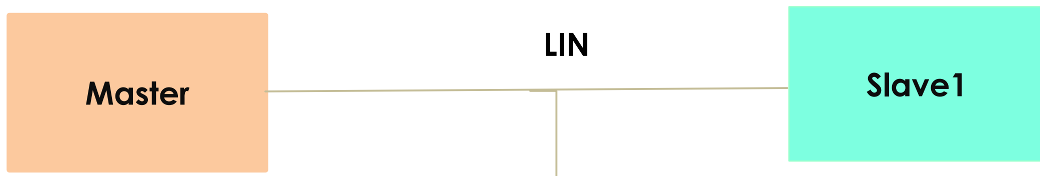
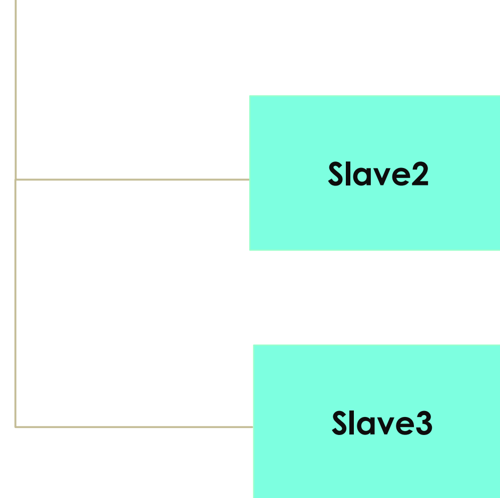
LIN Basics

Lipowsky Industrie-Elektronik GmbH

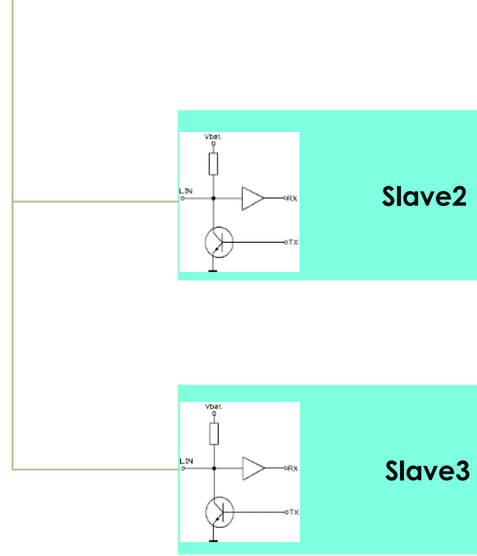
LIN-Basics



- 1 wire bus (+ Gnd and Vbat)
- Master / Slave concept
- Only one master at a time
- Master allocates the bus, nobody speaks without permission
- Bus speed 9600...19200 Bit/s
- Often there are several LIN buses in one vehicle, e.g. each door can have its own LIN bus, further buses for air conditioning actuators and seat adjustment can be available.



LIN Single Wire Hardware

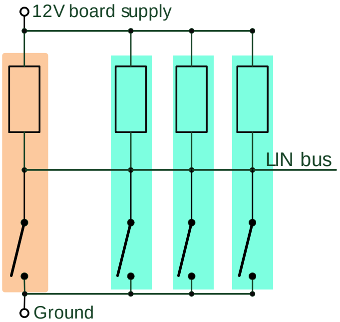


- Bi-directional communication via a line through open-collector output stages.
- The necessary pull-up is distributed over the nodes:

Master Pull-Up	1 Kiloohm
Slave Pull-Up	30 Kiloohm

- In order to avoid collisions right from the start, LIN works with a time slot method.
- The master assigns the bus itself or a node for a defined time by sending a certain feature.
- During this time the bus is then available to this node, which can place data on the bus.

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 (Dominant State), only if all output switches are open, the bus will be pulled up to its high level (Recessive State).

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 only the low level is determined by an active switch, the rising edge of the LIN bus signal also depends on the resulting value of the total pull-up resistance.

The lower the pull-up resistance, the steeper the rising edge and vice versa.

Too steep edges can lead to EMC problems and too flat edges can lead to misinterpretation by the UART. Therefore a correctly dimensioned pull-up resistor is very important!

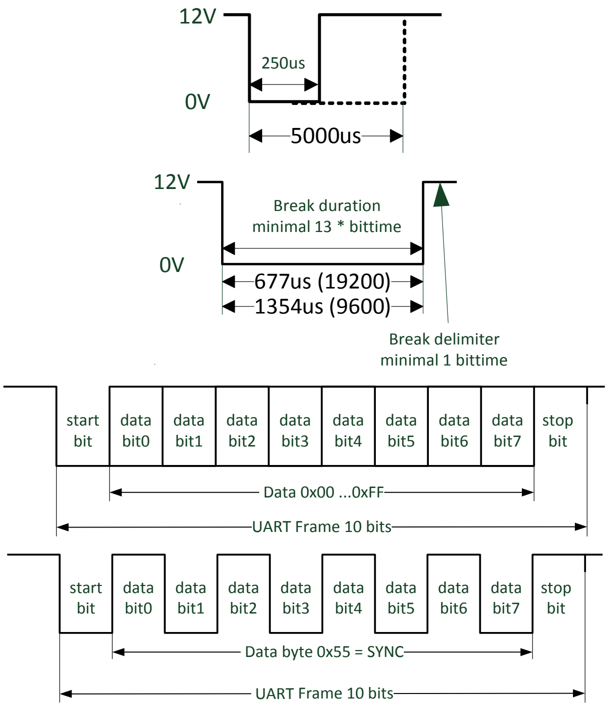
The LIN bus has only 2 states:

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 communication primitives



There are 3 basic signal patterns on the LIN bus:

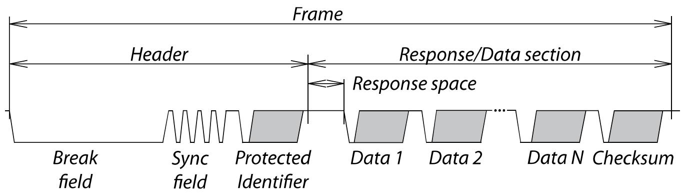
1. Wake up Event
Low level pulse with 250us…5 ms length
Slave recognition Low pulse >= 150 us, Slave should be able to process commands 100 ms after the rising edge of the bus.

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:

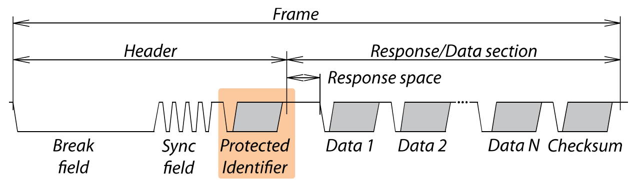
➢Break field	Indicates the beginning of a new frame, at least 13 bit times long, in order to be able to distinguish it reliably from all other characters.
➢Sync field	Allows the resynchronization of slave nodes with imprecise clock sources by measuring the bit times and reconfiguring the UART baud rate. 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 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.

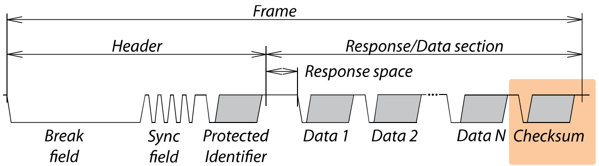
It is 8 bits in size, but 2 bits of it are used as parity bits, leaving only 6 bits for frame identification. Thus there are only 64 different frames on a LIN bus.

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 Bus Hardware

Most LIN nodes contain the following 2 components:

- Microcontroller with integrated UART
- LIN transceiver

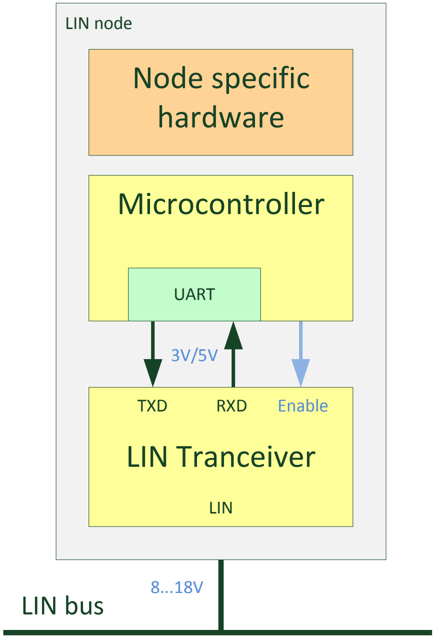
The **UART** converts data bytes into asynchronous serial patterns for transmission and decodes data bytes from the received serial data stream.

It also generates break and wake-up signal patterns; this can be implemented either by special LIN functions of the UART or 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.

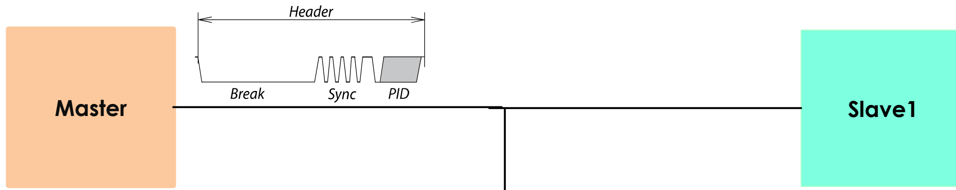
Further functions of a typical LIN transceiver are:

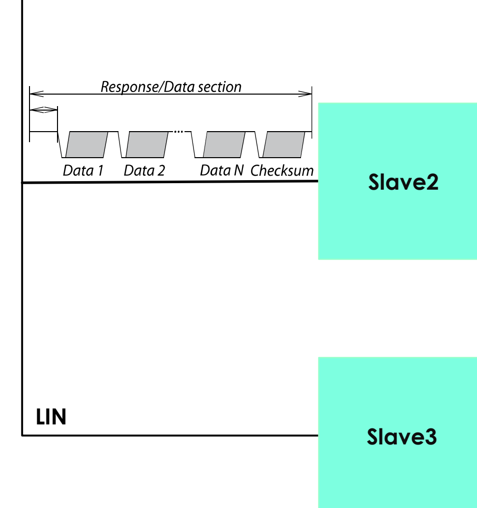
- Timeout Monitoring of the dominant level
- Slope control of the signal edges
- Switching to a high-speed mode to enable baud rates greater than 20 Kbit (e.g. ECU flashing)



2nd generation Baby-LIN systems use NXP MC33662 LIN transceiver

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.

