Vector Interaction Layer

Technical Reference

Il_Vector

Version 2.10.01

Authors	Klaus Emmert, Gunnar Meiss, Heiko Hübler
Status	Released

1 Document Information

1.1 History

Author	Date	Version	Remarks
P. Jost	2000-05-05	1.0	creation
P. Jost	2000-06-29	1.1	some corrections
P. Jost	2000-07-13	1.2	changes in Figure 4 and some further corrections
P. Jost	2000-08-06	1.3	correction of the First-Value Class
P. Jost	2000-09-13	1.4	little corrections in the description of the TxTask and IlInit
P. Jost	2001-03-01	1.5	message related transmission modes example for timeout monitoring multi channel support known problems integration example
P. Jost	2001-06-22	1.6	some names of attributes changed DataChanged flag Tx timeout monitoring Rx and Tx default values new screen shots of the current Gentool changes in the state machine and further little corrections
S. Hoffmann	2001-07-05	1.61	some corrections and branch for an OEM
P. Jost	2001-07-13	1.62	adapted the corrections of version 1.61 for general IL
P. Jost	2002-04-05	1.63	Signal groups Multiple physical and virtual ECU support Multiplex Signals Rx timeout monitoring: reload of timer and message related notification Notification in interrupt and task context (IL Polling) IL<Tx/Rx>StateTask Attributes for Rx timeout monitoring updated Configuration Tool pictures updated
P. Jost	2002-08-16	1.7	Name of this document changed from User Manual to Technical Reference Multiple Indication Flags per Signal Macro to Get and Clear at once Chapter for Configuration Tool updated ”New Style” API Data Type Prefix for Signal Access Further Callbacks for State Machine Initialization – IlInitPowerOn ECU Timeout
H. Hörner	2003-06-16	1.8	Several wording and spelling issues corrected List of abbreviations and glossary removed, replaced by

			an own document Implementation details moved to an Annex
K. Emmert	2003-09-02	1.9	Some design and link modifications.
H. Hörner	2004-05-14	2.0	Add usage of VStdLib Documented return value of flag get macros Difference between GenMsgDelayTime and GenMsgStartDelayTime clarified Some clarifications about signal groups Wording enhanced for multiplexed signals
Klaus Emmert Gunnar Meiss	2005-06-10	2.01	Added support for GENy Added new feature dynamic timeout handling Added raw API for multiplex signals Reworked dbc attributes chapter Added matrix with transmission modes
Gunnar Meiss	2005-08-02	2.02	Adapted GenMsgFastOnStart Added GENy Multiplex Support
Klaus Emmert Gunnar Meiss	2005-11-04	2.03	Added AUTOSAR API for GENy, configuration and signal access. Added GenMsgFastOnStart for multiplex messages in GENy Added ESCAN00014120 CANGen Added ESCAN00008602 CANGen Added ESCAN00008604 CANGen Reworked ESCAN00010718
Gunnar Meiss	2006-02-16	2.04	Added GENy Multiple ECU Reference Added ESCAN00013633 DynRxTimeout API postfix and data types have changed.
Klaus Emmert	2006-03-13	2.05	Signal Groups for GENy
Gunnar Meiss	2006-04-06	2.06	Added Indexed API discontinuation for GENy. Corrected ApplIlFatalError Prototype Improved GenSigTimeoutMsg_<ECU> Corrected GenSigSendType description Removed GenSigTimeoutMsg_<ECU> for GENy
Gunnar Meiss	2007-05-16	2.07	Opaque Data Types ESCAN00016935 GENy Improved documentation of call contexts of API functions ESCAN00017472, ESCAN00018014, ESCAN00014156, ESCAN00013962, ESCAN00013423, ESCAN00008047, ESCAN00008755
Gunnar Meiss	2007-12-17	2.08	Added GenSigSuprvResp, GenSigSuprvRespSubValue and GenSigTimeoutMsg_<ECU> for GENy Updated API descriptions Updated GenMsgStartDelayTime

			Updated GenMsgIlSupport ESCAN00024092
Gunnar Meiss	2008-04-21	2.08.01	ESCAN00024091
Gunnar Meiss	2008-07-17	2.09.00	Reworked Document Structure ESCAN00024902 Added Node Mapped dbc Attributes Updated Abbreviations and Glossary with CIWI ESCAN00028781 Added IlTxRepetitionsAreActive and IlTxSignalsAreActive ESCAN00028787 Reset Timeout Flags On Release Added Geny attribute descriptions ESCAN00023799 Added Limitation ESCAN00025371 Updated Dynamic Timeout Monitoring ESCAN00029109 Added Documentation of Generated APIs
Gunnar Meiss	2008-10-17	2.09.01	ESCAN00030172 The description of IlRxWait() is incorrect
Gunnar Meiss	2011-05-19	2.09.02	ESCAN00049272 OnChangeAndIfActive and OnChangeAndIfActiveWithRepetition is described incorrect in Table 3-6 "Send Type Matrix" ESCAN00049615 Incorrect Enumeration Values of the dbc attribute "ILUsed" ESCAN00048272 Incorrect Timing Diagram of the Transmit Fast if Signal Active Transmission Mode
Heiko Hübler	2012-03-13	2.10.00	Added Signal status information (UpdateBits)
Heiko Hübler	2012-05-14	2.10.00	Added description for the GENy GUI attribute “timeout time”
Heiko Hübler	2012-09-13	2.10.01	Added description for PreConfig Switch “Enable UpdateBit Support” Changed “Send on Init” description

Table 1-1 History of the Document

1.2 Reference Documents

No.	Source	Title	Version
[1]	Vector	Vector CAN driver. Technical Reference	
[2]	Vector	Vector Multiple ECUs. Technical Reference	1.00.00
[3]	Vector	Vector Configuration Tool. Online Documentation. (no printed manual available)	
[4]	OSEK	OSEK/COM, Version 3.0.3	3.00.03
[5]		Z.120 (1996). Message Sequence Chart (MSC). ITU-T, Geneva	April.1996
[6]	Vector	Interaction Layer User Manual	

[7]	AUTOSAR	AUTOSAR Specification of Module COM 2.0.0	2.00.00
[8]	AUTOSAR	AUTOSAR Specification of Module COM 3.1.0	3.1.0

Table 1-2 Reference Documents

⚠	**Please note** We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

2 Introduction

Nowadays cars are growing to become more and more complex systems. The functionality of a modern car is not dominated by mechanical components anymore. Electrical Control Units (ECU), sensors and actors became irreplaceable parts of a car. They are responsible for the reasonable functions of the power train, the chassis and the body of a car. An example for some ECUs is shown in Figure 2-1 Example for Some ECU's in a Modern Vehicle.

In many ways the functionality of an ECU in a car depends on information provided by other ECUs. For example the ECU of the dashboard needs the number of revolutions per time of the wheels to display the car's speed. As a result communication between the ECUs is a significant component of a modern vehicle.

The communication between ECUs should essentially remain encapsulated. The application working on an ECU should not need to know how to transmit or receive data from other ECUs. Therefore Vector Informatik GmbH provides a set of components for the communication of ECUs by the CAN bus.

