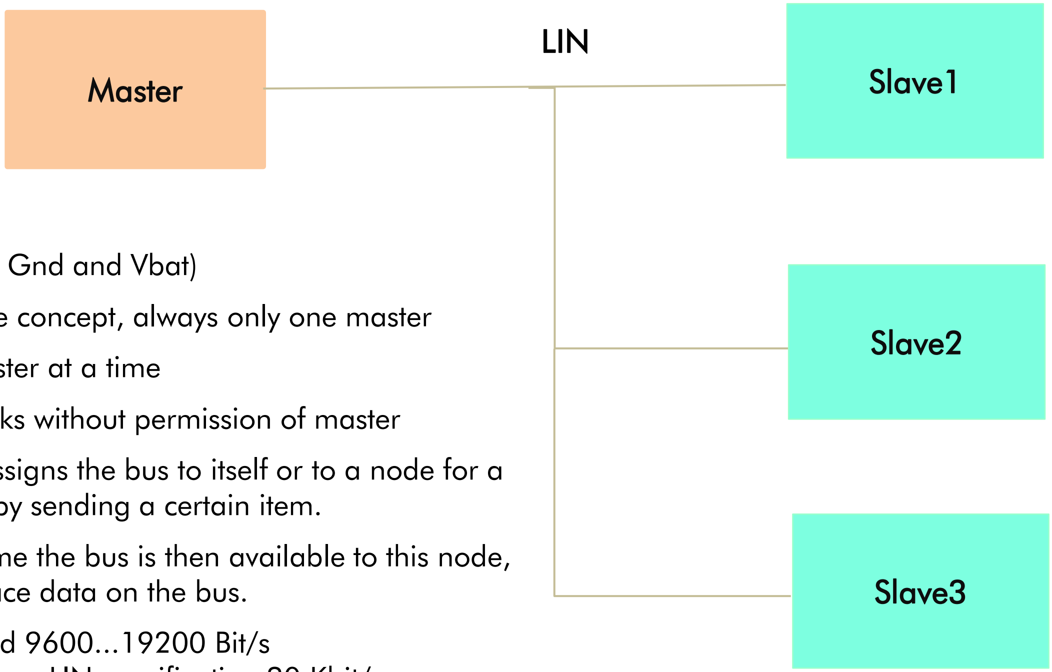
LIN-Basics

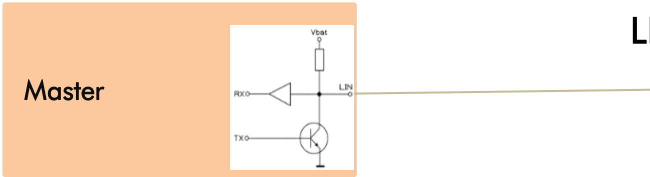
Lipowsky Industrie-Elektronik GmbH

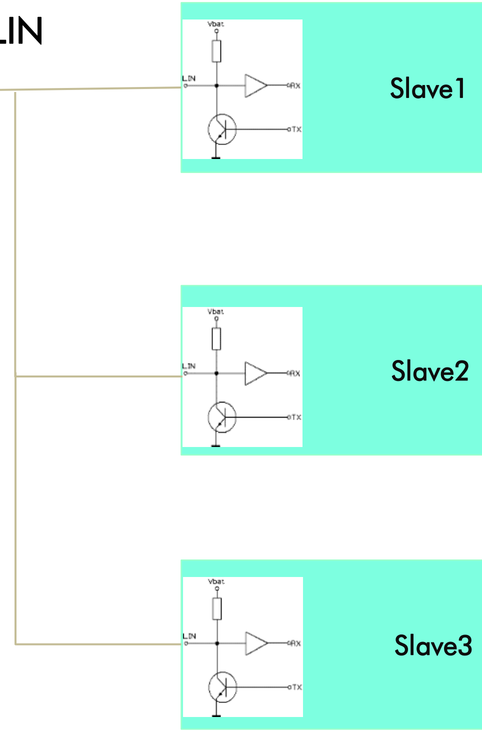
LIN-Basics



- 1 wire bus (+ Gnd and Vbat)
- Master / Slave concept, always only one master
- Only one master at a time
- No one speaks without permission of master
- The master assigns the bus to itself or to a node for a defined time by sending a certain item.
- During this time the bus is then available to this node, which can place data on the bus.
- Typ. bus speed 9600...19200 Bit/s
  Maximum as per LIN specification 20 Kbit/sec
- Often there are several LIN buses in one vehicle.
  Then the function for ambient lightning, buttons, climatic actuators, seat control are often to different busses.

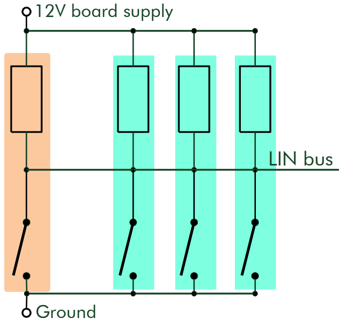
LIN Single Wire Hardware





- Bi-directional communication over one implemented by open-collector output stages.
- Each node has a switch (Tx), which can pull the bus line to GND.
- If all switches are open, the bus line is pulled high to plus (typ. 12 V) by a pull-up resistor.
- The necessary pull-up is distributed over the nodes:
  Master Pull-Up 1 KiloOhm
  Slave Pull-Up 30 KiloOhm
- Each node can read back the state of the LIN bus line (RX)

LIN-bus dominant / recessive states



A LIN bus with one master and 3 slaves can be reduced to the simplified circuit diagram shown on the left.

As soon as one of the nodes activates its output switch, the bus will have a low level, only if all output switches are open, the bus will be pulled up to its high level.

Since a single node is sufficient to determine the low level by closing its switch, this is called the **dominant level**.

Accordingly, the High level, for which all nodes must have their switches open, is called **recessive level**.

All pull-up resistors are connected in parallel, so the effective pull-up resistance value corresponds to the parallel connection of all pull-up resistors.

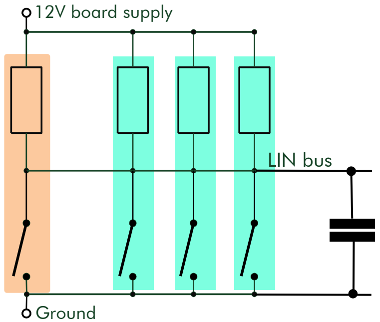
The LIN bus has only 2 states:

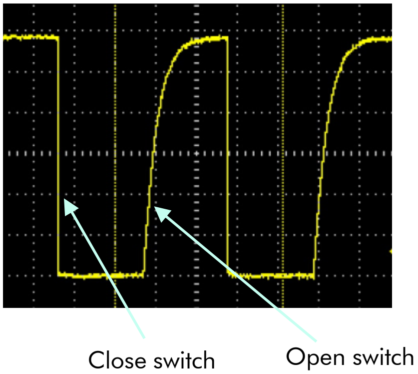
Recessive high state (all switches open)

Dominant low state (at least 1 switch closed)

All information that is transferred via the bus is coded by the chronological sequence of these two states.

LIN-Bus Signal Flanken





Recorded with deactivated Slope Control

For a better understanding of the effect of the pull-up on the signal edges, the equivalent circuit diagram is supplemented by a capacitance (capacitor).

This is formed by summing up the input capacitances of all LIN nodes and the line capacitance of the LIN bus line.

All pull-up resistors are connected in parallel so that the effective pull-up resistance value corresponds to the parallel connection of all pull-up resistors.

Since the line is actively pulled against GND when the dominant level is switched, the falling edge is correspondingly steep.

When the switches are opened, the discharged input capacitance must be reloaded to the high level via the pull-up.

The higher the pullup resistance is, the flatter the rising edge becomes.

Edges that are too steep can lead to EMC problems and edges that are too flat can lead to misinterpretation by the UART.

Therefore a correctly dimensioned pull-up resistor is very important!

LIN Bus Hardware

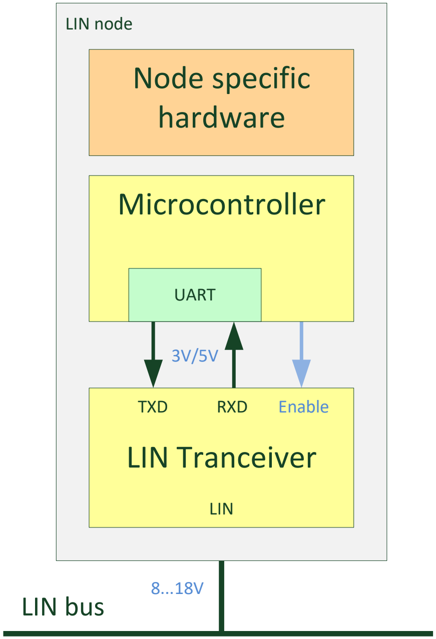
Most LIN nodes contain the following 2 components:

- Mikrocontroller with integrated UART
- LIN-Transceiver

The UART converts data bytes into asynchronous serial patterns for transmission and decodes data bytes from the received serial data stream.

It also generates break and wake-up signal patterns; this can either be implemented by special LIN functions of the UART or must be implemented by sending a binary 0x0 at a different baud rate or by bit-banging the TXD port under timer control.

The LIN transceiver translates the logic levels of the microcontroller (typ. 3...5V) into the LIN voltage range (8...18V) and converts the full-duplex RXD/TXD interface into a 1-wire half-duplex interface.



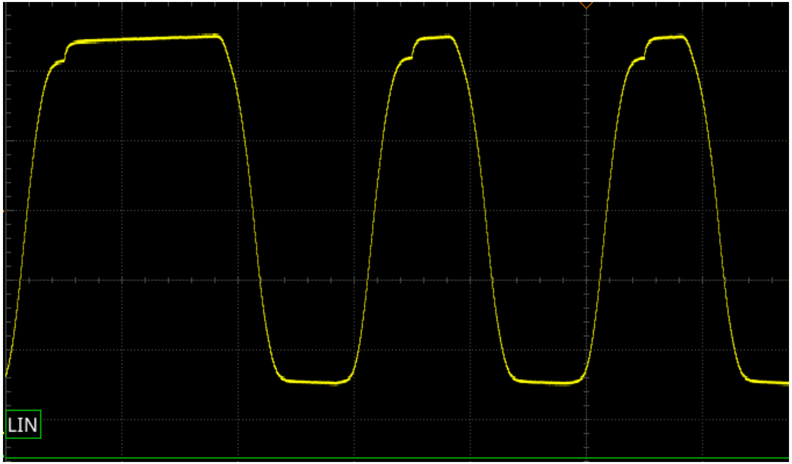
Baby-LIN systems (Generation 2) use a NXP MC33662 LIN Transceiver

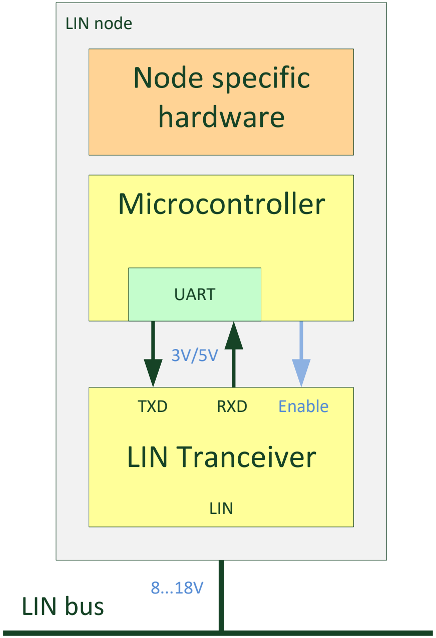
LIN Bus Hardware

Further functions of a typical LIN transceiver are

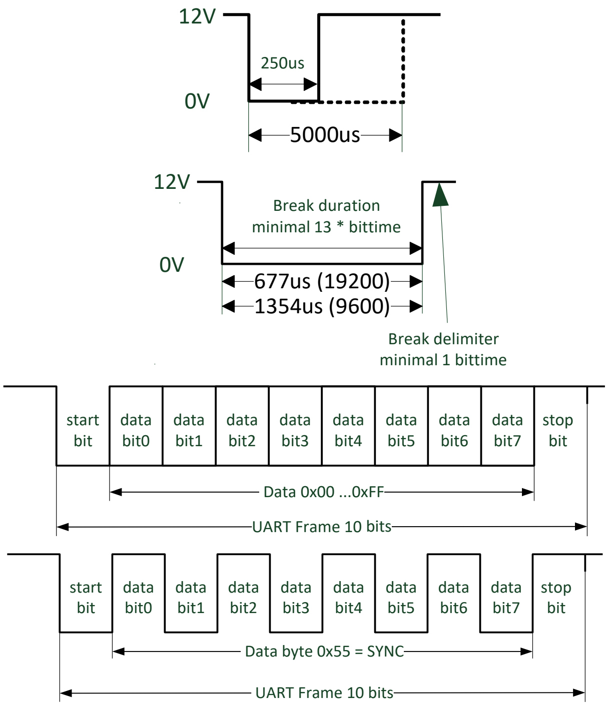
- Timeout monitoring of the dominant level
- Slope control of the signal edges
- Switch to a high speed mode to allow baud rates higher than 20 Kbit (e.g. for ECU flashing)
  => Disable Slope Control

LIN signal trace with slope control:





LIN-bus communication primitives



There are 3 basic signal patterns on the LIN bus:

1. Wake up Event
Low level pulse with 250us...5 ms length Slave recognition Low pulse >= 150 us, Slave should be able to process commands 100 ms after the rising edge of the bus.

2. Break
Low level with a length of at least 13 bit times followed by a high level (break delimiter) with a minimum duration of 1 bit time, is always sent by the master to mark the start of a new transmission (frame).

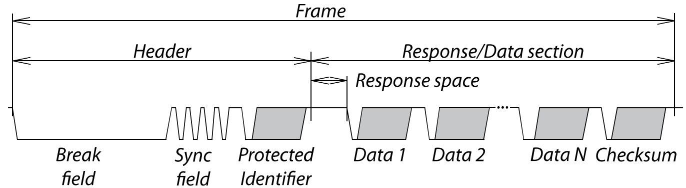
3. Asynchronous transmitted character (0....255)
Any 8 bit character (UART transmission) with 1 start bit, 8 data bits, 1 stop bit, no parity

The **LIN Sync field** corresponds to the character 0x55.

LIN Frame Bestandteile

Data transfer on the LIN bus

The smallest unit is a frame.



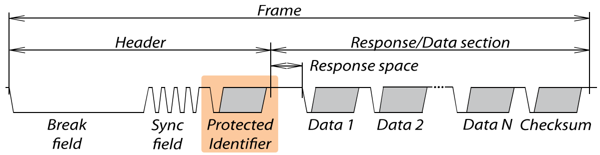
Frame Header:

- Break field — Indicates the beginning of a new frame, at least 13 bit times long, in order to be able to distinguish it reliably from all other characters
- Sync field — Allows the resynchronization of slave nodes with imprecise clock sources by measuring the bit times and reconfiguring the UART baud rate. Sync field is always sent by the master.
- Protected Identifier — A character with the frame ID. The 8-bit character contains 2 parity bits to protect the identifier, resulting in a total frame id range of 0…63.

Data Section

- Data1…Data N — 1…8 Data bytes which contain the information that will be transmitted.
- Checksum byte — Contains the inverted 8 bit sum with Carry handling over all data bytes (Classic checksum) or over data bytes and Protected Id (Enhanced checksum)
  LIN V.1.x => Classic Checksum
  LIN V.2.x => Enhanced Checksum

LIN frame security - Protected Id



Protected Id

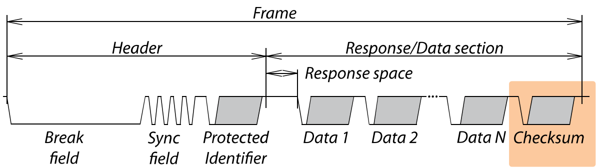
The frame ID identifies the frame. It is 8 bits in size, but 2 bits are used as parity bits, leaving only 6 bits for frame identification. Thus there are only 64 different frames on a LIN bus.

Paritybit P1 (ID.7)	Paritybit P0 (ID.6)	Identifier Bits ID.5 - ID.0
!(ID.1 ^ ID.3 ^ ID.4 ^ ID.5)	ID.0 ^ ID.1 ^ ID.2 ^ ID.4	0…63

Id dec	Id hex	PID	Id dec	Id Hex	PID	Id dec	Id hex	PID	Id dec	Id hex	PID
0	0x00	0x80	16	0x10	0x50	32	0x20	0x20	48	0x30	0xF0
1	0x01	0xc1	17	0x11	0x11	33	0x21	0x61	49	0x31	0xB1
2	0x02	0x42	18	0x12	0x92	34	0x22	0xE2	50	0x32	0x32
3	0x03	0x03	19	0x13	0xD3	35	0x23	0xA3	51	0x33	0x73
4	0x04	0xc4	20	0x14	0x14	36	0x24	0x64	52	0x34	0xB4
5	0x05	0x85	21	0x15	0x55	37	0x25	0x25	53	0x35	0xF5
6	0x06	0x06	22	0x16	0xD6	38	0x26	0xA6	54	0x36	0x76
7	0x07	0x47	23	0x17	0x97	39	0x27	0xE7	55	0x37	0x37
8	0x08	0x08	24	0x18	0xD8	40	0x28	0xA8	56	0x38	0x78
9	0x09	0x49	25	0x19	0x99	41	0x29	0xE9	57	0x39	0x39
10	0x0A	0xCA	26	0x1A	0x1A	42	0x2A	0x6A	58	0x3A	0xBA
11	0x0B	0x8B	27	0x1B	0x5B	43	0x2B	0x2B	59	0x3B	0xFB
12	0x0C	0x4C	28	0x1C	0x9C	44	0x2C	0xEC	60	0x3C	0x3C
13	0x0D	0x0D	29	0x1D	0xDD	45	0x2D	0xAD	61	0x3D	0x7D
14	0x0E	0x8E	30	0x1E	0x5E	46	0x2E	0x2E	62	0x3E	0xFE
15	0x0F	0xCF	31	0x1F	0x1F	47	0x2F	0x6F	63	0x3F	0xBF

LIN frame security - Checksum

According to the LIN specification, the checksum is formed as an inverted 8-bit sum with overflow treatment over **all data bytes (classic)** or **all data bytes plus protected id (enhanced)**:



C-sample code:

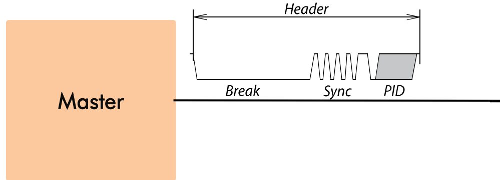
```
uint8_t checksum_calc (uint8_t ProtectedId, uint8_t * pdata, uint8_t
len, uint8_t mode){
            uint16_t tmp;
            uint8_t i;
            if (mode == CLASSIC)
              tmp = 0;

            else
                                    tmp = ProtectedId;
            for (i = 0; i < len; i++)
            {
              tmp += *pdata++;
              if (tmp >= 256)
                tmp -= 255;
            }

            return  ~tmp & 0xff; }
```

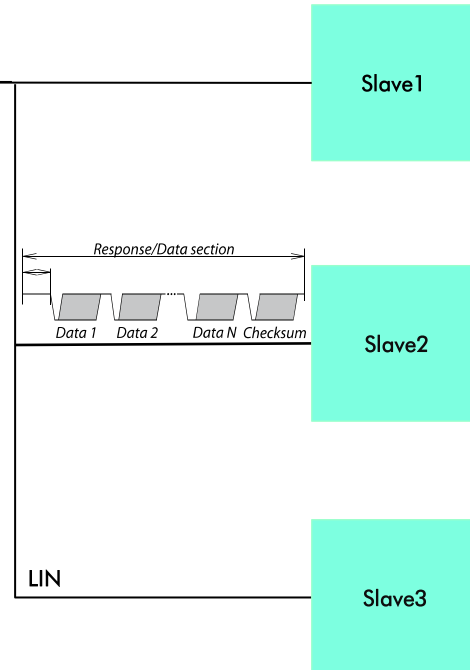
The 8 bit sum with overflow treatment corresponds to the summation of all values, with 255 being subtracted each time the sum >= 256.

Whether the Classic or Enhanced Checksum is used for a frame is decided by the master on the basis of the node attributes defined in the LDF when sending / receiving the data.

Classic checksum for communication with LIN 1.x slave nodes and **Enhanced** checksum for communication with LIN 2.x slave nodes.

LIN frame transmission / reception





We now know how a LIN frame is structured.

Now we look at how a LIN frame is used to transfer information on the bus.

The frame header is always sent by the master.

It is received by all connected nodes and they check the frame ID.

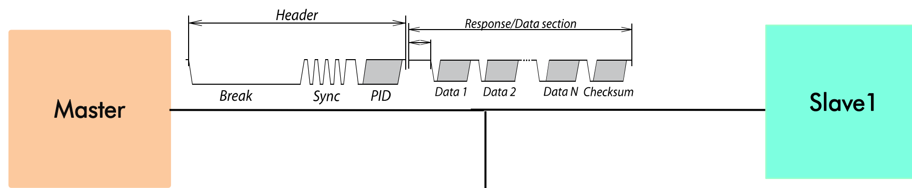
If a node determines that it is the publisher for this frame ID, it places the data for this frame on the bus.

So there is always only one sender (publisher) for the data of a particular frame.

The master waits for the data from the slave, these must appear within a certain maximum time.

So the master can recognize a missing slave by the missing data.

LIN frame transmission / reception



Of course, there are also frames that transfer data from the master to a slave, e.g. to transmit a command to a slave. In these cases the master is defined as publisher for this frame.

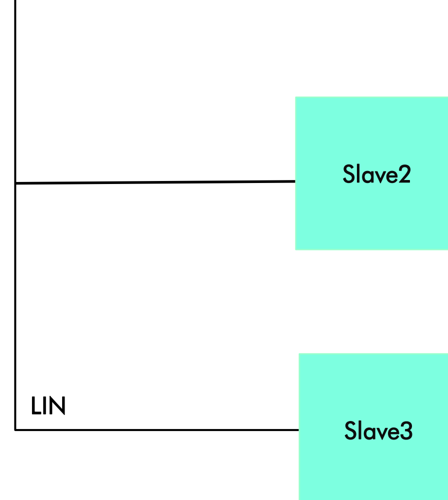
Here the master sends the frame header and the data section.

The master cannot recognize whether the addressed slave has received the frame or not.

Therefore, there is no confirmation mechanism for the LIN frame transmission, which can be found, for example, on the CAN bus.

Of course, the whole concept only works if every node (Master/Slave) connected to the bus knows whether it is the publisher for a certain frame (=ID) or not.

The assignment of the frames to the nodes is defined in the LIN Description File (LDF). Each frame (frame identifier) is assigned a node as publisher.



LDF - Lin Description File

- Format and syntax of the LDF (LinDescriptionFile) are described in the LIN specification. This specification has been developed by the LIN Consortium, in which various parties such as car manufacturers, suppliers and tool suppliers were involved. This means that the LDF specification is not dependent on a single manufacturer.
- Each LIN bus in a vehicle has its own LDF.
- This LDF summarizes all the characteristics of this specific LIN bus in one document.
- Which nodes are there on the bus?
- Which frames are defined for the bus (PID, number of data bytes, publisher)?
- Which signals are contained in a frame (signal size, signal mapping)?
- In which order should the frames appear on the bus (Schedule Table)?
- How to interpret the values of the contained signals, translation into physical units (signal encodings).

