CANMiniDriver for CODESYS V3

Implementation Guidelines

Version: 1.0
Template: templ_tecdoc_en_V2.0.docx
File name:
CODESYSControlV3_CANMiniDriver_Guidelines.docx

CONTENT

1 Introduction

CODESYS V3 provides a generic CAN L2 (CL2) layer for accessing the CANbus.
It implements hardware-independent functions for both sending and receiving messages and CANbus diagnosis. These functions are accessible from C as well as IEC. All CAN-based I/O drivers (e.g. CANopen Stack, J1939 Stack) are based on this layer.
The concrete access to the respective CAN chip is implemented by CANMiniDrivers. At runtime start, a CANMiniDriver registers each CAN interface at the CL2 layer.
Here a unique Network ID will be assigned and used for addressing the interface later.

CODESYS V3 already provides ready-to-use CANMiniDriver implementations for SJA 1000 (RTE), PCAN Basic (Control Win), SocketCAN (Linux), and EL6751 (EtherCAT/CAN Gateway).
For other CAN chips and platforms, a specific CANMiniDriver component has to be implemented.
This document serves as an implementation guideline for CANMiniDrivers and describes basic concepts.

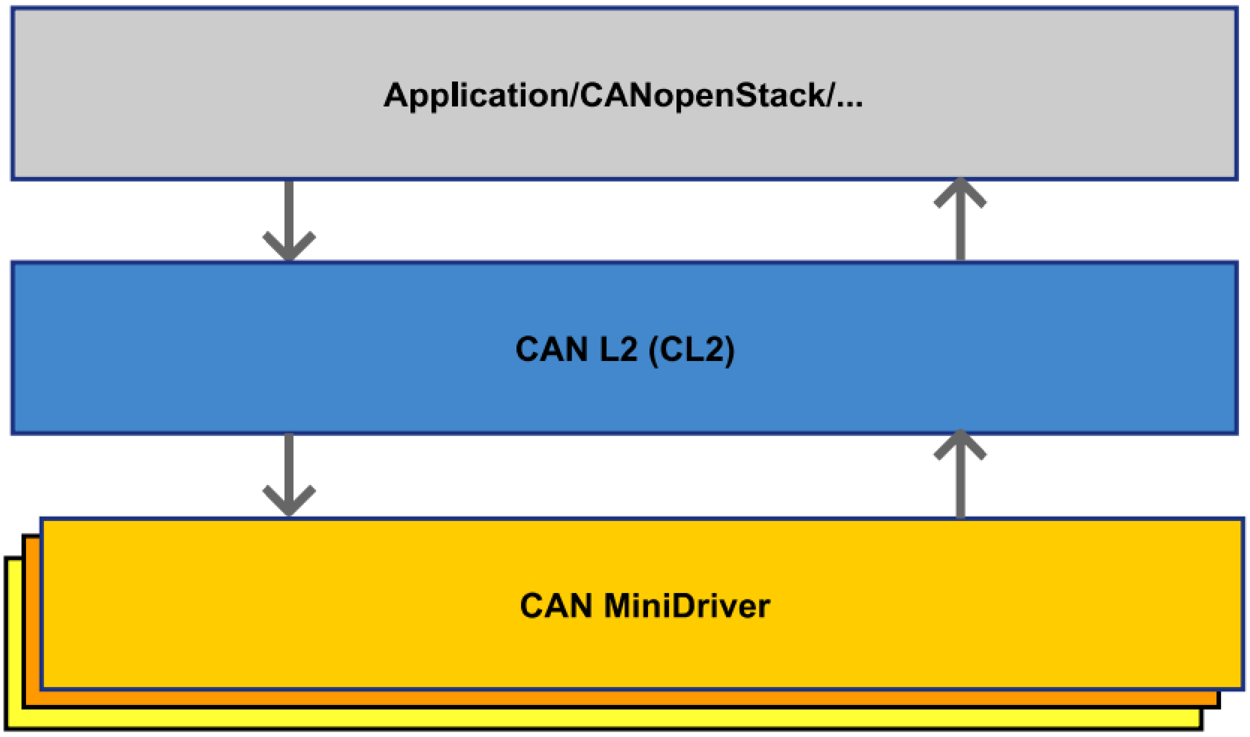


Figure 1: Overview

2 CANMiniDriver interface

A CANMiniDriver has to implement a certain interface. This interface is defined in CAACanMiniDriver.h (located in the Component folder) and will be described in the following chapters.

```
typedef struct cmd_cmdrv_tag
{
    CAA_ERROR    (*Setup)(CL2I_BYTE byNet);
    CAA_ERROR    (*Init)(CL2I_BYTE byNet, CL2I_WORD wBaudrate);
    CAA_BOOL     (*Send)(CL2I_BYTE byNet, CAA_HANDLE hBlock, CL2I_BYTE byPrio, CAA_ERROR* pError);
    CAA_BOOL     (*Receive)(CL2I_BYTE byNet, CAA_HANDLE hBlock, CAA_ERROR* pError);
    CAA_ERROR    (*GetInfo)(CL2I_BYTE byNet, CMD_INFO* pInfo);
    CAA_ERROR    (*Dispose)(CL2I_BYTE byNet);
    CAA_ERROR    (*Identify)(CL2I_BYTE byNet, CL2I_BYTE byCount);
    CAA_HANDLE   (*SetBlock)(CL2I_BYTE byNet, CL2I_BYTE byIndex, CAA_HANDLE hBlock, CAA_ERROR* peError);
    CAA_ERROR    (*SetCycle)(CL2I_BYTE byNet, CL2I_BYTE byIndex, CL2I_DWORD dwCycle);
    CAA_ERROR    (*SetMask)(CL2I_BYTE byNet, CL2I_BYTE byIndex, CL2I_DWORD dwValue, CL2I_DWORD dwMask);
    CAA_ERROR    (*ResetAlarm)(CL2I_BYTE byNet);
    CAA_ERROR    (*SetStatus)(CL2I_BYTE byNet, CL2I_BYTE byIndicator);
} CMD_CMDRV;
```

Figure 2: CANMiniDriver interface

2.1 Required functions

There is a minimum set of functions which have to be at least implemented by a CANMiniDriver.

2.1.1 CMD_GetInfo: Supported functions

Each CANMiniDriver implements the function `CMD_GetInfo` which provides information about the supported CANMiniDriver functionalities. This information is packed into a CMD_Info structure defined in CAA_CanMiniDriver.h:

```
typedef struct cmd_info_tag
{
    CL2I_BYTE    bySupport;
    CL2I_BYTE    byMaxCycleIndex;
    CL2I_BYTE    byMaxMaskIndex;
    CL2I_BYTE    byMaxPrio;
    CL2I_BYTE    byNLed;
} CMD_INFO;
```

Figure 3: Info structure

- **bySupport**: A bit field with following possible bits:
```
#define CMD_SUPPORT_IDENTITY    0x01
#define CMD_SUPPORT_SETBLOCK    0x02
#define CMD_SUPPORT_SETCYCLE    0x04
#define CMD_SUPPORT_SETMASK     0x08
#define CMD_SUPPORT_EXTCOBID    0x10
#define CMD_SUPPORT_RTRFRAME    0x20
#define CMD_SUPPORT_BUSALARM    0x40
#define CMD_SUPPORT_STATUSLED   0x80
```
  Figure 4: Supported functions bits

- **byMaxCycleIndex**: Contains the maximum number of cyclic messages (if SetCycle is supported).
  ➔ If one cyclic message is supported: set to 1.
- **byMaxMaskIndex**: Not yet used by current implementation. Use 0.
- **byMaxPrio**: Not yet used by current implementation. Use 1.
- **byNLed**: Number of LEDs. 0: No LED, 1: Bicolor LED, 2: red and green LED.
  Not yet used by current implementation.

Example:

```
static CMD_INFO cmdInfo    = {
  CMD_SUPPORT_RTRFRAME | CMD_SUPPORT_EXTCOBID | CMD_SUPPORT_BUSALARM
/* support RTR, Busalarm and 29 bit */,
  0        /* maxCycleIndex */,
  0        /* maxMaskIndex */,
  1        /* maxPrio  */,
  0        /* maxLed */
};

static CAA_ERROR CMD_GetInfo(CL2I_BYTE byNet, CMD_INFO* pInfo)
{
  if(pInfo)
  {
     memcpy((void*)pInfo, (void*)&cmdInfo, sizeof(CMD_INFO));
  }
  return CMD_NO_ERROR;
}
```

2.1.2 CMD_Setup and CL2_CmdRegister: Hardware identification, registration, and initialization

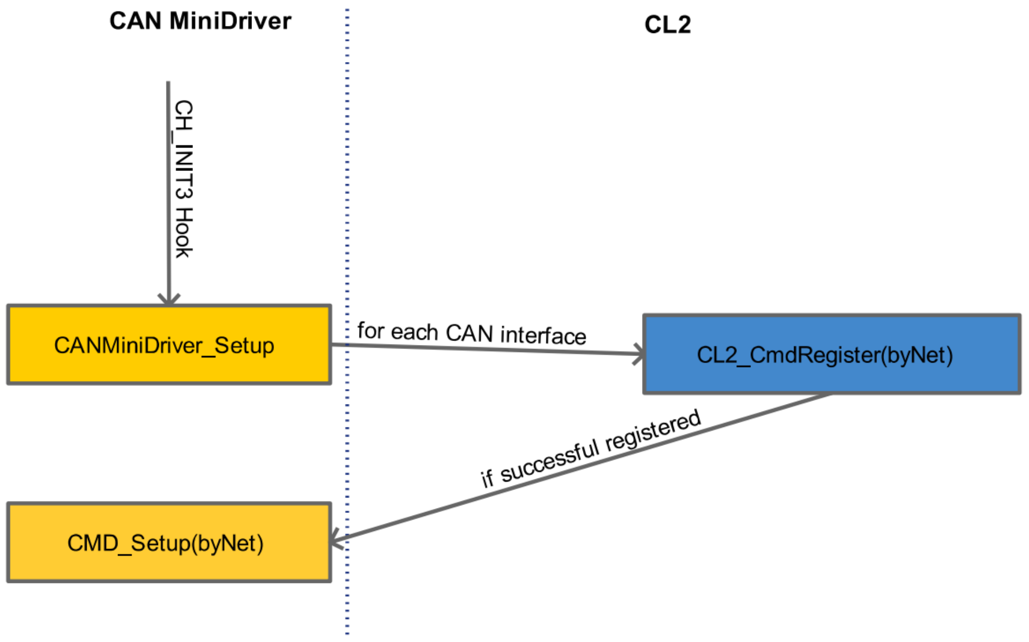


Figure 5: Init sequence

At startup (CH_INIT3 hook), a CANMiniDriver identifies all available CAN interfaces and registers them at CL2 layer by calling CL2_CmdRegister.

```
CAA_ERROR CAAFKT CL2_CmdRegister(unsigned char ucNetId, CMD_CMDRV* pCMDRV, CAA_COUNT ctMessages, CL2I_INFO** ppInfo)
```