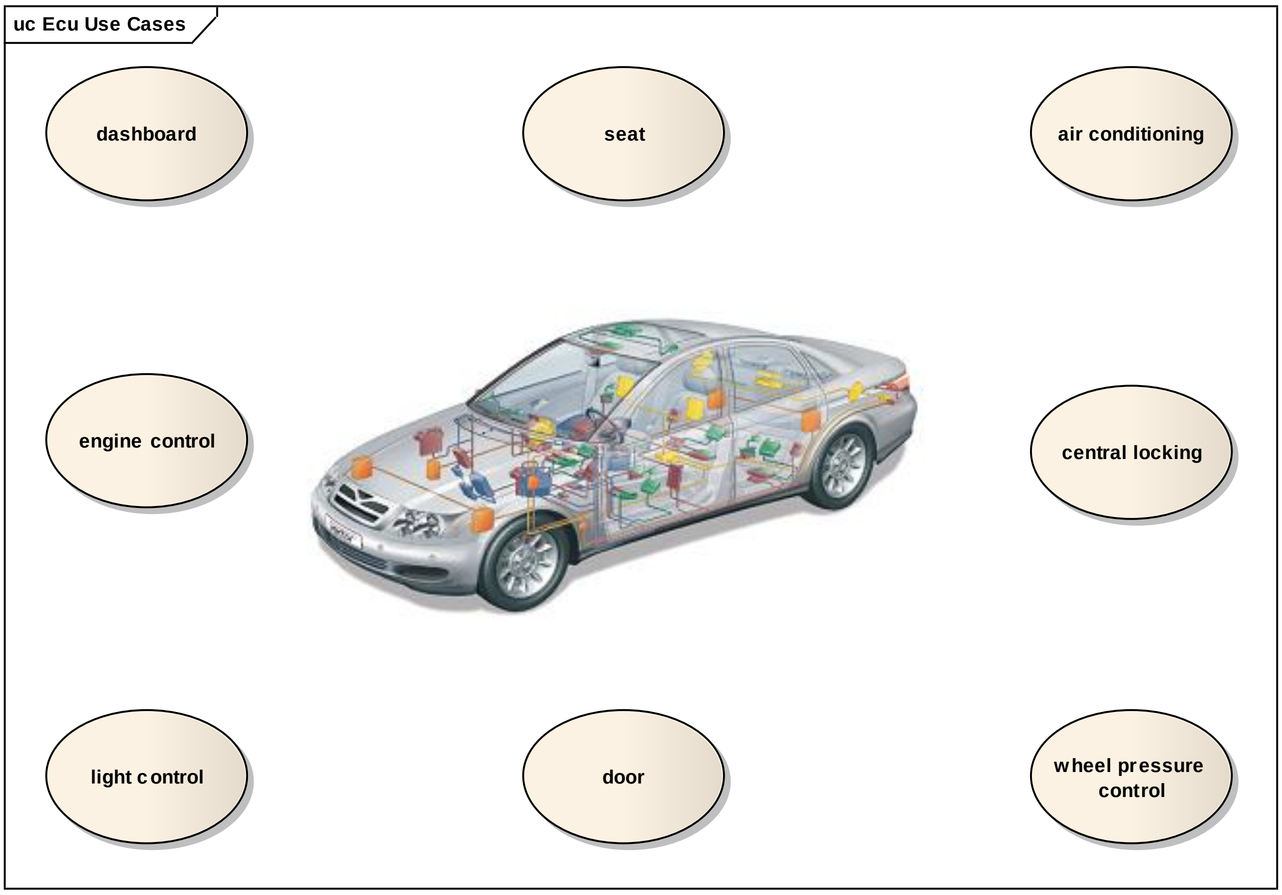


Figure 2-1 Example for Some ECU's in a Modern Vehicle

These communication components are called CANbedded. They relieve the application of its communication assignment including the exchange of simple data, diagnostic data, network management data, calibration data and more. This document is concerned with the data exchange via an Interaction Layer.

The Vector Interaction Layer hides all the communication related parameters and physical values from the application to ease the workload of the application. In case of transmission the application just needs to pass data to be transmitted to the Interaction Layer. The Interaction Layer decides when to transmit the data. Because the transmission depends of the chosen Transmission Mode which defines for example a periodic or an event triggered transmission of data. The other way round, in case of reception of data the Interaction Layer will notify the application about the arrival of data. Then the application could decide whether to read the updated data.

In any case the application does not need to know how the data is transmitted or received by the lower communication layers. It follows that the data structures of the application will be independent of the communication data structures (e.g. bus frames). The result is a higher reusability of the application software.

To control the Interaction Layer by the means of starting, suspending or deactivating a further API is provided. It is intended to be used for example by the Network Management. By this API it is possible to realize some CAN related modes like Sleep Mode, Bus-off Mode or Low-Voltage-Mode.

This manual is divided into three main chapters. The Overview introduces the features and concepts of the Interaction Layer. Next, the Functional Description explains the state machine, the communication flow, the data access and the transmission and reception of data. Technical Description, the last of the main chapters concerns with details of code generation, the API of the Interaction Layer and the usage or the Interaction Layer with or without an operating system.

2.1 Architecture Overview

The implementation of the Interaction Layer is intended to relieve the application of communication tasks. The Interaction Layer is one of the communications components of CANbedded offered by Vector Informatik GmbH. In Figure 2-2 Layer model of the Vector CAN communication components CANbedded it is shown how the Interaction Layer is embedded in the CANbedded protocol stack.

The communication software user has to be supplied with suitable access mechanisms to permit the adaptation of the ECU's behaviour to the network. Furthermore, the application needs mechanisms which permit a structured access to the data of the network. Such mechanisms are provided by the Interaction Layer of Vector Informatik GmbH and will be described in this manual. The Interaction Layer is responsible for separation of the Data Link Layer dependent low-level driver (CAN driver) and the application task which is independent of the underlying bus system.

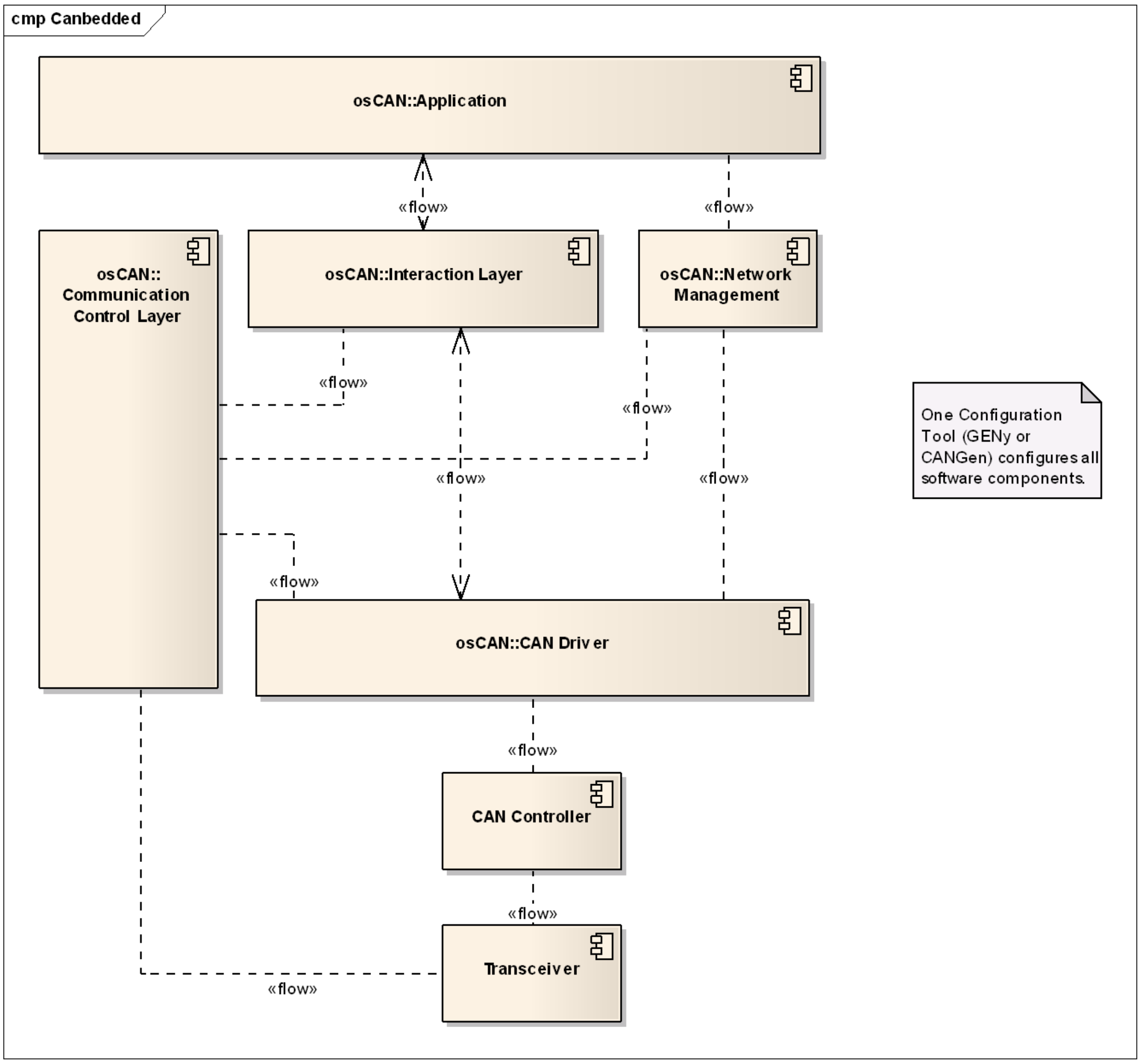


Figure 2-2 Layer model of the Vector CAN communication components CANbedded

The CAN Driver provides a mostly hardware independent interface to the higher communication layers. This enables the hardware independent implementation of the latter components and the target platform independent reuse of them.