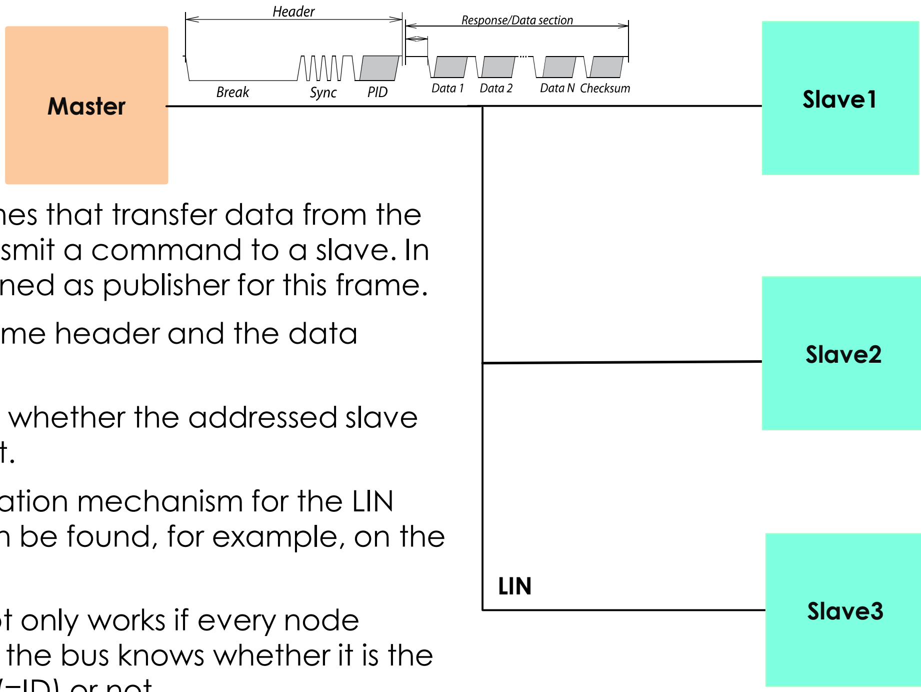
If a node determines that it is the publisher for this frame ID, it places the data for this frame on the bus.

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 mapping)?
- In which order should the frames appear on the bus (Schedule Table)?

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,Slave2Se
nsor;

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,0x00,"Off" ;
                logical_value,0x01,"Speed1" ;
                logical_value,0x02,"Speed2" ;
                logical_value,0x03,"Interval" ;}
   EncodingTemp {
                physical_value,0,253,0.8,-
                35,"degrees C" ;
                logical_value,0xFE,"Signal not
                supported" ;
                logical_value,0xFF,"Signal not
                available" ;}
}
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)

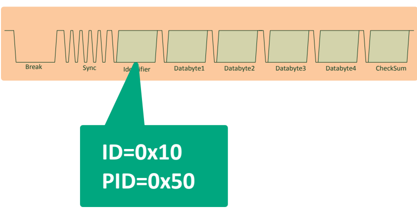
Databyte1.bit 0…7	message counter
Databyte2.bit 0	IgnitionOn (Klemme15)
Databyte2.bit 1…3	wiper speed

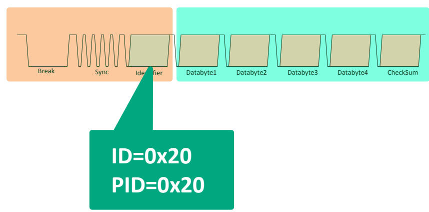
Frame with ID 0x20 has 4 data bytes
Publisher = Slave1Motor

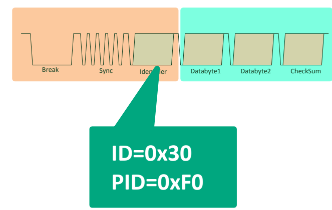
Databyte1.bit 0	wiper active
Databyte1.bit 1	park position
Databyte2.bit 0…7	CycleCounter LSB
Databyte3.bit 0…7	CycleCounter MSB

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

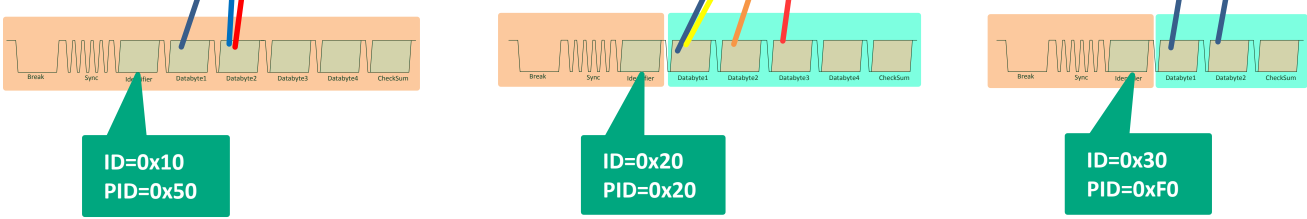
LDF definition:

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



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


```
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 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.

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.

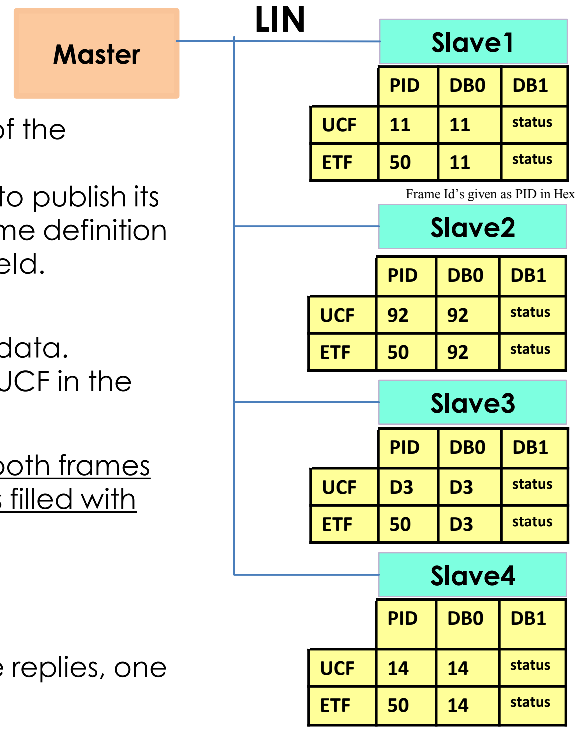
With UCF, the slave always sends the data.
With ETF, the slave only sends data if there is changed data.
In addition, the slave places the PID of the associated UCF in the first data byte.

<u>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.</u>

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

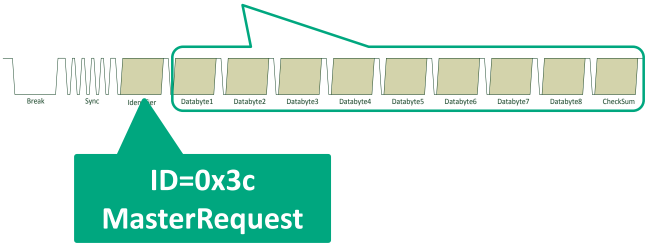
Collision triggers switch to Collision Schedule Table

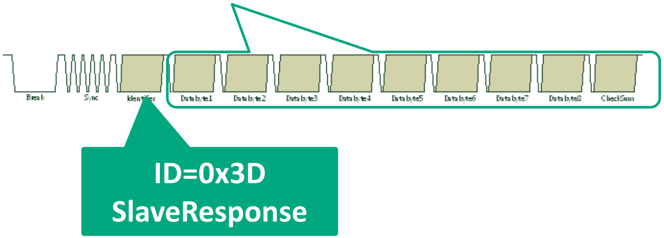
Timestamp	FrameId	FrameData	Checksum	
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 0x10	0x5e	V2 OK
+20	0x31 [0xb1]	0x21 0x07 0x00	0x26	V2 OK
+20	0x10 [0x50]			No Response
+10	0x00 [0x80]	0xf0 0x64 0x32 0x99 0x0c	0x52	V2 OK
+20	0x30 [0xf0]	0xa0 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

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:

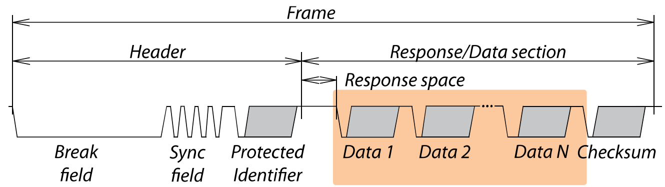
- **Keyword 2000 Protocol** (ISO 14230 -1 to 4)
- **UDS** (Unified Diagnostic Services) (ISO 14229-1:2013)

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).

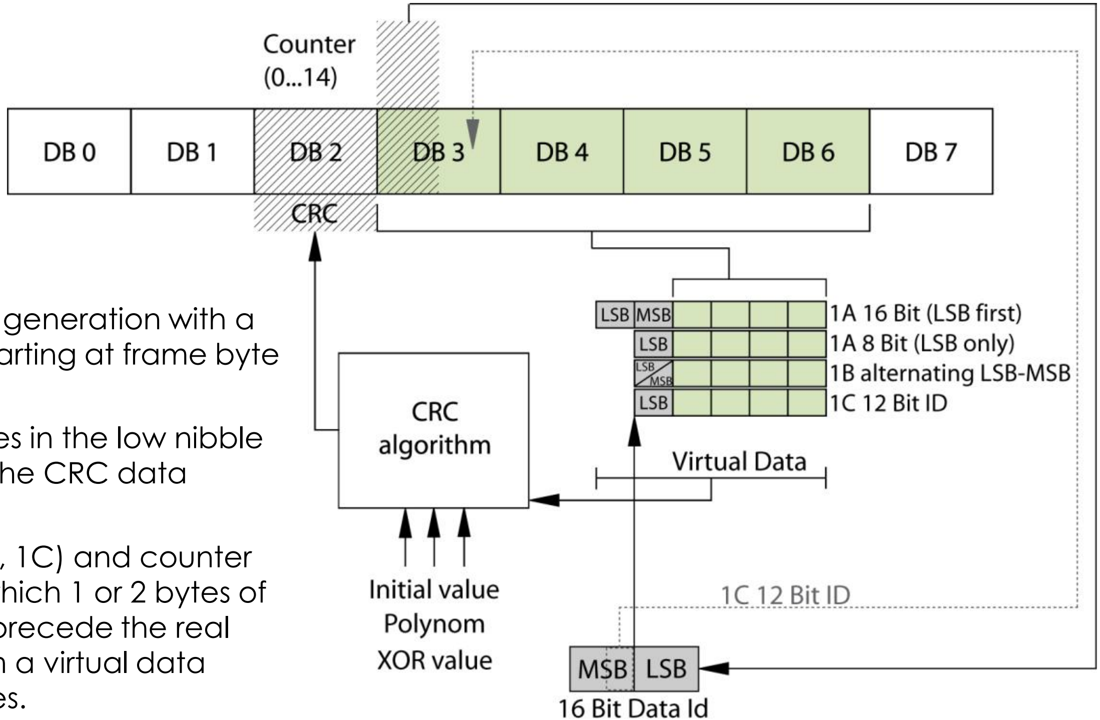
In addition to numerous proprietary implementations, a standard according to the Autosar E2E Specification is currently establishing itself, whereby there are several profiles here. However, first implementations deviating from the standard have already been viewed (e.g. BMW).

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



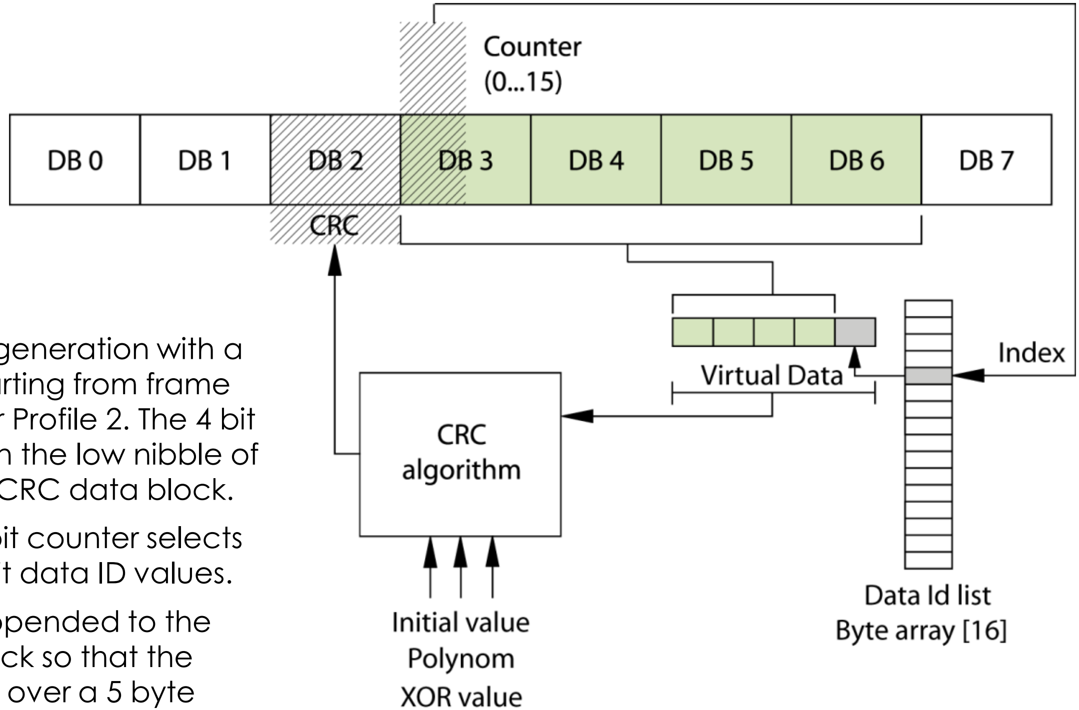
Example of a CRC generation with a CRC data block starting at frame byte DB3.

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.

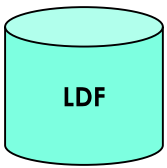
The LINWorks extension for importing new description formats such as Fibex or ARXML is planned for the future.