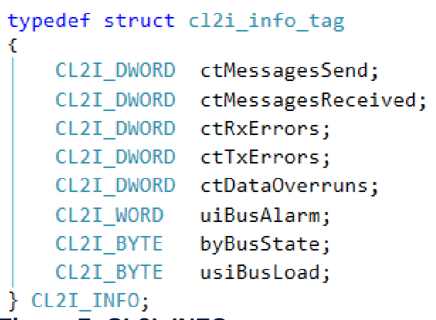
Figure 6: CmdRegister

It passes the following parameters to CL2:

- **ucNetID:** A unique number which identifies the CAN interface in CL2 layer. If Net ID is already used by another CAN interface (e.g. occupied by another CANMiniDriver instance) or exceeds maximal network number (define CL2_NNET) the function returns an appropriate error code.
- **pCMDRV:** A pointer to the CANMiniDriver interface. Basically this is a structure with function pointers used by CL2 to call into the CANMiniDriver (see Figure 7: CL2I_INFO).
- **ctMessages:** Number of messages that should be allocated initially for the Rx Message Pool.
  If 0, then CL2_NRXMSG from CAADefinesGeneric.h will be used (by default: 100).

Note: If Rx messages run out and define CL2_QUEUE_ADAPTION in CAADefinesGeneric.h is set to 1 (default), then the number of Rx messages will by dynamically increased at runtime.

- **ppInfo:** A pointer to a CL2I_INFO pointer. On successful registration, CL2 passes a pointer to a structure which serves as exchange memory for diagnostics.



```
typedef struct cl2i_info_tag
{
    CL2I_DWORD  ctMessagesSend;
    CL2I_DWORD  ctMessagesReceived;
    CL2I_DWORD  ctRxErrors;
    CL2I_DWORD  ctTxErrors;
    CL2I_DWORD  ctDataOverruns;
    CL2I_WORD   uiBusAlarm;
    CL2I_BYTE   byBusState;
    CL2I_BYTE   usiBusLoad;
} CL2I_INFO;
```

Figure 7: CL2I_INFO

Normally hardware is identified and registered by a function called `xxx_CanMiniDriver_Setup`, where xxx is the name of the driver (e.g. `SJA_CanMiniDriver_Setup`).

Example:

```
//interface definition
static CMD_CMDRV cmdInterface =
{
  CMD_Setup,
  CMD_Init,
  CMD_Send,
  CMD_Receive,
  CMD_GetInfo,
  CMD_Dispose,
  CMD_Identify,
  CMD_SetBlock,
  CMD_SetCycle,
  CMD_SetMask,
  CMD_ResetAlarm,
  CMD_SetStatus
};

#define XXX_NDRIVER CL2_NNET

static CL2I_INFO* s_pInfo[XXX_NDRIVER];       // exchange memory for diagnosis

static CL2I_BYTE s_byDriver[XXX_NDRIVER];     // lookup table for driver number

static CL2I_BYTE s_byNet[CL2_NNET];           // lookup table for NetID
static RTS_RESULT CDECL HookFunction(RTS_UI32 ulHook, …)
{
       switch (ulHook)
       {
              …

              case CH_INIT3:
                 xxx_CanMiniDriver_Setup();
                 break;

              …
       }
       return ERR_OK;
}


int xxx_CanMiniDriver_Setup(void)
{
  CL2I_BYTE byDriver;
  CL2I_BYTE byNet;
  CL2I_BYTE byLastNet = 0;
  CAA_ERROR eError;

  // TODO: search for available hardware interfaces

  // Register each driver
  for (byDriver = 0; byDriver < XXX_NDRIVER; byDriver++)
  {
      for (byNet = byLastNet; byNet < CL2_NNET; byNet++)
      {
          s_byDriver[byNet] = byDriver;
          s_byNet[byDriver] = byNet;
          eError = CL2_CmdRegister(byNet,
              &cmdInterface,
              CL2_NRXMSG,     // Size of RX-Msg Pool, you can also use 0
              &s_pInfo[byNet]

          );
          if (eError == CL2_NO_ERROR)
          {
              byLastNet = byNet + 1;
              break;
          }
          else
          {
              s_pInfo[byNet] = CAA_pNULL;
              s_byDriver[byNet] = XXX_NDRIVER;
              s_byNet[byDriver] = XXX_NNET;
          }
      }
  }
  return 0;
}
```

After successful registration of a CAN network, CL2 calls CMD_Setup:

```
static CAA_ERROR CMD_Setup(CL2I_BYTE byNet)
{
// TODO: Initialize the particular CANbus chip so that it reacts to the
// connected CANbus in a completely passive manner. The necessary memory should be //
allocated so that a subsequent call to CMD_Init starts the CANbus service
// without further expense. Interrupts should not be enabled yet.
}
```

2.1.3 CMD_Init: Initialize chip

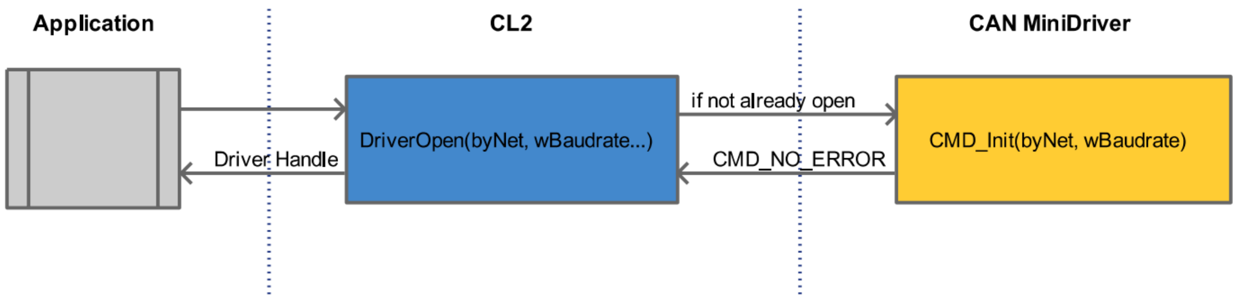


Figure 8: Driver Open

The first time an application calls the CL2 Driver Open function for a specific network, CL2 calls CMD_Init and passes NetID and desired baud rate to the particular CANMiniDriver instance. In this function, the CANMiniDriver prepares the chip for sending and receiving CAN messages with the given baud rate.

```
Example Implementation:
static CAA_ERROR CMD_Init(CL2I_BYTE byNet, CL2I_WORD wBaudrate)
{
  if (byNet < CL2_NNET)
  {
      CAA_ERROR eResult = CMD_NO_ERROR;
      CL2I_INFO* pCL2Info = s_pInfo[s_byDriver[byNet]];
      // TODO: init chip for driver instance s_byDriver[byNet] with wBaudrate
      // TODO: Reset pCL2Info ➔ set counters to 0, reset bus state, …
      return CMD_NO_ERROR;
  }

  return CMD_SETUP_ERROR;
}
```