2.2 Data Access Concept

The CAN bus uses messages to transmit user data. A message is 0 to 8 bytes long. The user information often does not match exactly 1, 2, 3 ... or 8 bytes. For example to transmit the state of a switch only 1 bit will be needed. This single bit has to be send in a 1 byte CAN frame. The 7 remaining bits will be left blank. To prevent such an overhead the remaining bits could be used to transmit other short user information. The user information combined to a message is called signals. Figure 2-3 Signal-oriented Access to Data provided by the Interaction Layer shows the combination of signals in the transmit section and the splitting of messages in the receive section of the Interaction Layer. This mechanism, of cause, also lowers the bus load and raises the efficiency of data exchange.

The key aspect of the Interaction Layer is the transmission and the reception of data. Therefore the Interaction Layer provides a signal oriented data interface called Signal Interface. The Signal Interface offers an API to write and read data. If data was written by the application, the Signal Interface decides depending on the Transmission Mode whether to transmit the data. The Signal Interface is located on the top of the Message Manager which is responsible for the transmission and reception of messages. By these options the application will be relieved of this area of responsibility.

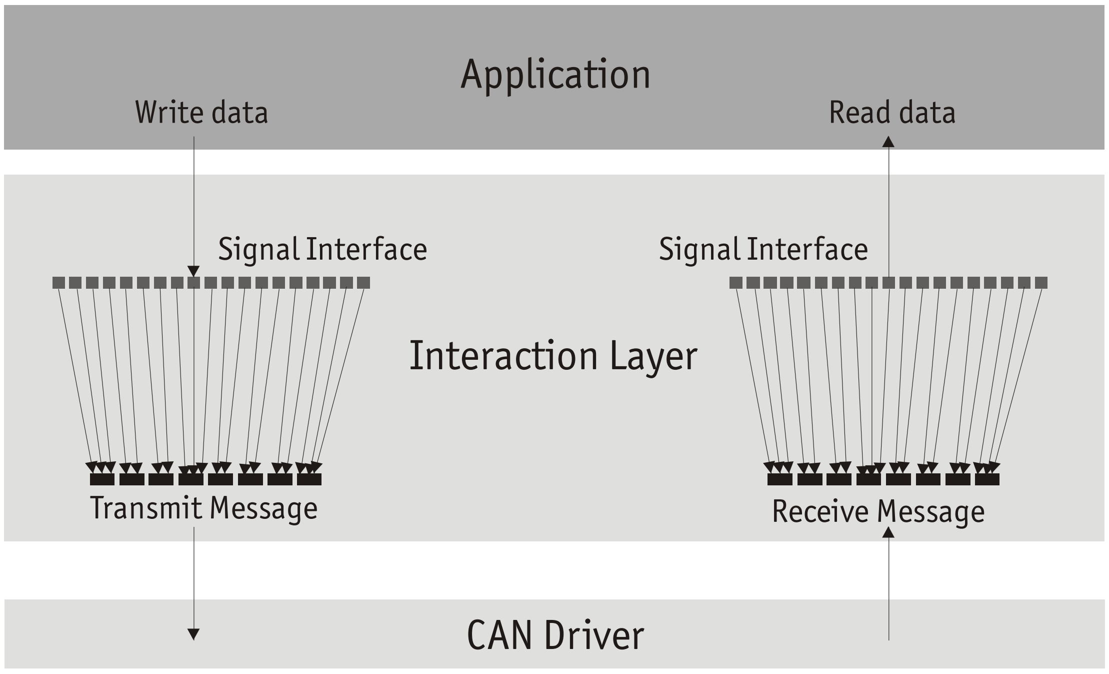


Figure 2-3 Signal-oriented Access to Data provided by the Interaction Layer

To transmit a signal the application just needs to write it to the Interaction Layer by calling `ILPut`. The Interaction Layer will decide what to do with the updated data. The decision depends on the chosen Transmission Mode and the delay timer. More information about Transmission Modes and delay timer could be found in chapter 3.5 Data Transmission. However, the application does not need to care about any further steps. The Vector Interaction Layer results in a supplementary support for the application.

For data exchange the Interaction Layer and the Data Link Layer need to copy the data several times. For example in case of reading a signal by the application the data has to be copied to the application's memory. But the reception of messages is handled by an interrupt which could intermit the copying of data. In case of reception, for example, the data has to be copied from the receive buffer of the CAN interface to the memory of the ECU by the interrupt handler. However, the Interaction Layer guarantees the consistency of any transmitted or received data and relieves the application of this area of responsibility, too. A further description of this problem will follow in chapter 3.4.1 Data Consistency.

2.3 Adapt the Vector Interaction Layer

To allow the application to access data in a signal-oriented manner, the Signal Interface provides macros and functions. The macro and function names are derived directly from the signal short names in the network database (also known as data dictionary or communication database). The signal names have prefixes and suffixes which can be defined by the user.