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

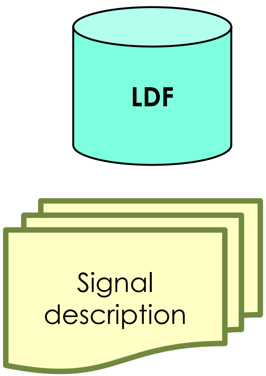
Operate LIN-node for

- 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

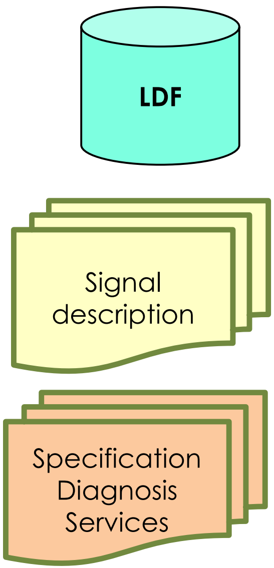
Operate LIN-node for

- 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

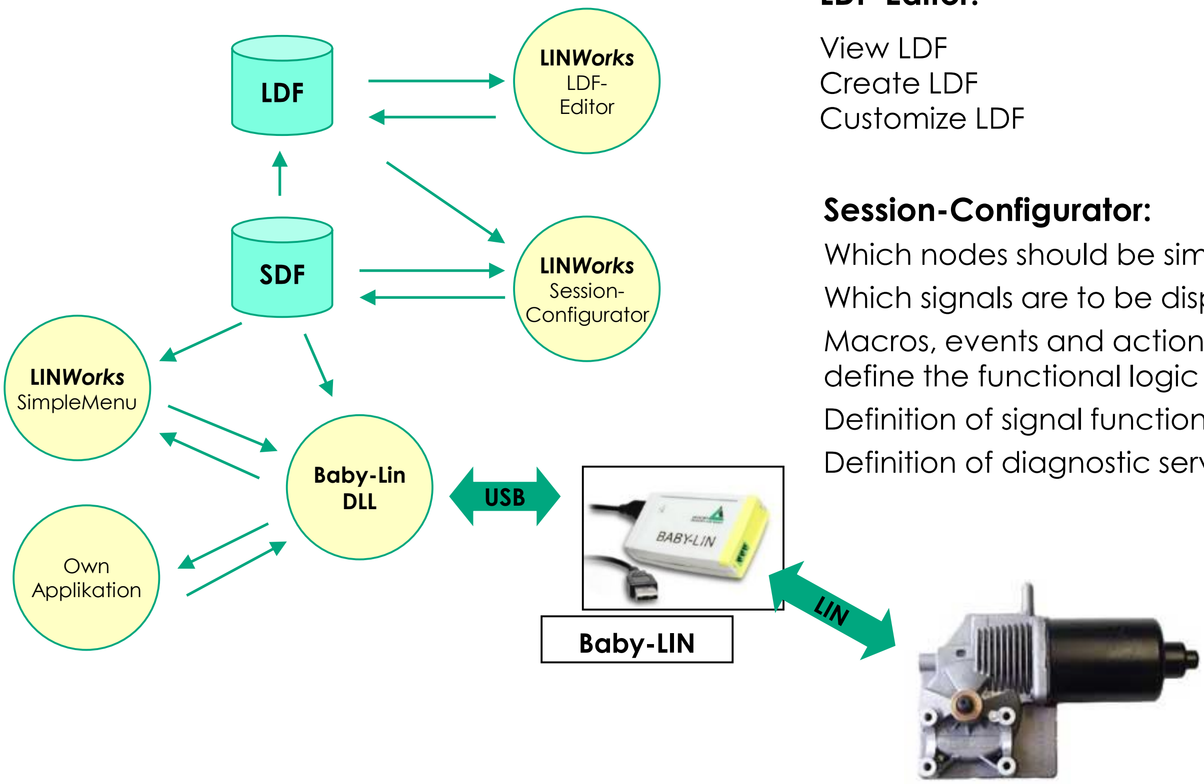
USB, Digital IO, RS-232, LAN connection via DLL or ASCII API

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?

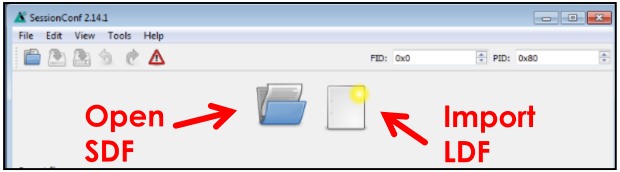
Which signals are to be displayed?

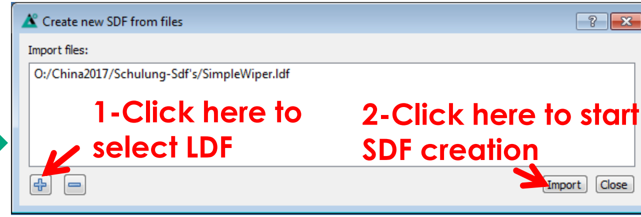
Macros, events and actions to define the functional logic

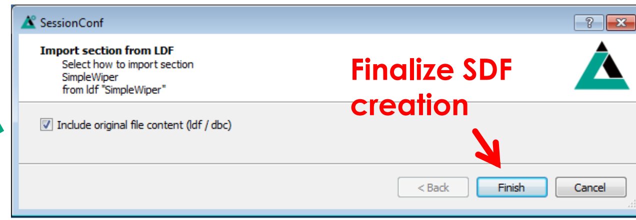
Definition of signal functions

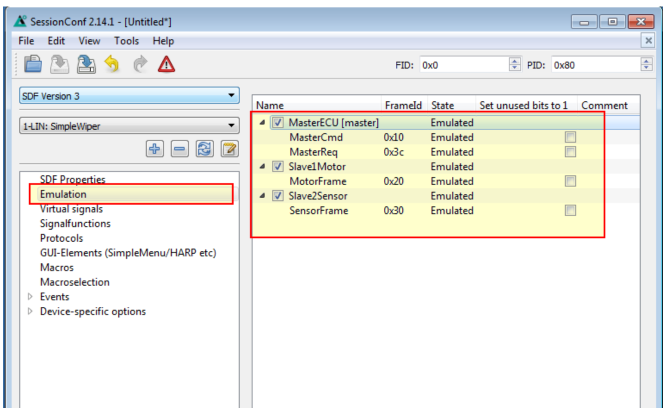
Definition of diagnostic services

LINWorks SessionConf





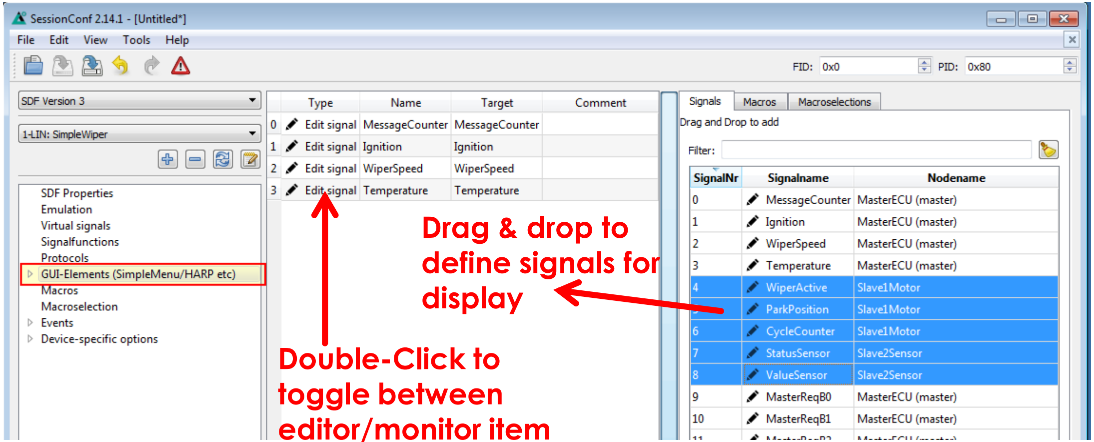




Minimal setup:

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

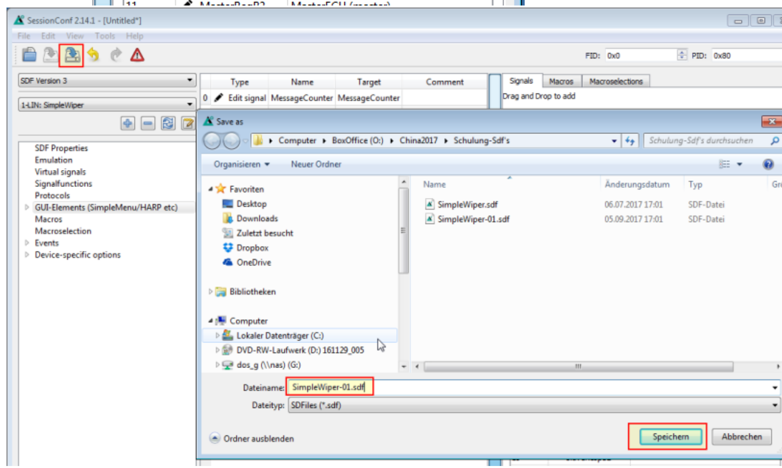
LINWorks SessionConf



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

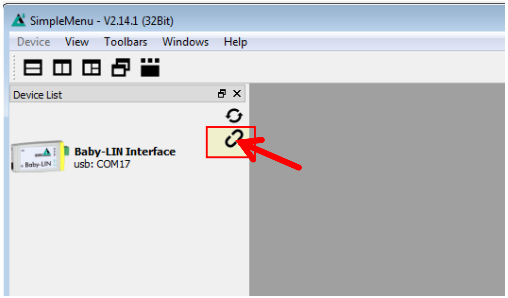
Save as SDF file

=> The first SDF is created!

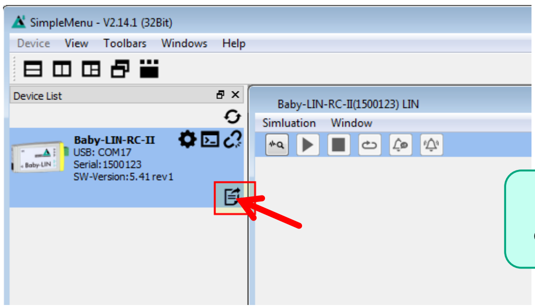


LINWorks Simple Menu

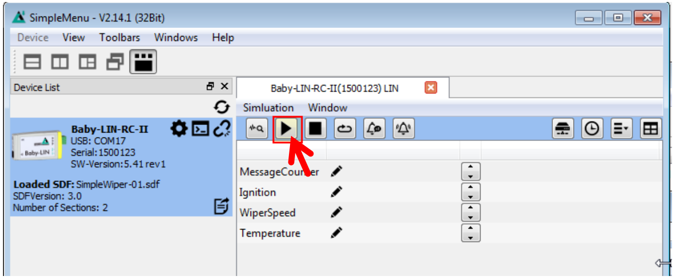
Step 1: Open SimpleMenu application
Step 2: Connect with Baby-LIN



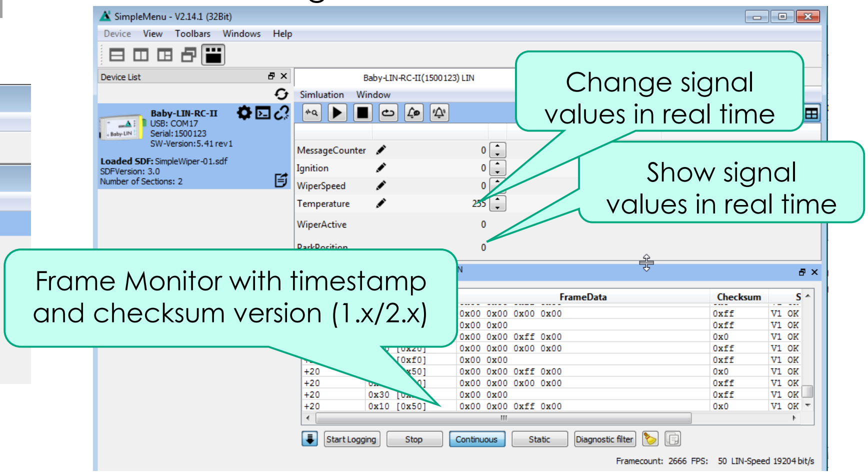
Step 3: Load SDF into Baby-LIN



Step 4: Start simulation



LIN-Bus running!



LINWorks Simple Menu

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.

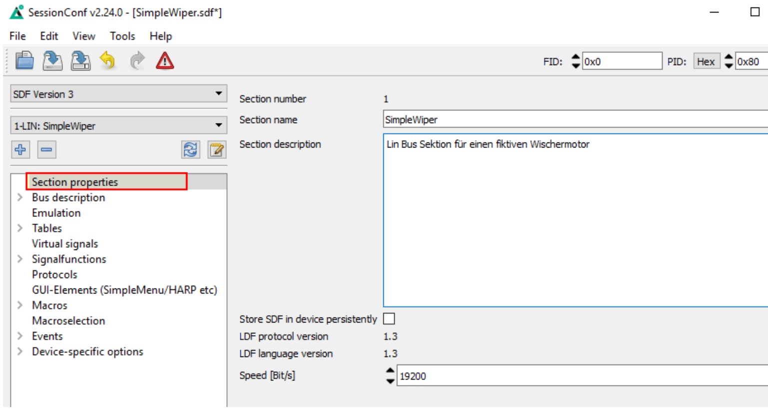
Emulate	NodeNr	Nodename
☑	0	MasterECU (master)
☑	1	Slave1Motor
☑	2	Slave2Sensor

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.

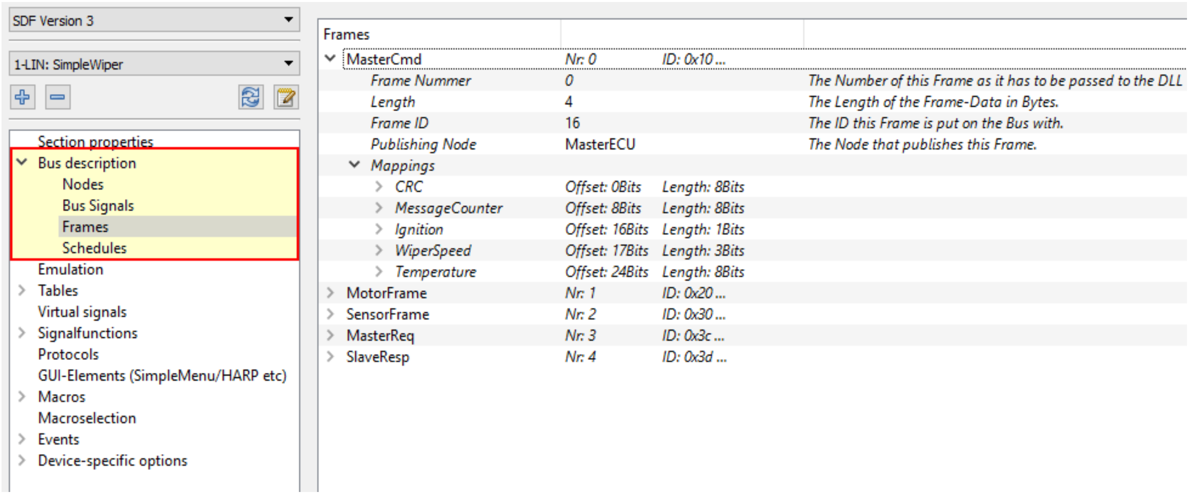
The baud rate must be entered here in a CAN section, since it cannot be taken over from the DBC and is therefore set to 0 after the DBC import.

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.



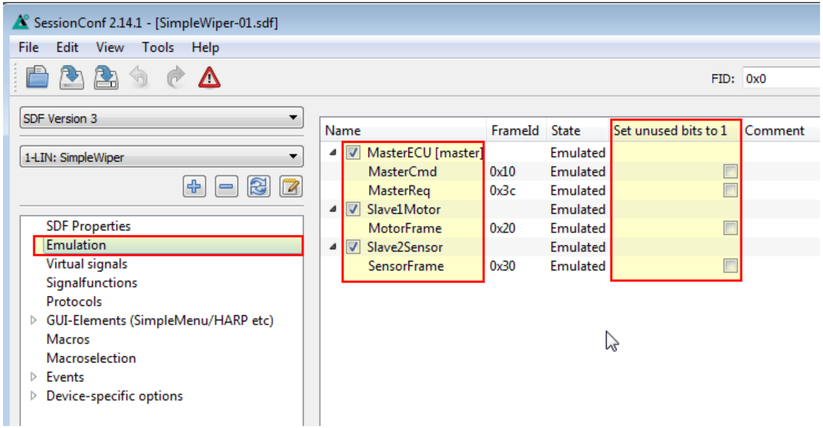
SessionConf - Emulation Setup

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.

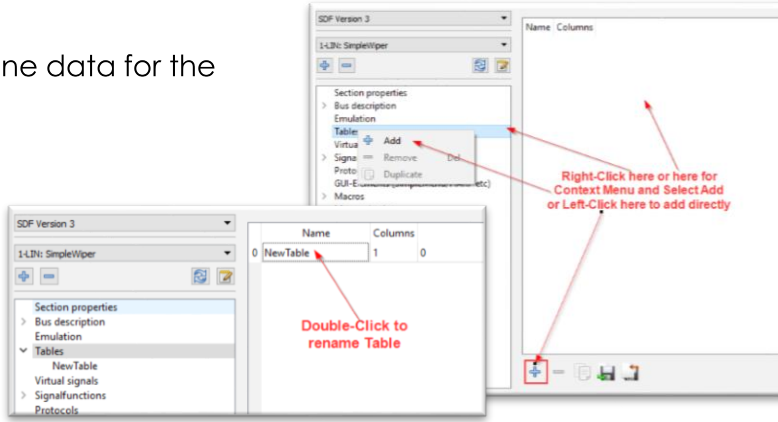