Example: Byte Value Temperature (0...255)

0..253	temp [°C] = 0.8 * value - 35	
0	=>	-35°C
100	=>	45°C
253	=>	167.4°C
254	means sensor not installed, signal not available	
255	means sensor error, no valid value available	

Sample LDF file

LDF header

Node section

Signal section

LIN_description_file ;
LIN_protocol_version = "1.3" ;
LIN_language_version = "1.3" ;
LIN_speed = 19.200 kbps ;

Nodes {

Master: MasterECU,1.0000 ms,0.1000 ms ;
Slaves: Slave1Motor,Slave2Sensor;

}

Signals {

MessageCounter: 8, 0x00, MasterECU, Slave1Motor,
Slave2Sensor;
Ignition: 1, 0x0, MasterECU, Slave1Motor,
Slave2Sensor;
WiperSpeed: 3, 0x0, MasterECU, Slave1Motor;
Temperature: 8 ,0xFF, MasterECU, Slave1Motor,
Slave2Sensor;
WiperActive: 1, 0x0, Slave1Motor, MasterECU;
ParkPosition: 1, 0x0, Slave1Motor, MasterECU;
CycleCounter:16, 0x00, Slave1Motor, MasterECU;
StatusSensor: 8, 0x00, Slave2Sensor, MasterECU;
ValueSensor: 8, 0x00, Slave2Sensor, MasterECU;

}

Sample LDF file

Frame section

Schedule table

Signal encoding section

Encoding to signal mapping

```
Frames {
    MasterCmd: 0x10, MasterECU, 4
                        { MessageCounter, 0;
                               Ignition, 8;
                               WiperSpeed, 9;
                               Temperature,16; }
    MotorFrame: 0x20, Slave1Motor, 4
                        {WiperActive, 0;
                         ParkPosition,1;
                         CycleCounter,16; }
    SensorFrame: 0x30, Slave2Sensor, 2
                        {StatusSensor, 0;
                         ValueSensor, 8; }
}
Schedule_tables {
   Table1 {      MasterCmd delay 20.0000 ms ;
                 MotorFrame delay 20.0000 ms ;
                 SensorFrame delay 20.0000 ms ;}
}
Signal_encoding_types {
EncodingSpeed {   logical_value, 0, "Off" ;
                  logical_value, 1, "Speed1" ;
                  logical_value, 2, "Speed2" ;
                  logical_value, 3,“ Interval" ;}
 EncodingTemp {   physical_value, 0, 253, 0.8, -35, "degrees C" ;
                  logical_value, 254, "Signal not supported" ;
                  logical_value, 255, "Signal has error" ;}
}
Signal_representation {
    EncodingSpeed: WiperSpeed;
    EncodingTemp: Temperature;
}
```

LIN application frames

LDF definition:

MasterECU = master
Slave1Motor = slave (wiper motor)

Frame with ID 0x10 has 4 data bytes
Publisher = MasterECU (master)

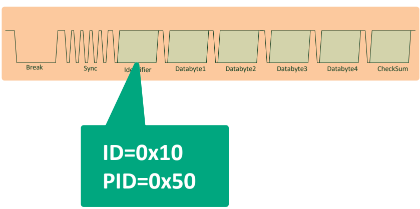
Databyte1.bit 0…7	message counter
Databyte2.bit 0	IgnitionOn (Klemme15)
Databyte2.bit 1…3	wiper speed

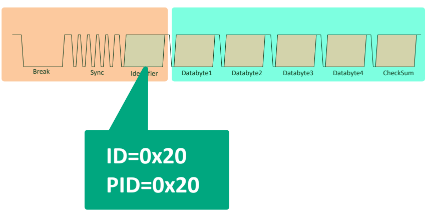
Frame with ID 0x20 has 4 data bytes
Publisher = Slave1Motor

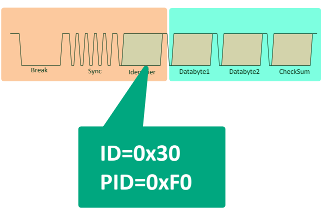
Databyte1.bit 0	wiper active
Databyte1.bit 1	park position
Databyte2.bit 0…7	CycleCounter LSB
Databyte3.bit 0…7	CycleCounter MSB

Frame with ID 0x30 has 2 data bytes
Publisher = Slave2Sensor

Databyte1	Sensor Status
Databyte2	ValueSensor







With the information from an LDF, you can assign all frames that appear on the bus to your publisher using the PID.
You can also interpret the data regarding the signals it contains…

LIN application frames

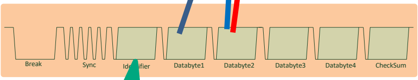
LDF definition:

MasterECU = master
Slave1Motor = slave (wiper motor)

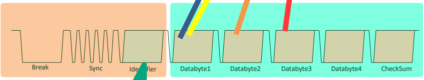
Frame with ID 0x10 has 4 data bytes
Publisher = MasterECU (master)
Databyte1.bit 0…7 message counter
Databyte2.bit 0 IgnitionOn (Klemme15)
Databyte2.bit 1…3 wiper speed

Frame with ID 0x20 has 4 data bytes
Publisher = Slave1Motor
Databyte1.bit 0 wiper active
Databyte1.bit 1 park position
Databyte2.bit 0…7 CycleCounter LSB
Databyte3.bit 0…7 CycleCounter MSB

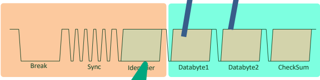
Frame with ID 0x30 has 2 data bytes
Publisher = Slave2Sensor
Databyte1 Sensor Status
Databyte2 ValueSensor



ID=0x10
PID=0x50



ID=0x20
PID=0x20



ID=0x30
PID=0xF0

With the information from an LDF, you can assign all frames that appear on the bus to your publisher using the PID.
You can also interpret the data regarding the signals it contains…

LIN Scheduling

The order in which the frames are sent to the LIN bus is defined in a so-called Schedule Table. One or more Schedule Table(s) are defined in each LDF.

Each table entry describes a frame by its LDF name and a delay time, which is the time that is made available to the frame on the bus.

A Schedule Table is always selected as active and is executed by the master.

```
Schedule_tables {
Table1        {MasterCmd delay 20.0000 ms ;
               MotorFrame delay 20.0000 ms ;
               SensorFrame delay 20.0000 ms ;}
SensorFast    {MasterCmd delay 10.0000 ms ;
               SensorFrame delay 10.0000 ms ;
               MotorFrame delay 10.0000 ms ;
               SensorFrame delay 10.0000 ms ;}
MotorFast     {MotorFrame delay 10.0000 ms ;}
}
```

The master places the corresponding frame headers on the bus and the publisher assigned to this frame places the corresponding data section + checksum on the bus.

Several schedules can help to adapt the communication to certain operating states.

The 3 Schedule Tables in the example above can optimize the acquisition of data from the engine so that it contains the corresponding frame with different repetition rates.

In TableFast, a motor signal would be updated every 10 ms, while in Standard Table (Table1), the signal would only be updated every 60 ms.

Only the master can switch the Schedule Table. Thus the master application determines which frames appear on the bus in which time sequence.

LIN Frame Typen

Auf dem LIN Bus gibt es die folgenden Frame Typen:

In der Beispiel LDF haben wir die Unconditional Frames gesehen. Diese haben genau einen Publisher und erscheinen dann auf dem Bus, wenn sie gemäß dem aktuell laufenden Schedule wieder dran sind.

Unconditional frame (UCF)

The data always comes from the same node (Publisher) and are transmitted with a constant time grid (Deterministic timing).

Event triggered frame (ETF)

A kind of alias FrameId, which combines several Slave UCF's to an own FrameId. If there is such an ETF in the schedule, only one node with changed data will put it on the bus. This saves bandwidth - but with the disadvantage of possible collisions. Due to the collision resolution, the bus timing is no longer deterministic.

Sporadic frames (SF)

This is actually more a schedule entry type than a frame type, because this SF combines several UCF's, which all have the master as publisher, in one schedule entry. The master then decides which frame to actually send, depending on which frame has new data

Diagnostic frames

A pair of MasterRequest (0x3C) and SlaveResponse (0x3D) frames. Used to send information that is not described in the LDF. No static signal mapping as with UCF, ETF and SF.

Frame type Event triggered Frame

Event triggered Frames (ETF)

ETF's were introduced to save bus bandwidth.

Example: 4 slave nodes in the doors detect the states of the window lift buttons.

Each node has a frame definition (unconditional UCF) to publish its key state, and it also has a second event triggered frame definition (ETF) to publish the same frame data via another FrameId.

With UCF, the slave always sends the data.
With ETF, the slave only sends data if there is changed data.
In addition, the slave places the PID of the associated UCF in the first data byte.

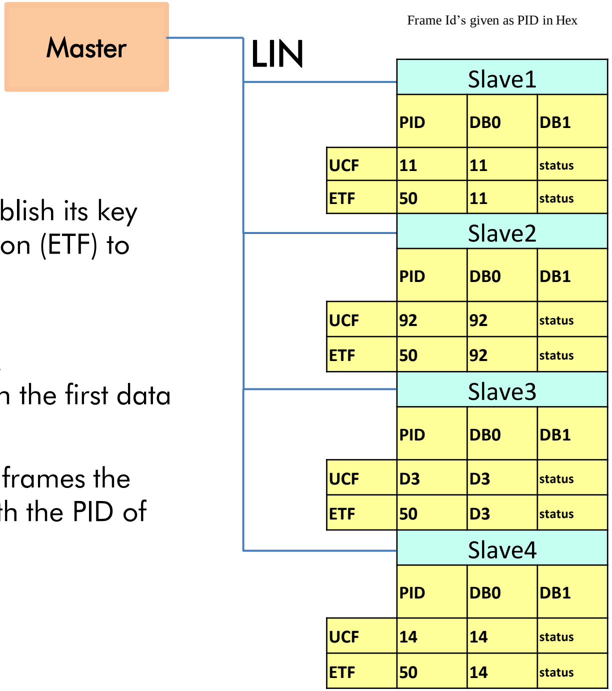
UCF / ETF have identical signal mappings, whereby in both frames the first byte is not occupied with a signal, but is always filled with the PID of the UCF.

So there are 2 possibilities to query the key states.

Via UCF frames, always works, but needs 4 frames.

Via ETF frame - this has then 3 answer variants:
No slave replies,
one slave replies or
several replies (collision).

ETF's are therefore slave frames with several possible publishers.



Frame type Event triggered Frame

The advantage of the larger bus bandwidth is bought with the possible collisions that can occur with ETF's if more than 1 node has new data for the same ETF.

The master recognizes such a collision by an invalid checksum.

In Lin 1.3/2.0 collision resolution without own collision table is defined.

Here the master will now fill the running schedule, the ETF slot, with the UTF ID's one after the other until it has queried all publishers possible for this ETF.

After that the master uses the ETF in this schedule slot again.

No Answer

1 Answer

Collision

Switching to UCF frames in ETF slot

Timestamp	FrameId	FrameData	Checksum	
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]	0x92 0x07	0x16	V2 OK
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			Collision
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x11 [0x11]	0x11 0x06	0xd7	V2 OK
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x12 [0x92]	0x92 0x06	0xd4	V2 OK
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x13 [0xd3]	0xd3 0x07	0x51	V2 OK
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x14 [0x14]	0x14 0x06	0xd1	V2 OK
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response

Frame type Event triggered Frame

With the LIN specification V.2.1 an additional mechanism for collision resolution was introduced - the Collision Schedule Table.

This Schedule Table can be assigned to the ETF definition in the LDF.

After detecting a collision, the master switches directly to the assigned Collision Schedule Table.

Typically, all UCF's of the ETF are listed there one after the other.

This means that the master can query the data of all nodes potentially involved in a collision much faster after a collision.

A possible disadvantage of this new method might be that the Collision Schedule does not provide a completely deterministic timing of the original schedule anymore, because the Collision Schedule is inserted additionally!

No Answer

1 Answer

Collision triggers switch to Collision Schedule Table

Timestamp	FrameId	FrameData	Checksum	
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]	0x92 0x07	0x16	V2 OK
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			Collision
+10	0x11 [0x11]	0x11 0x06	0xd7	V2 OK
+20	0x12 [0x92]	0x92 0x06	0xd4	V2 OK
+20	0x13 [0xd3]	0xd3 0x07	0x51	V2 OK
+20	0x14 [0x14]	0x14 0x06	0xd1	V2 OK
+5	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK

Frame type Event triggered Frame

This is how the LDF sections for the Event Triggered Frames look like.

Frames are defined as UCF's.

There the first 8 bits are not mapped with signals.

The Event Triggered Frame Definition combines several UCF's under one not yet used frame ID.

The optional specification of a Schedule Table name identifies it as a collision table for these Event Triggered Frames.

```
Frames {
    Master_DoorControl_Frame:0x00,master,5{
        Master_WindowHightFL,8;
        Master_FrameCounter,0;
        Master_Lock_DoorFR,5;
        Master_Lock_DoorFL,4;
        Master_WindowHightBR,29;
        Master_WindowHightBL,22;
        Master_Lock_DoorBL,6;
        Master_Lock_DoorBR,7;
        Master_WindowHightFR,15;
    }
    Door_FL_State:0x11,Door_FrontLeft,2{
        Door_FL_isOpen,8;
        Door_FL_isWindowUP,9;
        Door_FL_isLocked,10;
    }
    Door_FR_State:0x12,Door_FrontRight,2{
        Door_FR_isWindowUP,9;
        Door_FR_isOpen,8;
        Door_FR_isLocked,10;
    }
    Door_BL_State:0x13,Door_BackLeft,2{
        Door_BL_isOpen,8;
        Door_BL_isWindowUP,9;
        Door_BL_isLocked,10;
    }
    Door_BR_State:0x14,Door_BackRight,2{
        Door_BR_isOpen,8;
        Door_BR_isWindowUP,9;
        Door_BR_isLocked,10;
    }
}
Event_triggered_frames {
    DoorStates: ColTable, 16, Door_FR_State, Door_FL_State, Door_BR_State, Door_BL_State;
}

Schedule_tables {
    CTSchedule {
        Master_DoorControl_Frame delay 50.0000 ms ;
        DoorStates delay 50.0000 ms ;
    }
    ColTable {
        Door_FL_State delay 20.0000 ms ;
        Door_FR_State delay 20.0000 ms ;
        Door_BL_State delay 20.0000 ms ;
        Door_BR_State delay 20.0000 ms ;
    }
}
```

Frame type sporadic Frames

The purpose of the sporadic frames is to build some dynamic behaviour into the deterministic and real-time schedules without losing the determinism in the rest of the schedule.

A sporadic frame group shares the same frame slot. When it is ready for transmission, it first checks whether there has been a signal update. This results in 3 scenarios:

1. No signal has changed:

   - no frame is sent and the slot remains empty.

2. One signal has changed:

   - corresponding frame is sent

3. More than one signal has changed:

   - The frame with the highest priority is sent first. The other frames are not lost and are sent according to the order of prioritisation with each call of the sporadic frame slot.

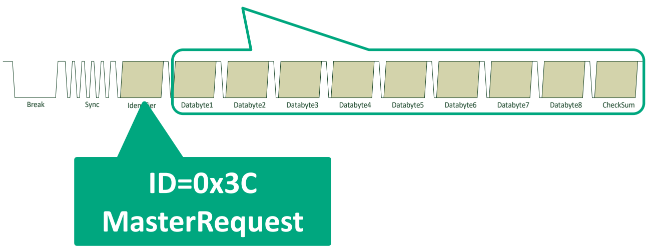
The prioritisation of the frames results from the order in which the frames are defined in the LDF.

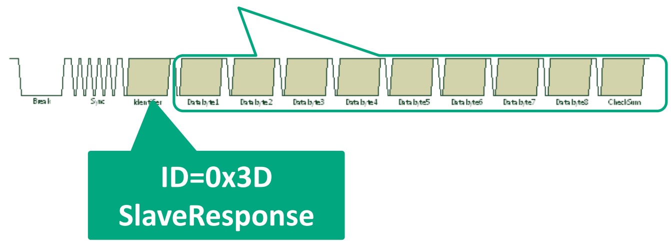
```
Frames {
    MasterSub1:0x04,BSG,2{
        MasterSporadicF1,0;
    }
    MasterSub2:0x05,BSG,4{
        MasterSporadicF2,0;
    }
    MasterSub3:0x07,BSG,3{
        MasterSporadicF3,0;
    }
}

Sporadic_frames {
    SporadicA:MasterSub1,MasterSub2,MasterSub3;
 }

Schedule_tables {
    Master_ST1 {
        LsRLS_1 delay 100.0000 ms ;
        LsMaster_1 delay 100.0000 ms ;
        LsWWS_1 delay 100.0000 ms ;
        AliasEVT delay 100.0000 ms ;
    }
    }
    Main_sporadic {
        LsRLS_1 delay 200.0000 ms ;
        LsMaster_1 delay 200.0000 ms ;
        LsWWS_1 delay 200.0000 ms ;
        SporadicA delay 200.0000 ms ;
    }
```

LIN diagnostic frames 0x3C/0x3D

0x3C MasterRequest:
Request Data define the node and the requested action.

0x3D SlaveResponse:
Data generated by the addressed slave; content depends on request





Master Request and Slave Response have special properties

- They are always 8 bytes long and always use the Classic Checksum.
- No static mapping of frame data to signals; frame(s) are containers for transporting generic data.
- Request and response data can consist of more than 8 data bytes. For example, the 24 bytes of 3 consecutive slave responses can form the response data. You then need a rule for interpreting the data. This method is also used for the DTL (Diagnostic Transport Layer).

LIN Protokolle mit Diagnoseframes

The MasterRequest - SlaveResponse mechanism can be used to transmit a wide variety of data because it is a universal transport mechanism.

A main application is the diagnosis and End of Line (EOL) configuration of nodes.

In the field there is a whole range of different protocols, depending on the vehicle and ECU manufacturer.

- A lot of proprietary diagnostics or EOL protocols
- **DTL** based protocols (**D**iagnostic **T**ransport **L**ayer)

Other protocols are typically based on the DTL layer:

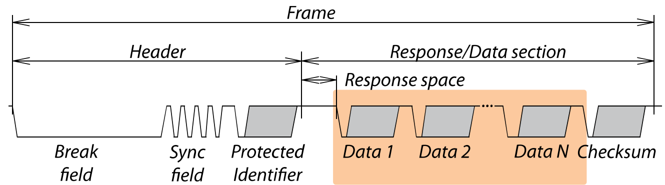
- **Standard LIN Diagnostics**
- **UDS** (Unified Diagnostic Services) (ISO 14229-1:2013)

These protocols are not part of the LDF definition.

Only the two frames 0x3C (MasterRequest) and SlaveResponse (0x3D), which serve as transport containers for the actual protocol data, are defined in the LDF.

More details about the Diagnostic Frames and related protocols will be discussed in the 2nd part of the LIN Workshop.

LIN frame security – In data CRC (optional)



Currently, the use of an additional security/safety feature for LIN frames can be observed with an increasing tendency.

It is an 8 bit CRC, which is formed by a certain block of data (e.g. Data2..Data7) and then also placed in the data section (e.g. in Byte Data1).

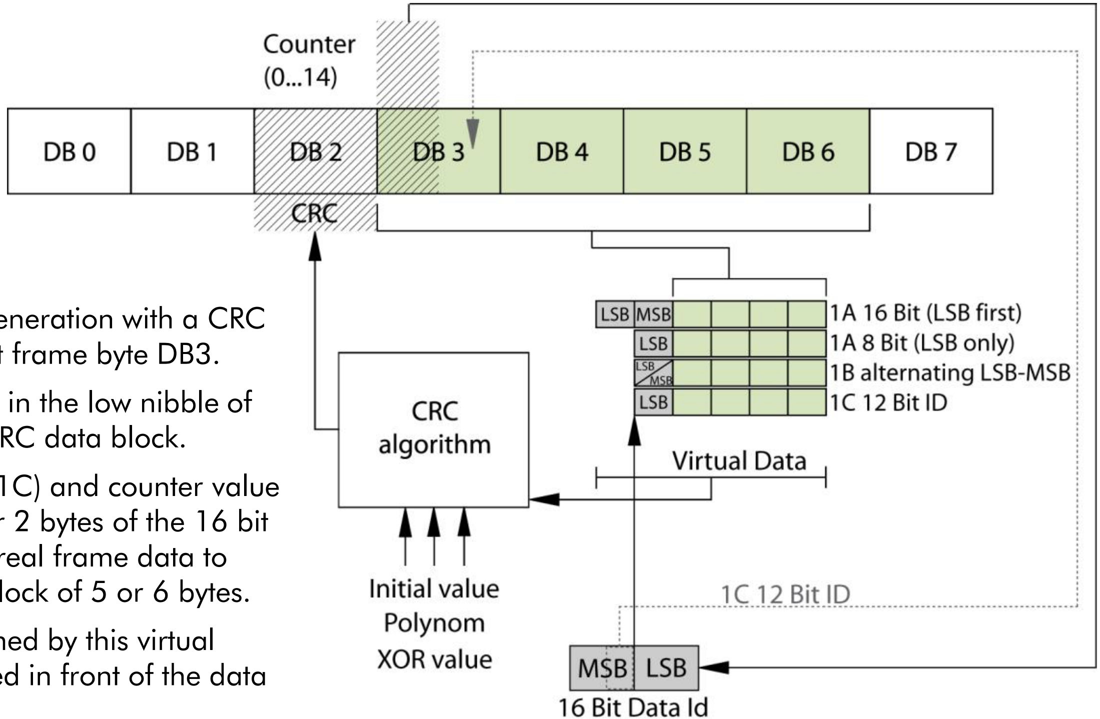
In addition to numerous proprietary implementations, a standard according to the Autosar E2E Specification is currently establishing itself, whereby there are several profiles here. However, first implementations deviating from the standard have already been viewed (e.g. BMW).

In contrast to the LIN Checksum calculation, which is disclosed in the LIN specification, the special parameters for these InData CRC's are usually only available against NDA (non disclosure agreement) from the manufacturer.

The CRC not only ensures transmission security, but is also a security feature because it can be defined in such a way that certain functions of a system can only be accessed by authorized remote peers.

All CRC Autosar implementations share an additional 4 bit counter in the data. This counter is incremented every time a frame is sent.

LIN frame security – Autosar E2E Profile1



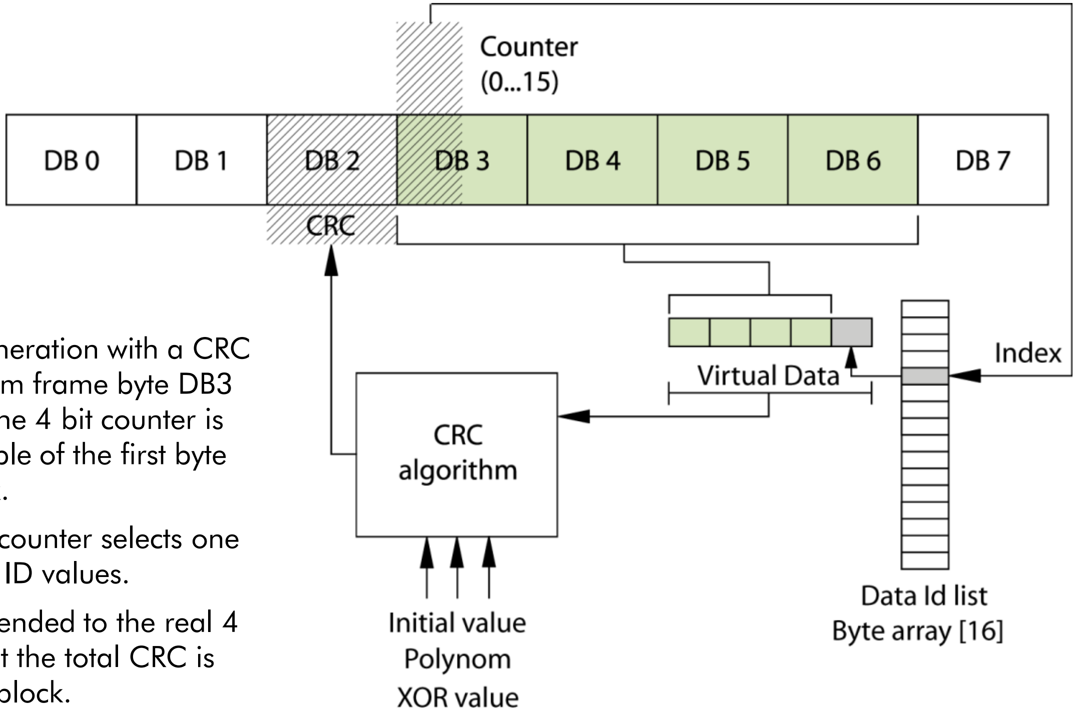
Example of a CRC generation with a CRC data block starting at frame byte DB3.

The 4 bit counter lies in the low nibble of the first byte of the CRC data block.

Profile type (1A, 1B, 1C) and counter value determine which 1 or 2 bytes of the 16 bit data ID precede the real frame data to form a virtual data block of 5 or 6 bytes.

The CRC is then formed by this virtual data block and placed in front of the data block in the frame.

LIN frame security – Autosar E2E Profile 2



Example of a CRC generation with a CRC data block starting from frame byte DB3 to Autosar Profile 2. The 4 bit counter is located in the low nibble of the first byte of the CRC data block.

The value of the 4 bit counter selects one of 16 given 8 bit data ID values.

This value is then appended to the real 4 byte CRC block so that the total CRC is formed over a 5 byte block.

In contrast to profile 1, the counter here runs from 0...15 (with profile 1 0...14).

LIN frame security – In data CRC

The definition of the parameters for a particular Indata CRC's definition is not part of the LDF specification.

In practice, there are different ways of documenting the CRC parameter specifications in a concrete project.

Sometimes they are stored as comments in an LDF file.

Or they are given in a description of the signals and frames (message catalog) of a vehicle manufacturer (PDF/HTML file).More recent description formats for bus systems such as Fibex (Asam) or ARXML (Autosar) already contain syntax elements for defining such Indata CRCs.

If necessary, a file in one of these formats can be obtained from the client.

Here one must observe the market further, in order to see what establishes itself here as mainstream.

With the LINWorks PC software the necessary parameters for the CRC's can be included in a simulation description.

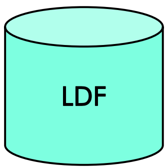
The LINWorks extension for importing new description formats such as Fibex or ARXML is planned for the future.

How to create a LIN application

Typical LIN application:

A LIN node (slave) and a suitable LDF file are available.

An application is to be implemented in which a simulated LIN master allows the node to be operated in a certain way.