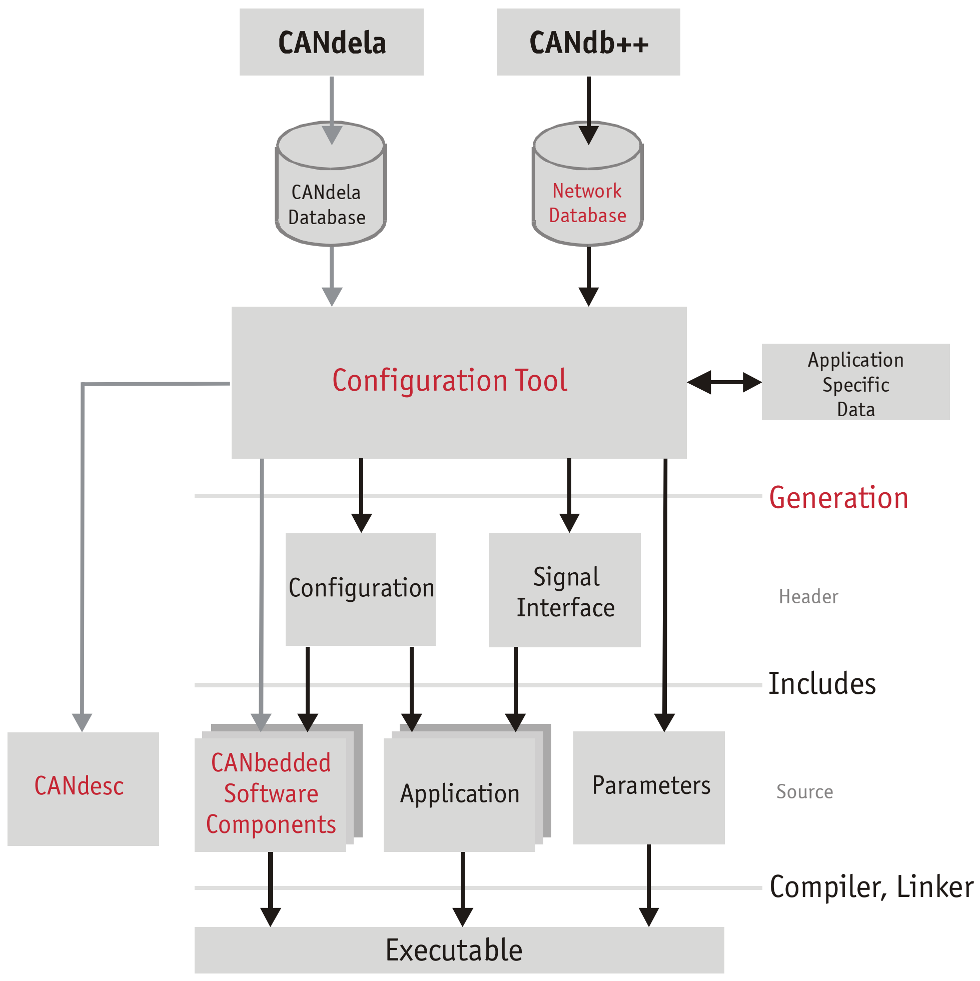


Figure 2-4 Usage of the network database to generate parts of the Interaction Layer

These functions and macros will be generated by the Configuration Tool using the network database related to a project. For this purpose, the car manufacturer makes the latest version of the network database available to all suppliers. Each ECU producer receives - in addition to the network component implementation for "her/his" processor - the Configuration Tool by which the supplier generates the parts of the communication components that are relevant to the particular ECU.

All application specific data which are made available by the supplier is saved in a separate file to preserve this information in the case of a network database update. The Configuration Tool checks for consistency of application specific data and the network database.

Figure 2-4 Usage of the network database to generate parts of the Interaction Layer shows the usage of the Configuration Tool.

If the communication components use more than one CAN-channel, the macros and functions have to be generated for each CAN-channel with the corresponding network database. In the Configuration Tool, a channel-index must be chosen that indicates on which CAN-channel the data dictionary is used.

Syntactically, data is always accessed by a function. Nevertheless, this might actually involve a macro. Whether there is a function call or a macro for direct access to the data buffer underlying the command will depend on the signal's data type. This will make it possible to change the implementation between functions and macros without having to change the syntax in the source code of the application. The use of macros is preferable with regard to run time. However, if two or more bytes have to be read for the signal access a function has to be used. This would allow including synchronization mechanisms (see section 3.4.1) for the data access within these functions. For reasons of efficiency, values with more than 32 bits are passed by data pointers, i.e. if the function is called the application passes a pointer to a memory location where the function stores the signal value. The application is responsible for providing sufficient memory space. If a signal is entered in the network database, e.g. as a 34 bit signal, the application must pass a pointer to a memory area with 5 bytes. When signals are transmitted compressed in a message, i.e. they only use a few bits within a byte, the Signal Interface expands appropriately for reading by the application and compresses appropriately for writing by the application. For example, reading of compressed signals take up 1 to 7 bits, the signals are mapped to one byte each for the application. Internally the Signal Interface continues to store these data in its memory in compressed form.

3 Functional Description

3.1 Features

The interface is divided into different communication layers. The Interaction Layer including the Signal Interface and the Message Manager put on the Data Link Layer represented by the Vector CAN driver. These layers, as described in chapter 2.1 Architecture Overview, are responsible for a number of tasks listed below.

The following features are supported:

Supported Feature
Receive Messages:
Provide signal-oriented access to data which arrived over the CAN bus.
Notify the application on signal level, if a message has arrived (indication).
Monitors receive messages to determine whether they arrive periodically (timeout monitoring).
Use default values in case of timeout or when the reception was stopped.
Transmit Messages:
Provide signal-oriented access to data to be sent over the CAN bus.
Provide different Transmission Modes to offer the application various mechanisms to transmit data.
Notify the application on signal level, if a message was transmitted (confirmation).
Monitor transmit messages to determine whether they had been actually transmitted (timeout monitoring).
Always keep a delay time between send requests of a message. This should delimit the bus load.
Use default values in case of timeout or when the transmission was stopped.

Table 3-1 Supported features

3.2 Initialization

If the CCL is not used in the software stack, the application has to initialize the components.

Example

Here is an example, if the initialization has to be implemented by the application.

```
/* Disable interrupts during the initialization of the
Components */
DisableInterrupts();

/* Initialize all components */
CanInitPowerOn();
IlInitPowerOn();
TpInitPowerOn();
DiagInit();

/* Enable interrupts */
EnableInterrupts();
```

3.1 Interaction Layer State Machine

An ECU can be in several states. These are normal-operation, sleep mode, bus-off mode and others. In different states the communication components have to meet different requirements. Therefore a state machine is defined for the Interaction Layer which consists of the states uninit, running, waiting and suspended (See in Figure 3-2 State Machine of the Interaction Layer). The state machine is instantiated per channel and for each communication direction (See in Figure 3-1 Rx and Tx State Machines).

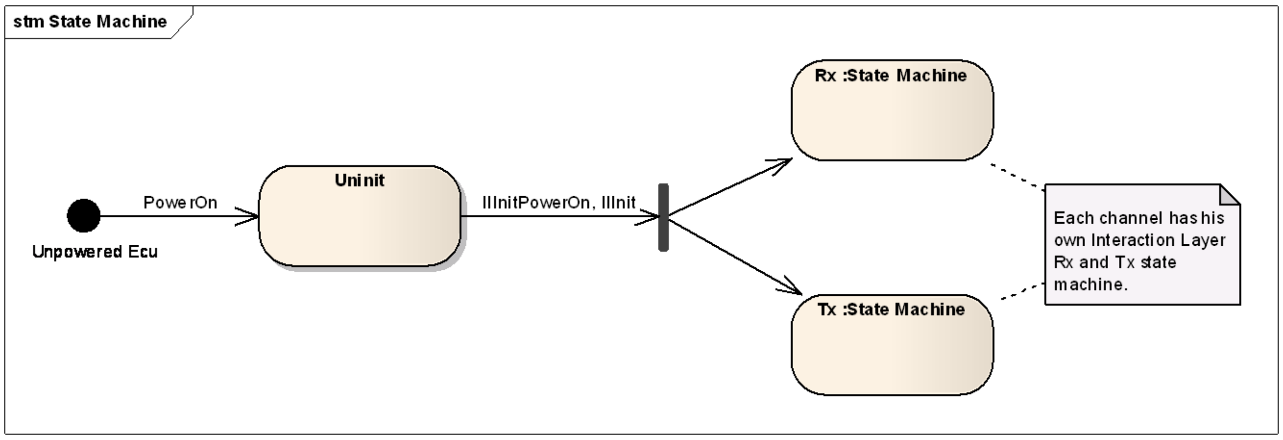


Figure 3-1 Rx and Tx State Machines

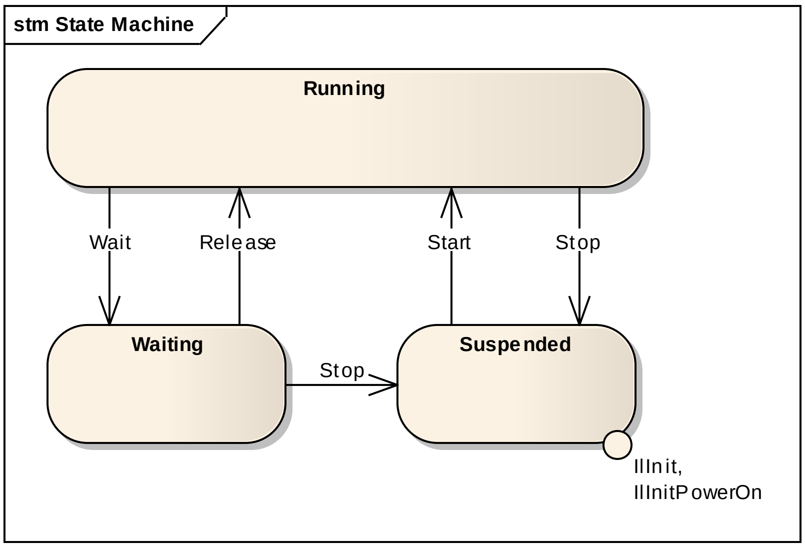


Figure 3-2 State Machine of the Interaction Layer

3.1.1 States

3.1.1.1 Uninit

After power-on the Interaction Layer will be in the uninit-state. Initializing the Interaction Layer will lead to a state transition to the state suspended.