Session Conf – Tables

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



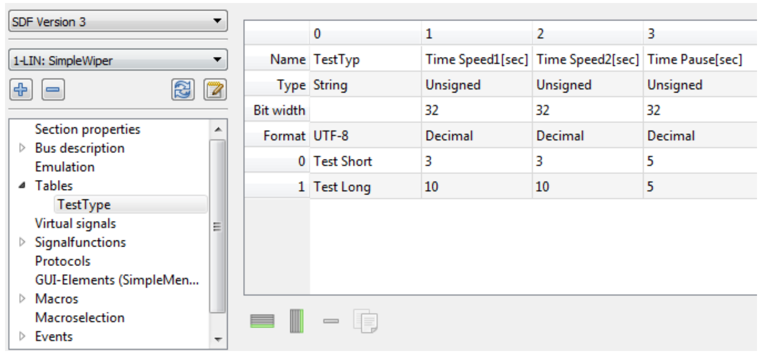
A column can contain text (String) or numbers (Signed/Unsigned Integer).

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

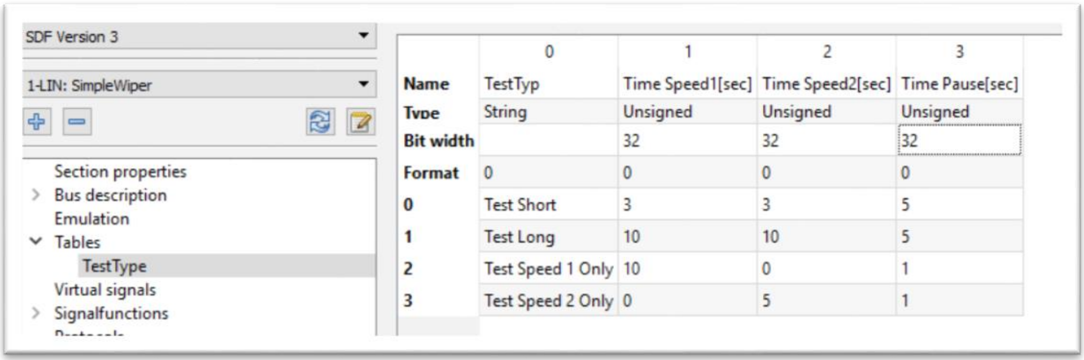
Here is an example table for defining test variants for a wiper endurance run.



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

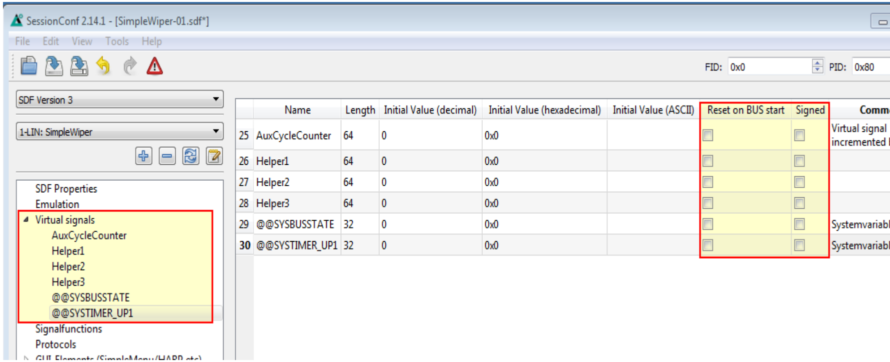
Allows to emulate the behavior of SDF-V2 files.

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.



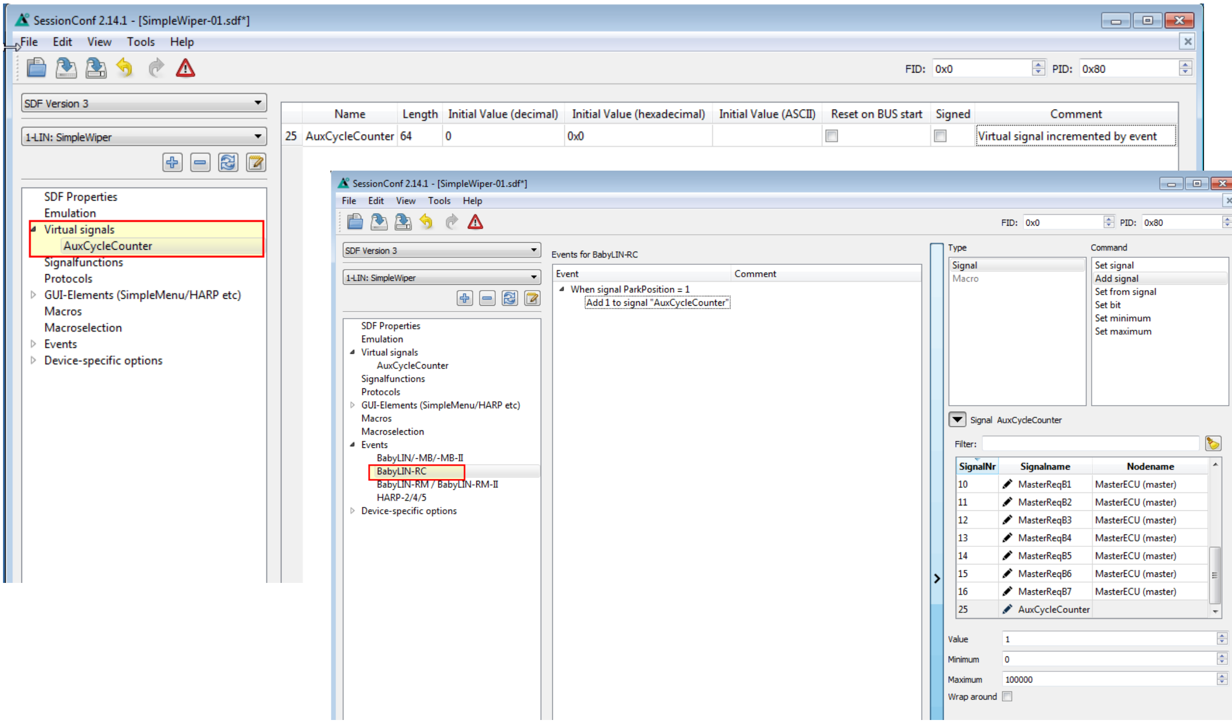
The comment column allows you to enter notes and explanations about the variable.

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 – Virtual signals system variables

Special virtual signals => system variables

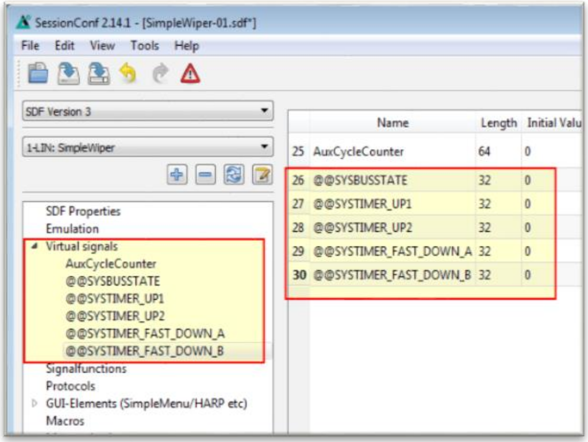
There are virtual signals with reserved names.

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 variables 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 equal to 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

Weitere @@SYSxxx Systemvariablen zur I/O Kontrolle

@@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 variable 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 variable allows to define the fullscale value (corresponding to 100%). By default, this is set to 200 by the system.

Man kann zum Beispiel die @@SYSDIGIN1…x und die @@SYSPWMIN1..2 Systemvariable sehr gut mit einem ONCHANGE Event kombinieren.

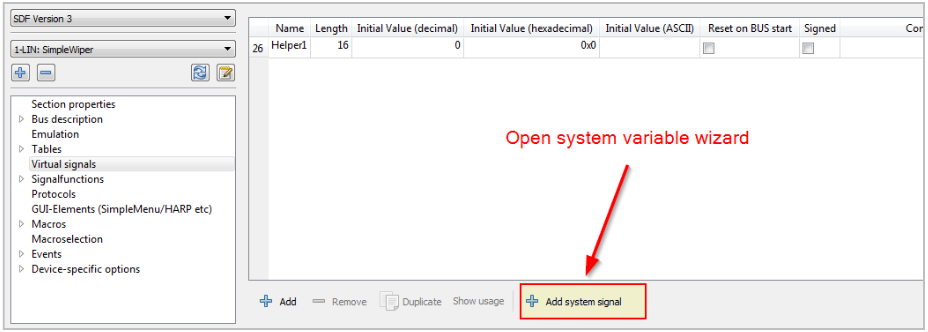
So kann man zum Beispiel den Wert eines digitalen Eingangs mit nur einer Eventdefinition auf ein LIN Bus Signal übertragen.

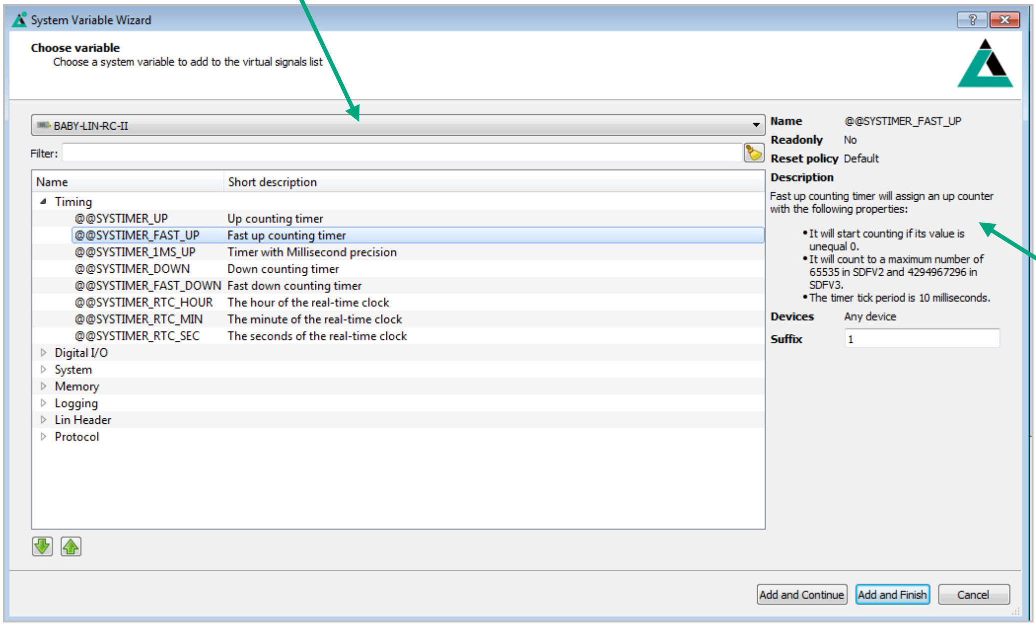
Damit man sich diese reservierten Namen für die Systemvariablen und deren Schreibweise nicht alle merken muss, gibt es im SessionConf einen System Variablen Wizard.

SessionConf – System variables wizard

Easy creation of system variables with the wizard.

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





