint32_t tmp = 0;
	if (message->DLC == 4u)
	{
		/* copy the data from the CAN frame to the display variable */
		tmp |= message->Data[0];
		tmp |= message->Data[1] << 8;
		tmp |= message->Data[2] << 16;
		tmp |= message->Data[3] << 24;
	}
	CAN_M_ReceivedTestData = tmp;
}
```

To be able to receive the message, the CAN hardware filter must now be configured. We configure a FilterBank in Listen mode to let only the identifier of our test message pass to the software.

The filter is configured only once in the **canApi_UserInitCallback** function.

```
/**
 * @brief Initialize the user implementation layer. Setup the API buffer structure.
 * This function is called once when the firmware initialized the CAN peripheral.
 */
void canApi_UserInitCallBack(void)
{
     /* Init the buffer structure. We use a ringbuffer implementation in this example */
     canApi_SetupBuffer(RINGBUFFER, RINGBUFFER);
     canApi_ClearTransmitBuffer();
     canApi_ClearReceiveBuffer();

     /* Set filter */
     canApi_FilterSetFourStdIdListMode(FilterBank01, 0x111, 0, 0x1B6, 0, 0x171, 0, 0x172, 0);
     canApi_FilterSetFourStdIdListMode(FilterBank02, 0x176, 0, 0x178, 0, 0x310, 0, 0x521, 0);
     canApi_FilterSetTwoStdIdListMode(FilterBank03, 0x50C, 0, 0x0, 0);

     canApi_FilterSetOneStdIdListMode(FilterBank04,0x600,0);
}
```

To be able to view / edit a variable with the **Enable Tool**, we must declare it in the **CAN_variables.xml** file. Open it with your editor and add the description for our new variable **CAN_M_ReceivedTestData**. https://friwo.link/ag/enable-tool