2.1.4 CMD_Send, MsgSendAckn: Message Sending

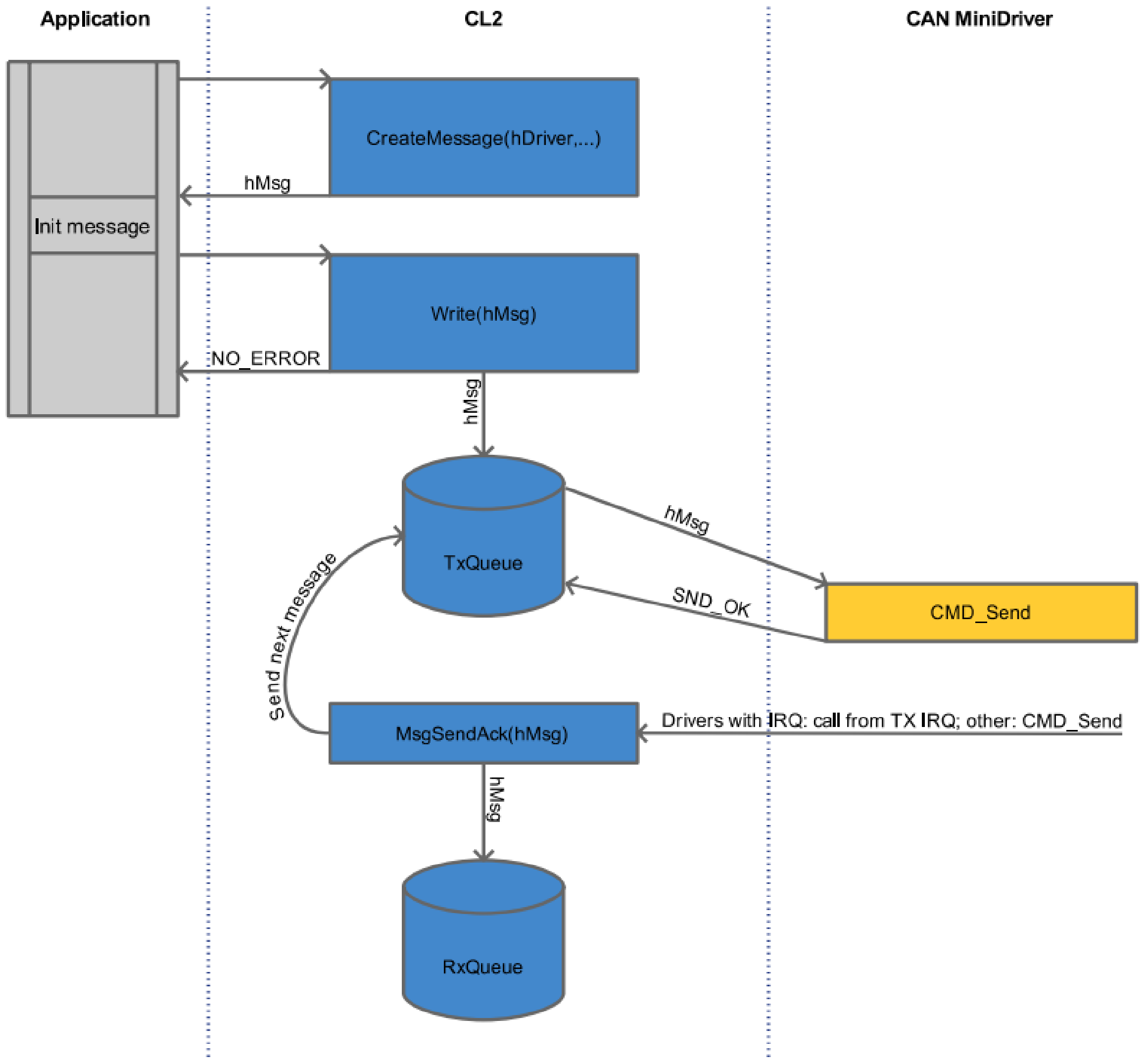


Figure 9: Message sending

If an application wants to send a CAN message, then it allocates a message handle first from the Tx message pool and initializes the message. Afterwards, it calls CL2.Write and passes the message handle to CL2 layer. If the Tx Queue is empty, then CL2 passes the message handle directly to the CANMiniDriver by invoking CMD_Send thread safe (synchronized by CAA_ENTER_CRITSEC). Then CMD_Send performs the execution of all steps necessary for the specific CANbus chip to send the message.
The function is defined as follows:

```
static CAA_BOOL CMD_Send(CL2I_BYTE byNet, CAA_HANDLE hBlock, CL2I_BYTE byPrio, CAA_ERROR* peError)
```

- Returns `CMD_SND_OK` and eError = `CMD_NO_ERROR` if message was sent successfully.
- Returns `CMD_SND_NOT_OK` and eError = `CMD_NO_ERROR` if message cannot be currently send (for example if chip is busy). CL2 will send the message back to the front of Tx Queue.
- Returns `CMD_SND_NOT_OK` and any error code for eError if message cannot be sent and should be dismissed by CL2.

If Tx Queue is not empty, then CL2.Write sends the message handle to the Tx Queue for later processing. In this case, CMD_Send will not be called.
When a CANMiniDriver finished sending for the current message, it calls CL2_MsgSendAckn with the current Tx message handle. (For further details, see following chapters). At this point, the Tx Queue will be checked by CL2 and another CMD_Send may be triggered.
As CL2 implements a Tx loopback functionality, MsgSendAckn checks if someone is interested in the Tx message. If so, then it will be sent to the Rx Queue. Otherwise the message handle will be released and sent back to the Tx message pool.

2.1.4.1 Drivers with Tx IRQ

Drivers with Tx Interrupt store the current Tx driver handle (CMD_Send) and call MsgSendAcknIRQ for that handle in Tx Interrupt handler. Then they signal that message was successfully sent and chip is ready for sending the next message. Furthermore, the interrupt service routine may update the time stamp of the message just sent.

Note: It is important that processing Tx interrupts before Rx interrupts. Otherwise there could be problems with synchronous PDOs when using CANopenStack and Motion.

Example:

```
static CAA_BOOL CMD_Send(CL2I_BYTE byNet, CAA_HANDLE hBlock, CL2I_BYTE byPrio,
CAA_ERROR* peError)
{
  CAA_BYTE byDriver = s_byDriver[byNet];
  CL2I_INFO* pInfo = s_pInfo[byDriver];
  CL2I_BLOCK* pBlock = (CL2I_BLOCK*)CAL_CL2_MsgGetData(hBlock);

  CAA_SET_RESULT(peError, CMD_NO_ERROR);

  /* … */

  if (xxxDriverContext[byDriver].hBlock != CAA_hINVALID) /* chip is busy */
  {
      return CMD_SND_NOT_OK;
  }


  /* check TTL for CANopenSafety SIL2 */
  if (pBlock->dwTSP != 0)
  {
      /* check TTL */
      RTS_SYSTIME time;
      CAA_DWORD dwTime, dwTTL;

      CAL_SysTimeGetUs(&time);
      dwTime = (CAA_DWORD)time;
      dwTTL = pBlock->dwTSP;

      if ((dwTTL < dwTime && (dwTime - dwTTL) < 0x80000000) ||
          (dwTTL > dwTime && (dwTTL - dwTime) >= 0x80000000))
      {
          /* dwTTL < dwTime ==> message is not valid anymore ==> discard it */
          CAA_SET_RESULT(peError, CMD_TTL_ERROR);
          return CMD_SND_NOT_OK;
      }
  }

  /* copy message to chip */
  /* CAN ID: pBlock->cobId.X.ID*/
  /* RTR bit: pBlock->cobId.X.RTR */
  /* 29 bit: pBlock->cobId.X.EID */
  /* Data length: pBlock->byLen.X.DLC */
  /* Data pointer: &pBlock->byData[0] */
  /* store the actual block for TX-IRQ handling */
  xxxDriverContext[byDriver].hBlock = hBlock;

  return CMD_SND_OK;
}
```

```
/* Transmit Interrupt */
if (/* Tx-IRQ */)
{
   hBlock = xxxDriverContext[byDriver].hBlock;
   xxxDriverContext[byDriver].hBlock = CAA_hINVALID;
   pBlock = MBM_MsgGetData(hBlock);
   pBlock->dwTSP = 0; /* Set Timestamp to current time */
   pInfo->ctMessagesSend++;
   CL2_MsgSendAckn(byNet, hBlock);
}

/* Rx Interrupt */
if (/* Rx-IRQ */)
   …
```

2.1.4.2 Drivers without TxIRQ

Drivers without Tx IRQ have to call MsgSendAcknNoIRQ directly from CMD_Send if the message was set successfully (➔ CMD_Send returns CMD_SND_OK and no error). If Tx Queue contains remaining messages, then CL2 triggers further CMD_Send calls until all messages are processed or CMD_Send signals that no further messages can be processed (return value: CMD_SND_NOT_OK). In this case, CL2 will stop sending messages until CL2_MsgSendAcknNoIRQ with CAA_hINVALID is called from driver. Then the driver signals that sending may be working again.
Note: In CMD_Send implementation, you can call CL2.GetTxQueueLength for retrieving the remaining number of Tx messages. This may be useful if the driver wants to collect some messages first before sending it to the chip (e.g. useful for gateway implementations where messages are copied to Ethernet frames).

```
static CAA_BOOL CMD_Send(CL2I_BYTE byNet, CAA_HANDLE hBlock, CL2I_BYTE byPrio,
CAA_ERROR* peError)
{
  CAA_BYTE byDriver = s_byDriver[byNet];
  CL2I_INFO* pInfo = s_pInfo[byDriver];
  CL2I_BLOCK* pBlock = (CL2I_BLOCK*)CAL_CL2_MsgGetData(hBlock);

  CAA_SET_RESULT(peError, CMD_NO_ERROR);

  /* … */

  /* check TTL for SIL2 */
  if (pBlock->dwTSP != 0)
  {
      /* check TTL */
      RTS_SYSTIME time;
      CAA_DWORD dwTime, dwTTL;

      CAL_SysTimeGetUs(&time);
      dwTime = (CAA_DWORD)time;
      dwTTL = pBlock->dwTSP;

      if ((dwTTL < dwTime && (dwTime - dwTTL) < 0x80000000) ||
      (dwTTL > dwTime && (dwTTL - dwTime) >= 0x80000000))
      {
          /* dwTTL < dwTime ==> message is not valid anymore ==> discard it */
          CAA_SET_RESULT(peError, CMD_TTL_ERROR);
          return CMD_SND_NOT_OK;
      }
  }
```

```
        /* copy message to chip */
        /* CAN ID: pBlock->cobId.X.ID*/
        /* RTR bit: pBlock->cobId.X.RTR */
        /* 29 bit: pBlock->cobId.X.EID */
        /* Data length: pBlock->byLen.X.DLC */
        /* Data pointer: &pBlock->byData[0] */
        error = CAN_Write(msg);

        if (error)
        {
            xxxDriverContext[byDriver].bTxError = CAA_TRUE;
            /* We have to call MsgSendAcknNoIRQ with CAA_hINVALID later on!*/

            return CMD_SND_NOT_OK;
        }
        else
        {
            pInfo->ctMessagesSend++;
            CAL_CL2_MsgSendAcknNoIRQ(byNet, hBlock);
            return CMD_SND_OK;
        }
    }

    /* the function which polls the chip for received messages */
    static void xxxRxHandler(void)
    {
        for (each CAN Driver context)
        {
            /* process Rx messages */
            /* …  */

            if (xxxDriverContext[byDriver].bTxError)
            {
               /* In case of a tx error, we try to empty the Tx Queue here.
                * This is just an example. MsgSendAckn may be also called from another
                * method. E.g. if the hardware offers the possibility getting an event
                * when sending becomes possible again.
                */
               xxxDriverContext[byDriver].bTxError = CAA_FALSE;
               CAL_CL2_MsgSendAcknNoIRQ(s_byNet[byDriver], CAA_hINVALID);
            }
         }
    }
```