3.1.1.2 Running

The running state is used for normal operation.

- Receive Section
  Reception of data is enabled as well as timeout monitoring and notification.
- Transmit Section
  Transmission of data is enabled. Signal Interface and Message Manager are working. The notification and the timeout monitoring are activated.

3.1.1.3 Waiting

This state was designed for example to support bus-off mode or low-voltage mode.

- Receive Section
  Reception of data is enabled as well as the notification for indication. The timeout monitoring will be turned off to prevent timeout detection of messages from an ECU which is in bus-off mode and does not transmit data.
- Transmit Section
  Transmission of data and the timeout monitoring will be disabled and the API will keep on working. So the application could request the transmission of data, but the Interaction Layer won’t follow immediately. The transmit requests will be stored and executed, when the state transits to Running. Transmission bursts are avoided if GenMsgStartDelay timings are defined, if this state is used in the bus-off mode.

3.1.2 State Transitions

The transitions of the state machine are divided into transmit and receive sections of the Interaction Layer and will usually be initiated by the Network Management. The application does not need to get involved here.

3.1.2.1 Init

The component variables are initialized. There is an option for initializing the transmit buffer and/or the receive buffer with default values. If no default value was configured, the buffers will be initialized with 0. The content of the default values is defined at compile time within the Configuration Tool. The flags will all be reset. If configured in the Configuration Tool, the application will be notified by invoking signal related callback functions.

3.1.2.2 Start

The receive section and the transmit section respectively are started within this transition.

Communication Section	Description
Receive Section	> The flags used for notification will be reset. These are in particular: the first value flag, the data changed flag, the indication flag and the Rx timeout flag. > Timeout monitoring will be started by a reload of the timers. > The values of the messages will be set to their default values, if defined at compile time. > If configured in the Configuration Tool, the application will be notified by invoking ApplIlRxStart() and/or signal related callback functions.
Transmit Section	> The flags used for notification will be reset. These are in particular: the confirmation flag and the Tx timeout flag. > The timers used for cycle times (e.g. for Send Periodic) will be initialized and started after the start delay time. > Timeout monitoring will be enabled by a reset of the timers. > The values of the signals will be set to their default values, if defined at compile time. > If configured in the Configuration Tool, the application will be notified by invoking ApplIlTxStart() and/or signal related callback functions.

Table 3-2 Start transition events

3.1.2.3 Stop

The reception and the transmission of messages respectively are stopped within this transition.

Communication Section	Description
Receive Section	> The flags used for notification won’t be changed. > The timer used for timeout monitoring will be stopped. > If configured, the values of the signals will be set to their default values. Otherwise they won’t be changed. > If configured in the Configuration Tool, the e application will be notified by invoking ApplIlRxStop() and/or signal related callback functions.
Transmit Section	> The flags used for notification won’t be changed. > If configured, the values of the messages will be set to their default values. Otherwise they won’t be changed. > Transmission and timeout monitoring will be stopped. > If configured in the Configuration Tool, the application will be notified by invoking ApplIlTxStop() and/or signal related callback functions.

Table 3-3 Stop transition events

3.1.2.4 Wait

The receive section and the transmit section respectively are deactivated within this transition.

Communication Section	Description
Receive Section	> Timeout monitoring will be stopped.
Transmit Section	> The flags used for notification won't be changed. > The values of the messages won't be changed but could be updated by the application. > Transmission and timeout monitoring will be stopped. > Requests for direct transmissions are stored in the waiting state. The related messages are transmitted after leaving the waiting state (release).

Table 3-4 Wait transition events

3.1.2.5 Release

The receive section respectively the transmit section are activated again within this transition.

Communication Section	Description
Receive Section	> The timeout monitoring will be restarted by reloading the timers. > The timeout flags are cleared, if configured (only GENy).
Transmit Section	> The timers used for cycle times and timeout monitoring will be continued. The values won't be changed to avoid interference between periodic transmitted messages on the bus (bursts). Pending requests for direct transmissions are performed. This can lead to a burst of messages after leaving the waiting state.

Table 3-5 Release transition events

3.2 Main Functions

The Interaction Layer provides two functions (IlRxTask and IlTxTask) that have to be called cyclically as configured in GENy by the Application, OS or CCL.

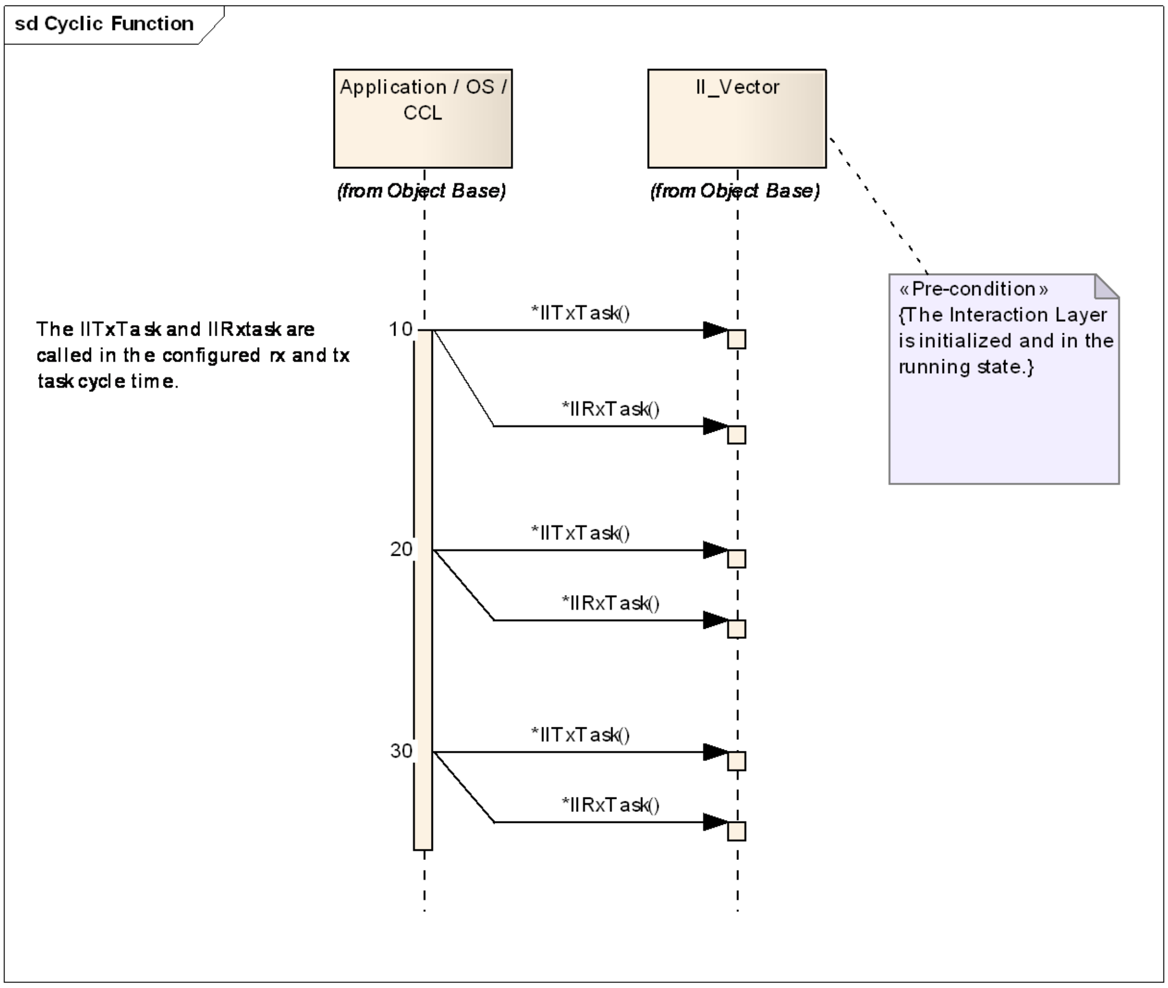

sd Cyclic Function
Application / OS / CCL
(from Object Base)
Il_Vector
(from Object Base)
«Pre-condition» {The Interaction Layer is initialized and in the running state.}
The IlTxTask and IlRxtask are called in the configured rx and tx task cycle time.
10
20
30
*IlTxTask()
*IlRxTask()
*IlTxTask()
*IlRxTask()
*IlTxTask()
*IlRxTask()