Tasks

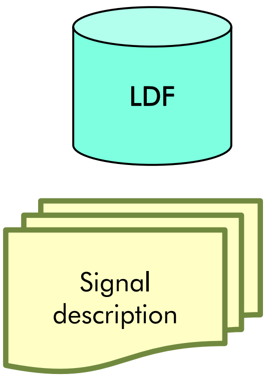
Operate LIN-node for

- functional test
- endurance run
- software validation
- demonstration
- production,
  EOL (End of Line)

How to create a LIN application

However, the information in the LDF is usually not sufficient. The LDF describes the access and interpretation of the signals, but the LDF **does not** describe the functional logic behind these signals.

Therefore you need an additional signal description which describes the functional logic of the signals (XLS signal matrix or other text file).



Tasks

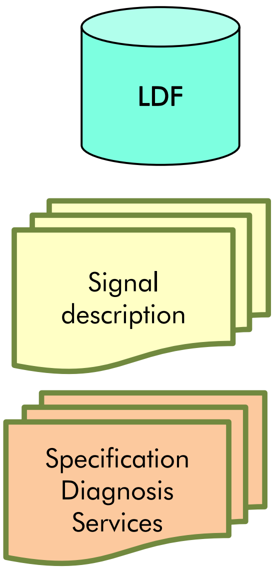
Operate LIN-node for

- functional test
- endurance run
- software validation
- demonstration
- production,
  EOL (End of Line)

How to create a LIN application

If the task also requires diagnostic communication, an additional specification of diagnostic services supported by the nodes is required (protocol type and services).

Only the two frames 0x3C/0x3D with 8 data bytes each are defined in the LDF, but not their meaning.



Tasks

Operate LIN-node for

- functional test
- endurance run
- software validation
- demonstration
- production,
  EOL (End of Line)

Lin Workflow in LINWorks

Optional hosting system
PC or PLC

LDF

Signal description

Specification Diagnosis Services

SDF

SessionDescriptionFile

Bus simulation based on LDF data

Implementation of functional logic through macro and event programming

Implementation of diagnostic services via protocol feature

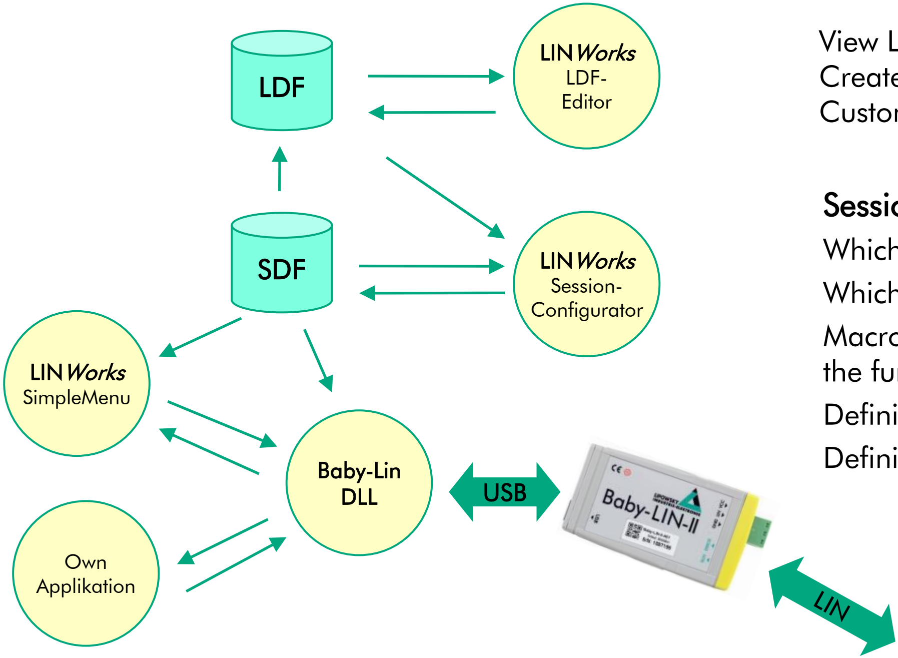
USB, Digital IO, RS-232, LAN connection via DLL or ASCII API

SDF is loaded to device

LIN-Bus

SDF
The linchpin in LINWorks-based applications

LINWorks components



LDF-Editor:

View LDF
Create LDF
Customize LDF

Session-Configurator:

Which nodes should be simulated?

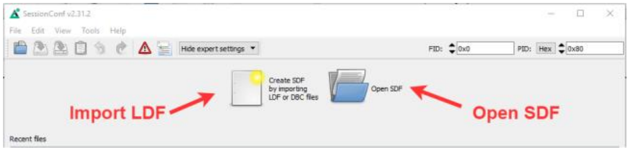
Which signals are to be displayed?

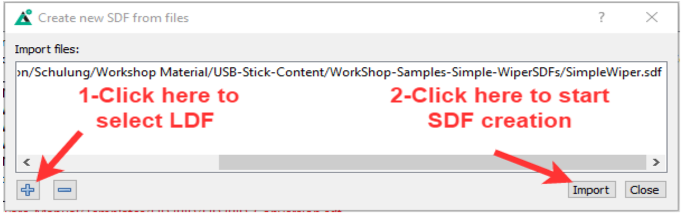
Macros, events and actions to define the functional logic

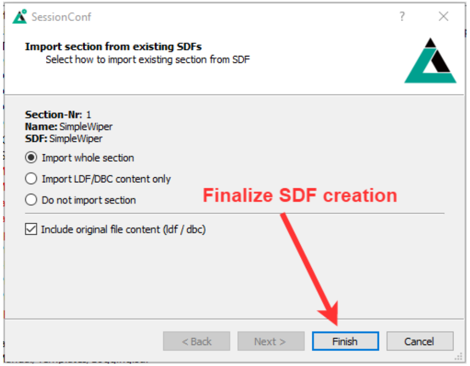
Definition of signal functions

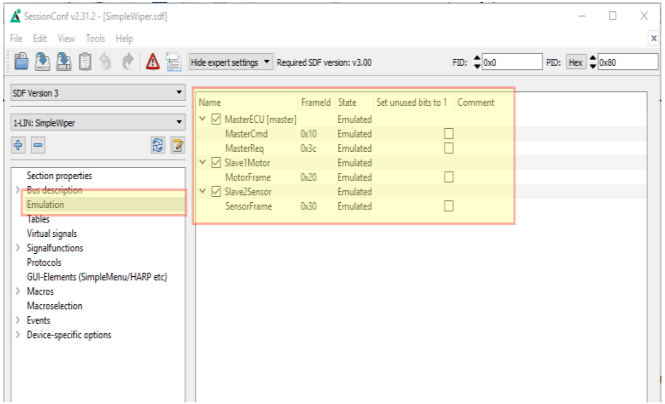
Definition of diagnostic services

LINWorks SessionConf



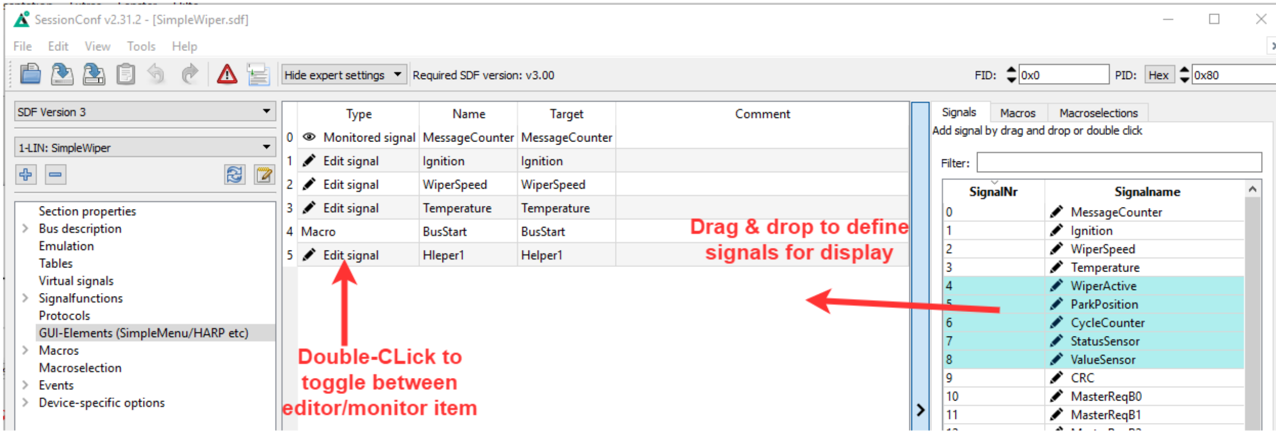






Minimal setup:

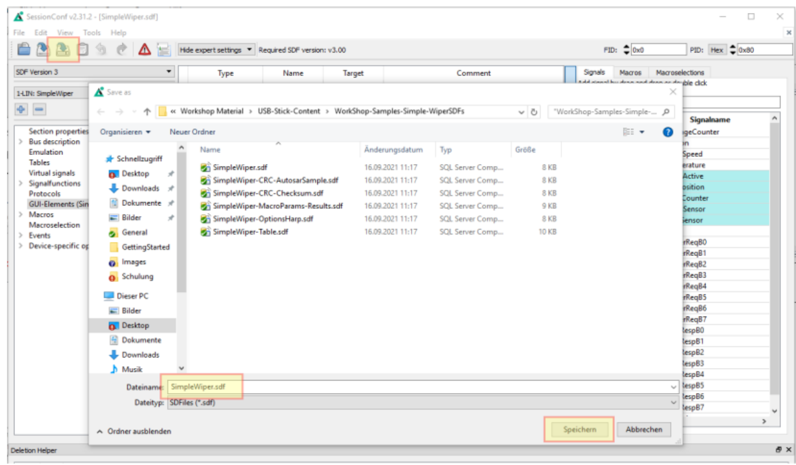
- Import LDF file into Session Configurator.
- Define emulation setup.



Defining the display contents for the PC software SimpleMenu (optional)

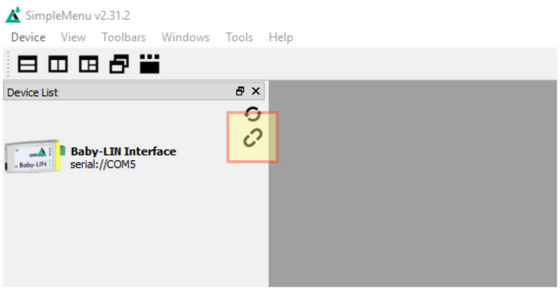
Save as SDF file

=> The first SDF is created!

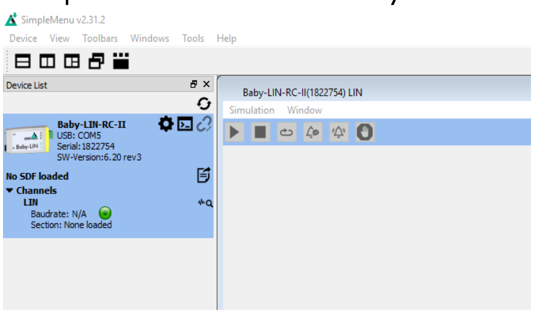


LINWorks Simple Menu

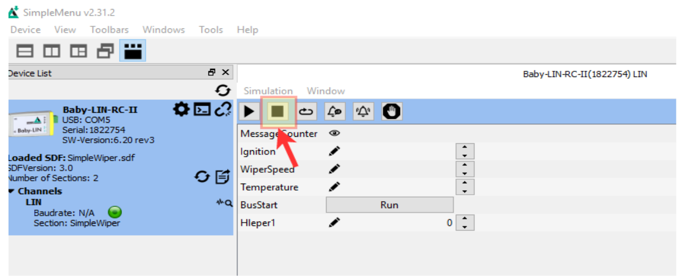
Step 1: Open SimpleMenu application
Step 2: Connect with Baby-LIN



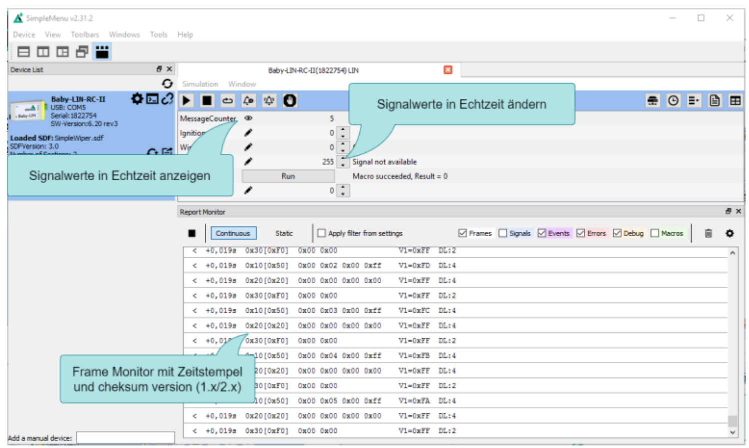
Step 3: Load SDF into Baby-LIN



Step 4: Start simulation

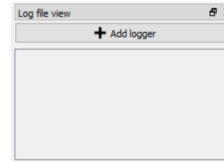


LIN-Bus running!

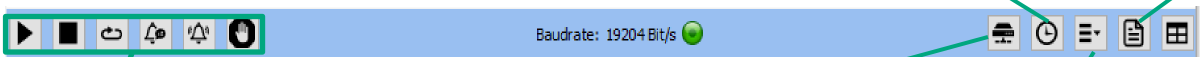


LINWorks Simple Menu

Adding a logger to record bus communication



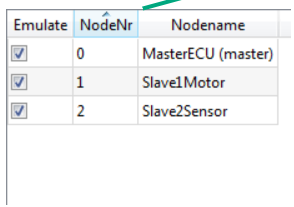
Switching to another schedule



Start, Stop, Wakeup and Sleep command

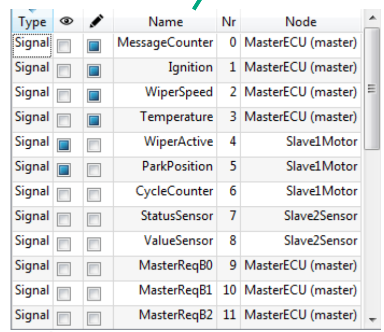
Restart command allows to start the bus without resetting the signals to the default values from the LDF/SDF.

This happens when using the Start function.



Emulate	NodeNr	Nodename
☑	0	MasterECU (master)
☑	1	Slave1Motor
☑	2	Slave2Sensor

Nodes can be dynamically switched on and off during simulation.



Type			Name	Nr	Node
Signal	☐	■	MessageCounter	0	MasterECU (master)
Signal	☐	■	Ignition	1	MasterECU (master)
Signal	☐	■	WiperSpeed	2	MasterECU (master)
Signal	☐	■	Temperature	3	MasterECU (master)
Signal	■	☐	WiperActive	4	Slave1Motor
Signal	■	☐	ParkPosition	5	Slave1Motor
Signal	☐	☐	CycleCounter	6	Slave1Motor
Signal	☐	☐	StatusSensor	7	Slave2Sensor
Signal	☐	☐	ValueSensor	8	Slave2Sensor
Signal	☐	☐	MasterReqB0	9	MasterECU (master)
Signal	☐	☐	MasterReqB1	10	MasterECU (master)
Signal	☐	☐	MasterReqB2	11	MasterECU (master)

The screen content can also be configured here as a supplement to the definition from the SDF.

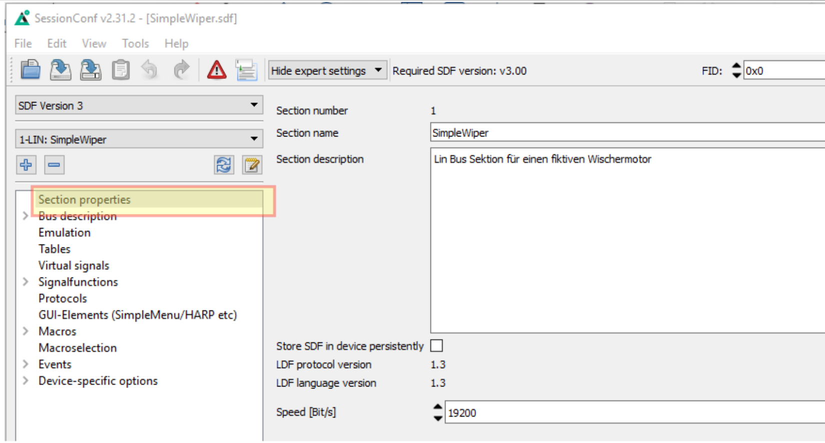
SessionConf – Section Properties

Section properties

Here you can enter a name and a description for the section.

The flag "Store SDF in device persistently" is important for stand-alone operation.

If it is set, the SDF is automatically stored in the dataflash of the device during the download.



If it is not set, the SDF is stored in the RAM of the device and is then deleted again after a Power-OFF-ON cycle.

Speed[Bit/s]

Here the LIN baud rate is displayed, which was taken over from the LDF, you can overwrite this baud rate with another value if necessary.

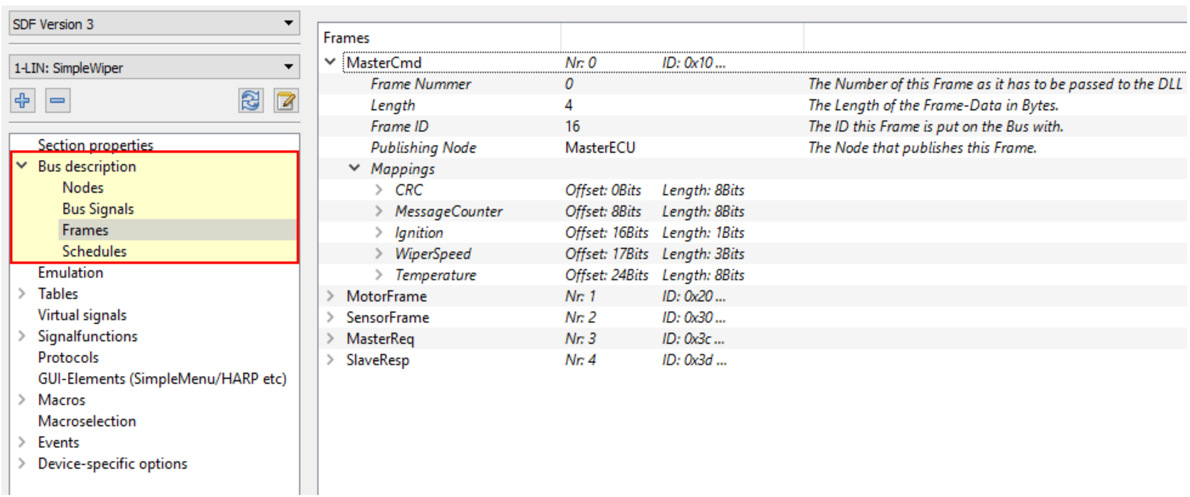
The baud rate must be entered here in a CAN section, since it cannot be taken over from the DBC and is therefore set to 0 after the DBC import.

SessionConf – Bus Description

Bus description

This area is used to display all objects taken over from the LDF such as nodes, frames, signals, schedules, etc.

You can also change some of them here. Frame id's or slot times can be adjusted in Schedule Tables.



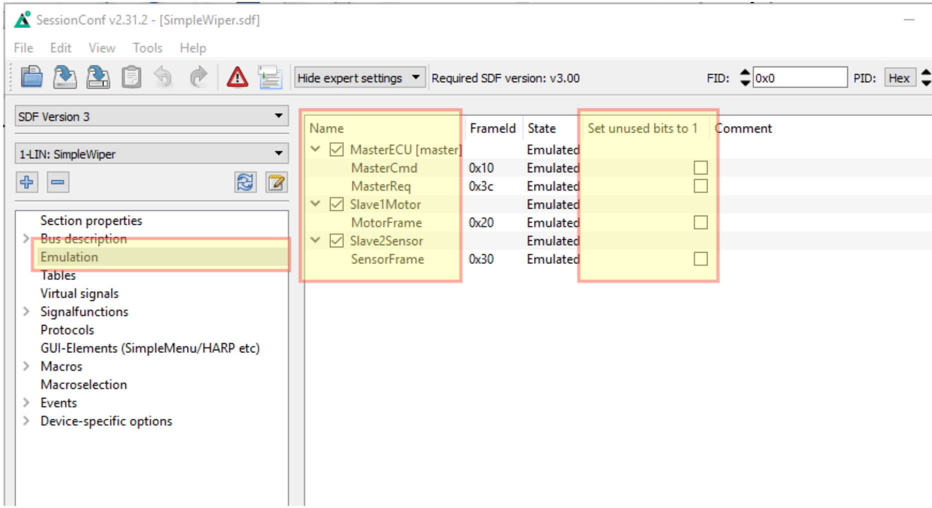
SessionConf - Emulation Setup

Emulation setup

Here you define which of the nodes defined in the LDF is to be simulated by the Baby-LIN.

Depending on which nodes are connected, you should only select nodes that are not physically present.

In our SimpleWiper example we have not connected any real nodes, so we simulate all three nodes.



Set unused bit to 1 checkbox

If not all bits in a frame are occupied with a signal, you can decide here whether these unoccupied bits are set with a 1 or a 0 during transmission.

In SDF-V2 this option did not exist yet, because unmapped bits were always set to 0.

Session Conf – Tables

The new SDF feature '*Tables*' allows to define data for the functional logic in tabular form.

1.) Creating a table

2.) Enter a name for the table

3.) Definition of columns

A column can contain text (String) or numbers (Signed/Unsigned Integer).

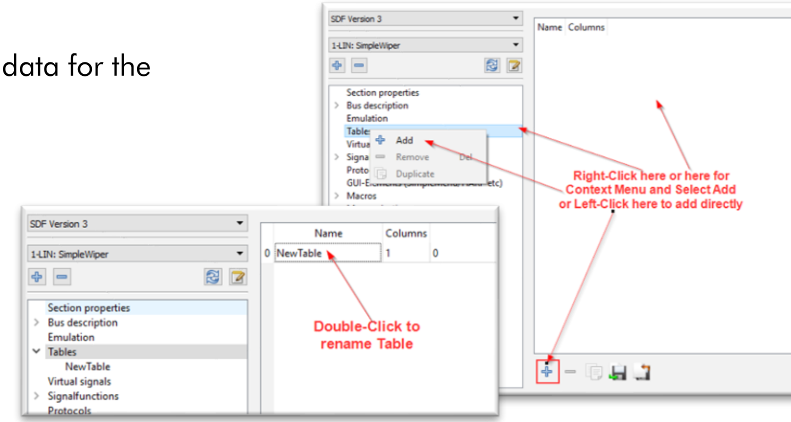
For numbers, the size (1…64 bit) can be defined for memory space optimization.

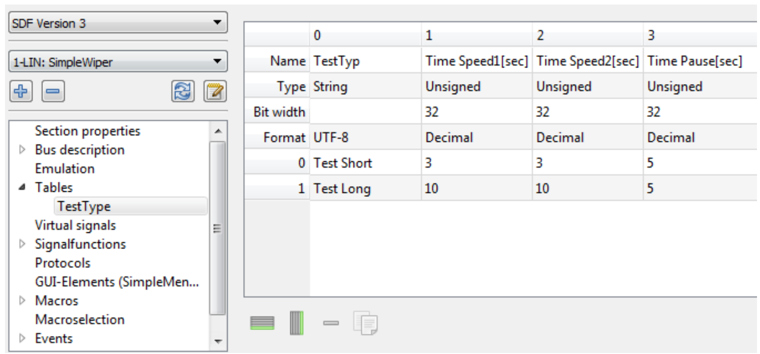
Format defines the display or input format for number columns.

Decimal	Number 32 => 32
Hexadecimal	Number 32 => 0x20
Binary	Number 32 => 0b100000

Here is an example table for defining test variants for a wiper endurance run.

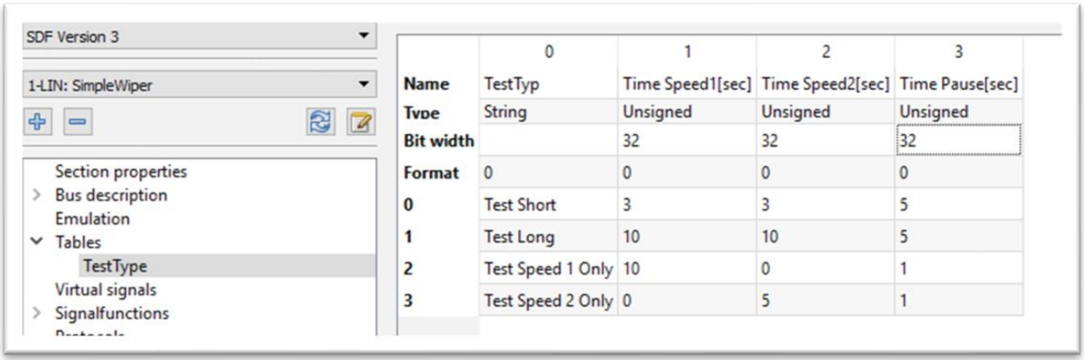
Column 0 contains the name of the test, columns 1…3 define specific time specifications for the individual test variants.





Session Conf – Tables

Here the completed example table with 5 test variants, column 0 contains the name of the test, columns 1…3 define certain time specifications for the individual test variants.