2.1.5 Receiving Messages

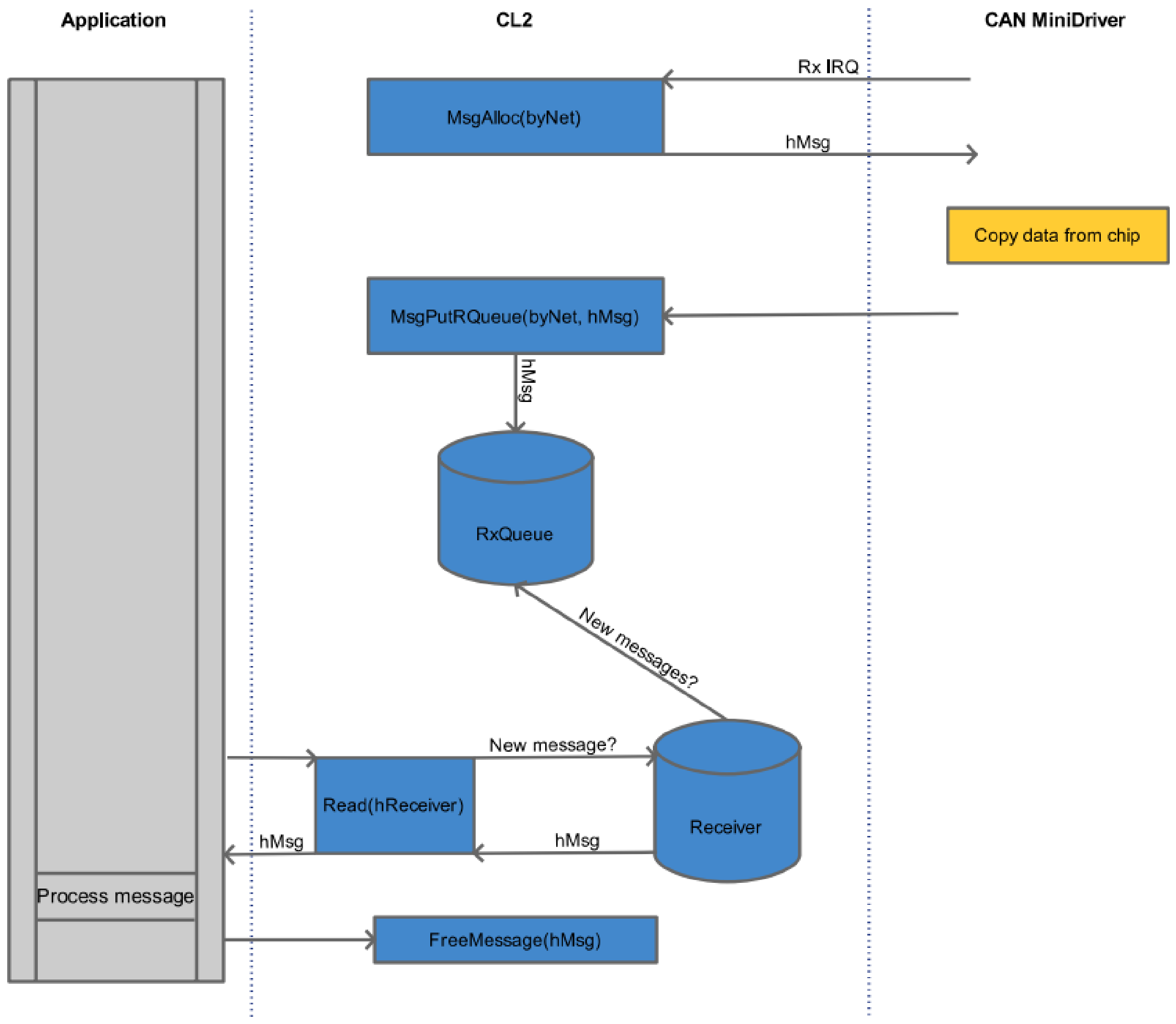


Figure 10: Message receiving

When a driver receives a message from chip, it allocates an Rx Handle from CL2 by calling MsgAlloc for the specific network. Then the driver copies all message data from chip (usually done in a function called CMD_Receive) and passes the message to CL2 by calling MsgPutRQueue. If someone is interested in this message, CL2 will send it to the Rx Queue.
Note: The interface function CMD_Receive is not used by CL2. It may be used internally by the CANMiniDriver for copying the message data, but it can be also left blank.

Example:
Rx IRQ Handler or Rx receive task:

```
if (new message received)
{
   hBlock = CAL_CL2_MsgAlloc(s_byNet[byDriver], CAA_pNULL);
   CL2I_BLOCK* pBlock = (CL2I_BLOCK*)CAL_CL2_MsgGetData(hBlock);

   if (pBlock)
   {
        /* copy message data */
        pBlock->byLen.B = byLen;
        pBlock->byNet = byNet;
        pBlock->dwTSP = CAL_SysTimeGetMs();
        pBlock->cobId.D = dwId;
        pBlock->cobId.X.EID = xType;
        pBlock->cobId.X.RTR = xRtr;

        if (message copy successful)
        {
           s_pInfo[s_byDriver[byNet]]->ctMessagesReceived++;
           CAL_CL2_MsgPutRQueue(s_byNet[byDriver], hBlock);
        }
        else
        {
           CAL_CL2_MsgFree(hBlock);
           s_pInfo[s_byDriver[byNet]]->ctDataOverruns++;
        }
   }
   else
   {
        s_pInfo[s_byDriver[byNet]]->ctDataOverruns++;
   }
}
```

2.1.6 CMD_Dispose: Driver Close

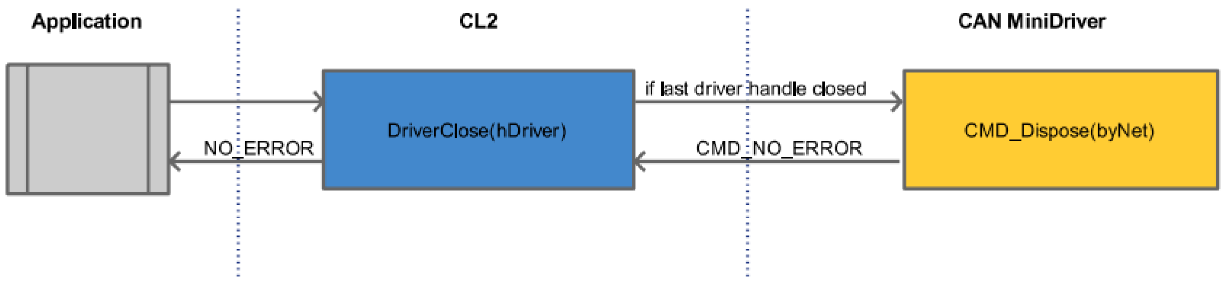


Figure 11: Driver close

If the last driver handle for a specific network is closed by the application, CL2 calls CMD_Dispose and the driver has to deinitialize the chip.

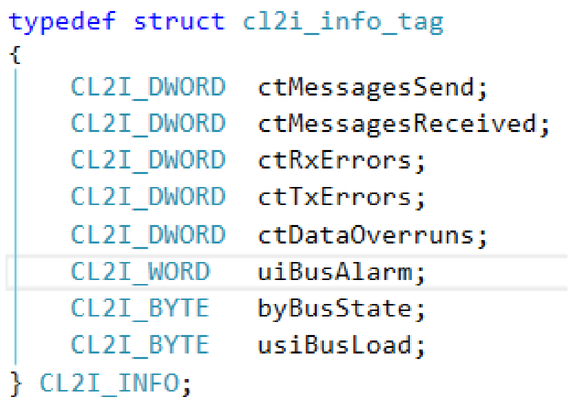
```
static CAA_ERROR CMD_Dispose(CL2I_BYTE byNet)
{
     /* deinitialize chip */
}
```

2.2 Optional Functions

The following chapter describes all optional functions of a CANMiniDriver.

2.2.1 Diagnosis

Bus diagnosis is an optional functionality. In general, a driver will also work without implementing it. But it is highly recommended to implement it. With diagnosis information about the CAN bus, a CAN application (e.g. CANopen stack) can react to bus errors and give diagnosis feedback to the user.
If a driver registers at CL2, then a pointer to a CL2I_Info structure is returned. This structure contains diagnosis counters and other diagnosis information, such as bus state.



```
typedef struct cl2i_info_tag
{
    CL2I_DWORD  ctMessagesSend;
    CL2I_DWORD  ctMessagesReceived;
    CL2I_DWORD  ctRxErrors;
    CL2I_DWORD  ctTxErrors;
    CL2I_DWORD  ctDataOverruns;
    CL2I_WORD   uiBusAlarm;
    CL2I_BYTE   byBusState;
    CL2I_BYTE   usiBusLoad;
} CL2I_INFO;
```

Figure 12: Diagnosis structure

2.2.1.1 Diagnostic Counter

The following diagnostic counters are defined (see also Figure 12: Diagnosis structure):

- **ctMessagesSend:** Should be incremented when a message was successfully sent.
  Should be set to 0 when driver will be disposed.

  For drivers with Tx IRQ: Increment it in TX IRQ handler;
  For drivers without IRQ: Increment it in CMD_Send

- **ctMessagesReceived:** Should be incremented when a message was successfully received and passed to CL2 by calling MsgPutRQueue.

- **ctRxErrors:** Should be set to the value of the CAN chip’s Rx Error register.

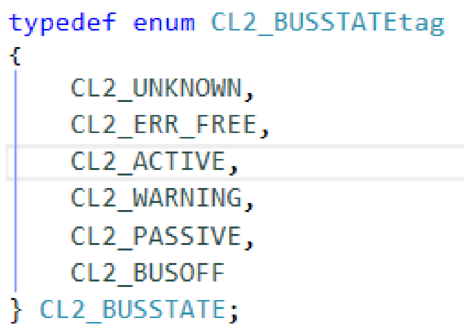
- **ctTxErrors:** Should be set to the value of the CAN chip’s Tx Error register.