Figure 3-3 Call of the Interaction Layer cyclic function

Example

Here is an example, if the task calls have to be implemented by the application.

```
for(;;)
{
  /* periodic call of IlRxTask() and IlTxTask() */
  if (flag_10ms)
  {
    IlRxTask();
    IlTxTask();
    flag_10ms = 0;  /* clear flag which was set by a timer
*/
  }
```

	}

3.3 Interaction Layer Communication Concept

3.3.1 Interface Concept

The Interaction Layer as placed in the layer model has got two interfaces - one to the upper layer represented by the application and one to the layer below, the Data Link Layer. The receipt and the transmission of data is the main task of the Interaction Layer. To fulfil both of these tasks the Interaction Layer represented by its interfaces need to interact in different ways with its communication partners. Therefore different techniques like functions, interrupts and periodic tasks are used.

To enable the transmission of data for the application, functions and macros are provided by the Interaction Layer. The application could call these functions and macros whenever it needs to. The Interaction Layer usually copies the data to be transmitted to its local memory, sets a transmit request and leaves the processor to the application. The data actually will be transmitted later by a periodic task. This task checks at a defined period of time for transmit requests and executes them by calling the Data Link Layer. However, the application does not need to know anything about the process of transmission. It just needs to call the function respectively macro related to a signal to start the transmission process. A detailed description of this proceeding is given in chapter 3.5 Data Transmission.

The received data has to be treated immediately when arrived. This will be done by a receive interrupt. Inside the interrupt handler only the time critical work is done. The remaining not time critical work will be done by a periodic task which for example is responsible for the notification of the application. To get the signal values the application has to poll these values by calling functions respectively macros related to the signals. To decide whether to get new data the application will be notified about the arrival of new data. The reception is described more detailed in chapter 3.6 Data Reception.

3.3.2 Notification Mechanisms

Two mechanisms are provided for the notification of the application. These are: flag-interface and function-interface.

If the flag interface is used, the application has to poll the flags which were set by the Interaction Layer. To maintain as much separation as possible between a message and the signals of a message each signal can have its own flags or functions. Parameterization for this is performed in the Configuration Tool. Flag access is done by C macros. The macro name is comprised of the signal name and a postfix.

If the function interface is used, the application has to provide callback functions which are called by the Interaction Layer. The function name comprises the signal name and its indication-function's pre- and postfixes.

3.4 Data Access

3.4.1 Data Consistency

Since the Data Link Layer operates interrupt-driven, the read and write access to CAN data can be interrupted by a write or read request of the Data Link Layer. Under some circumstances this can lead to incorrect data if the data access in the Interaction Layer involves multiple bytes.
Figure 3-4 Synchronization Problem of Data Access describes a write operation of two bytes performed by the application. After the first byte is read (r 0x01) from the memory the read operation was interrupted by the write operation (w 0xAB and w 0xFF) of the IRQ of a receive message. However, the application continues reading the second byte (r 0xAB) after the interrupt routine finished. This causes an inconsistency of data read by the application. The same problem could occur during any access to shared memory.

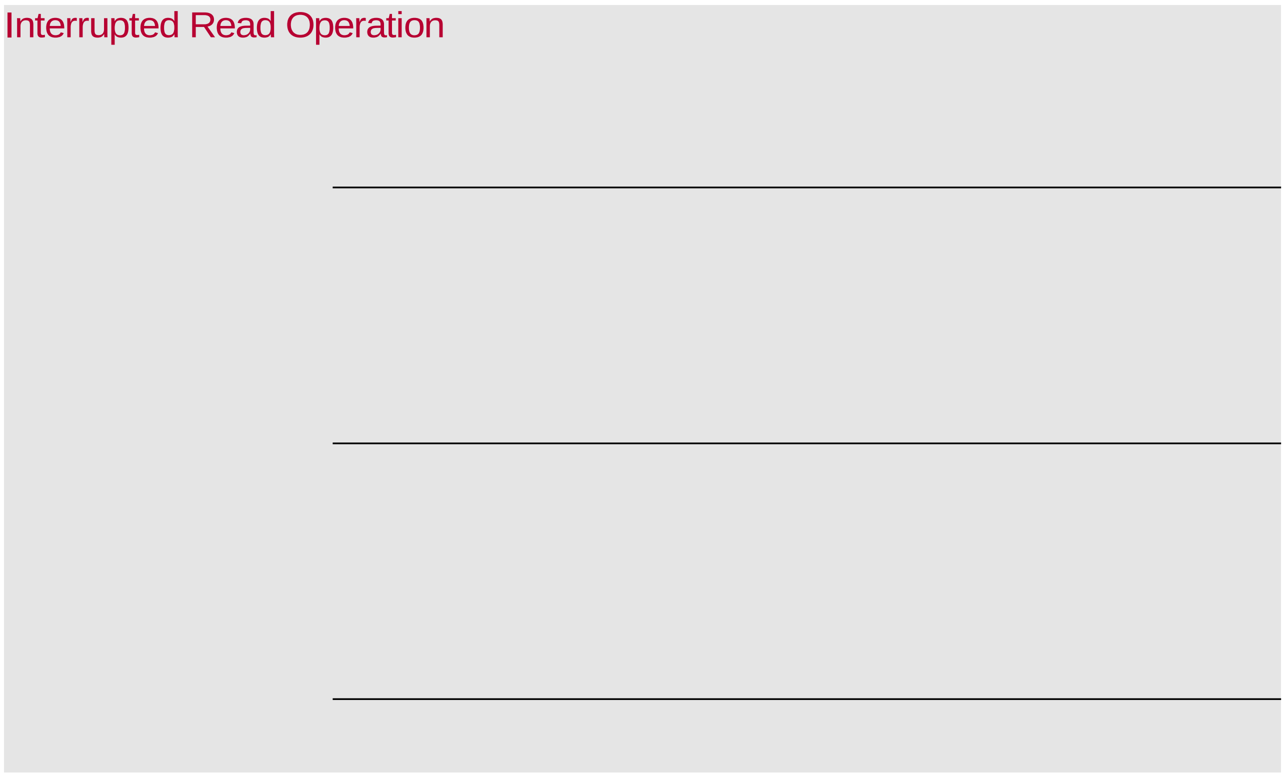


Figure 3-4 Synchronization Problem of Data Access

To prevent this, synchronization mechanisms were inserted for read/write into the Interaction Layer. Accesses on signals which do not need any synchronization (e.g. bit signals) could be executed as macros. The access to other signals must be routed through functions, because suitable synchronization mechanisms can be inserted there. The signal functions are generated by the Configuration Tool, where suitable synchronization mechanisms are included.

The choice of the synchronization method depends on the particular processor type and will be made by the Configuration Tool. One possibility for example is to disable interrupts while reading or writing data. So the interrupts can't disturb the access mechanism.