	0	1	2	3
Name	TestTyp	Time Speed1[sec]	Time Speed2[sec]	Time Pause[sec]
Type	String	Unsigned	Unsigned	Unsigned
Bit width		32	32	32
Format	0	0	0	0
0	Test Short	3	3	5
1	Test Long	10	10	5
2	Test Speed 1 Only	10	0	1
3	Test Speed 2 Only	0	5	1

Macros contain commands for accessing these table values.

You can implement procedures that differ only in parameter values in a single macro and read and use the parameters from the corresponding table line, depending on the test type you have set.

How to access the values is described in the explanation of the macro commands in the Table section.

The tables occupy much less memory space than virtual signals and are a better alternative for applications with many identical nodes (ambient lighting, climate actuators).

Session Conf – Virtual signals

Virtual signals can be defined in addition to the signals defined in the LDF. These do not appear on the bus, but can be used in macros and events.

These signals are very useful for implementing functional logic.

They can also be mapped to Protocol Frames (Protocol Feature).

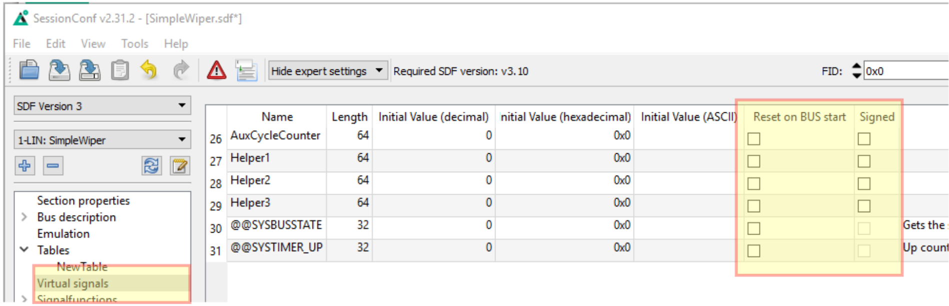
The size of a virtual signal is 1…64 bit adjustable - important when used in the protocol feature.

Each signal has a default value that is set when the SDF is loaded.

Checkbox Reset on Bus start

Allows to emulate the behavior of SDF-V2 files.

There all signals (also the virtual ones) were loaded with the default values at every bus start.



Check box signed

By default, a signal is always treated as unsigned.

With this checkbox you can turn it into a signed signal.

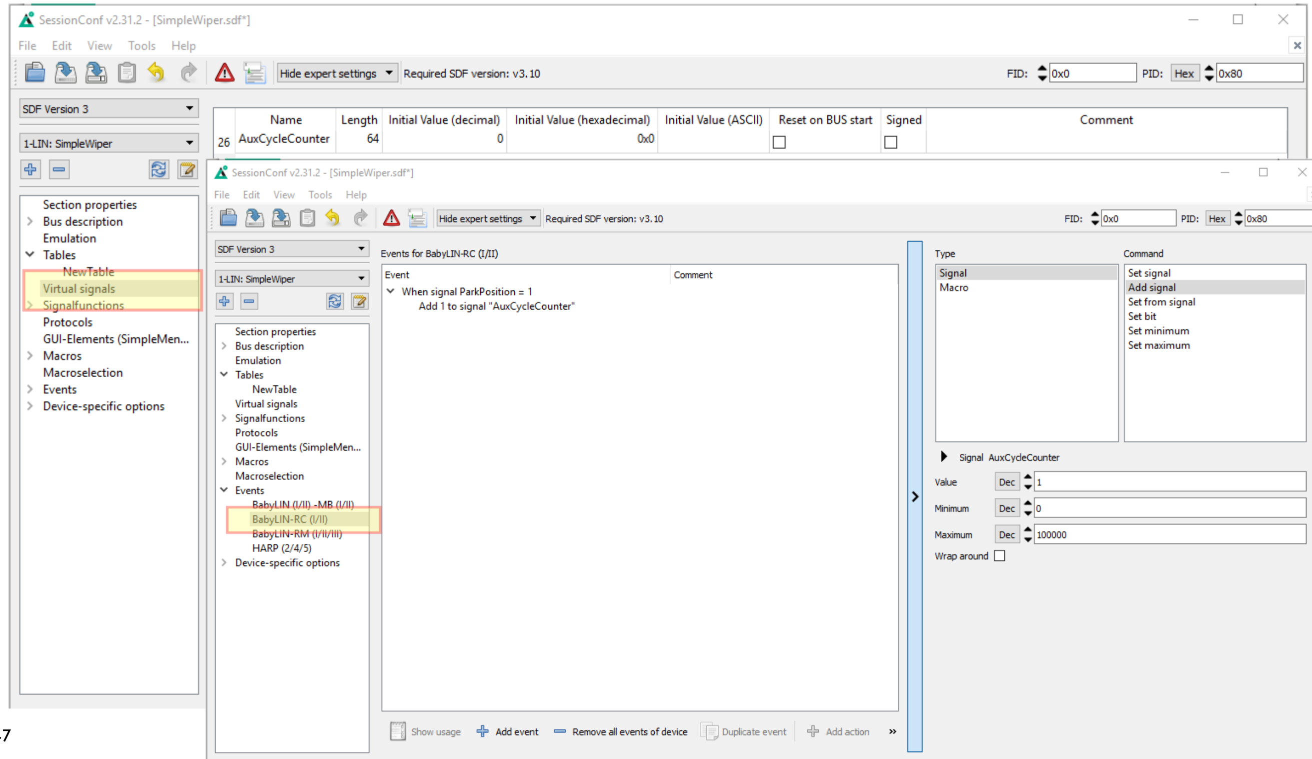
The comment column allows you to enter notes and explanations about the variable.

Session Conf – Virtual signals

Use case example

Implementation of a cycle counter by using the motor signal parking position.

Each time the signal state changes from 0 to 1, the event increments the virtual signal AuxCycleCounter.



SessionConf – System signals as spezial virtual signals

Special virtual signals => system signals

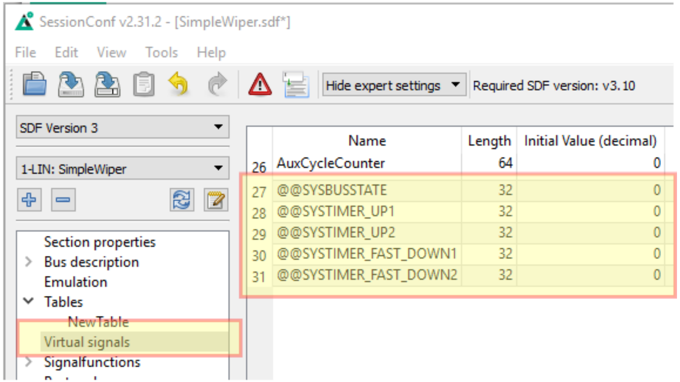
There are virtual signals with reserved names.

If these are used, a virtual signal is created once and at the same time a certain behavior is associated with this signal.

This way you have access to timer, input and output resources and system information.

Depending on the hardware version, there may be a different number of supported system variables.

All names of system signals start with prefix **@@SYS**



Often used system variables (timing functions/system information):

@@SYSBUSSTATE	gives information about LIN communication: 0 = no bus voltage, 1 = bus voltage, but no schedule is running, 2 = schedule is running and frames are sent
@@SYSTIMER_UP	generates an up counter that counts as soon as its value is not equalto 0. The counter tick is one second.
@@SYSTTIMER_DOWN	creates a down counter that counts every second until its value is 0.
@@SYSTIMER_FAST_UP @@SYSTIMER_FAST_DOWN	like SYSTIMER_UP or _DOWN, but the timer tick here is 10 ms.

SessionConf – Virtual signals system variables

More system signal for I/O control

@@SYSDIGIN1…x	Access to the digital inputs (e.g. Baby-LIN-RM-II or Baby-LIN-RC-II)
@@SYSDIGOUT1…x	Access to digital outputs (e.g. Baby-LIN-RM -II)
@@SYSPWMOUT1…4	Generation of PWM output signals on up to 4 outputs. The signal value between 0 and 100 [%] defines the pulse/pause ratio.
@@SYSPWMPERIOD	This system signal defines the fundamental frequency for the PWM output. It can be set between 1 and 500 Hz.
@@SYSPWMIN1..2	The two inputs DIN7 (@@SYSPWMIN1) and DIN8 (@@SYSPWMIN2) are supported as PWM inputs (Baby-LIN-RM-II).
@@SYSPWMINFULLSCALE	This system signal defines the full scale value (corresponding to 100%). By default, this is set to 200 by the system.

For example, the @@SYSDIGIN1…x and the @@SYSPWMIN1..2 system signal can be combined with an ONCHANGE event.

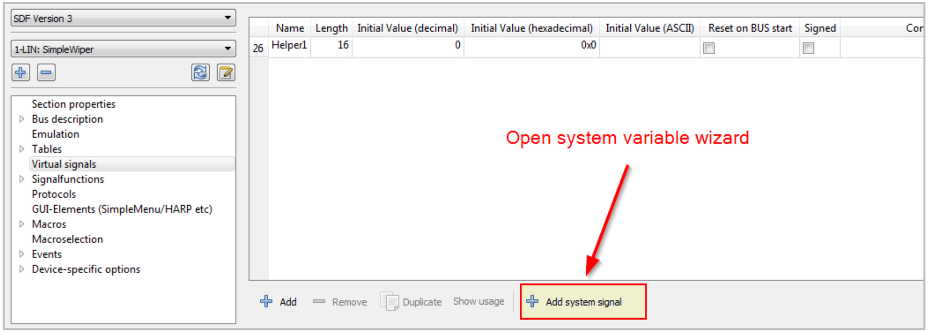
So the input value a digital input can be transferred to a LIN bus signal with only one event definition.

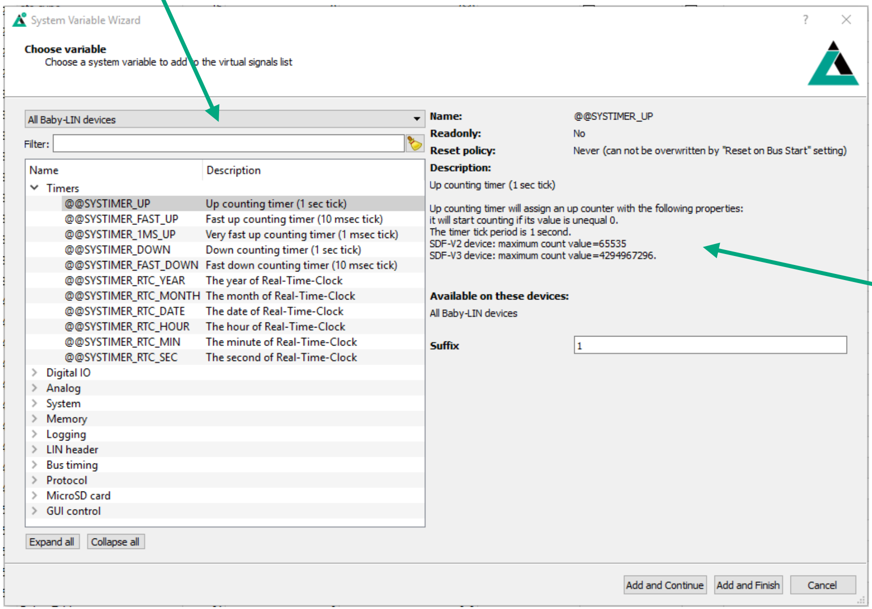
To avoid having to remember all the reserved names for the system signals and their notation, SessionConf provides a system signal wizard in the virtual signal section.

SessionConf – System variables wizard

Easy creation of system signals with the wizard.

Drop-down selection menu for restricting the display to the system signals that are available for this device type.





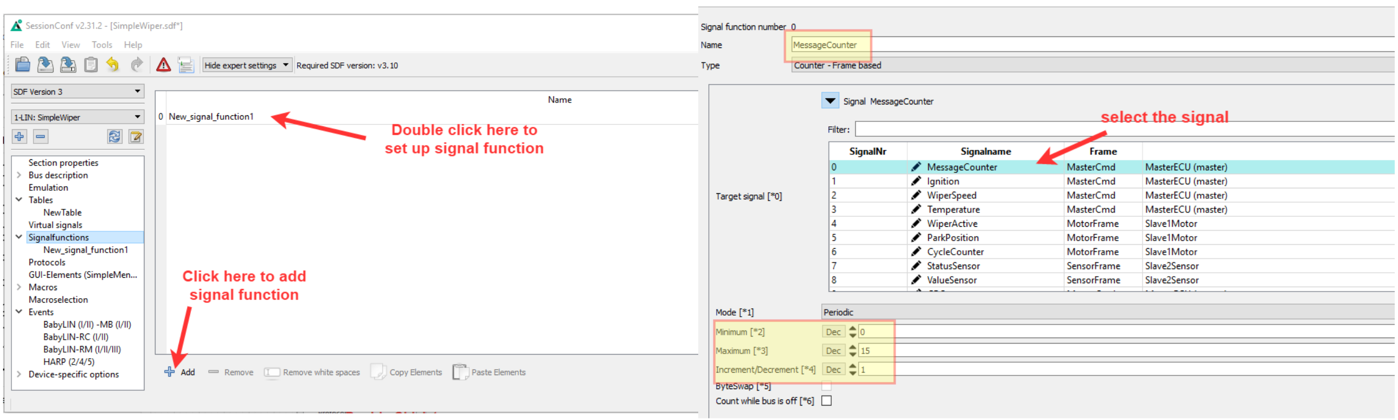
Information on the function of the selected system signal

Signal functions - Counter

If the Baby-LIN replaces the LIN bus master, it should generate the frames and signals exactly as the original control unit in the vehicle does (residual bus simulation).

There are signals in real applications that need special handling, e.g. message counters that increment their value every time they are sent on the bus, and when they reach their maximum value, they start at 0 again.
This function can be automated in the SDF via a signal function.

Another example of signal functions are CRC's in the data.



Signal functions - CRC

Signal Function CRC

With this signal function you can define an Indata checksum or CRC for specific frames according to various algorithms

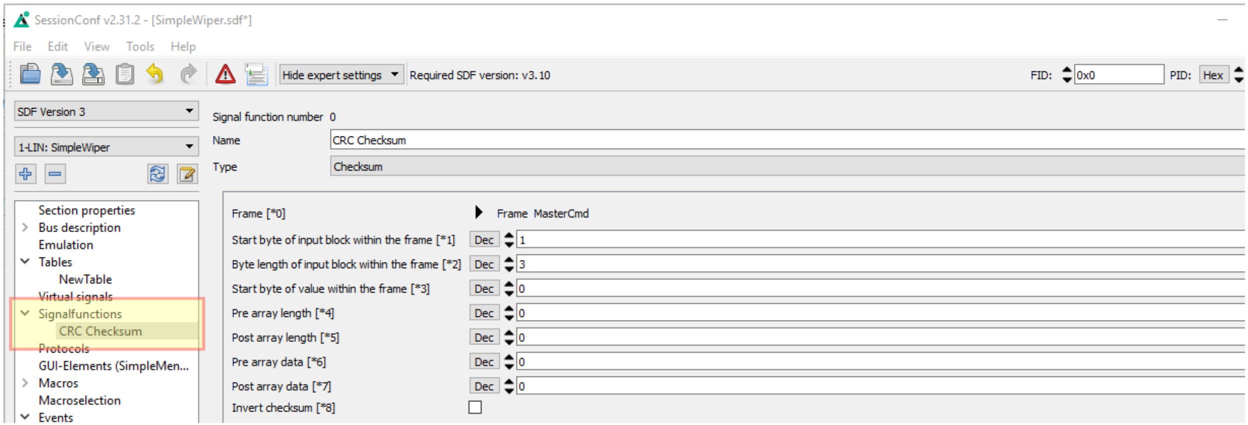
- **Checksum 8 Bit Modulo** — adds all bytes belonging to the data block and uses the LSB of the sum.
- **CRC-8** — forms an 8 bit CRC over the data block according to the specified parameters
- **CRC-16** — forms a 16 bit CRC via the data block according to the specified parameters.
- **XOR** — links all bytes of the data block via XOR.
- **CRC AUTOSAR Profile1/2** — forms a CRC according to Autosar specification E2E Profile 1/2 and other implementations.

The CRC algorithm can be freely configured with initial value, polynomial and XOR value.

For the standard Autosar variants the correct default values are suggested.

Signal functions – CRC example Checksum

Here the checksum is formed in a frame with a length of 4 bytes (= length of Frame MasterCmd) over the second to fourth data byte (Param *1 = 1 => block starts with 2nd data byte, Param *2 = 3 => block length 3, block thus comprises 2nd data byte...4th data byte) and then stored in the first data byte (Param *3 = 0 => 1st data byte).



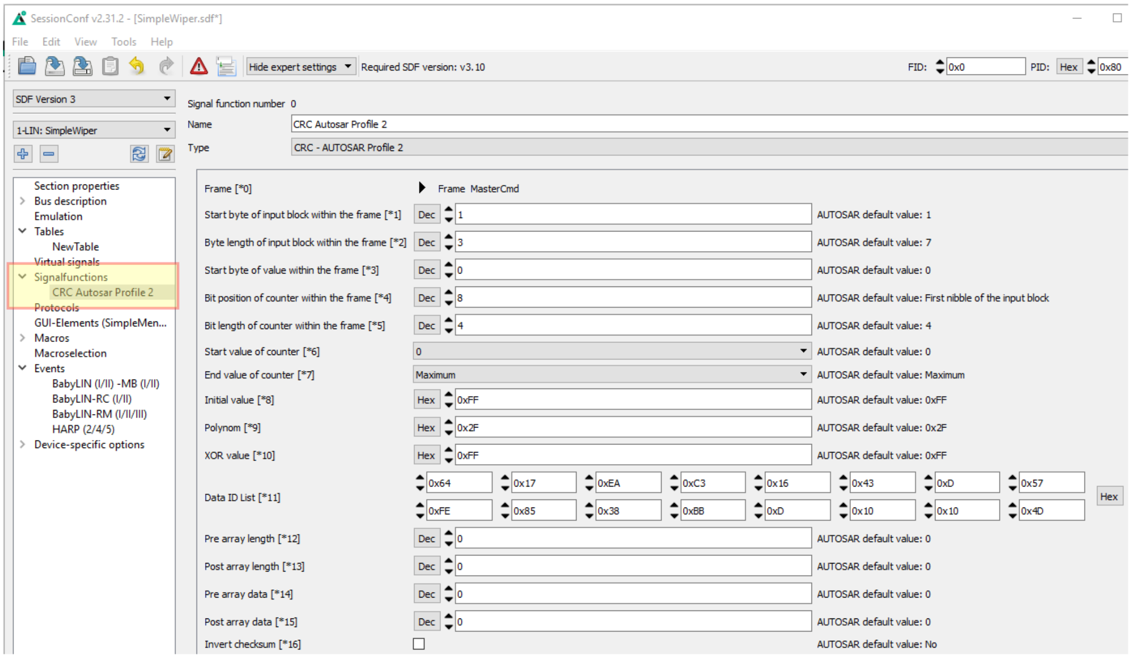
The parameters *4 to *7 define an optional prepend and postpend buffer with up to 8 byte values, which are then prepended or appended to the data of the real frame before the calculation.

This is used to implement special cases in which, for example, the FrameId is to be included in the CRC calculation.

Signal functions – CRC example Autosar

Here an Autosar CRC according to profile 2 is formed in a frame with 4 bytes length (= length of Frame MasterCmd) over the second to fourth data byte. Here too, the data block over which the CRC is formed comprises the 2nd data byte to the 4th data byte.

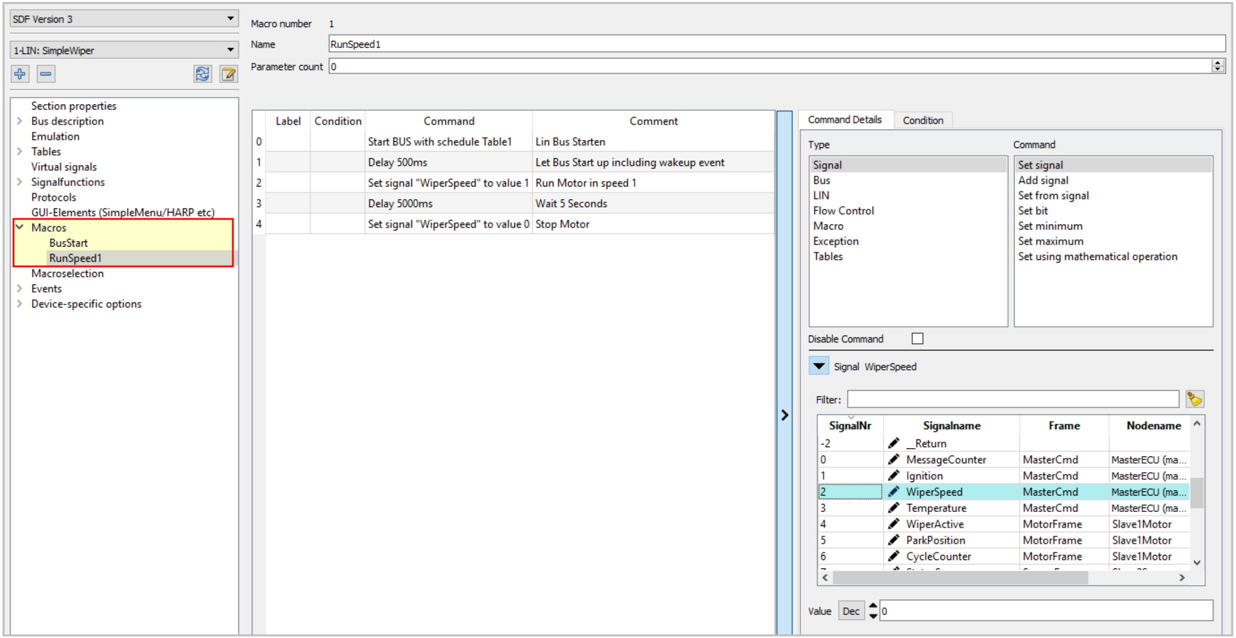
For Autosar CRC there is then a whole series of parameters.



Session Conf - Macros

Macros are used to combine multiple operations into a sequence.

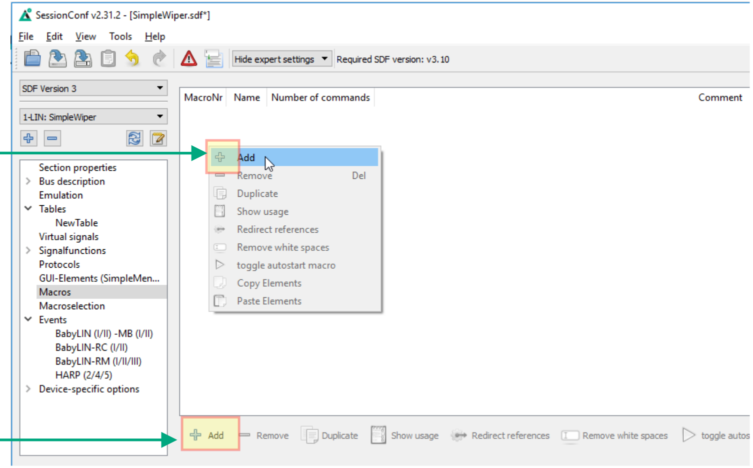
Macros can be started by events or, with SDF-V3, can also be called from other macros in the sense of a Goto or Gosub. The DLL-API calls a macro with the macro_execute command.



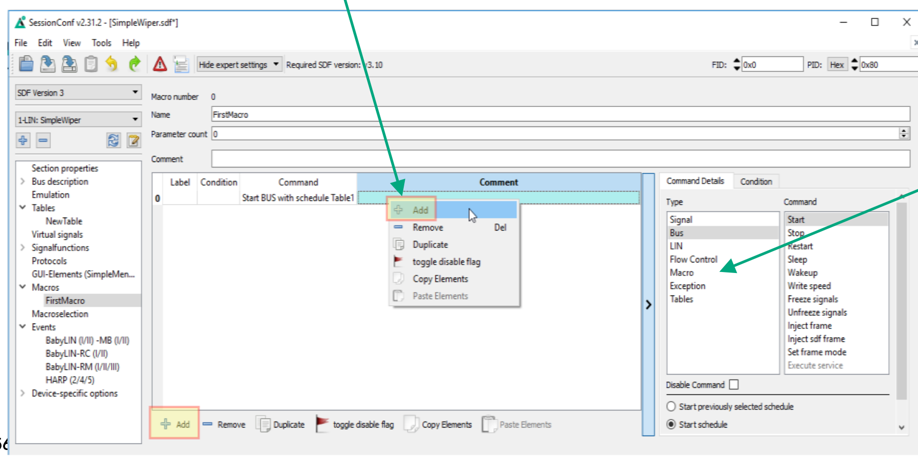
Macros play an important role in the implementation of functional logic in an SDF.

Session Conf - Macros

First you have to create a new macro, either with the
context menu (right-click)
or with the plus button.