- **ctDataOverruns:** Number of dismissed receive messages.
  Should be incremented when message was received by chip, but no message handle could be allocated and if the chip signals a data overrun situation (if supported).

- **usiBusLoad:** current bus load in percent.
  Currently not implemented on drivers provided by CODESYS.

2.2.1.2 Bus State

Another very important component of the CL2I_INFO structure is the variable byBusState. It holds the current state of the associated CANbus. The following values can be set by the CANMiniDriver implementation:



```
typedef enum CL2_BUSSTATEtag
{
    CL2_UNKNOWN,
    CL2_ERR_FREE,
    CL2_ACTIVE,
    CL2_WARNING,
    CL2_PASSIVE,
    CL2_BUSOFF
} CL2_BUSSTATE;
```

Figure 13: Bus State

- **UNKNOWN**
  The state of the network is not known or the bus state is not implemented by driver.

- **ERR_FREE**
  No occurrence of CANbus errors so far. The error counters of the chip are zero.

- **ACTIVE**
  Only a few CANbus errors so far. The error counters of the chip are below the warning level.

- **WARNING**
  Occurrence of some CANbus errors. The error counters are above the warning level.

- **PASSIVE**
  Too many CANbus errors. The error counters are above the error level.

- **BUSOFF**
  The node has been separated from the CANbus. The error counter has exceeded the permitted maximum.

2.2.1.3 CMD_ResetAlarm: Bus Alarm Handling

A CANMiniDriver should detect bus alarm events such as Bus Off. A bus alarm is an error situation which cannot be resolved without reinitializing the chip. A CANMiniDriver signals a bus alarm to the higher software layers by incrementing uiBusAlarm variable of CL2I_INFO structure. An application calls CL2.GetBusAlarm which returns the result of the Boolean expression: uiBusAlarm > 0.
An application can try to fix the bus alarm by calling CL2.ResetBusAlarm which leads to a call of CMD_ResetAlarm. The CANMiniDriver should initialize the CAN chip to normal operating mode. If a bus alarm is fixed, then uiBusAlarm should be set to 0 by CANMiniDriver.
Note:

- A CANopenStack calls ResetAlarm each bus cycle as long as GetBusAlarm returns true. This has to be considered when implementing ResetAlarm.
- If a CANMiniDriver supports the detection of bus alarms and CMD_ResetAlarm is implemented, then set CMD_SUPPORT_BUSALARM flag in CMD_GetInfo.
  Otherwise, CMD_ResetAlarm has to return CMD_NOT_IMPLEMENTED.

Example:
This is an example of a driver without IRQ. There are many different ways to implement bus alarm handling. This is just one of many possible solutions.

```
static CAA_ERROR CMD_ResetAlarm(CL2I_BYTE byNet)
{
  CAA_BYTE byDriver = s_byDriver[byNet];
  CL2I_INFO* pInfo = s_pInfo[byDriver];
  XXX_INFO pDriver = &xxxDriverContext[byDriver]

  if(!pInfo)
  {
      return CMD_SETUP_ERROR;
  }

  /* This driver  */
  xxxDriverContext[byDriver].reset_current = CAL_SysTimeGetMs();

  if (pDriver->reset_last != 0 &&
     (pDriver->reset_current - pDriver->reset_last < 1000) )
  {
      return CMD_NO_ERROR;
  }

  if (pInfo->uiBusAlarm > 0)
  {
      pDriver->reset_last = pDriver->reset_current;
      /* This is just an example for getting a driver into normal operating mode */
      CMD_Dispose(byNet);
      CMD_Init(byNet, pDriver->wBaudrate);
      /*
         For drivers without IRQ: Empty Tx Queue after bus alarm.
         For IRQ driver: Call MsgSendAcknIRQ in IRQ Handler when bus error is fixed.

      */
      CAL_CL2_MsgSendAcknNoIRQ(byNet, CAA_hINVALID);
  }

  pInfo->uiBusAlarm = 0;
  return CMD_NO_ERROR;

}
```

2.2.1.4 CMD_SetStatus: LEDs

A CANMiniDriver may implement LEDs according to CiA 303-3 indicator specification. This can be signaled in CMD_GetInfo by setting the CMD_SUPPORT_STATUSLED flag.
If CL2 detects a LED state change, then CL2 calls CMD_SetStatus:

```
static CAA_ERROR CMD_SetStatus(CL2I_BYTE byNet, CL2I_BYTE byIndicator)
```

byIndicator contains information for two LEDs:

```
/* CiA 303-3 ERR Led (Red) */
#define CMD_INDICATOR_ERR_OFF 0x00
#define CMD_INDICATOR_ERR_ON  0x01
#define CMD_INDICATOR_ERR_FLK 0x02 /* flickering */
#define CMD_INDICATOR_ERR_BLK 0x03 /* blinking */
#define CMD_INDICATOR_ERR_SFL 0x04 /* single flash */
#define CMD_INDICATOR_ERR_DFL 0x05 /* double flash */
#define CMD_INDICATOR_ERR_TFL 0x06 /* triple flash */
#define CMD_INDICATOR_ERR_QFL 0x07 /* quadruple flash */

/* CiA 303-3 RUN Led (Green) */
#define CMD_INDICATOR_RUN_OFF 0x00
#define CMD_INDICATOR_RUN_ON  0x10
#define CMD_INDICATOR_RUN_FLK 0x20 /* flickering */
#define CMD_INDICATOR_RUN_BLK 0x30 /* blinking */
#define CMD_INDICATOR_RUN_SFL 0x40 /* single flash */
#define CMD_INDICATOR_RUN_DFL 0x50 /* double flash */
#define CMD_INDICATOR_RUN_TFL 0x60 /* triple flash */
#define CMD_INDICATOR_RUN_QFL 0x70 /* quadruple flash */
```

Figure 14: LED flags

2.2.2 CMD_SetBlock, CMD_SetCycle

A CANMiniDriver may support the cyclic sending of CAN messages. Therefore, CL2 provides two functions: one for setting a specific transmit message handle (SetBlock) and one for activating cyclic sending for this handle (SetCycle). These two function calls are redirected to the corresponding CANMiniDriver implementation.
This feature can be used for CANopenStack to get a better jitter for Sync messages (see the next section).
If a CANMiniDriver supports this function, then CMD_GetInfo has to return the CMD_SUPPORT_SETCYCLE flag and the number of supported messages (for byMaxCycleIndex, see also 2.1.1).

```
static CAA_HANDLE CMD_SetBlock(CL2I_BYTE byNet, CL2I_BYTE byIndex, CAA_HANDLE hBlock, CAA_ERROR* peError)
```

byIndex: It is possible to register several transmit messages. They are identified by an index starting with 0.
hBlock: The message handle.
peError: Optional pointer to an error enumeration. Return CMD_NOT_IMPLEMENTED if function is not supported.

```
static CAA_ERROR CMD_SetCycle(CL2I_BYTE byNet, CL2I_BYTE byIndex, CL2I_DWORD dwCycle)
```

byIndex: Index of message
dwCycle: cycle time for message in microseconds.

Return CMD_NOT_IMPLEMENTED if function is not supported.

2.2.2.1 Usage of CMD_SetBlock, CMD_SetCycle for CANopen Sync messages

By default, CANopen Sync messages are sent cyclically from the IEC bus cycle task. This means that, under poor circumstances, the CANopen Sync messages will have a jitter that is equal to the jitter of this cyclically called task.

In order to generate a more accurate sync signal, it is possible to generate those packets externally from a hardware timer and to trigger the CAN task from this timer.

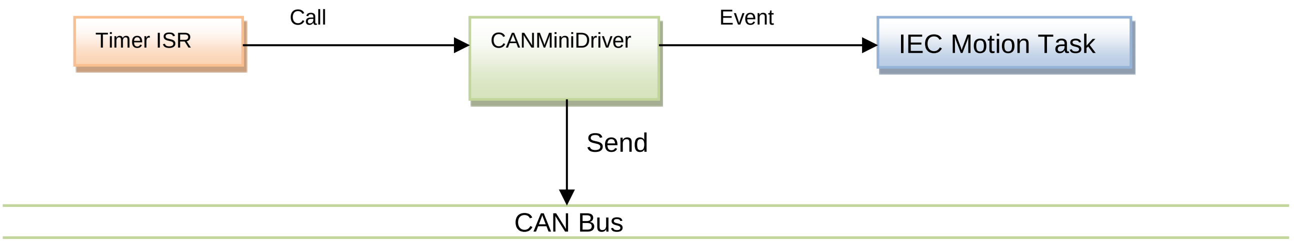


Figure 15: External Sync Overview

When sync is enabled in a CANopen project, the 3S CANopen stack attempts to enable the external sync mechanism of the CAN driver. This will be successful if the driver implements the following two functions of the CAN L2 driver API:

- CMD_SetBlock()
- CMD_SetCycle()

The function CMD_SetBlock() is called to pass the sync message to the driver. CMD_SetBlock() receives a block handle from a message block that contains the sync message. This function takes control of the passed block handle. This control is given back to the calling function at the next call of CMD_SetBlock().

The function CMD_SetCycle() is called by CL2 to define the cycle time in which a sync packet will be generated. This time value is given in microseconds and should be used to program a hardware timer.

Within the timer ISR, the CAN driver needs to send the packet to the CAN. After a send interrupt has signaled that the sync packet was sent, the driver needs to send an event to the motion task.