3.4.2 Signal Interface

The Signal Interface provides functions/macros for read access and writes access to the shared CAN data memory. Each signal has its own function/macro whose argument is the signal value or a data pointer. The function name comprises the signal name from the network database and suitable application-specific prefixes and suffixes. The data access functions can be implemented as macros or as actual functions as receive functions can be. The two variants do not differ syntactically. The implementation reserves the option of implementing signal access as a macro or as a function.

For read access to the transmit buffer and write access to the receive buffer, respectively, macros are provided. E.g. the usual operation on a receive signal is reading new data. Therefore, a function or a macro will be provided. Additionally a macro for write access to this receive signal can be configured. By default these macros are switched off, but can be configured by the Configuration Tool. If not needed, we recommend not switch the macros on, because of the resources functions need.

Depending on the size of the signal, the type of argument for data access can differ. If the signal length is lower than or equal to 8 bit, the signal argument is treated as a vuint8 (8-bit unsigned char). Signals, which are between 9 bit and 16 bit are treated as vuint16 (16-bit unsigned short) values. If the signal size is between 17 bit and 32 bit a vuint32 (unsigned long) will be used as signal argument. For signals greater than 32 bit, the function requests a pointer to the source data buffer. I.e. the Application has to pass a data pointer to a memory area of sufficient size.

3.4.3 AUTOSAR Signal Interface

The Vector Interaction Layer can be configured to support the signal access as defined in the AUTOSAR COM specification Version 2.0 [8].

The signal access is realized by using one function for write access and one function for read access. The signal that has to be written or read is given as parameter, as well as the value for a write access. The signal that is read is stored to the location given with the second parameter. See in the API below and the examples in chapter 3.4.4 Example: Writing and reading a signal value.

Example
A signal is written using

	■ Signals defined in the dbc file can be accessed. ■ No local communication is supported. ■ The return value is always E_OK. ■ Restrictions of the standard API are inherited. ■ Opaque Data types are mapped to unit data types according to the following list: Bit length 1..8 -> vuint8 Bit length 9..16 -> vuint16 Bit length 17..32 -> vuint32

.

3.4.4 Example: Writing and reading a signal value

The database contains the signals "EngineSpeed" (36 bit), "InteriorLight" (1 bit signal), "NewTemperature" (36 bit), “NewWindowPos” (16bit) and “EngineRPM” (30bit). The user configures the prefixes "ILPut" and “ILGet” and the suffix "_Sig”. By this configuration the Configuration Tool derives the functions shown below.

```
/* Applicatioe.mke*emre.scie*v(t)38(l)-24cbpe*******/m( )] TET EMC  /P <</
```

[illegible]

```
Com_SendSignal(WindowPos_Sig, &NewWindowPos);      /* Signal value
<= 16 Bit  */
Com_ReceiveSignal(EngineRPM_Sig, &EngineRPB);     /* Signal value
<= 32 Bit */
```

	Caution All generated signal access only provides **unsigned integer** values. Signed, float and the scaling factors (as adjustable in CANdb++) are not supported and have to be interpreted by the application.

3.4.5 Signal Groups

Physically dependent signals often need to be transmitted together. Therefore, the Interaction Layer provides the possibility to combine signals to a signal group. With a signal group the application can collect the data of dependent signals and invoke the transmission when the group is complete. The definition of the groups is done in the data base file (DBC), the data collection in a reserved buffer.

With the configuration tool GENy settings for a whole group can be done, e.g. the way of how to react in case of a group reception or the usage of a buffer provided by the application.

The strategy for transmitting a signal group is to update all group signals and then sending the group.

And vice versa is the strategy for receiving a signal group. The group is updated and then every signal can be read.

There must be distinguished between two different APIs:

- IL API (with data buffer provided by the application or data buffer provided by the IL)
- AUTOSAR API

3.4.5.1 Il API

The IL API can be used with a buffer provided by the application or the buffer provided by the IL. This can be selected on the configuration view of each signal group in GENy.

IL Buffer used

If the GENy checkbox **Use Appl SignalGroupBuffer** on the configuration view of the single groups is **not** checked the standard buffer defined by the IL is used. The Interaction Layer provides a structure in which the related signals are combined.

	Example
	Transmission of a signal group with IL API.
	`IlGetTx<groupname>();`
	`IlPutTx<signalname>(data)`

```
...
IlPutTx<groupname>();
```

Example

Reception of a signal group with IL API.

```
IlGetRx<groupname>()
value = IlGetRx<signalname>(dataPtr);
...
```

Appl SignalGroupBuffer used

With the GENy checkbox **Use Appl SignalGroupBuffer** on the configuration view of the single groups the usage of the buffer provided by the application is activated. You have to provide this buffer.

Example

Transmission of a signal group with IL API and buffer provided by the application.

```
/* declare the buffer */
V_MEMRAM0 V_MEMRAM1 _c_<groupname>_buf V_MEMRAM2
<groupname>;
/* initialize the buffer */
IlGetTx<groupname>ShadowBuffer(&<groupname>);



IlPutTx<signalname>SigShadowBuffer(&<groupname>, data);
...
IlPutTx<groupname>ShadowBuffer(&<groupname>);
```

Example

Reception of a signal group with AUTOSAR API and buffer provided by the application.

```
/* declare the buffer */
V_MEMRAM0 V_MEMRAM1 _c_IlRxGroup00_buf V_MEMRAM2
<groupname>;
/* initialize the buffer */
IlGetRx<groupname>ShadowBuffer(&<groupname>);
value = IlGetRx<groupname>SigShadowBuffer(&<groupname>,
```

	`dataPtr);`

3.4.5.2 AUTOSAR API

Using the AUTOSAR API there is no way to define an own shadow buffer for the storage of the signal groups. The predefined shadow buffer is used.

Updating the buffer for transmission:

Example
Transmission of a signal group with AUTOSAR API.

```
Com_UpdateShadowSignal(<signalname>, &data);
...
Com_SendSignalGroup(<groupname>);
```

Example
Reception of a signal group with AUTOSAR API.

```
Com_ReceiveSignalGroup(<groupname>);
Com_ReceiveShadowSignal(<signalname>, &ret);
...
```

3.4.5.3 GENy configuration

Almost any setting that is available for a single signal also is available for a signal group and can be selected on the configuration view of the signal group in GENy. In detail this is:

- Put and get macros
- Indication flag and function
- Confirmation flag and function
- Timeout flag and function
- Notification in case of state machine transition: init, start, stop
- Default values

Caution
Signal groups and multiplex messages cannot be combined in one message!.

3.4.6 Default Values

Each signal may have a default value which is defined in the Configuration Tool at compile time. These default values are used for

- Initializing the Interaction Layer (`IlInitPowerOn`)
- Starting the receive section (`IlRxStart`)
- Suspending the receive section (`IlRxStop`)
- Starting the transmit section (`IlTxStart`)
- Suspending the transmit section (`IlTxStop`)
- Replacing signal values in the case of receive errors (time out)

By initializing the Interaction the signal values will be set to 0 (zero), if no default values were defined. The user can define a single default value for each signal and configure, if it should be used when the Interaction Layer is initialized or when the receive section or the transmit section were started or stopped.

At compile time the user could define what the Interaction Layer should do, if a receive error occurred. It is possible to keep the old signal values or to replace them by a default value (only if defined).

The largest default value which can be set in the configuration tool is 0xffffffff (32 Bit). If signals greater than 32 Bits are used, they have to be set by the application e.g. in ApplIlTxStart().

3.5 Data Transmission

There are many ways data could be sent e.g. cyclic or triggered by a change of an initial value, etc. The concept behind transmitting data with the Vector Interaction Layer is explained in the following.

3.5.1 Transmission Concept

The Vector Interaction Layer offers a set of so-called transmission modes. According to these modes the signals and messages are being sent. The setting of the modes has to be done in the DBC file using the CANdb++ editor for any signal.