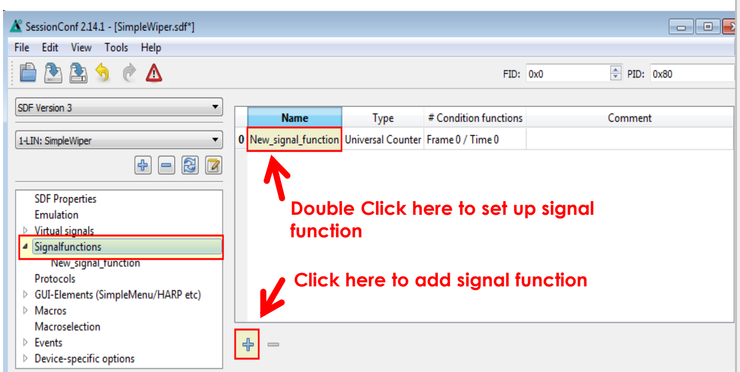
Information on the function of the system variable in focus

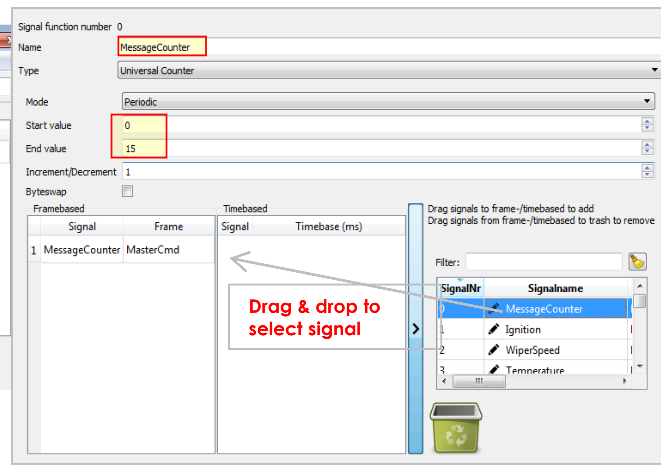
Signal functions - Counter

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

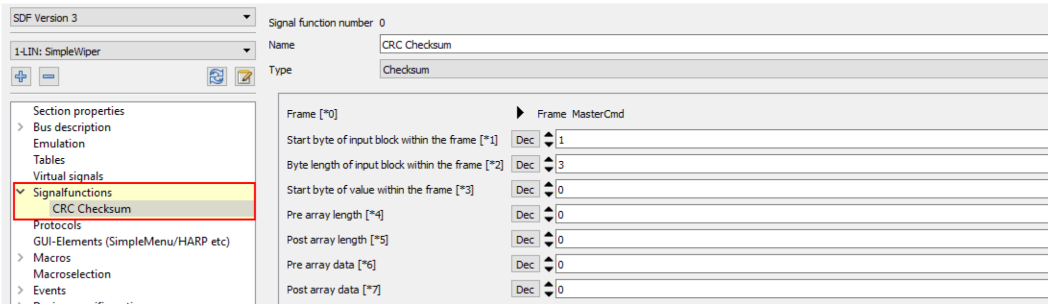
- **Checksum 8 Bit Modulo** adds all bytes belonging to the data block and uses the LSB of the sum.
- **CRC-8** forms an 8 bit CRC over the data block according to the specified parameters
- **CRC-16** forms a 16 bit CRC via the data block according to the specified parameters.
- **XOR** links all bytes 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).



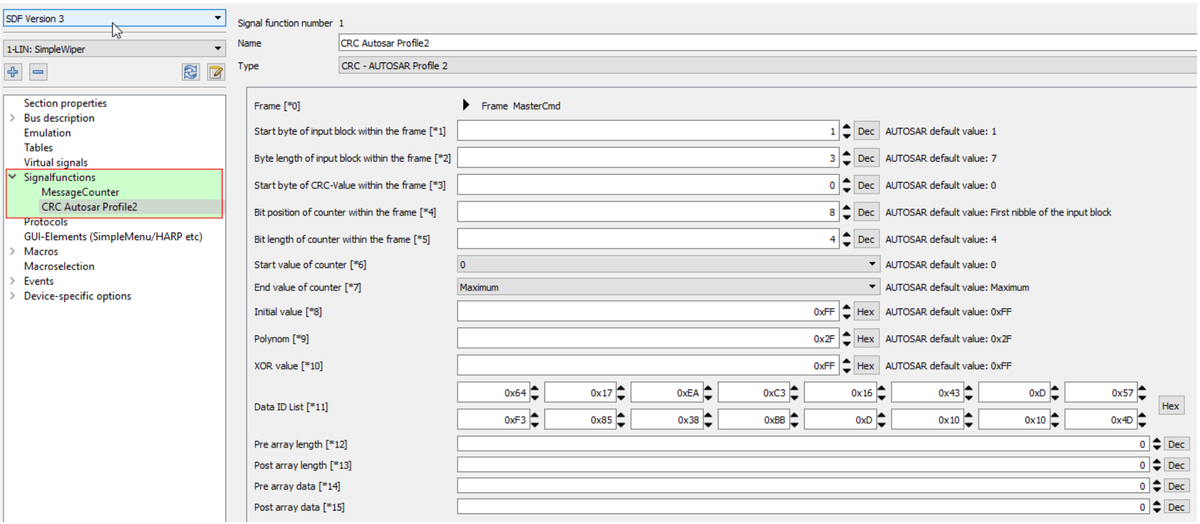
The parameters *4 to *7 define an optional prepend and postpend buffer with up to 8 byte values, which are then prepended or appended to the data of the real frame before the calculation.

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.

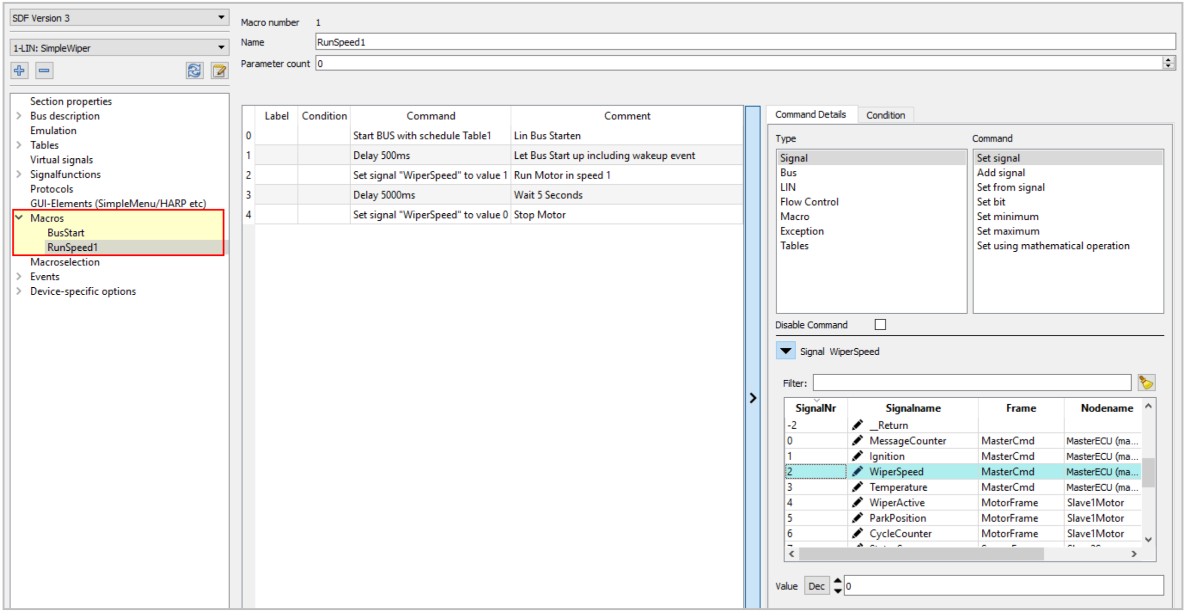
For Autosar CRC there is then a whole series of parameters.



Session Conf - Macros

Macros are used to combine multiple operations into a sequence.

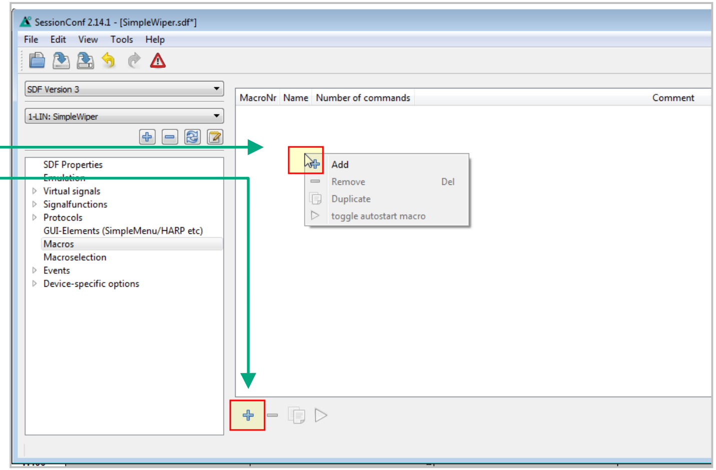
Macros can be started by events or, with SDF-V3, can also be called from other macros in the sense of a Goto or Gosub. The DLL-API calls a macro with the macro_execute command.



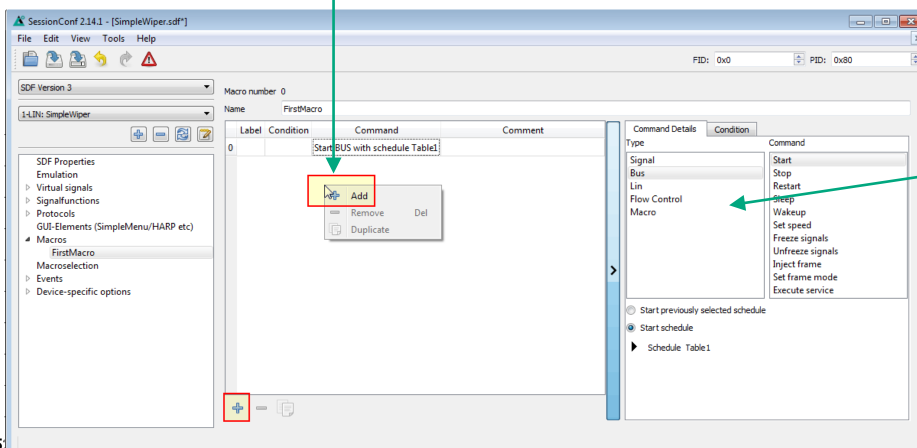
Macros play an important role in the implementation of functional logic in an SDF.

Session Conf - Macros

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

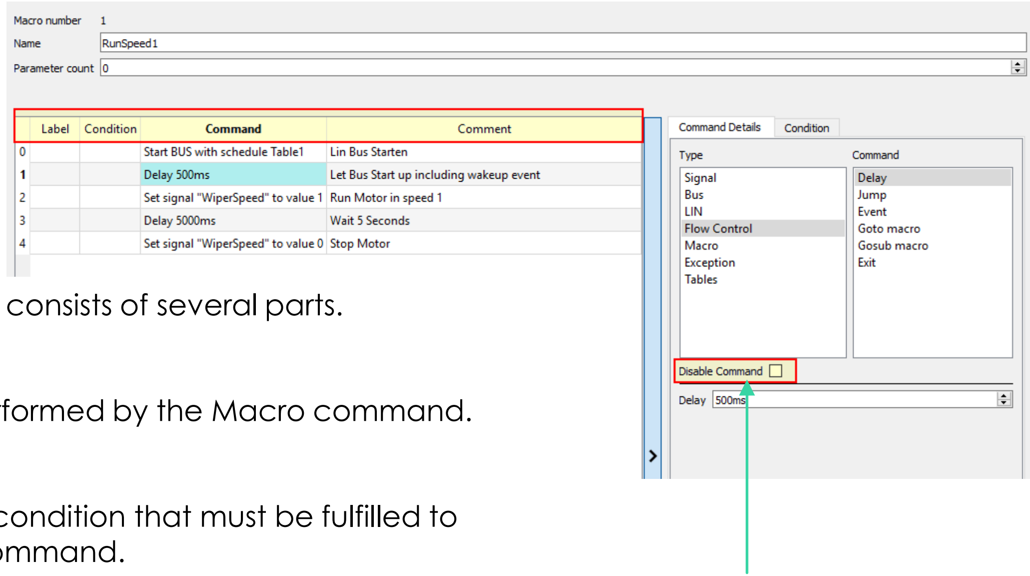


Then you add commands to this macro. The command Start Bus is always inserted; it is then changed to the desired command.



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.

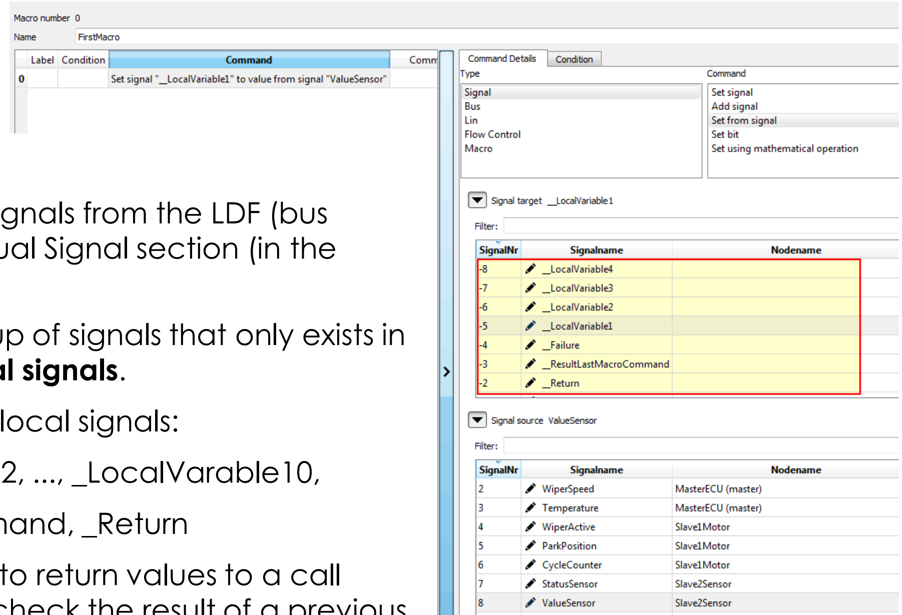
Condition
Here you can define a condition that must be fulfilled to actually execute the command.

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



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, ..., _LocalVarable10,

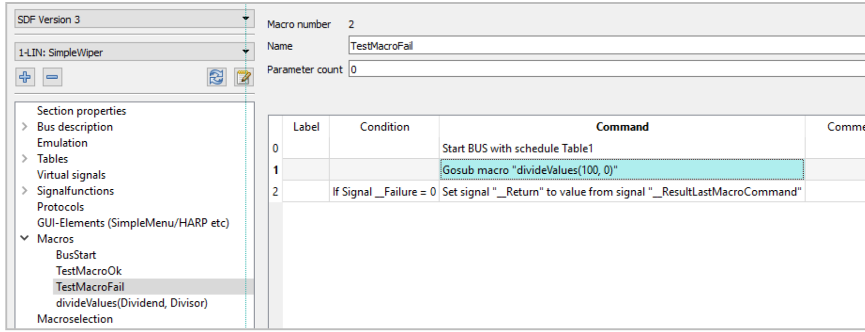