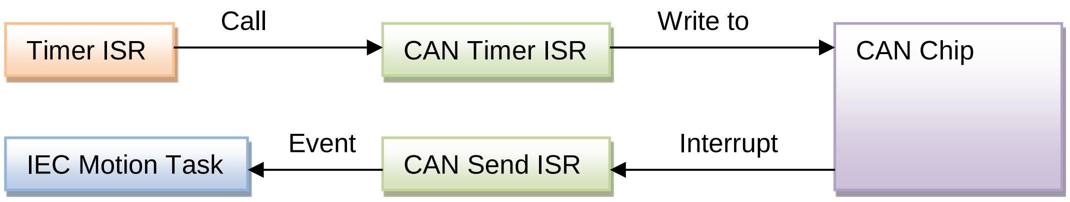


Figure 16: Timer ISR

CAN Timer ISR example:

```
s_pClonedSyncBlock = CAL_CL2_MsgClone( CanNet, pSavedSyncBlock, &error );

if(error == CL2_NO_ERROR)
{
   CMD_Send( CanNet, s_pClonedSyncBlock, 0, 0);
}
```

CAN Send ISR:

```
if(s_pSyncBlockCloned == hBlock)
{
   /* Wake up the Motion Task */
   if((s_hEventCanSync != RTS_INVALID_HANDLE))
   {
      CAL_SysEventSet(s_hEventCanSync);
   }

 }
```

When you are using a target with external sync, you are creating a motion task which is triggered by an external event. But this also means that you will not define a cycle time for this task. This time will be implicitly defined with the external sync period.

However, because the SoftMotion stack needs to know the cycle time of the task on which it is running, you need to set this time within your CAN driver manually. This can be done directly in the function CMD_SetCycle() because you get the sync period in this function.

To get the task handle of the motion task, as well as the sync event, you need to register on the event "TaskCreateDone" of "CmpSchedule". You should search for your event name and save a handle to this event, as well as to the task handle, in a static variable of the driver.

Example of CMD_SetCycle():

```
/* set cycle time of IEC task */
if(s_hTaskCanSyncInfo != NULL)
s_hTaskCanSyncInfo->tInterval = dwCycle;
/* program hardware timer */
…
```

2.3 Non-implemented functions

The functions CMD_Identify and CMD_SetMask are currently not used by CL2.
Return CMD_NOT_IMPLEMENTED.

3 Important Runtime Defines

There are some important runtime defines that can be adapted to optimize the memory usage and behavior of a CAN runtime system (see CAADefinesGeneric.h).

CL2_NNET: Maximum number of CAN interfaces. If your runtime has a maximum number of 2 CAN interfaces and there should be no support for other CAN drivers, then set this define to 2 to optimize the memory footprint.

CL2_NRXMSG: Default number of initially allocated receive messages per driver instance. Will be used if driver calls CmdRegister with ctMessages = 0.

CL2_QUEUE_ADAPTION: If set to true, then the number of receive messages will automatically extended if receive message handles are running short. It is recommended to activate this feature. Otherwise there could be lost receive messages.

CL2_NPRIO: Number of message priorities supported by CL2.

CL2_SZRIDP: Number of pre-allocated bytes for CL2 receivers. Memory will be extended dynamically if the user creates more receivers.

CL2_CALLBACK: Enables the callback support of CL2. You can disable this for saving memory as it is not used by our CAN-based stacks.

4 Conformance Test

The following section describes test procedures for CANMiniDrivers.

4.1 Overview

There are two test procedures available: a semi-automatic CANMiniDriver Conformance Test and a manual CAN Echo test.

Q: Which test should I use?
A: The semi-automatic test needs a Peak CAN Interface (e.g. PCAN USB dongle) for testing. If you have this kind of interface (e.g. PCAN USB dongle), then we recommend using the automatic test.
Otherwise you can use the old-style CAN Echo Test. Therefore, only a CAN Analyzer with message sending functionality is required.

4.2 CANMiniDriver Conformance Test (automatic test)

4.2.1 Overview

The CANMiniDriver Conformance Test consists of two applications: a test server application that runs on the device under test (DUT) and a client application designed for a CODESYS Control Win V3 with PCAN interface.

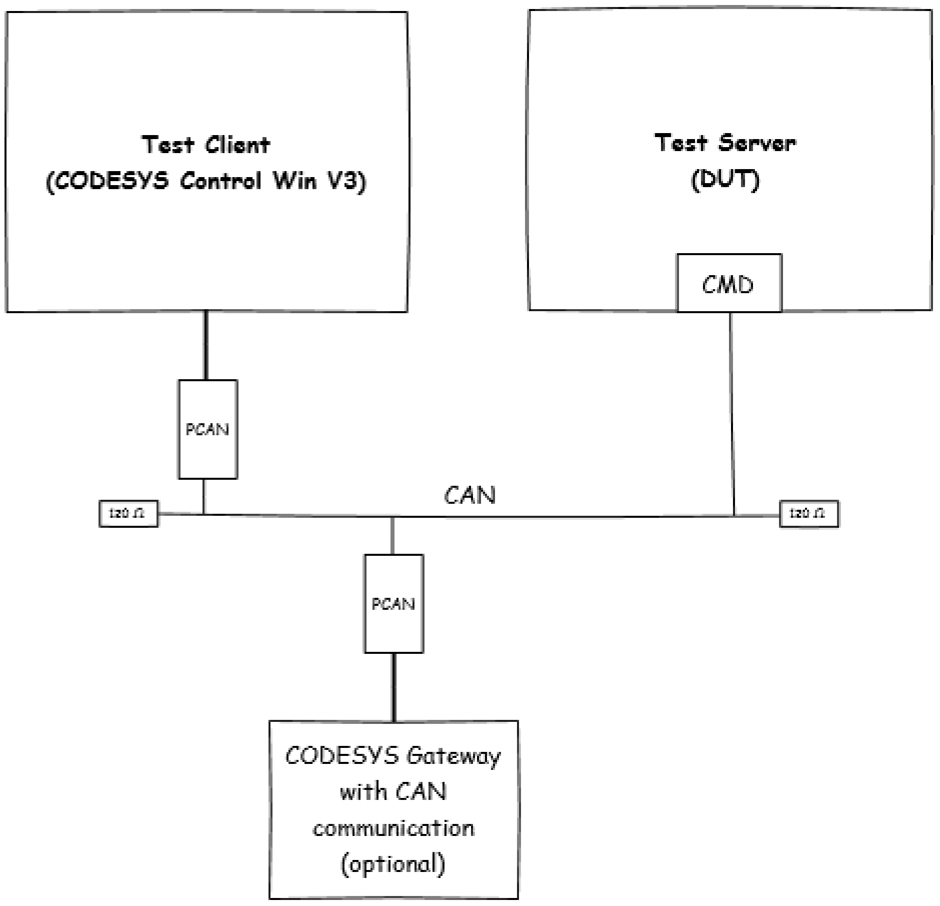


Figure 17: Test setup

The test server is controlled by test control messages on CAN-ID 0 sent by the test client.
The test server application performs the following tasks:

- It echoes all CAN messages with an incremented CAN-ID except of following IDs:
  16#0 (Test Control message IDs)
  16#80-16#FF (CANopen Emergency IDs)
  16#580-16#67F (CANopen SDO IDs)
  Note: CAN IDs used for Blockdriver communication are not echoed. Therefore, block driver communication can be used in parallel.
- It checks the message data consistency: All received message must have a sequence counter in the first byte (if DLC is greater 0). The sequence counter is incremented with every message.
  All other bytes contain the incremented value of their predecessor.
  Example of a valid message sequence:
  ID 0x30, DLC=8: **00** 01 02 03 04 05 06 07
  ID 0x700, DLC=1: **01**
  ID 0x703, DLC=0; (Sequence counter = 2)
  ID 0x20; DLC=3; **03** 04 05
- It logs error information to the PLC logger.
- It provides an integrated visualization with CAN Hardware configuration settings and diagnostic information.

The test client application performs the following tasks:

- It provides a visualization for test configuration.
- It generates bus traffic over a PCAN interface and checks the echoed messages received from the test device:
  - Are the sequence counters in the correct order?
  - Is each message information echoed correctly (Data, DLC, RTR, EID,…)?
  - Are all messages echoed? (➔ Tx count should be equal to Rx count).
- It shows the test result in the visualization.

4.2.2 Requirements

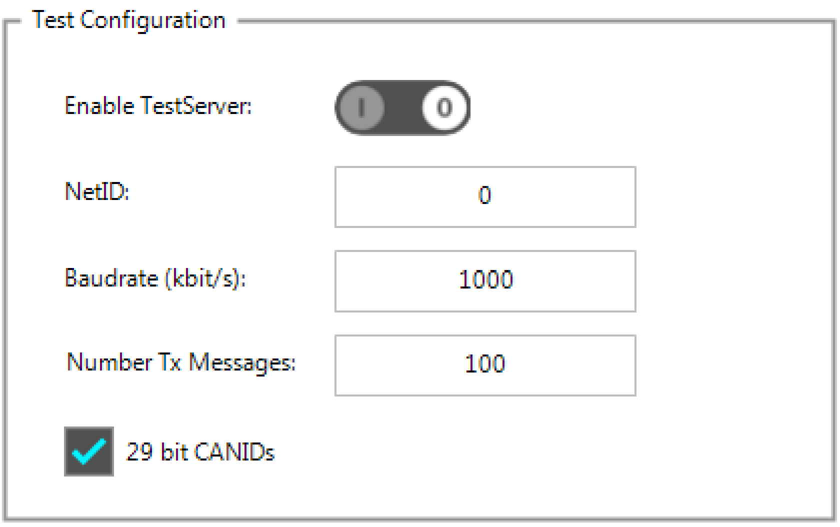
- CODESYS 3.5 SP9 or higher
- CODESYS Control Win V3 with Peak CAN interface (e.g. Peak PCAN USB Dongle)
- CMDConformanceTest project (included in runtime delivery: Templates/CANMiniDriverConformanceTest)

4.2.3 Test procedure

Open CMDConformance project (Templates/CANMiniDriverConformanceTest/CMDConformanceTest.project) and press "Update all" if Project Environment dialog opens.