```
<ddObj Name="CAN_M_ReceivedTestData" Kind="Variable">
        <ddProperty Name="Description">
                CAN-bus Test data, received value via CAN in demo code</ddProperty>
        <ddProperty Name="Type">UInt32</ddProperty>
        <ddProperty Name="Scaling">./LocalScaling</ddProperty>
        <ddProperty Name="Value">0</ddProperty>
        <ddProperty Name="Min">0</ddProperty>
        <ddProperty Name="Max">4294967295</ddProperty>
        <ddProperty Name="Address"></ddProperty>
        <ddObj Name="LocalScaling" Kind="Scaling">
                <ddProperty Name="LSB">1</ddProperty>
                <ddProperty Name="Unit">s</ddProperty>
        </ddObj>
</ddObj>
```

CREATING THE NEW FIRMWARE

At this point we have successfully prepared the c- and variable description files for implementation of the CAN demo code. In the following we use FRIWO SDK to compile our individual module CAN_custom and integrate it into the basic software framework.

- **Open** FRIWO_SDK (if not already open)
- In main view press **Select Project**
- Press **Load Project**
- Navigate to the **project folder** in your SDK workspace (i.e. .\SDK_Workspace\YourProjectName)
- Select the **project file** (i.e. CAN_demo.sdkproj) and press open

Now the default project settings are loaded. We can check if the right module (CAN_custom) is selected by pressing Select Module in the main view.

- Press **Compile** to build the firmware

The compile process is finished if the process bar is fully loaded and the status shows "Finished – Click here to view your firmware!". By clicking on the text, you are navigated directly to the output build folder of the firmware file (*.eef). This folder is generated inside the project path (i.e. .\YourProjectName\FirmwareRelease\RELEASEID)

If an error occurs during the compilation process, refer to **section „Troubleshooting" in our Manual** using the indicated error code.
https://friwo.link/ag/manual

UPDATE MOTOR CONTROLLER

For updating the Motor Controller with the generated firmware we have to switch from FRIWO SDK to FRIWO **Enable Tool**.
https://friwo.link/ag/enable-tool

Please refer to **Enable Tool Manual** for further information about updating the Motor Controller.
https://friwo.link/ag/et-manual

TEST THE NEW FIRMWARE

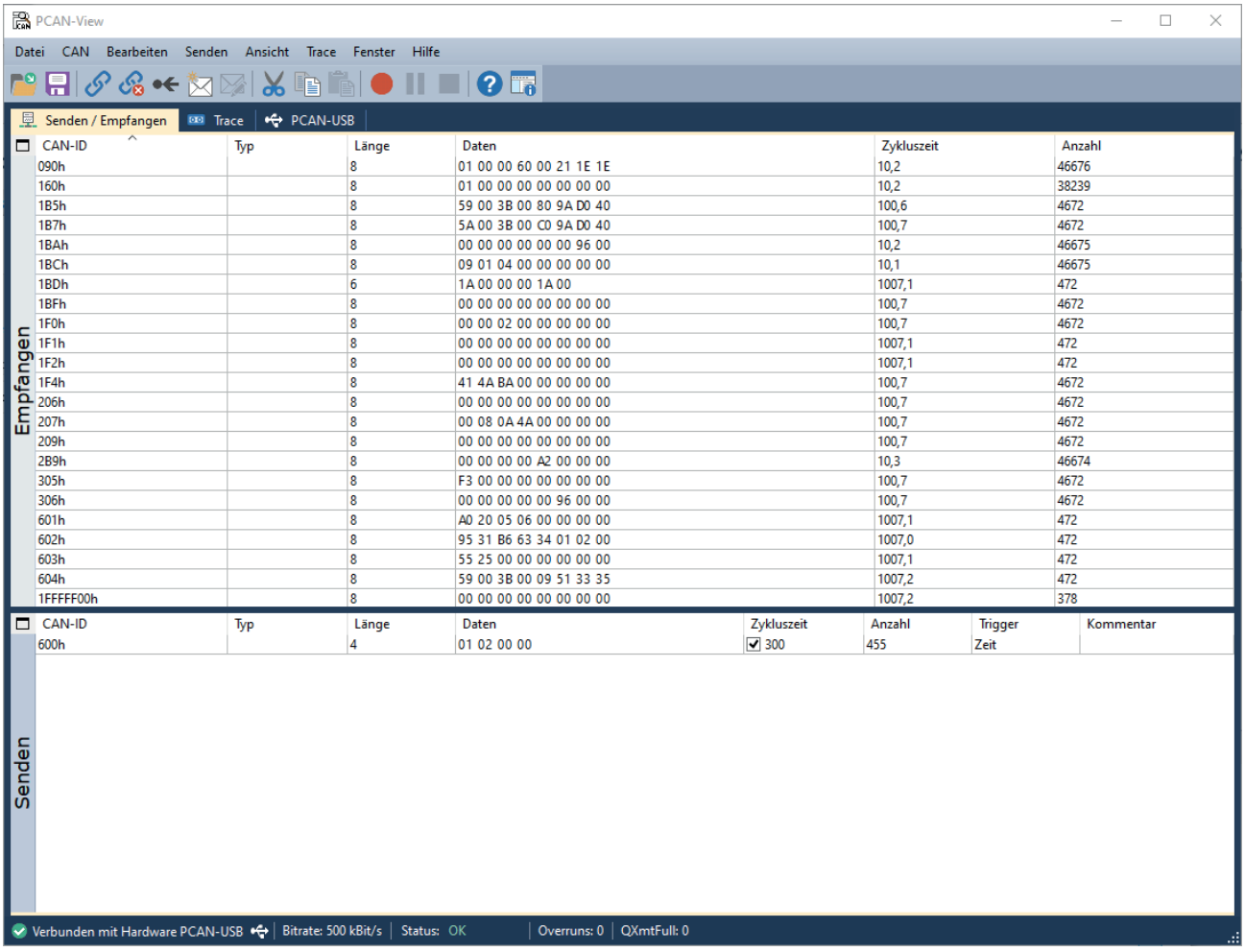
In order to be able to monitor the messages on the CAN, suitable hardware is required. For this example a PEAK CAN USB adapter was used to verify the functionalities.

The screenshot shows the freeware **PCAN-View** which can be used to send and receive CAN frames to test the CAN bus of our system.

https://friwo.link/ag/pcan

Our desired test message on the ID 0x1FFFFF00 is sent by the motor controller in a one second interval as intended. If we use the **PCAN-View** tool to send test frames on ID 0x600 with our configured format in a suitable interval (here 300 milliseconds) we can view the transmitted values in the **Enable Tool** in real time by observing our new parameter **CAN_M_ReceivedTestData**.

The value of the parameter switches to 0 after reaching the timeout if we disconnect the CAN connection.



You can implement additional functions, messages and interfaces to your other CAN devices in this fashion, even with significantly more complex functions.

Have fun programming!

Feedback

We are working very hard to improve our products and therefore **feedback** is indispensable! Please send us your valuable feedback as contact form or via Mail to feedback@friwo.com

https://friwo.link/ag/feedback