_Failure, _ResultLastMacroCommand, _Return

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.

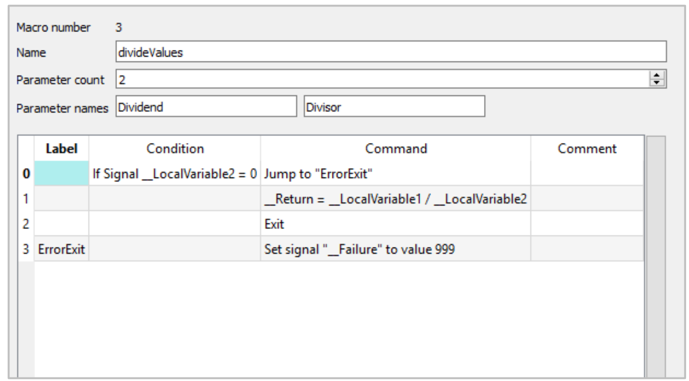
E.g. to save intermediate results when performing a calculation with several calculation steps.

Macro Parameter handover



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 signal 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 variable of the calling macro.

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

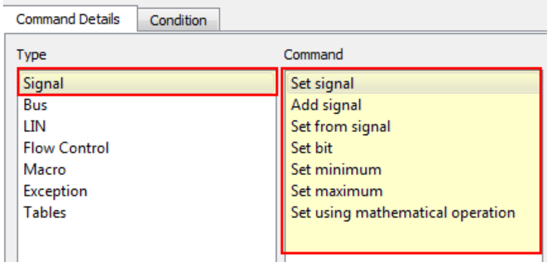
	Label	Condition	Command	Comment
0		If Signal _LocalVariable2 = 0	Jump to "ErrorExit"	
1			_Return = _LocalVariable1 / _LocalVariable2	
2			Exit	
3	ErrorExit		Set signal "_Failure" to value 999	

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.

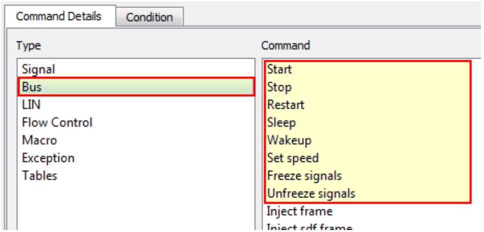
Important note: The value of _**ResultLastMacroCommand** is only valid in the Macro command line directly after the Gosub command, because this signal always contains the result of the previous command.
The _Failure variable has a different behavior. It is automatically transferred to the calling macro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

Macro Signal commands



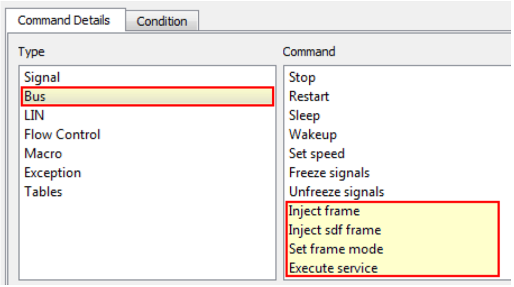
Macro command	Description
Set signal	Assign a constant value to a signal.
Add signal	Add a constant to a signal value (constant can also be negative).
Set from signal	Set a signal with the value of another signal.
Set bit	Set or delete a specific bit of a signal.
Set Minimum	Assignment of the smallest 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



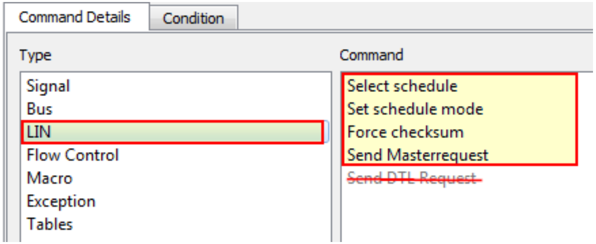
Macro command	Description
Start	Resets all bus signals to the LDF default values.
Stop	Stops the Lin Bus communication.
Restart	Starts the LIN bus, but receives all signal values. **No reset to LDF default values.**
Sleep	Sends a Sleep Frame to the bus and stops Schedule.
Wakeup	Sends a wakeup event 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



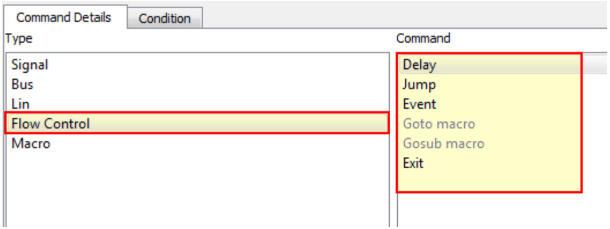
Macro command	Description