4.2.3.1 Preparations for the test server

- Update DeviceToTest Device to your PLC device description.
- Download `DeviceToTest.Application` to your test device. Start the application.
- Open the integrated visualization (➔ double-click Visualization in the device tree).


Test Configuration

Enable TestServer: 0

NetID: 0

Baudrate (kbit/s): 1000

Number Tx Messages: 100

✓ 29 bit CANIDs

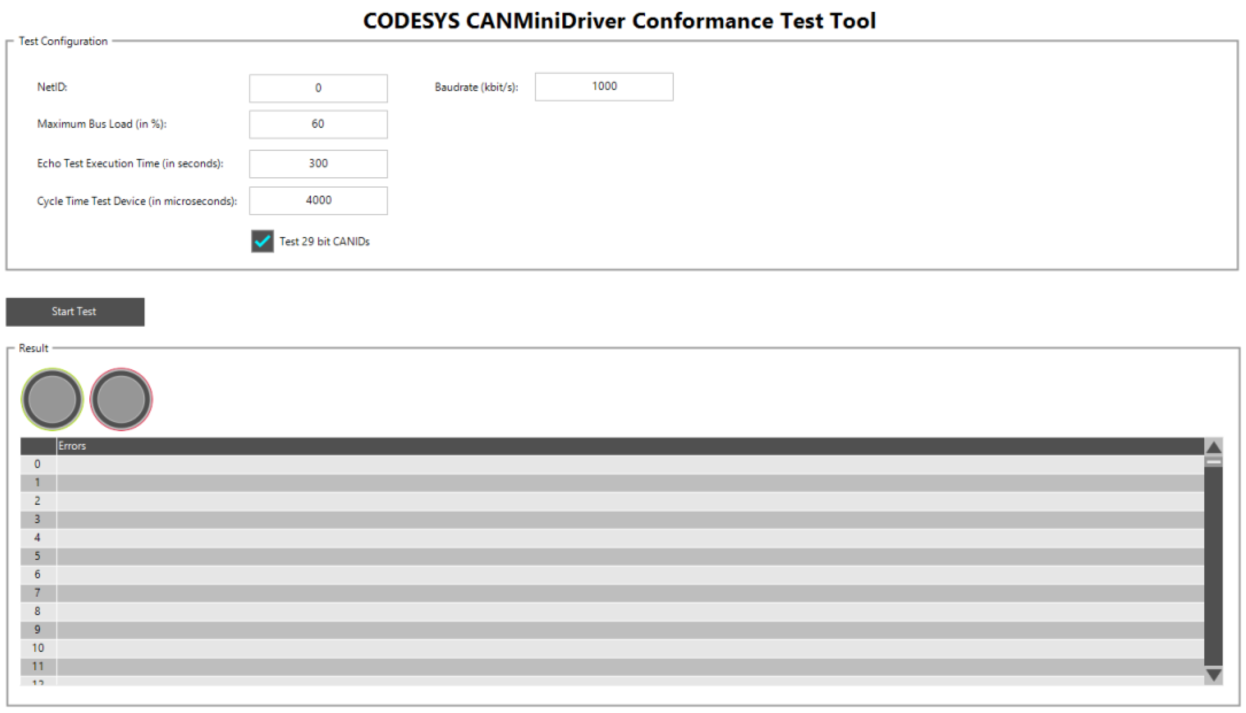
Figure 18: Test configuration

- Set the following settings:
  - NetID of the CAN interface, e.g. 0
  - Baud rate (recommended: 1000)
  - Number of Tx Messages (recommended: 100)
  - 29 bit CANID: Does your driver support 29-bit CANIDs (e.g. needed for J1939)
- Switch on "Enable Testserver"
  - The driver will be opened by calling `CMD_Init` and the visualization shows all diagnostic counters.
  - If an error occurs (no diagnostic counters will be shown), then check your test configuration. If NetID was correct, then check the implementation of `CMD_Setup` (normally called in `CH_INIT3` hook) and `CMD_Init` (called on `Cl2.DriverOpenH`).

4.2.3.2 Preparations for the test client

- Install the PCAN USB interface driver and activate the `CmpPCANBasicDrv` runtime system component in your CODESYS Control Win V3 installation. For more information and detailed instructions, see CODESYS Online Help topic "Inserting a PCAN USB adapter".
- Connect the PCAN interface to the test server (test setup see chapter 4.2.1).
- Update Device "Tester" to the corresponding CODESYS Control Win V3 device.

- Download `Tester.Application` to your CODESYS Control Win V3. Start the application. The visualization opens:


CODESYS CANMiniDriver Conformance Test Tool

Test Configuration

NetID:	0
Maximum Bus Load (in %):	60
Echo Test Execution Time (in seconds):	300
Cycle Time Test Device (in microseconds):	4000
Baudrate (kbit/s):	1000

✓ Test 29 bit CANIDs

Figure 19: Conformance Test

4.2.3.3 Automatic test execution

- Setup the Test Client:
  - Enter the NetID of PCAN interface (default: 0; see PLC logger).
  - Enter the same baud rate as configured on test server.
  - Enter maximal bus load (Recommended: 50-70%).
  - Enter "Echo test execution time" (default: 300 seconds).
  - Enter the test server cycle time (default: 20000us = 20 ms).
  - Select 29-bit CAN identifiers test if supported by MiniDriver.
- IF CODESYS communication over CAN is used on the same CAN Bus, then it is recommended to log out from application to minimize bus traffic.
- Press Start Test
  - The following tests are performed:
    - Message Test:
      Different kind of messages are sent to the Test Server. It will be checked if all information is echoed correctly.
      - RTR messages
      - 11-bit and 29-bit CANIDs
      - Different DLCs
    - Echo Test:
      The bus load is generated and echoed messages will be checked for sequence and data errors. Furthermore, the message roundtrip time is measured.

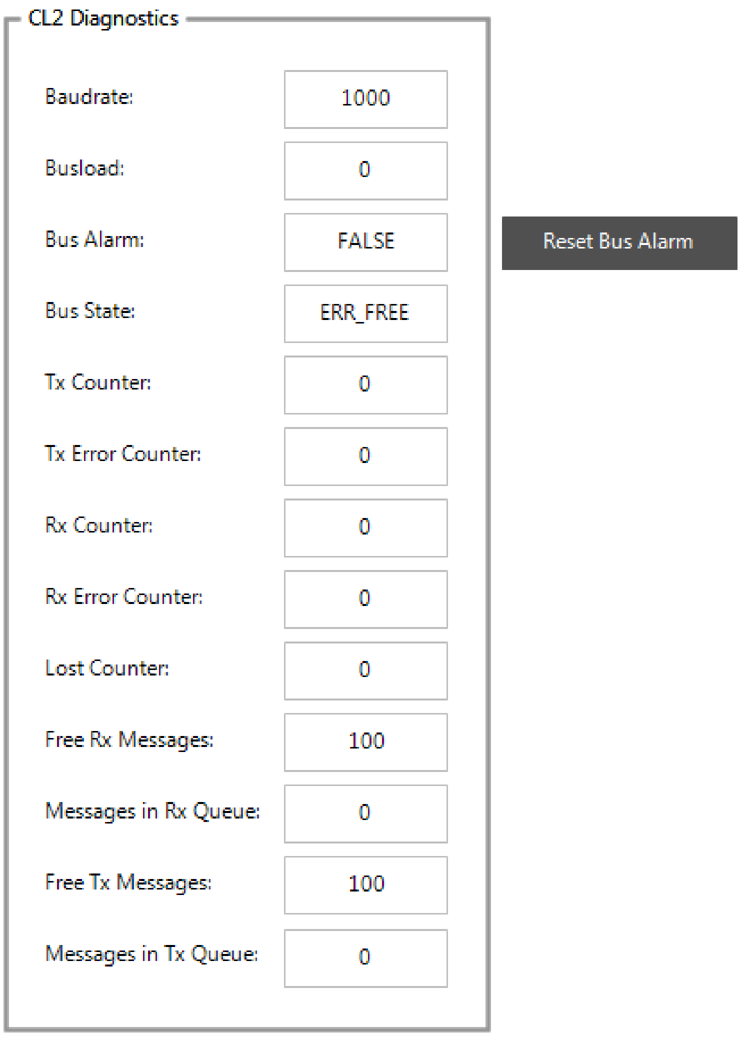
Errors will be shown in the error table of the visualization. Additional information can also be found in the PLC log of the Test Server and integrated visualization (diagnostic counters).
If test was successful, then the green LED illuminates in the Result Group Box.

If an error occurs in Message test, then the test prints the corresponding CAN message in the error table.
You can reproduce this error manually by sending this message with a CAN Analyzer and checking the echoed message (CANID + 1, DLC, DATA, RTR, 29 bit, …). The echoed message should be exactly the same message but with an incremented CAN ID.
If an error occurs in Echo Test, it is useful to record a CAN trace, so you can analyze the message flow. CAN messages with CAN ID ending on 0 (e.g. 0x890) are always sent by the tester. Messages with CAN ID ending on 1 are sent by the tested device.