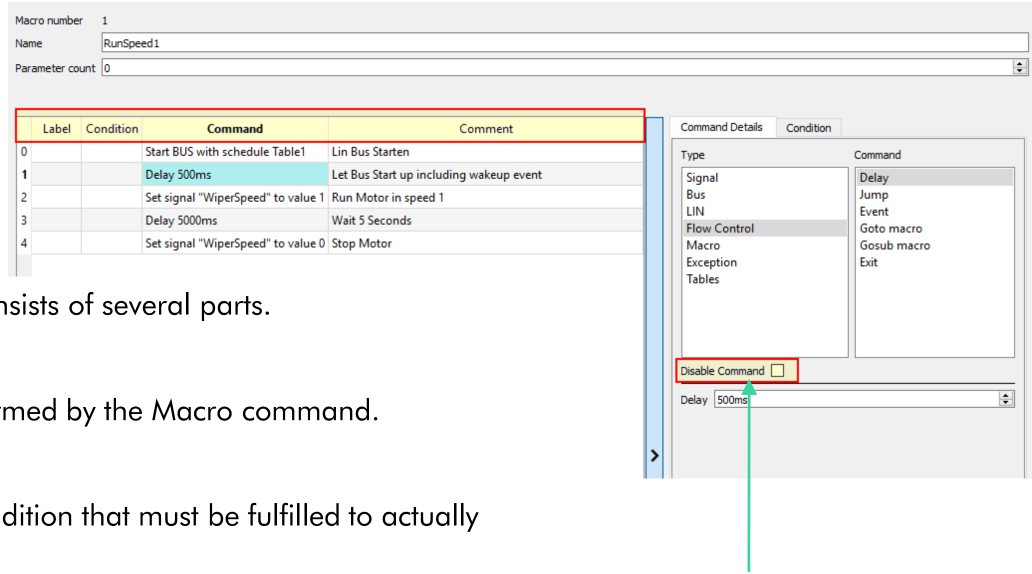


Then you add commands to this macro. The command Start Bus is always inserted; it is then changed to the desired command.



There are several categories from which you can select macro commands, such as signals, bus, LIN etc..

Session Conf - Macros



Each macro command consists of several parts.

Command
The operation to be performed by the Macro command.

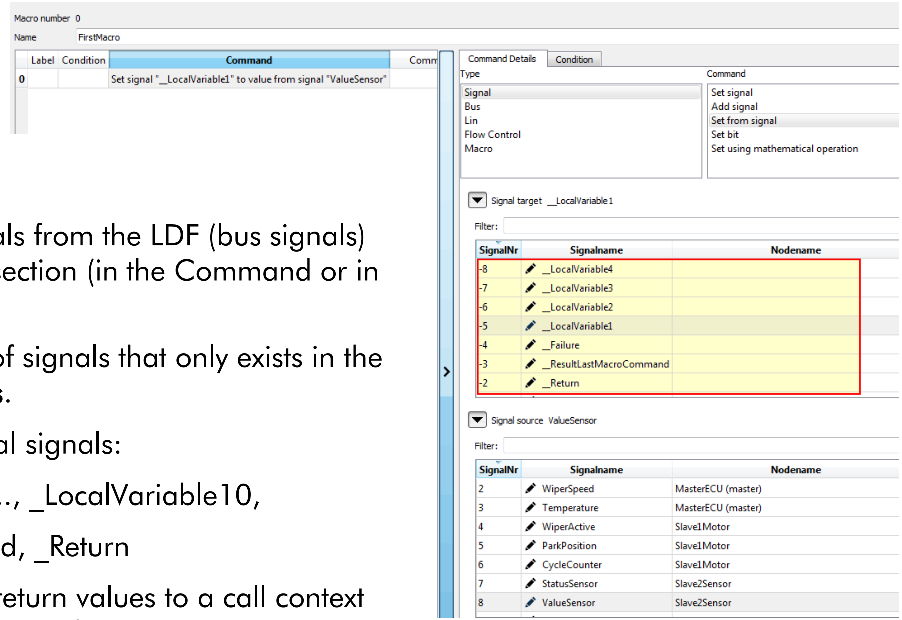
Condition
Here you can define a condition that must be fulfilled to actually execute the command.

Comment
A comment that allows you to make notes about the macro command, e.g. what to do with it on the bus.

Label
This marking of a macro command line can be used when selecting a jump command.

With the latest LINWorks version and Baby-LIN firmware every macro command can be disabled. Then it will be treated as if it were not present.

Macro Local signals/Local variables



All Macro Commands can use signals from the LDF (bus signals) and signals from the Virtual Signal section (in the Command or in the Condition).

In addition, there is another group of signals that only exists in the context of a macro: **the local signals**.

Each macro always provides 13 local signals:

_LocalVariable1, _LocalVariable2, …, _LocalVariable10,

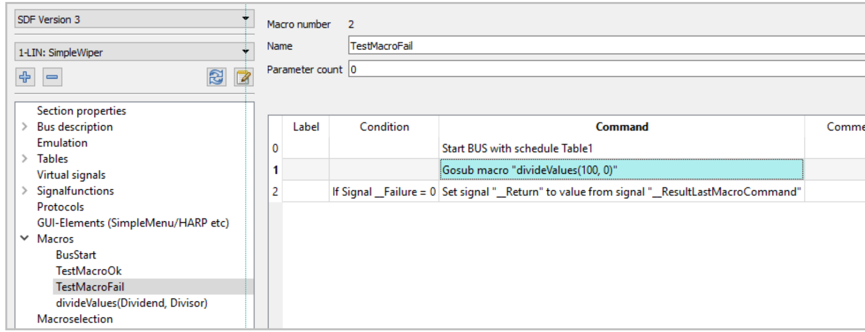
_Failure, _ResultLastMacroCommand, _Return

The last 3 provide a mechanism to return values to a call context (_Return, _Failure) or to check the result of a previous macro command. (_ResultLastMacroCommand).

The signals _LocalVariableX can be used e.g. as temporary variables in a macro.

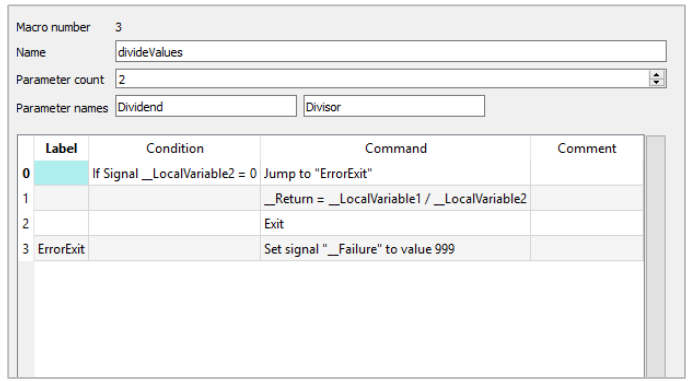
E.g. to save intermediate results when performing a calculation with several calculation steps.

Macro Parameter Passing



A macro can have up to 10 parameters when called.

In the macro definition these parameters can be given names, which are then displayed in brackets behind the macro name on the left side of the menu tree.



The parameters end up in the signals _LocalVariable1…10 of the called macro.

If no or less than 10 parameters are passed, the remaining _LocalVariableX signals get the value 0.

To return the result of a macro to the caller, the local signals _Return and _Failure are available.

Macro Result return

The local signals **_Failure** and **_Return** are used to return results to a call context.

Call by other macro (Gosub)

The calling macro can use the _LastMacroResult Command variable to access the return value of the called macro which it has stored in the _Return command.
If the signal failure in the called macro was set to a value other than 0, this value is also automatically transferred to the _Failure signal of the calling macro.

Call by MacroExec Cmd for Baby-LIN-MB-II

A macro called by the Ascii API returns the value of the _Return variable as a positive result.
If the _Failure variable is set in the executed macro, the return value is @50000+<_Failure>.
Attention: Result return only with blocking Macro call.

Macro number 2
Name TestMacroFail
Parameter count 0

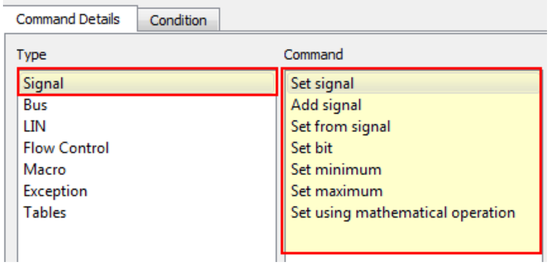
	Label	Condition	Command	Comment
0			Start BUS with schedule Table1	
1			Gosub macro "divideValues(100, 0)"	
2		If Signal _Failure = 0	Set signal "_Return" to value from signal "_ResultLastMacroCommand"	

Macro number 3
Name divideValues
Parameter count 2
Parameter names Dividend Divisor

	Label	Condition	Command	Comment
0		If Signal _LocalVariable2 = 0	Jump to "ErrorExit"	
1			_Return = _LocalVariable1 / _LocalVariable2	
2			Exit	
3	ErrorExit		Set signal "_Failure" to value 999	

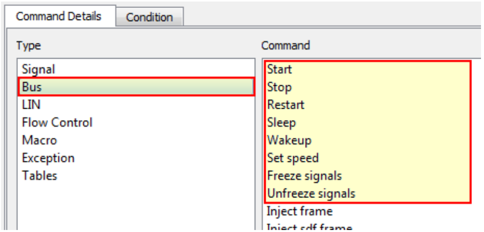
Important note: The value of **_ResultLastMacroCommand** is only valid in the Macro command line directly after the Gosub command, because this signal always contains the result of the previous command.
The _Failure variable has a different behavior. It is automatically transferred to the calling macro when setting in the called macro when returning if it has a value unequal to 0.

Macro Signal commands



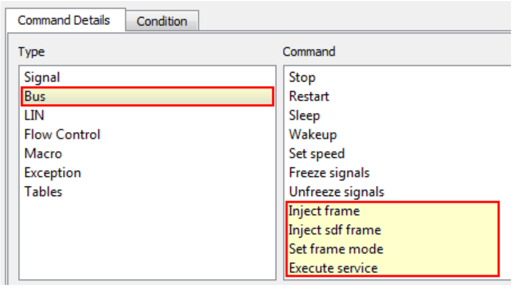
Macro command	Description
Set signal	Assign a constant value to a signal.
Add signal	Add a constant to a signal value (constant can also be negative).
Set from signal	Set a signal with the value of another signal.
Set bit	Set or delete a specific bit of a signal.
Set Minimum	Assignment of the smallest value (corresponding to bit length and signed property).
Set Maximum	Assignment of the largest value (corresponding to bit length and signed property).
Set using mathematical operation	Define the value of a signal by a mathematical operation between 2 signals or a signal and a constant. (+, -, *, /, >>, <<, XOR, AND, OR)

Macro Bus commands



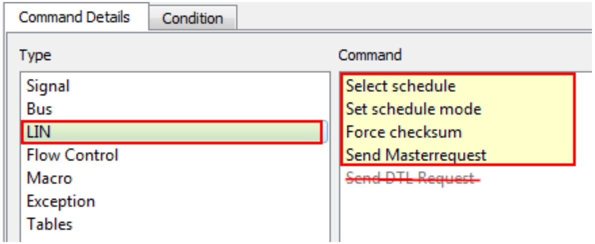
Macro command	Description
Start	Resets all bus signals to the LDF default values.
Stop	Stops the Lin Bus communication.
Restart	Starts the LIN bus, but receives all signal values. **No reset to LDF default values.**
Sleep	Sends a Sleep Frame to the bus and stops Schedule.
Wakeup	Sends a wakeup event and starts Schedule.
Set speed	Sets the baud rate of the LIN bus to the entered value.
Freeze signals	Blocks all subsequent signal changes until an unfreeze occurs. Allows atomic signal changes in a frame.
Unfreeze signals	Applies all accumulated signal changes since the last freeze.

Macro Bus commands



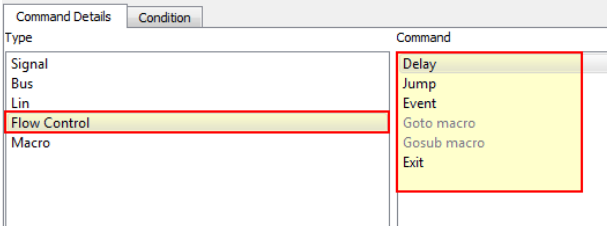
Macro command	Description
Inject frame	Allows to send any frame without LDF definition. With the latest LINWorks/Firmware version a blocking execution is also supported.
Inject SDF frame	**New:** Allows to send an SDF frame (LDF/DBC) without a schedule; the bus must be started and the frame must be sent independently from the current schedule and the bus signals must be updated accordingly (with the ReadFrame).
Set frame mode	Deactivate and activate LIN frames in a schedule or toggle between no, single shot or periodic transmission (CAN)
Execute service	Execution of a Protocol Service defined in the Protocol section. Request/Response Frame pairs can be defined and virtual signals can be mapped into request and response data.

Macro LIN-Bus commands



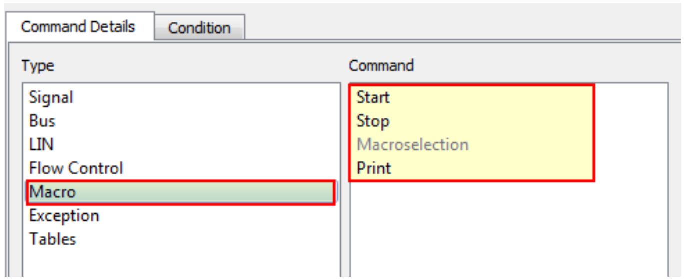
Macro command	Description
Select schedule	Schedule switching optionally, Schedule mode can also be transferred.
Set schedule mode	Permanently assign an execution mode to a schedule table: • Cyclic • Single run • Exit on complete
Force checksum	Force a certain checksum type: Automatic, V1(Classic Checksum), V2 (Enhanced Checksum)
Send Master Request	Send a Master Request (Frame ID 3C), a Schedule with suitable 0x3C Frame must run! Due to Inject and Execute Service Commands rather obsolete.
Send DTL Request	Deactivated: If the protocol feature has become unnecessary, it will disappear in one of the next updates.

Macro Flow Control commands



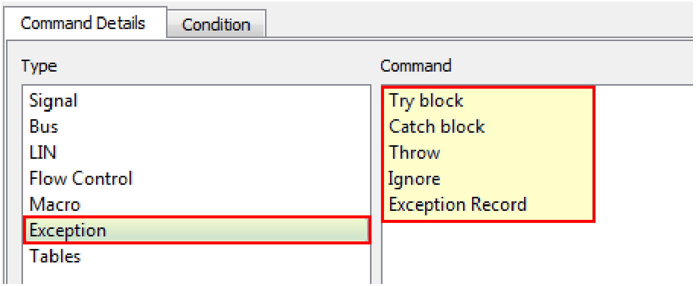
Macro command	Description
Delay	Delays macro execution by the specified time (ms).
Jump	Branches to another command in the same macro. Used for loops or branches, often in conjunction with a condition.
Event	Deactivates and activates events.
Goto macro	Branches to another macro; the remaining commands of the running macros are no longer executed.
Gosub macro	Call another macro. The running macro is continued after the Gosub command, if the called macro was terminated. The called macro can return a result (_Return/_Failure).
Exit	Ends the execution of the current macro. If the macro was called by another command via Gosub command, control is returned to the calling macro.

Macro Macro commands



Macro command	Description
Start	Starts another macro. This runs independently and parallel to the current macro.
Stop	Stops the processing of another macro.
Macroselection	Starts a macro from a Macro Selection (group of macros) There are several options for selecting the macro from the Selection group.
Print	Output of texts, signal values on the debug channel in the Simple Menu. Very helpful for troubleshooting macro programming. Further information and output to additional channels in the future.

Macro Exception commands



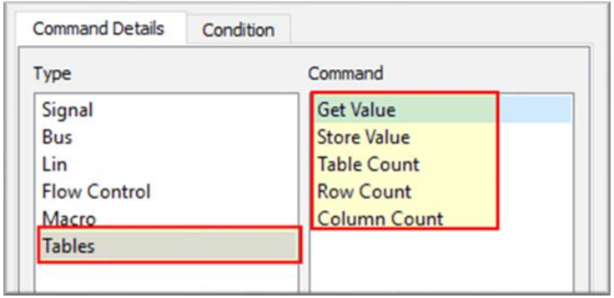
Macro command	Description
Try Block	Defines the beginning or end of a Try block.
Catch Block	Defines the beginning or end of a Catch block.
Throw	Triggers an exception with the given exception code anywhere (in the try block or outside the try block).
Ignore	Allows you to ignore certain exceptions for the following command. For example, if an Execute Service error is the expected situation due to a missing response.
Exception Record	When an exception is raised by __ResultLastMacroCommand != 0 in a try block or by a throw command, the exception code, macro number and macro command line are stored in an ExceptionRecord. With this command you can access these values.

Macro Table commands

If there are tables in the SDF, the following commands allow access.

The Get Value and Store Value operations are currently only supported on the device for cells of type Number.

The string values can already be read out via DLL.



Macro command	Description
Get Value	Loads the value of a Table Cell (Table : Row : Col) into a signal.The table, column and row selection can be defined using constants or signal references.
Store Value	Stores a signal value in a Table Cell (Table : Row : Col) Table, column and row selection as constant or signal reference.
Table Count	Sets the specified signal with the number of tables in this SDF section.
Row Count	Sets the specified signal with the number of rows in the requested table. This allows you to iterate over all lines of a table in a macro, for example.
Column Count	Sets the specified signal with the number of columns in the requested table.

Macro Table example

Use the TestType table in a macro.

The parameters for the SubMacros RunSpeed1, RunSpeed2 and Pause are read from the appropriate table row for the selected test type (Signal TestSelection).

	0	1	2	3
Name	TestTyp	Time Speed1[sec]	Time Speed2[sec]	Time Pause[sec]
Type	String	Unsigned	Unsigned	Unsigned
Bit width		32	32	32
Format	UTF-8	Decimal	Decimal	Decimal
0	Test Short	3	3	5
1	Test Long	10	10	5
2	Test Speed 1 Only	10	0	1
3	Test Speed 2 Only	0	5	1

SDF Version 3

1-LIN: SimpleWiper

- Section properties
- Bus description
- Emulation
- Tables
  - TestType
- Virtual signals
- Signalfunctions
- Protocols
- GUI-Elements (SimpleMenu/HARP e...
- Macros
  - BusStart
  - RunTest
  - RunSpeed1(time)
  - RunSpeed2(time)
  - Pause(time)
- Macroselection
- Events
- Device-specific options

Macro number 1

Name RunTest

Parameter count 0

	Label	Condition	Command	Comment
0			_LocalVariable1 = Table[TestType]::Row Count	Check if TestSelection is in range
1		If Signal TestSelection >= Signal _LocalVariable1	Jump to "ErrorExit"	
2			Start BUS with schedule Table1	
3	TestLoop		_LocalVariable1 = Table[TestType]::Row[TestSelection]::Column[1]	
4			Gosub macro "RunSpeed1(_LocalVariable1)"	
5			_LocalVariable1 = Table[TestType]::Row[TestSelection]::Column[2]	
6			Gosub macro "RunSpeed2(_LocalVariable1)"	
7			_LocalVariable1 = Table[TestType]::Row[TestSelection]::Column[3]	
8			Gosub macro "Pause(_LocalVariable1)"	
9			Jump to "TestLoop"	
10	ErrorExit		Set signal "_Failure" to value from signal "ErrorCodeInvalidParam"	

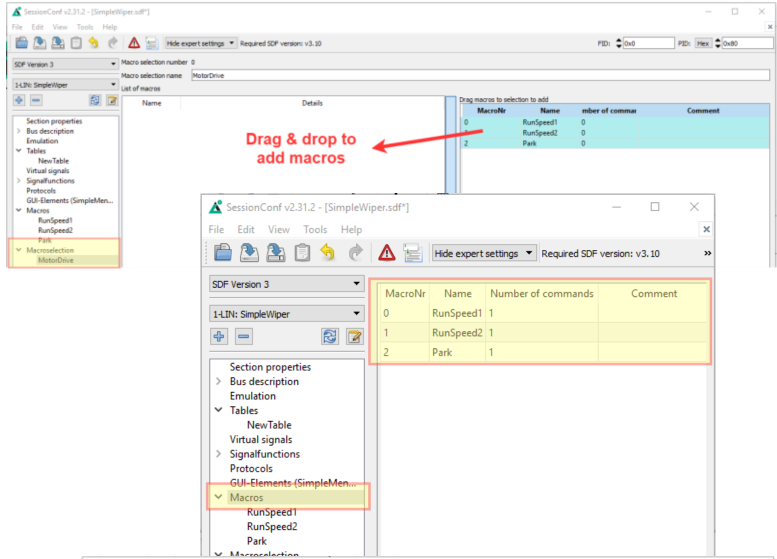
SessionConf – Macro selection

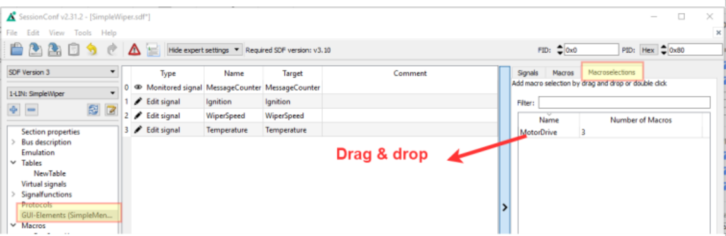
Macro selection

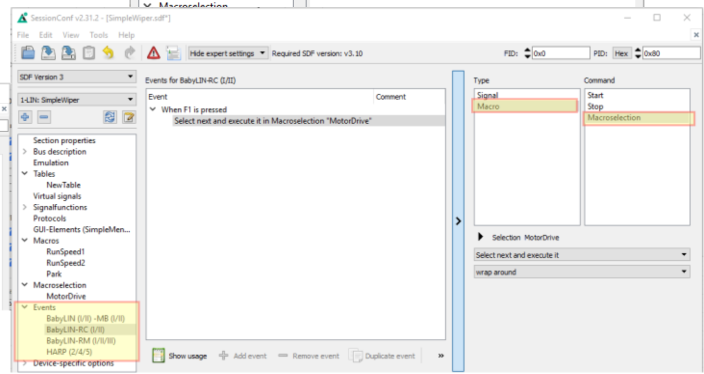
A macro selection defines a group of macros from which a macro can be selected for execution.

Example: A macro selection to choose between the macros RunSpeed1, RunSpeed2 and StopMotor.

The selection can then be made using a GUI Element, Event Action or Macro Command (SDF-V3).





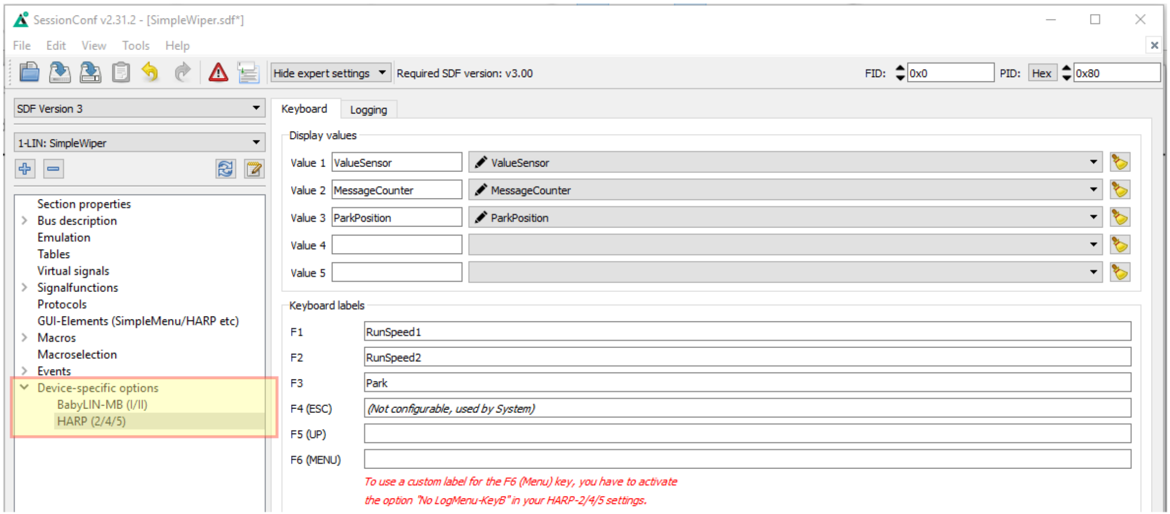


SessionConf – Device specific options

Device specific options

So far this section is only relevant for HARP users. Here you can define the signals and key labels for the HARP Keyboard Menu.

There are also setting options for custom variants (e.g. WDTS).