Inject frame	Allows to send any frame without LDF definition. With the latest LINWorks/Firmware version a blocking execution is also supported.
Inject SDF frame	**New:** Allows to send an SDF frame (LDF/DBC) without a schedule; the bus must be started and the frame must be sent 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



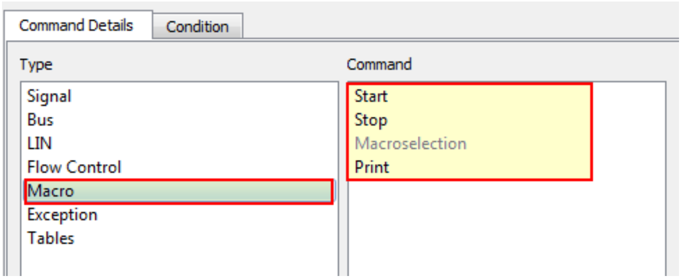
Macro command	**Description**
Select schedule	Schedule switching optionally, Schedule mode can also be transferred.
Set schedule mode	Permanently assign an execution mode to a schedule table: • Cyclic • Single run • Exit on complete
Force checksum	Force a certain checksum type: Automatic, V1(Classic 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.

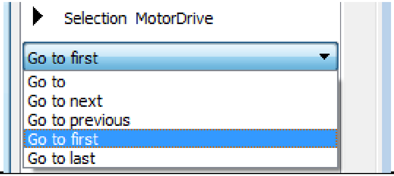
Macro Flow Control commands



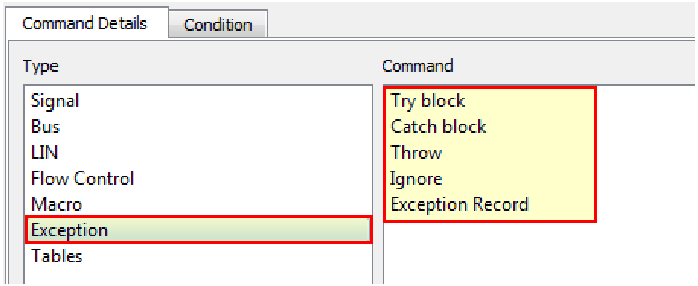
Macro command	**Description**
Delay	Delays macro execution by the specified time (ms).
Jump	Branches to another command in the same macro. Used for loops or branches, often in conjunction with a condition.
Event	Deactivates and activates events.
Goto macro	Branches to another macro; the remaining commands 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



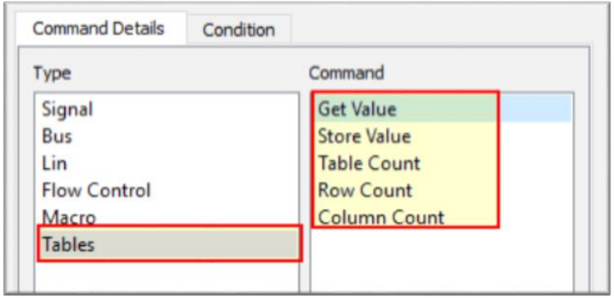
Macro command	Description
Try Block	Defines the beginning or end of a Try block.
Catch Block	Defines the beginning or end of a Catch block.
Throw	Triggers an exception with the given exception code anywhere (in the try block or outside the try block).
Ignore	Allows you to ignore certain exceptions for 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"	

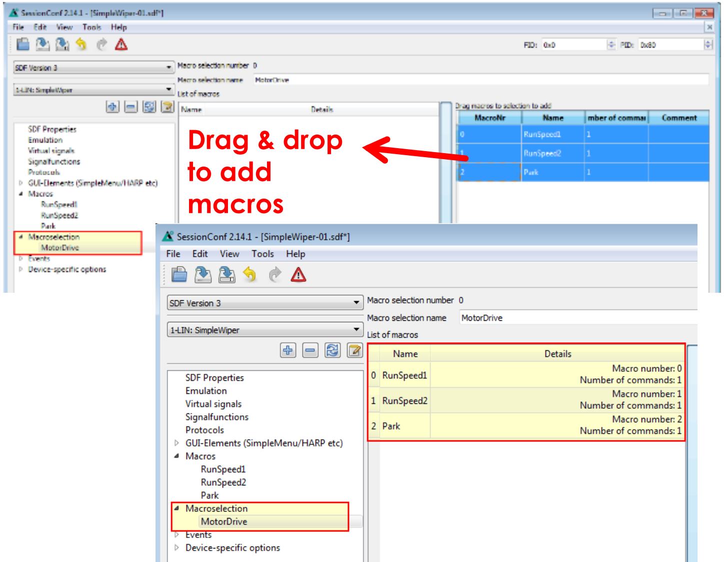
SessionConf – Macro selection

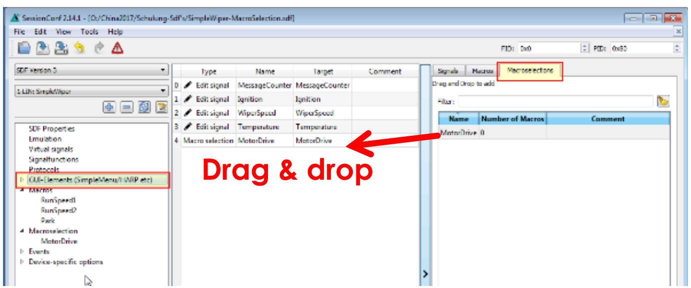
Macro selection

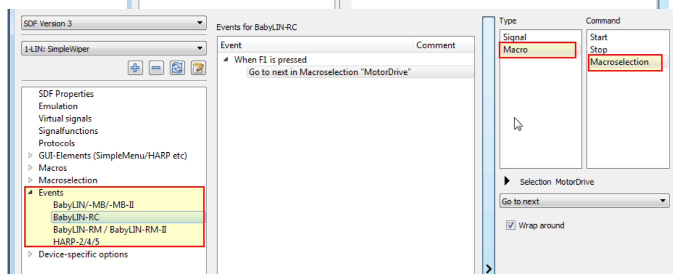
A macro selection defines a group of macros from which a macro can be selected for execution.

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).





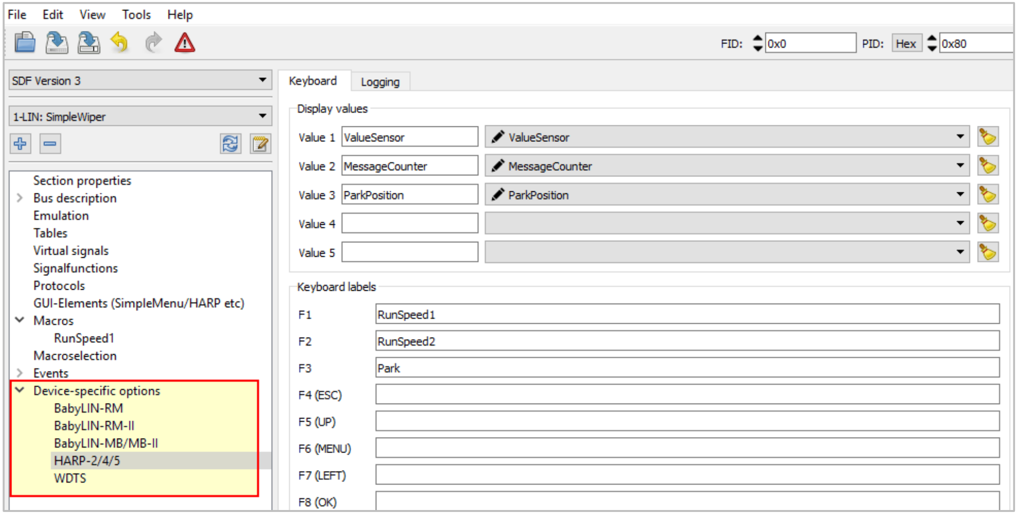


SessionConf – Device specific options

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.