4.2.3.4 Bus diagnostic test

In addition to the automatic test, it is also required to test bus error detection and BusOff recovery:

- Open the integrated visualization of DeviceToTest.
- The group box "CL2 Diagnostics" shows all MiniDriver diagnostic counters:



CL2 Diagnostics

Baudrate:	1000
Busload:	0
Bus Alarm:	FALSE
Bus State:	ERR_FREE
Tx Counter:	0
Tx Error Counter:	0
Rx Counter:	0
Rx Error Counter:	0
Lost Counter:	0
Free Rx Messages:	100
Messages in Rx Queue:	0
Free Tx Messages:	100
Messages in Tx Queue:	0

Figure 20: Bus Diagnostics

- The Bus State should be ERR_FREE and Bus Alarm FALSE.
- Now start the test on Test Client side. Check following diagnostic information:
    - Tx Counter and Rx Counter are incremented.
    - Lost Counter is 0.
    - Tx Error and Rx Error Counter is 0.
    - Bus State remains ERR_FREE and Bus Alarm FALSE.
- Now generate a bus failure: Short circuit CAN Low and CAN High some seconds:

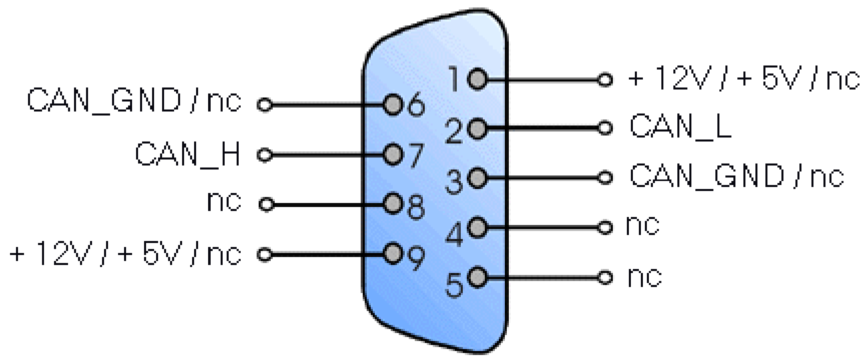


Figure 21: CAN Sub-D

- The Bus State signals a bus error – normally bus off. In bus off state, most CAN chips need a reset command. This will be signaled by Bus Alarm = TRUE. As long as Bus Alarm is TRUE, no messages will be sent or received (Tx an Rx Error Counter remain values).
  Now the application has to trigger a Reset Bus Alarm command which leads to a CMD_ResetAlarm call. In this function, the MiniDriver has to reset the chip.
  You can test this functionality by pressing the "Reset Bus Alarm" button.
  Then check the following:
  - Bus Alarm should change to FALSE.
  - Bus State is not equal to BUS_OFF.
  - RxCounter and TxCounter will be incremented again (if Test Client sends still messages). ➔ Driver sends and receives messages again ➔ Reset Bus Alarm implementation is correct.

4.3 CANEcho test (manual test)

4.3.1 Overview

If you have no Peak CAN interface, then you can use the manual CANEcho test. This test can also be used for testing more than one interface at the same time.
The test project echoes all CAN messages with an incremented CAN ID on the bus. Furthermore, it implements a message latency measurement.

4.3.2 Requirements

- CODESYS 3.5 SP9 or later
- CAN Analyzer with messages sending functionality
- CANEcho project (included in runtime delivery: Templates/CANMiniDriverConformanceTest)

4.3.3 Test procedure

4.3.3.1 Preparations

- Plug a CAN Analyzer into the CANbus of your PLC.
- Open CAN Echo project (Templates/CANMiniDriverConformanceTest/CANEcho.project), update PLC device to you device and download it.
- Start the application and open the integrated "Visualization".
- Enter NetID, baud rate, Number of Tx messages, and 29-bit capability. Enable the TestServer.

4.3.3.2 Message content test

- With CAN Analyzer, send the following messages and check if they are echoed correctly (same DLC, DATA, RTR…) by your PLC:
  - Message with 8 Bytes.
  - Message with 4 Bytes.
  - Message with 0 Bytes.
  - RTR Message with 0 Bytes.
  - 29 Bit CANID Message (if supported).

4.3.3.3 Message loss test and diagnosis counter

- Send several cyclic messages with CAN Analyzer.
- Let test run for a while (e.g. 5 minutes).
- Stop sending messages.
- Compare receive message number and transmit message count in CAN Analyzer.
  They have to be equal!
- Check diagnosis counter in integrated visualization. See documentation in CANL2 lib.

4.3.3.4 Message latency test

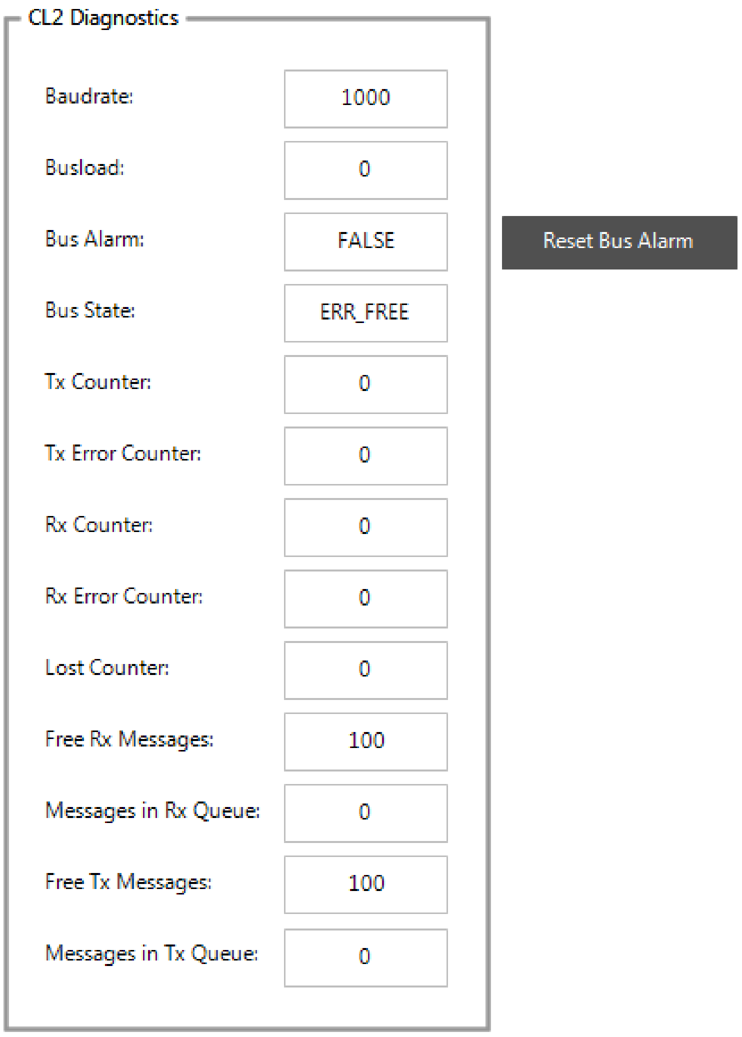
- In integrated visualization, enter a CANID in Message Latency Test. This CANID will be used for message latency test.
- Enable message latency test.
- Send a cyclic message on the configured CANID. E.g. every 100ms.
- In Visualization check:
  Minimum time span ➔ minimum time span between two CAN messages
  Maximum time span ➔ maximum time span between two CAN messages
  Should be:
  Minimum time span should be the message cycle time (e.g. 100ms)
  Maximum time span should be the message cycle time or message cycle time + 1 task cycle.

  Note: If the maximum time span is higher, then there is a jitter on message receiving. If this jitter is too high, then CANopen Heartbeat or Nodeguarding may not work (consequence: slaves will be reset).
- Repeat the test with higher bus load. Send additional fast cyclic messages and check the message jitter.

4.3.3.5 Bus diagnostic test

This section describes how to test bus error detection and BusOff recovery:

- Open the integrated visualization.
- The group box "CL2 Diagnostics" shows all MiniDriver diagnostic counters:



CL2 Diagnostics

Baudrate:	1000
Busload:	0
Bus Alarm:	FALSE
Bus State:	ERR_FREE
Tx Counter:	0
Tx Error Counter:	0
Rx Counter:	0
Rx Error Counter:	0
Lost Counter:	0
Free Rx Messages:	100
Messages in Rx Queue:	0
Free Tx Messages:	100
Messages in Tx Queue:	0

Figure 22: Bus diagnostics

- Bus State should be ERR_FREE and Bus Alarm FALSE.
- Now send cyclic messages with CAN Analyzer. Check following diagnostic information:
    - Tx Counter and Rx Counter are incremented.
    - Lost Counter is 0.
    - Tx Error and Rx Error Counter is 0.
    - Bus State remains ERR_FREE and Bus Alarm FALSE.
- Now generate a bus failure: Short circuit CAN Low and CAN High some seconds:

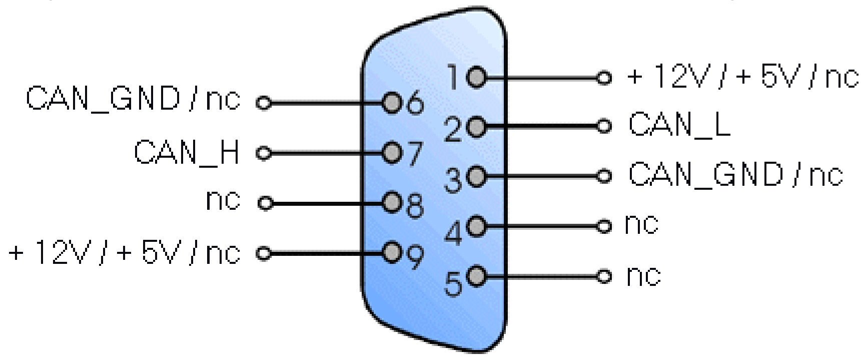


Figure 23: CAN Sub-D

- Bus State signals a bus error – normally bus off. In bus off state, most CAN chips need a reset command. This will be signaled by Bus Alarm = TRUE. As long as Bus Alarm is TRUE, no messages will be sent or received (Tx an Rx Error Counter remain values).
  Now the application has to trigger a Reset Bus Alarm command which leads to a CMD_ResetAlarm call. In this function the MiniDriver has to reset the chip.
  You can test this functionality by pressing the "Reset Bus Alarm" button.
  After that check the following:
  - Bus Alarm should change to FALSE
  - Bus State is unequal BUS_OFF
  - RxCounter and TxCounter will be incremented again ➔ Driver sends and receives messages again ➔ Reset Bus Alarm implementation is correct.

Change History

Version	Description	Author	Date
0.1	Creation	SB	07.07.2017
1.0	Maximal Busload added, CAN ID distribution Release	SB	22.02.2018