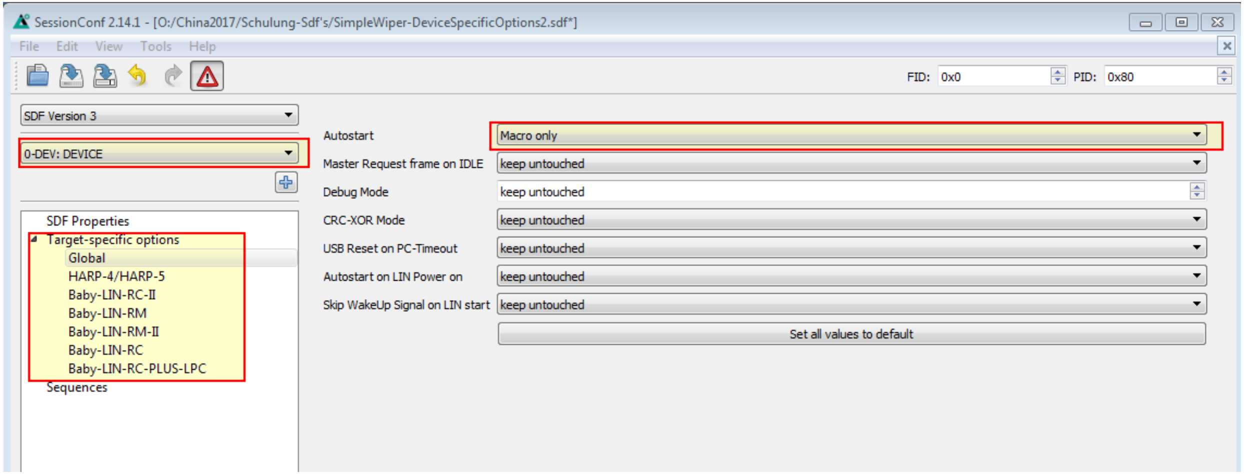
SessionConf – Device section

The Device Section (only in SDF-V3 files) allows to store the Target Configuration directly in the SDF file.

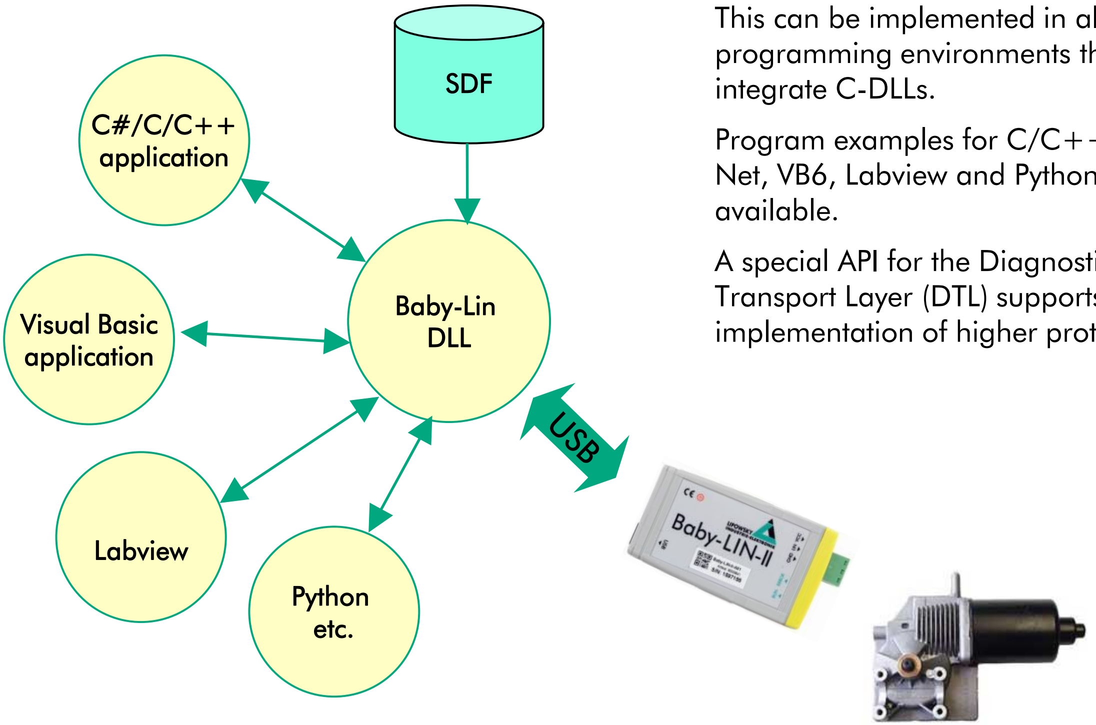
It is still possible to configure the target device in the SimpleMenu, as it was only possible in LINWorks V1.x.

If a SDF-V3 file contains a target configuration it is automatically transferred to the device during the download.

Previous problems with forgotten Target Configuration at the customer are now a thing of the past.



Baby-LIN DLL



The provided DLL allows to address the LIN bus from own PC applications.

This can be implemented in all programming environments that can integrate C-DLLs.

Program examples for C/C++, C#, VB-Net, VB6, Labview and Python are available.

A special API for the Diagnostic Transport Layer (DTL) supports the implementation of higher protocols.

Baby-LIN DLL

Baby-LIN DLL provides a whole range of API calls
The most important and most widely used are:

BLC_open (const char * port);
opens a connection to a Baby-LIN device

BLC_getChannelHandle (BL_HANDLE handle, int channelid);
gets a channel handle to a certain channel of a device (LIN/CAN, etc.)

BLC_loadSDF (BL_HANDLE handle, const char* filename, int download);
loads an SDF into the DLL and into the Baby-LIN (download = 1)

BLC_sendCommand (BL_HANDLE handle, const char* command);
sends an API command to the baby-LIN

BLC_close (BL_HANDLE handle);
closes a Baby-LIN connection

The list of all API commands can be found in the BabyLINDLL.chm help file.

There are a large number of commands that can be issued using the API call BLC_sendCommand(...).

The most important are:

start	Starts the bus communication; for LIN with optional Schedule index
schedule	Switch to another Schedule Table (LIN channel)
stop	Stops the bus communication on the given channel.
setsig	Setting a signal value
dissignal	Activation of the signal reporting for the specified signal.
disframe	Activation of frame reporting for the specified frame
macro_exec	Starts the execution of a macro stored in the SDF
inject	Allows the sending of frames independent of running schedules

The list of all commands can be found in the BabyLINDLL.chm help file

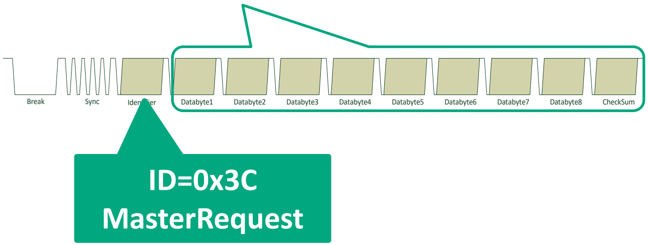
Baby-LIN DLL

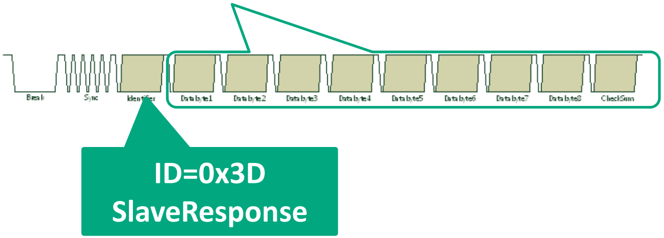
- The Baby-LIN DLL is a native library with C-interface.
- For an easy integration with .NET languages like C# and VisualBasic.NET additional wrappers are included.
- Also a Python and a VisualBasic 6 wrapper are available.
- For LabView there is an example VI collection.
- The Baby-LIN library is available as DLL under Windows and as Shared Library for PC-based and ARM-based (e.g. RaspberryPi) Linux systems.
- By accessing all signals, frames, macros etc. defined in the SDF, the distribution of tasks between your own application and the Baby-LIN device can be freely defined to a large extent.
- In addition to the SDF-based API, the DLL also offers a purely frame-based API (Monitor API). Contrary to its name, this API also supports writing operations such as sending frames.
- The Monitor API is also used for the new UDS protocol support..

LIN diagnostic frames 0x3C/0x3C

0x3C MasterRequest:
Request Data define the node and the requested action.

0x3D SlaveResponse:
Data generated by the addressed slave; content depends on request





Master Request and Slave Response have special properties

- They are always 8 bytes long and always use the Classic Checksum.
- No static mapping of frame data to signals; frame(s) are containers for transporting generic data.
- Request and response data can consist of more than 8 data bytes. For example, the 24 bytes of 3 consecutive slave responses can form the response data. You then need a rule for interpreting the data. This method is also used for the DTL (Diagnostic Transport Layer).

LIN Diagnostic Frames 0x3C/0x3D

Since a MasterRequest is received by all Slave nodes, but only one Slave is to respond to the following SlaveResponse Frame, the data in the MasterRequest must contain a kind of addressing so that the Slave can recognize that it is meant.

The connected nodes must then have different addresses according to this addressing method.

In addition, the data of the request must describe which action the master wants to execute with the addressed slave.

In order to reduce the effort for specification and implementation of these mechanisms in a LIN application, a general definition was created that is part of the LIN specification.

The protocol called DTL (DiagnosticTransportLayer) also allows larger data packets with more than 8 bytes (maximum frame size for LIN) to be transported.

The use of the Diagnostic Transport Layer (DTL) is also referred to as Cooked Mode.

However, there are still applications today that operate diagnostics without DTL; these are usually manufacturer-specific, which is referred to as raw mode.

Diagnostic transport layer DTL in detail I

Diagnostic Cooked mode

- MasterRequest and SlaveResponse Frames are the transport containers.
- Data Objects with up to 4095 bytes can be transmitted
- NAD and PCI are 2 elements that occur in each frame and provide information about the frame and its destination or origin.

Diag Frametypes

SF - Single Frame	NAD	PCI = SF	D0	D1	D2	D3	D4	D5
FF - First Frame	NAD	PCI = FF	LEN	D0	D1	D2	D3	D4
CF - ConsecutiveFrame	NAD	PCI = CF	D0	D1	D2	D2	D4	D5

PCI composition	B7...B4	B3...B0		
PCI-SF (Single Frame)	0	Length		Length 0...6, which is maximum payload length in SingleFrame Message
PCI-FF (First Frame)	1	Lenght/256	Length&0xff	In a FirstFrame (FF) PCI is always followed by additional LEN Byte.
				Maximum Payload length = 4095 (12 Bit)
PCI-CF (Consecutive Frame)	2	Framecounter		Framecounter in first CF = 1, in second CF = 2 etc. If more than 15 frames ,
				frame counter wraps around and continues with 0, 1, 2,...

LIN & CAN Tools for Test and Production

Diagnostic transport layer DTL in detail II

LIPOWSKY INDUSTRIE-ELEKTRONIK

Diag Frametypes

SF - Single Frame	NAD	PCI = SF	D0	D1	D2	D3	D4	D5
FF - First Frame	NAD	PCI = FF	LEN	D0	D1	D2	D3	D4
CF - ConsecutiveFrame	NAD	PCI = CF	D0	D1	D2	D2	D4	D5

PCI composition	B7...B4	B3...B0		
PCI-SF (Single Frame)	0	Length		Length 0...6, which is maximum payload length in SingleFrame Message
PCI-FF (First Frame)	1	Lenght/256	Length&0xff	In a FirstFrame (FF) PCI is always followed by additional LEN Byte.
				Maximum Payload length = 4095 (12 Bit)
PCI-CF (Consecutive Frame)	2	Framecounter		Framecounter in first CF = 1, in second CF = 2 etc. If more than 15 frames ,
				frame counter wraps around and continues with 0, 1, 2,...

Example: SF-Request with SF-Response

Request	0x3C	0x0A	0x03	0x22	0x06	0x2E	0xFF	0xFF	0xFF
Response	0x3D	0x0A	0x06	0x62	0x06	0x2E	0x80	0x00	0x00

Example: SF-Request (Wildcard Nad) with MultiFrame-Response (FF + 2*CF)

Request	0x3C	0x7F	0x03	0x22	0x06	0x5E	0xFF	0xFF	0xFF
Response	0x3D	0x0A	0x10	0x0E	0x62	0x06	0x5E	"3"	"C"
Response	0x3D	0x0A	0x21	"8"	"9"	"5"	"9"	"5"	"3"
Response	0x3D	0x0A	0x22	"7"	" "	" "	0xFF	0xFF	0xFF

FrameId	NAD	PCI-SF		Payload	FillByte
		PCI-FF	Length		
		PCI-CF			

Special case MasterRequest: Sleep

Even in systems that use DTL mode, a certain MasterRequest Frame can occur that differs from the DTL Frame Layout Schema.

ID	DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7
0x3C	0x00	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF

This is the Sleep Command Frame, which can be sent by the master.

It requests all connected nodes to go into sleep mode.

Usually the sleep mode in the slave is also linked to a power saving mode, depending on the slave implementation.

After sending this frame, the master stops sending further frames.

To wake up the bus again, the master sends a wakeup event and continues with the scheduling (start, restart, wakeup).It is also permissible for a slave to wake up a sleeping bus with a wakeup event.

This is also the only situation on the LIN bus where a slave can show activity on the bus without being requested to do so by the master.

Standard Diagnostic Services

In the LIN specifications V.2.0 and V.2.1 some standard diagnostic services are defined.

This standard diagnostic service is based on the DTL (Diagnostic Transport Layer).

Each service is identified by a service ID in the 1. payload byte, depending on the service further parameters follow

The table shows the available services

Only the services 0xB2 and 0xB7 must always be supported by a slave, the others are optional.

The service 0xB1 Assign Frame Identifier was only available in LIN V.2.0; it was replaced in LIN V.2.1 by the service 0xB7.

SID	Service	type
0 - 0xAF	reserved	reserved
0xB0	Assign NAD	Optional
0xB1	Assign frame identifier	Obsolete
0xB2	Read by Identifier	Mandatory
0xB3	Conditional Change NAD	Optional
0xB4	Data Dump	Optional
0xB5	Reserved	Reserved
0xB6	Save Configuration	Optional
0xB7	Assign frame identifier range	Mandatory
0xB8 - 0xFF	reserved	reserved

source: LIN specification V.2.2.

A master that controls nodes with LIN V.2.0 and LIN V.2.1 must support both services. In fact, many LIN slave nodes support both services alternatively.

DTL standard payload layout

DTL-Request Service Id SID always in 1. byte of payload

PAYLOAD Request								
DB0	DB1	DB2	DB3	DB4	DB5	DB6	......	DBn
SID	P1	P2	P3	P4	....	....	....	....

As a rule, a service consists of a request and a response, whereby there can be a positive and a negative response.

On positive response:
SID | 0x40 = RSID

PAYLOAD Positive Response								
DB0	DB1	DB2	DB3	DB4	DB5	DB6	......	DBn
RSID	[P1]	[P2]	[P3]	[P4]	....	....	....	.....

On negative response:
1. Byte 0x7F
2. Byte SID
3. Byte ErrorCode

PAYLOAD Negative Response								
DB0	DB1	DB2	DB3	DB4	DB5	DB6	......	DBn
0x7F	SID	Error code	Not used	Not used	Not used	Not used	....	Not used

Lin product identification

According to LIN specification, each LINV.2x node has a unique product identification.
The product identification consists of 3 values:

Supplier Id	16 bit number (most significant bit always 0), the Supplier Id is assigned to the manufacturer by the CIA (formerly LIN Consortium).
Function Id	16 bit Manufacturer-specific number that identifies a specific product. Products that differ in LIN communication or in their properties at the interfaces should have different Function Ids.
Variant	8 bit number, which should always be changed if the node does not experience functional changes.

Supplier Id and Function Id are required in some diagnostic services as parameters in the MasterRequest.

Wildcards have been defined so that these services can also be executed without knowledge of this ID.

Every node should support this wildcard, in practice this is not always the case.

Wildcards usually only work with a single connected slave.

However, there are exceptions, e.g. auto-addressing, but no response is evaluated.

Wildcards	
NAD	0x7F
Supplier Id	0x7FFF
Function Id	0xFFFF

Read data by Identifier Service

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xB2	Identifier	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB

Only identifier 0 must be supported by each LIN node.

Identifier	Interpretation	Length of response
0	LIN Product Identification	5 + RSID
1	Serial number	4 + RSID
2 - 31	Reserved	-
32 - 63	User defined	User defined
64 - 255	Reserved	-

The layout of the response data depends on the requested identifier:

Id	NAD	PCI	RSID	D1	D2	D3	D4	D5
0	NAD	0x06	0xF2	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB	Variant
1	NAD	0x05	0xF2	Serial 0, LSB	Serial 1	Serial 2	Serial 3, MSB	0xFF
32-63	NAD	0x02 - 0x06	0xF2	user defined	user defined	user defined	user defined	user defined

source: LIN specification V.2.2.

Diagnostic Services Assign Nad

Assign NAD Service

NAD	PCI	SID	D1	D2	D3	D4	D5
Initial NAD	0x06	0xB0	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB	New NAD

If only one slave is connected, you can also use the wildcard NAD 0x7F.
Not all slaves allow the reconfiguration of the NAD (e.g. VW-Led's from the exercises).

Wildcards	
NAD	0x7F
Supplier Id	0x7FFF
Function Id	0xFFFF

A positive answer then looks like this :

NAD	PCI	RSID	Unused				
Initial NAD	0x01	0xF0	0xFF	0xFF	0xFF	0xFF	0xFF

source: LIN specification V.2.2.

Service Assign Frame-Id

This service was deleted in LIN Spec V.2.1, but you often have to implement it in a master if you have LIN V.2.0 slaves connected there.

It is possible to use the wildcards for Supplier Id and NAD, but only if only one participant is connected.

The Message Id is a 16 bit identifier that uniquely references each frame of a node.

This Message Id / Frame assignment can be found in the node attributes of an LDF file.

Attention: the Protected Id is used in this request.

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xb1	Supplier ID LSB	Supplier ID MSB	Message ID LSB	Message ID MSB	Protected ID

If the service was successful, the slave gives a positive response as far as the master requests it by sending a slave response header.

RSID = 0xB1 | 0x40 = 0xF1

NAD	PCI	RSID	Unused				
NAD	0x01	0xf1	0xff	0xff	0xff	0xff	0xff

When using the wildcard NAD, the response is the real NAD.

source: LIN specification V.2.2.

Diagnostic Services Assign FrameId II

The Message Id used in the Assign Frame Id Service is a 16 bit number.

Each configurable frame of a LIN node is listed in the Configurable Frames section of the node attributes in the LDF. There the corresponding Message Id is also assigned to each frame. The message id is only unique within a node, but nodes of the same type have the same message id for the same frame.

Node Attribute Section from SDF

```
Node_attributes {
    LRL_1 {
        LIN_protocol = 2.0 ;
        configured_NAD = 0x01 ;
        product_id = 0x0002, 0x3000, 1 ;
        response_error = COMM_ERR_LRL_1_LIN ;
        configurable_frames {
            ST_LRL_1_LIN = 0x3001 ;
            BRC_DT_LRL_LIN = 0x3003 ;
            SU_LRL_1_LIN = 0x3002 ;
        }
    }
```

Supplier_id

Message_ids of frames supported by this node

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xb1	Supplier ID LSB	Supplier ID MSB	Message ID LSB	Message ID MSB	Protected ID

source: LIN specification V.2.2.

Diagnostic Services Assign FrameId range

Service Assign FrameId Range

This command was introduced in LIN V2.1 and replaces the obsolete Service Assign frame Id.

With this service you can assign new ID's to up to 4 frames.

The Start Index indicates to which frame the first PID in the list of up to 4 PID's belongs.

The order in the list is the same as the frames listed in the Node Attribute Section of the LDF.

If a frame is not to be supported at all, enter the value 0; if a frame is not to be reconfigured, but to retain the previous value, enter the value 0xff.

Unused PID's in the list are also set to 0xff.

```
FAN_2 {
    LIN_protocol = "2.2";
    configured_NAD = 0x02;
    initial_NAD = 0x02;
    product_id = 0x7FFF, 0xFFFF;
    response_error = COMM_ERR_FAN_2_LIN;
    P2_min = 50.0 ms;
    ST_min = 0.0 ms;
    configurable_frames {
        CTR_FAN_2_LIN;
        ST_FAN_2_LIN;
    }
```

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xB7	start index	PID (index)	PID (index+1)	PID (index+2)	PID (index+3)

Positive response

NAD	PCI	RSID	unused				
NAD	0x01	0xF7	0xFF	0xFF	0xFF	0xFF	0xFF

source: LIN specification V.2.2.

Diagnostic Service Assign FrameId range (0xB7)

Example : Configuration of 6 PID's

The slave node has 6 configurable frames, as shown in the LDF extract on the right. To assign all 6 FrameId's 2 B7 Services must be excuted.

NAD	PCI	SID	D1	D2	D3	D4	D5
0x02	0x06	0xB7	0x00	0x20	0x61	0xE2	0xA3

NAD	PCI	SID	D1	D2	D3	D4	D5
0x02	0x06	0xB7	0x04	0x64	0x25	0xFF	0xFF

```
FAN_2 {
        LIN_protocol = "2.2";
        configured_NAD = 0x02;
        initial_NAD = 0x02;
        product_id = 0x7FFF, 0xFFFF;
        response_error = COMM_ERR_FAN_2_LIN;
        configurable_frames {
                POWER_STATUS;
                CTR_FAN_2_LIN;
                ST_FAN_2_LIN;
                FAN_SPEED1;
                FAN_SPEED2;
                FAN_CURRENT_SPEED;
        }
```

Result of frameId assignment:

POWER_STATUS	ID: 0x20	PID: 0x20
CTR_FAN_2_LIN	ID: 0x21	PID: 0x61
ST_FAN_2_LIN	ID: 0x22	PID: 0xE2
FAN_SPEED1	ID: 0x23	PID: 0xA3
FAN_SPEED2:	ID: 0x24	PID: 0x64
FAN_CURRENT_SPEED:	ID: 0x25	PID: 0x25

The positive response for both service would look like this:

NAD	PCI	SID	unused				
0x02	0x01	0xF7	0xFF	0xFF	0xFF	0xFF	0xFF

Data Dump Service

This service can be used by the Manufacturers node to implement product-specific configuration services, for example, for the EOL.

So some actuator manufacturers use this service to configure with direction, EmergencyRun, EmergencyRunPosition, etc.

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xB4	User defined	User defined	User defined	User defined	User defined

Positive Response

RSID corresponds again to the rules of the DTL (0xB4 | 0x40 = 0xF4); all further data in the payload are defined manufacturer specifically.

NAD	PCI	RSID	D1	D2	D3	D4	D5
NAD	0x06	0xF4	User defined	User defined	User defined	User defined	User defined

source: LIN specification V.2.2.

LIN Diagnostic Service Save Configuration (0xB6)

Save Configuration (0xB6)

This service can be used by the node manufacturer to persistently save changes to the node configuration (NAD, FrameId, etc.) via the Data Dump Service..

However, this is not uniformly regulated because some nodes immediately write to a non-volatile memory when the corresponding change service is performed. Other nodes initially only make the change temporarily in RAM and then need this slave configuration service to store the values in non-volatile memory.

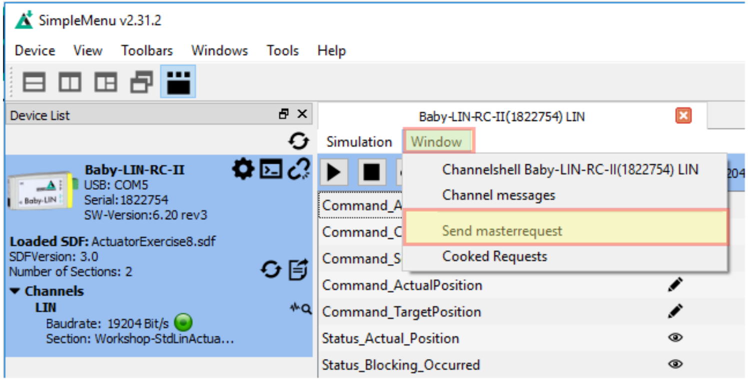
Save configuration requeste:

NAD	PCI	SID	Unused				
NAD	0x01	0xB6	0xFF	0xFF	0xFF	0xFF	0xFF

Upon successful execution of the service and correct NAD, the slave should respond with the following frame. It should be noted that there is no wait for the configuration to be saved

NAD	PCI	RSID	Unused				
NAD	0x01	0xF6	0xFF	0xFF	0xFF	0xFF	0xFF

Raw mode interactive Simple Menu



The Send Masterrequest function in the Simple Menu can be used to quickly and interactively check whether a node supports a particular diagnostic service.

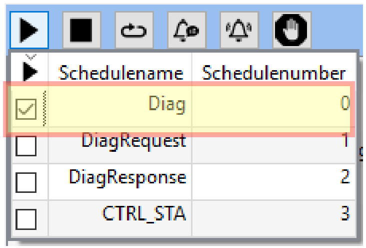
This Interactive MasterRequest mask only works if a Diagnosis Schedule has been started.

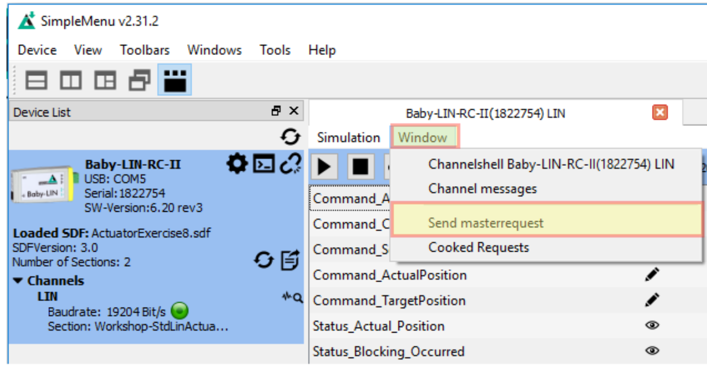
This must contain MasterRequest and SlaveResponse Frames.

Any diagnostic frames can be defined in this mask, even those that are not DTL compliant.

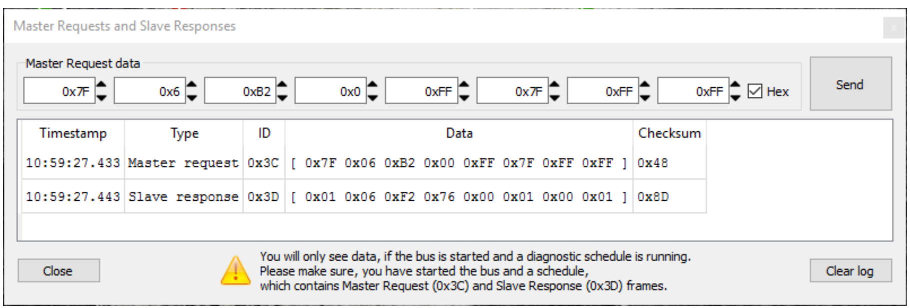
The slave response can also be displayed in this mask.

Raw mode interactive in Simple Menu





For this function a Schedule with MasterRequest and SlaveResponse Frames must be activated.
After entering the request data and setting RequestCount to 1, we see the answer when we press Send.



The actuator reports the values

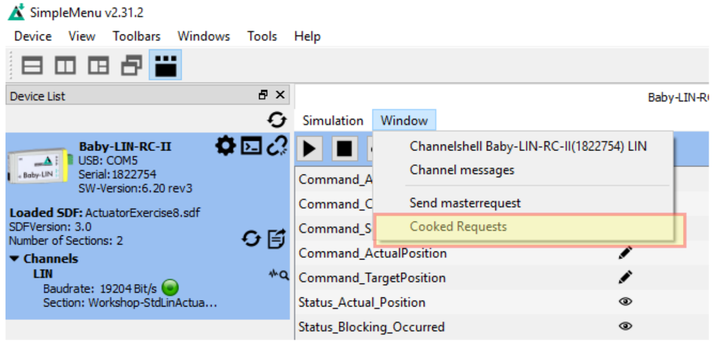
NAD	0x13
Supplier Id	0x0076
Function Id	0x0001
Variant	0x01

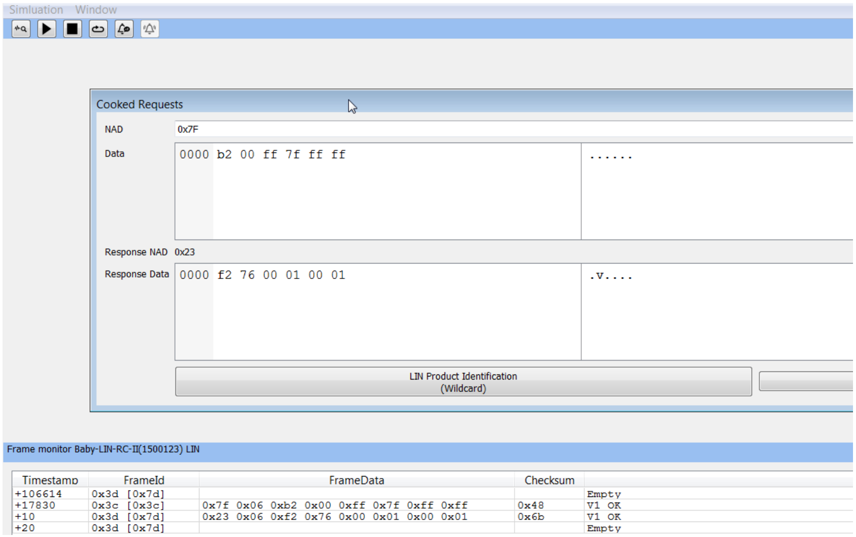
Cooked mode interactive Simple Menu

There is also an interactive mask for the Cooked Mode in the SimpleMenu.

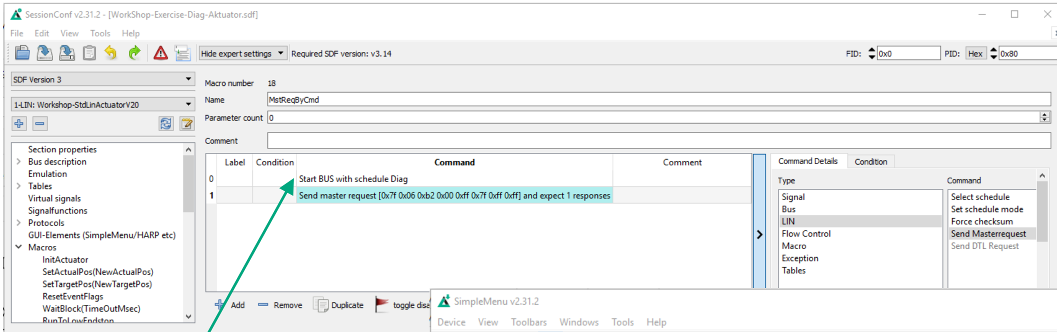
You also have to make sure that a schedule with MasterRequest and SlaveResponseFrames is running.

With the frame monitor you can also see the raw data on the bus.

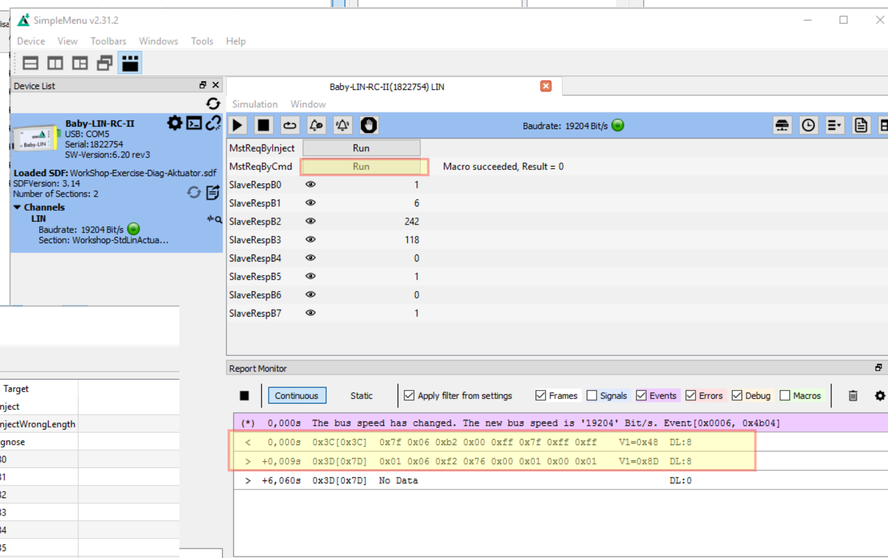


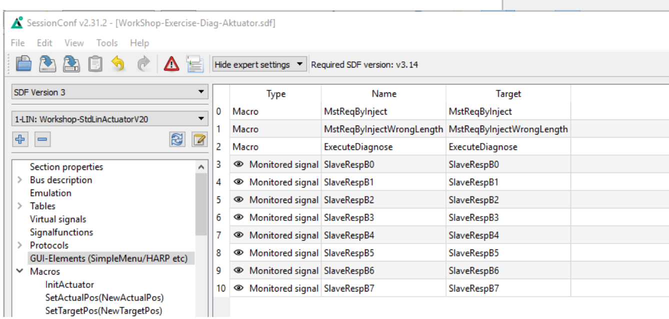


Raw mode MacroCommand MasterReq



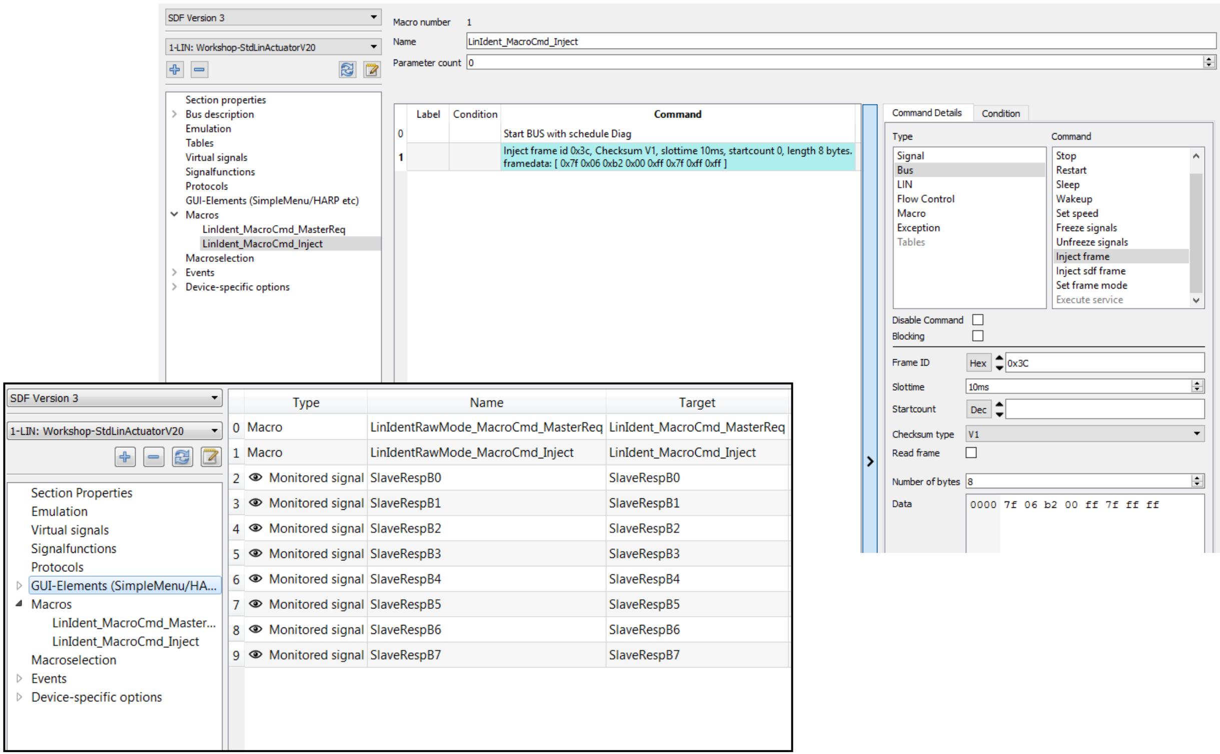
Diag Schedule necessary, therefore start bus first





Raw mode MacroCommand Inject

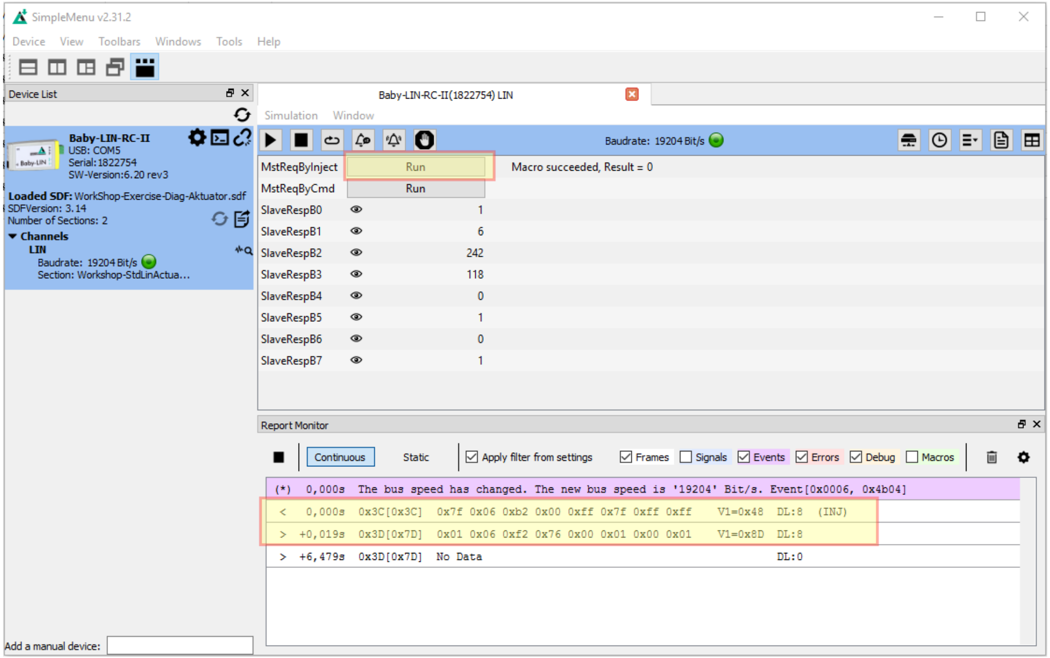
Alternativ Inject Command



Raw mode MacroCommand Inject

When sending a diagnostic request via Macro Command Inject you get the identical data.

In the Frame Monitor, Inject Frames are marked separately.

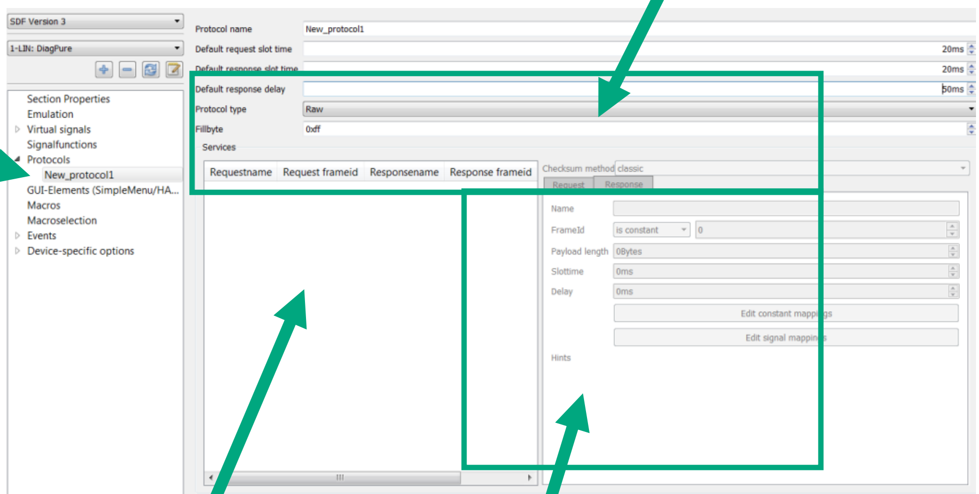


SDF Feature Protocol Service

New SDF Feature Protocols

2. setting basic protocol properties

1. add protocol here



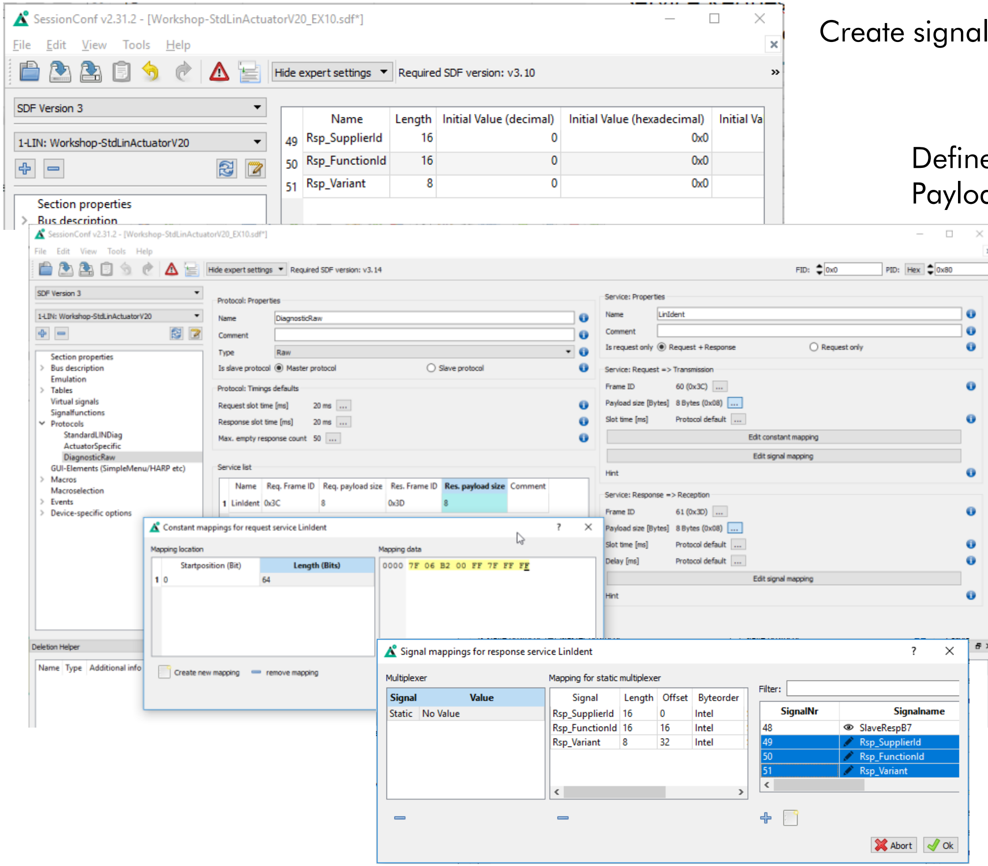
3. add service with request and optional response here

4. service details with constant data and/or signal mappings Only virtual signals can be mapped in a protocol service!

Raw mode MacroCommand Execute service

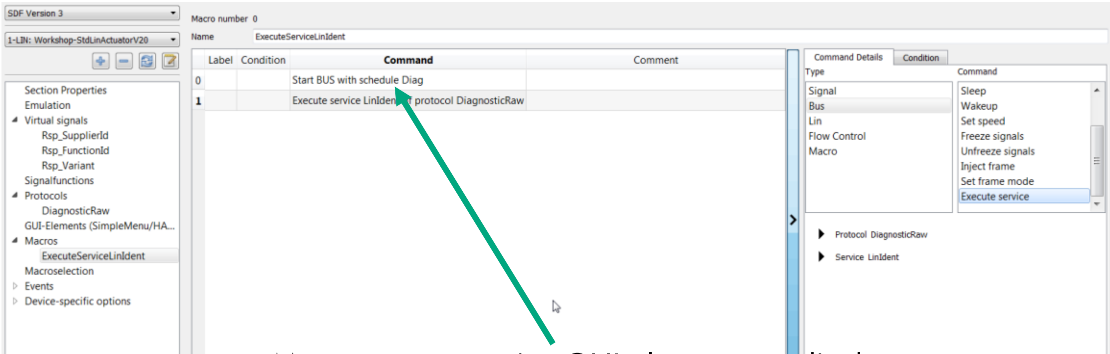
Create signals to record the response data

Define Service Request with Constant Payload

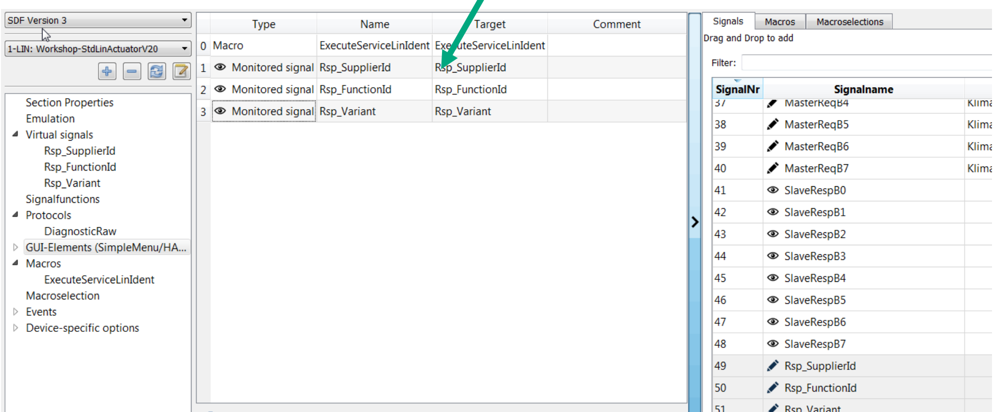


Define Service Response with Signal Assignments

Raw mode MacroCommand Execute service



Macro to run service GUI elements to display response data

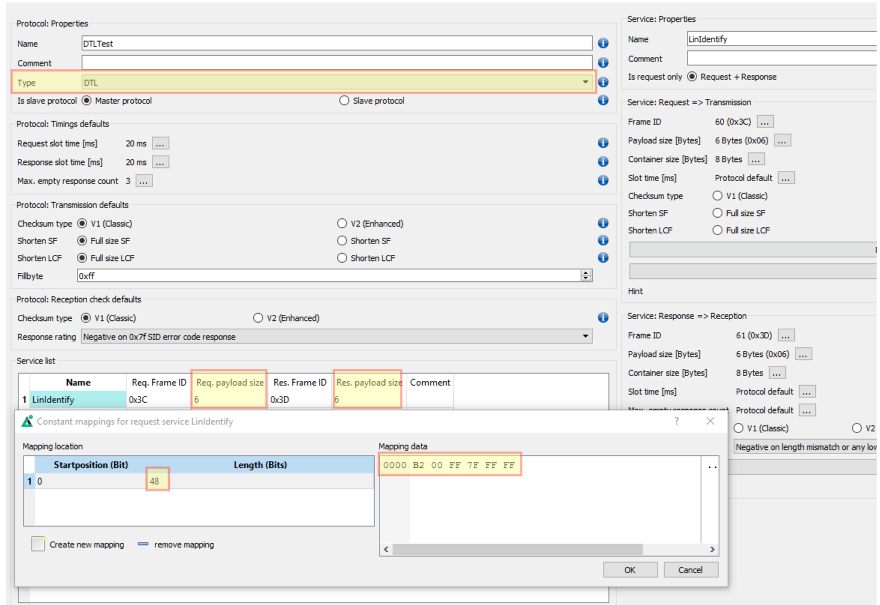


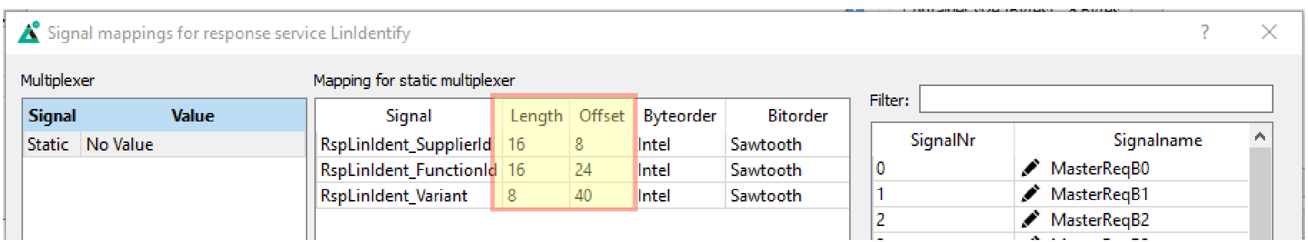
Cooked mode MacroCommand Execute service

We're now drawing up a protocol as DTL. (Copy and change the raw protocol)

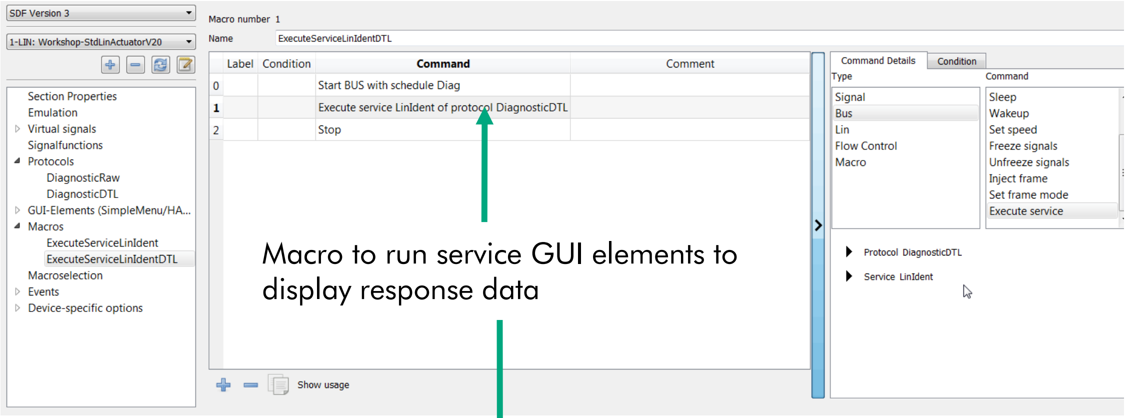
For DTL we need a virtual signal @@SYS_SERVICE_REQUEST_NAD

We set its default value to 0x7f (NAD Wildcard)

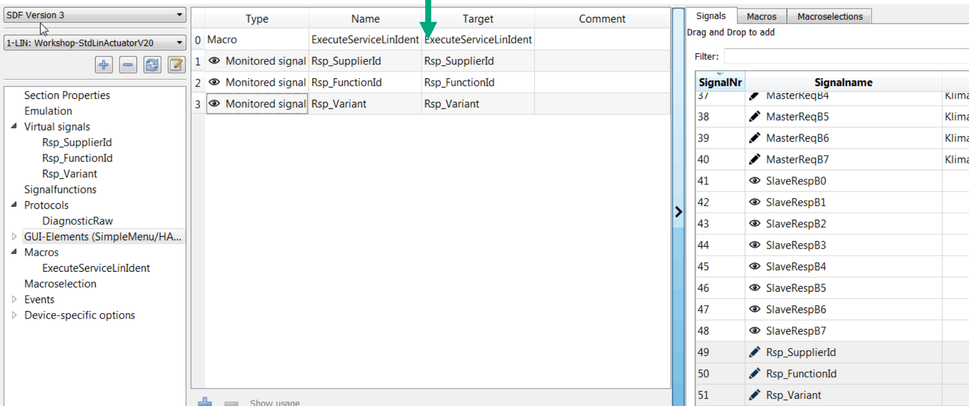




Cooked mode MacroCommand Execute service II



Macro to run service GUI elements to display response data



Advantages SDF Protocol Services

The use of protocol services offers many advantages, so that the older Macro commands SendMasterRequest or Inject will not be used anymore.

- Macro execution is synchronous to bus communication
  If a command "Execute Protocol Service" is finished, the frames were also on the bus.
- Any problems that occur when sending / receiving protocol frames are detected and reported back.
- Support of DTL/ISO TP Multiframe messages (Request and Response).
- With DTL/ISO-TP the negative return codes are evaluated and returned.
- A temporary NCR 0x78 is also handled correctly and the response request is repeated until a final positive or negative response is received.
- Return value of the Macro commands Execute Protocol Service allows error handling in the SDF.
- Access to the return value via the local signal __ResultLastMacroCommand.
- The protocol mechanism is not limited to diagnostic frames, it also allows the creation of applications with dynamic schedules, because then the frame dispatch is triggered by macro and not by schedule.
- The FrameId of a protocol service can also be defined via a signal.

Result Codes command Execute Protocol Services

Here is a list of the most common error codes that are available as __ResultLastMacroCommand after an Execute Protocol Service Command.

A complete list can be found in the respective user manual of the product.

Return code	Description (firmware version >= V.6.16)
0	The service was performed successfully
2	Service not successful, due to lack of LIN bus voltage
3	One slave did not respond in the required time
10	Too many services have been started (possibly from macros running in parallel).
14	The length of the slave response does not match the SDF protocol definition
20	Bus was not yet started
21	Bus level unexpected low
47	Bus level unexpected high
256…511	With DTL/ISO-TP the slave can give a negative response. This contains an 8 bit error code. This error code is returned here with an offset of 0x100. e.g. 0x12 => 0x112 => 274

Result Codes command Execute Protocol Services

In LIN Standard Diagnostics and UDS the same Negative Response Codes are usually applied in the negative response. (0x7F <SID> <ErrorCode=NRC>

Here is a list of the most important NCR's.

NRC	Meaning of NRC
0x10	generalReject
0x11	serviceNotSupported
0x12	subFunctionNotSupported
0x13	incorrectMessageLengthOrInvalidFormat
0x14	responseTooLong
0x21	busyRepeatRequest
0x22	conditionsNotCorrect
0x31	requestOutOfRange
0x33	securityAccessDenied
0x35	invalidKey

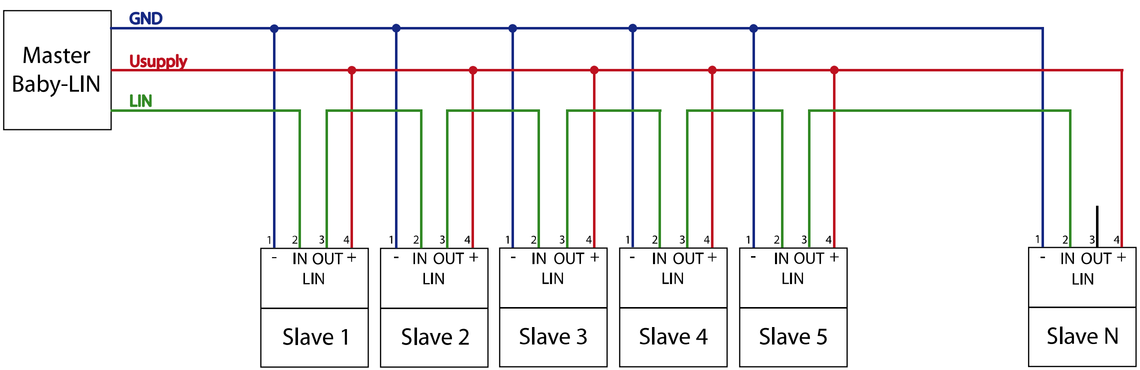
Diagnostic Frames-Auto-addressing

Auto addressing is needed if you have a LIN bus with several similar slaves, as is often the case with air-conditioning actuators or ambient LEDs.

Auto addressing uses different methods to make the identical slaves individually addressable for the master by a certain procedure, in order to be able to assign them a specific NAD and FrameId.

The 2 most common methods are the daisy chain and the bus shunt method.

Both methods use a bus wiring with a LIN-IN and LIN-Out pin at each slave.



Diagnostic Frames-Auto-addressing

	NAD Wildcard	PCI		Supplier ID Wildcard		SNPD sub function ID	SNPD method ID		
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x01	0x01	0xFF	**First MstReq to start Autoadressing**
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x01	**1. new Nad assignment**
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x02	**2. new Nad assignment**
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x03	**3. new Nad assignment**
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x04	**4. new Nad assignment**
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x03	0x01	0xFF	**Store new Nad's persistently**
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x04	0x01	0xFF	**Finalizing MstReq**

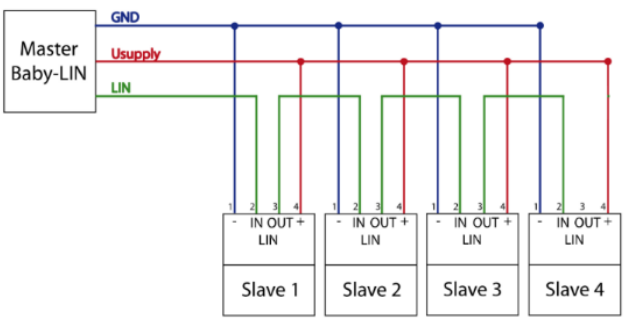
SNPD Subfunction	ID
All slaves go into the un-configured state.	0x01
Sets NAD of the next slave in the chain	0x02
All slaves save new NAD persistent	0x03
End of Auto Addressing for all Slaves. Slaves go into normal operation and use new NAD	0x04

SNPD Method	ID
Daisy Chain	0x01
Bus Shunt	0x02/0xF1

In practice, the bus shunt method is often identified by 0xF1 instead of 0x02!

Auto Addressing Daisy Chain Method

	NAD Wildcard	PCI		Supplier ID Wildcard		SNPD sub function ID	SNPD method ID	
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x01	0x01	0xFF
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x01
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x02
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x03
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x02	0x01	0x04
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x03	0x01	0xFF
0x3C	0x7F	0x06	0xB5	0xff	0x7f	0x04	0x01	0xFF



Nad 0x01 | Nad 0x02 | Nad 0x03 | Nad 0x04

Daisy chain is based on a switch integrated in the slave between the LIN-IN and the LIN-Out pin.

Cmd 0x01 0xff opens the switch for all slaves.

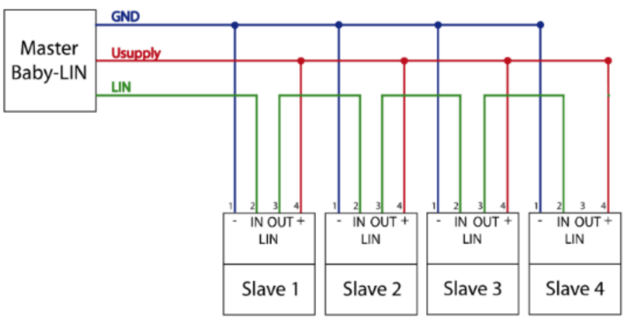
Thus only Slav1 is connected directly to the master.CMD 0x02 0x01 gives this slave a new NAD. Slave closes its switch and now waits for the 0x04 0xff command.

Next 0x02 0x02 CMD now goes to the 2nd slave which is now connected and so on.

Daisy chain mode: The slaves receive the distributed NAD's in the order from the next to the most distant slave.

Auto Addressing Bus Shunt Method

	NAD Wildcard	PCI		Supplier ID Wildcard		SNPD sub function ID	SNPD method ID	
0x3C	0x7F	0x06	0xB5	0xFF	0x7F	0x01	0xF1	0xFF
0x3C	0x7F	0x06	0xB5	0xFF	0x7F	0x02	0xF1	0x01
0x3C	0x7F	0x06	0xB5	0xFF	0x7F	0x02	0xF1	0x02
0x3C	0x7F	0x06	0xB5	0xFF	0x7f	0x02	0xF1	0x03
0x3C	0x7F	0x06	0xB5	0xFF	0x7F	0x02	0xF1	0x04
0x3C	0x7F	0x06	0xB5	0xFF	0x7F	0x03	0xF1	0xFF
0x3C	0x7F	0x06	0xB5	0xFF	0x7F	0x04	0xF1	0xFF



Nad 0x04	Nad 0x03	Nad 0x02	Nad 0x01

The bus shunt method requires a more complex hardware in the slave, consisting of switchable current sources and pull-up resistors.

These are controlled in the break signal phase of the subfunction 0x02 frames by the slave in such a way that at the end of the break, the most distant slave recognizes that he is the one.

Thus the slave furthest away from the master knows that it is to take over the NAD contained in this frame and then no longer participates in the further sequence.

At the break of the next 0x02 frame, the slave now furthest away will recognize its position and take over the NAD accordingly.

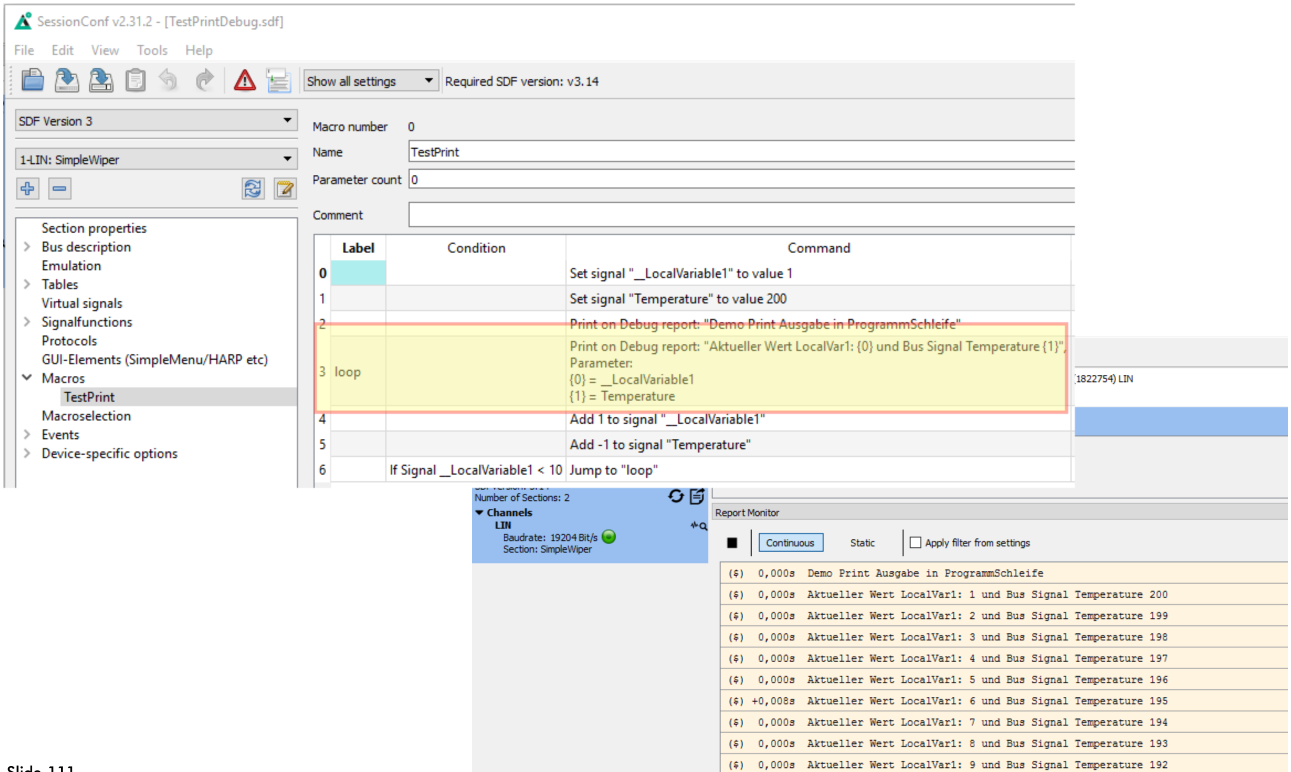
This will be repeated until all slaves were connected.

Shunt method: Here the NADs distributed in the auto addressing sequence are assigned from the farthest to the next slave, i.e. exactly the opposite as with the daisy chain method.

Macro programming - debugging

When programming more complex operations in macros, it is helpful to be able to track the operation of a macro to find programming or operation errors.

This is where the Macro Command Macro Print helps (example SDF TestPrintDebug.sdf).



Macro programming – error handling

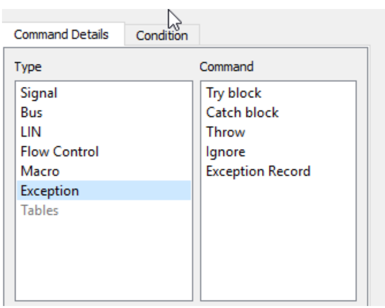
Even in a correctly programmed sequence, errors can occur during execution, for example because a defective test object does not respond at all. A carefully developed SDF application should be able to detect and handle these errors.

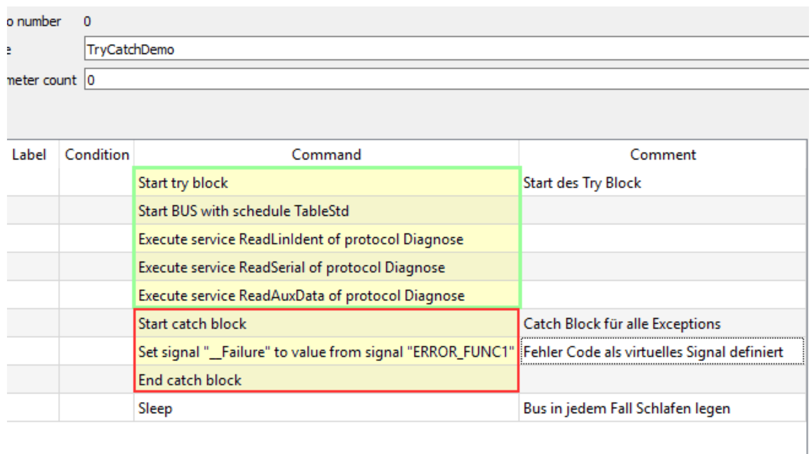
The result values of the individual Macro Commands (_ResultLastMacroCommand) already show whether a command worked or not. The prerequisite for this is that the command, if selectable, is executed blocking.

A TRY-CATCH mechanism has been implemented to avoid having to introduce an error handling after every command in a macro.

Every error in the try block (green marking) automatically branches to the catch block (red).

Without errors the catch block is skipped.



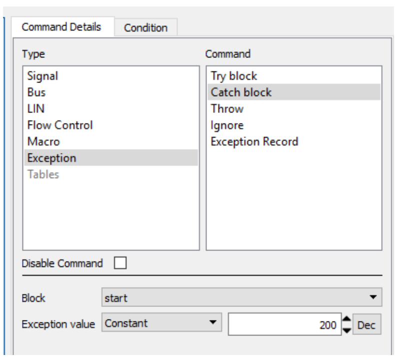


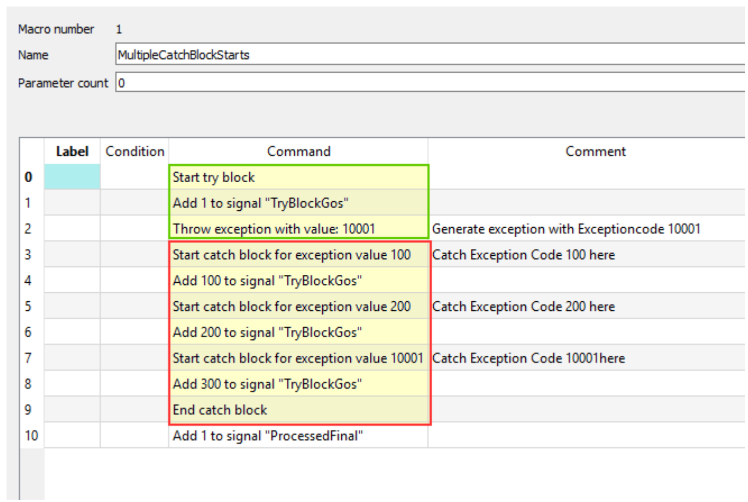
Macro programming – Try-Catch

You can specify several Catch Block Start Commands one after the other, so you can define areas in the Catch Block that are responsible for certain exceptions.

Therefore there is the option to define an Exception Value as filter in the Catchblock Start Command.

If two Catch Block Start Commands are directly behind each other, the area after the second CatchStart is executed for both Exception Values.





Macro number 1

Name MultipleCatchBlockStarts

Parameter count 0

	Label	Condition	Command	Comment
0			Start try block	
1			Add 1 to signal "TryBlockGos"	
2			Throw exception with value: 10001	Generate exception with Exceptioncode 10001
3			Start catch block for exception value 100	Catch Exception Code 100 here
4			Add 100 to signal "TryBlockGos"	
5			Start catch block for exception value 200	Catch Exception Code 200 here
6			Add 200 to signal "TryBlockGos"	
7			Start catch block for exception value 10001	Catch Exception Code 10001here
8			Add 300 to signal "TryBlockGos"	
9			End catch block	
10			Add 1 to signal "ProcessedFinal"	

Macro programming– Try-Catch

The Try-Catch command can also be used as a switch case construct, as known from other programming languages.

Macro number 2

Name: SwitchCaseDemo

Parameter count: 0

	Label	Condition	Command	Comment
0			Throw exception with value of signal: SwitchVar (55)	Entspricht dem Switch
1			Start catch block for exception value 1	Case 1:
2			Set signal "CaseValue" to value 111	
3			Start catch block for exception value 0	Case 0:
4			Set signal "CaseValue" to value 0	
5			Start catch block for exception value 2	Case 2:
6			Set signal "CaseValue" to value 222	
7			Start catch block for exception value 3	Case 3:
8			Set signal "CaseValue" to value 333	
9			Start catch block for exception value 4	Case 4:
10			Set signal "CaseValue" to value 444	
11			Start catch block for exception value 5	Case 5
12			Start catch block for exception value 6	Case 6:
13			Set signal "CaseValue" to value 6666	exececuted for Case 5 und Case 6
14			Start catch block	Default Zweig
15			Set signal "TryBlockGos" to value 9999	
16			End catch block	
17			Add 1 to signal "ProcessedFinal"	

The Throw command can also be used outside a Try Block to raise an exception.

Here it replaces the switch statement.

The catch block implements the different case branches.

The last catch block start without exception value serves as default branch and catches all switch values that are not handled by a case branch.

MB-II Log functions

After installing a micro-SD card, the MB-II can create log files which can be accessed via the integrated website of the device.

There are 2 application variants.

A.) Continuous logging

By creating and uploading a log configuration, logging is activated and data is permanently written to the log file as specified in the log configuration file.

B.) SDF controlled logging

Specially formatted macro print commands to control logging from the SDF.

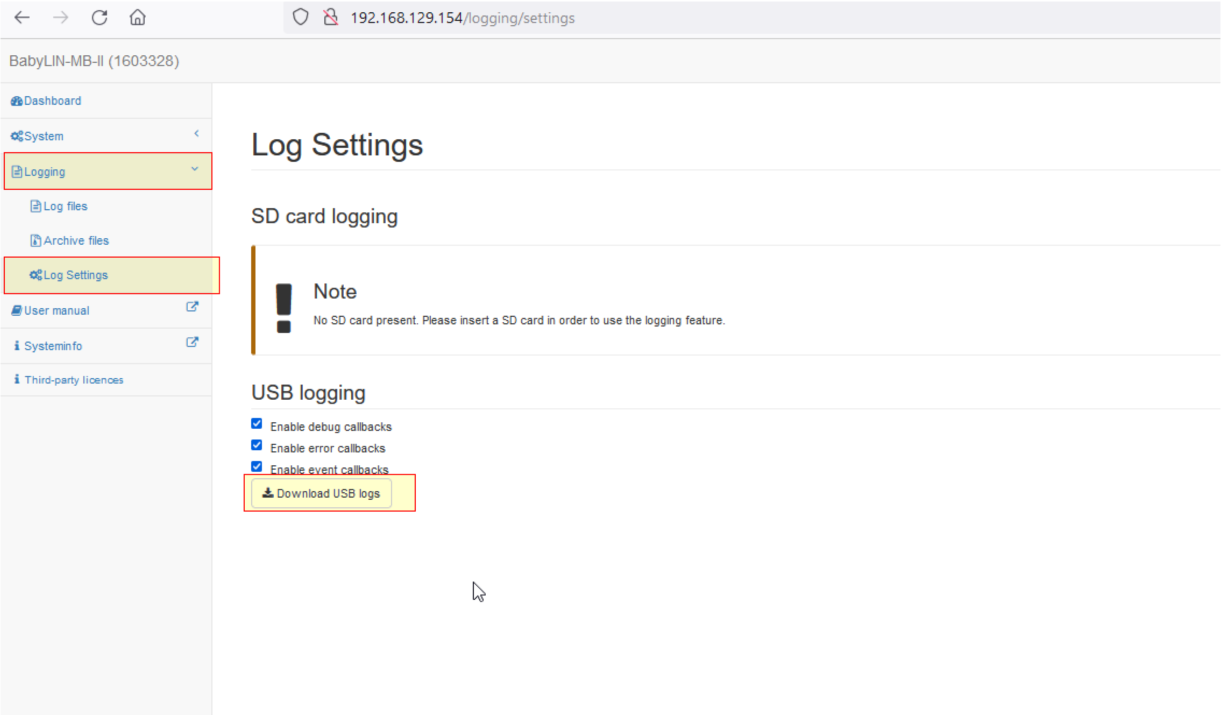
Open log file, close and discard.

The file name can be generated from the SDF.

For example, you can define the creation of a separate log file for each inspected part, and define the serial number read out as the file name.

A.) USB-Logging without USB stick

Logging is also started by creating and uploading a log file. The special feature, however, is that the USB logs can be downloaded directly. This means that the logging function can be implemented on existing devices even though no SD card is installed.



Stay up-to-date

The Baby-LIN firmware and the LINWorks software are constantly being further developed.

Both can be obtained free of charge in the current version directly from our customer portal.

(https://www.lipowsky.de/downloads/)

For the firmware update of the Baby-LIN devices the application **blprog.exe** is included in the download package.

This application takes over the update process largely automatically if the files have been unpacked from the ZIP into a separate directory.

New unit variants will be added in 2023

- New product base for Baby-LIN-III, Baby-LIN-RC-III
- First baby CAN device planned as entry-level variant

If you have any questions or suggestions, please feel free to contact us at any time by phone: 06151-93591-0 or by email: info@lipowsky.de

We are also happy to visit your computer via TeamViewer to support you on site in case of problems.

Baby-LIN feature matrix

Features	Baby-LIN-II	Baby-LIN-RC-II	Baby-LIN-RM-III	HARP-5	Baby-LIN-MB-II
LINWorks compatible	✓	✓	✓	✓	✓
SDF transfer	USB 2.0	USB 2.0	USB 2.0	USB 2.0 SDHC card	Ethernet web interface RS-232
LIN-Bus Interfaces	1 x LIN	1 x LIN	1 x LIN	1 x LIN	1 x LIN
Optional LIN-Bus Interfaces	✗	✗	1 x LIN	1 x LIN	5 x LIN
Optional CAN-Bus Interfaces Optional CAN-FD Interfaces	✗	✗	1 x CAN HS/FD 1 x CAN-LS/HS/FD	1 x CAN HS 1 x CAN LS	1 x CAN-HS 2 x MIF-CAN-FD
Digital Inputs/ Digital Outputs	✗	✗ 1 x Digital Output	8 x Digital Input 6 x Digital Output	1 x Digital Input 1 x Digital Output	1 x Digital Input 2 x Digital Output
Special features	✗	Option for SD Card support	Digital In – and outputs, analogue inputs,	LIN voltage switch, 12 V node supply generator	Logging on internal micro-SD card, logdata accessed by device webpage
Typical applications	PC-Interface	PC-Interface and hand-held commander	PLC-coupling or stand-alone bus simulator	Hand-held control with bus data display	PC/PLC coupling via LAN or RS-232