The following **signal** transmission modes are selectable:

- Cyclic
- OnWrite
- OnWriteWithRepetition
- OnChange
- OnChangeWithRepetition
- IfActive
- IfActiveWithRepetition
- NoSigSendType
- OnChangeAndIfActive

- OnChangeAndIfActiveWithRepetition

Additionally there are also transmission modes for **messages**:

- Cyclic
- IfActive
- NoMsgSendType

The resulting transmission mode is an OR between the message and the signal transmission mode. The greyed fields describe the attribute to be set to for this specific transmission mode.

	Signal Related Transmission	**Mixed Transmission**	**Advanced Transmission**
Message **Signal**	**NoMsgSendType**	**Cyclic**	**IfActive**
Cyclic	**Cyclic Transmission**		**Transmit fast if signal is active** (automatically set for all signals in this message) **or** **Cyclic Transmission**
	GenMsgCycleTime		GenMsgCycleTime GenMsgCycleTimeFast GenSigInactiveValue
OnWrite	**OnEvent** [Write] Will be sent immediately after a write access.	**Cyclic Transmission** **or** **OnEvent** [Write] (immediately)	**Transmit fast if signal is active** (automatically set for all signals in this message) **or** **OnEvent** [Write] Will be sent immediately after a write access.
		GenMsgCycleTime	GenMsgCycleTimeFast GenSigInactiveValue
OnWriteWithRepetition	**OnEvent** [Write] **with Repetition**	**Cyclic Transmission** **or** **OnEvent** [Write] **with Repetition**	**Transmit fast if signal is active** (automatically set for all signals in this message) **or** **OnEvent** [Write] **with Repetition**

	GenMsgCycleTimeFast GenMsgNrOfRepetition	GenMsgCycleTime GenMsgCycleTimeFast GenMsgNrOfRepetition	GenMsgCycleTimeFast

NoSigSendType	**No Transmission**	**Cyclic Transmission** (GenMsgCycleTime must be set)	**Transmit fast if signal is active** (automatically set for all signals in this message, GenMsgCycleTimeFast must be set)
		GenMsgCycleTime	GenMsgCycleTimeFast GenSigInactiveValue
OnChangeAndIfActive	**Transmit fast if signal is active** **or** **OnEvent** [Change] Will be sent immediately after value changed.	**Cyclic Transmission** **or** **Transmit fast if signal is active with Repetition** **or** **OnEvent** [Change] Will be sent immediately after value changed.	**Transmit fast if signal is active** (automatically set for all signals in this message)
	GenMsgCycleTimeFast GenSigInactiveValue	GenMsgCycleTime GenMsgCycleTimeFast GenSigInactiveValue	GenMsgCycleTimeFast GenSigInactiveValue
OnChangeAndIfActiveWithRepetition	**Transmit fast if signal is active with Repetition** **or** **OnEvent** [Change] Will be sent immediately after value changed.	**Cyclic Transmission** **or** **Transmit fast if signal is active with Repetition** **or** **OnEvent** [Change] Will be sent immediately after value changed.	**Transmit fast if signal is active with Repetition** (automatically set for all signals in this message)
	GenMsgCycleTimeFast GenSigInactiveValue GenMsgNrOfRepetition	GenMsgCycleTime GenMsgCycleTimeFast GenSigInactiveValue GenMsgNrOfRepetition	GenMsgCycleTimeFast GenSigInactiveValue GenMsgNrOfRepetition

Table 3-6 Send Type Matrix

It is the job of the data base engineer (or a suitable program) to assign the signals to the messages to get the desired transmission modes for any message and signal.

The application does not need to know the transmission mode of the signals. It just calls the function to write or read the signal value (`ILPutTx`**`signalname`** or `ILGetRx`**`signalname`**). Everything else will be done by the Signal Interface.

In case of periodic transmission modes only two different cycle times could be chosen for signals combined in the same message. Therefore, the cycle time of a periodically transmitted signal depends on the cycle time of other signals defined for periodic transmission related to the same message. The application developer is responsible for choosing sensible combinations of signals for a message. She/He will be supported by the Configuration Tool.

The transmission modes resulted from the combinations as shown in the table above are explained in detail in chapter 3.5.2 Signal Related Transmission Modes.

3.5.2 Signal Related Transmission Modes

This summarizes the first column of the table above. The message send type is set to `NoMsgSendType`.

3.5.2.1 Cyclic Transmission

A static period is used to transmit the signals cyclically using this transmission mode. This mode could be used to transmit signals which are frequently changing their values like the rpm of an engine for example. The period should be adapted to the speed the signals are changing their values. Short periods causes high bus load.

As shown in Figure 3-5 Timing Diagram of the Periodic Transmission Mode signals could be updated asynchronously to the period of transmission. Each time the transmission takes place the Interaction Layer checks for the current value of the message. This, of cause, could lead into the loss of data, if a signal was updated two or more times within a period.

This Cyclic Transmission Mode actually just copies the signal data. The cyclic transmission of the messages is done using the **`GenMsgSendType`**.

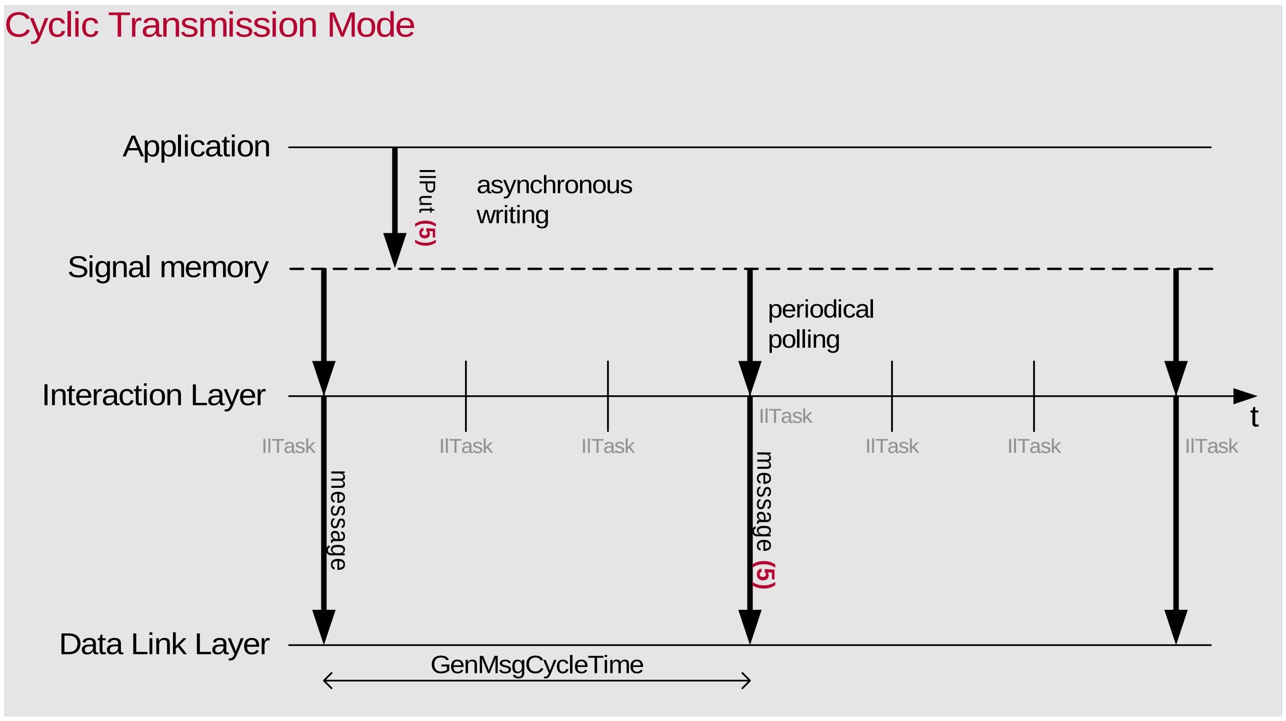


Figure 3-5 Timing Diagram of the Periodic Transmission Mode

3.5.2.2 OnEvent (OnWrite, OnChange)

Signals using this transmission mode will be transmitted once each time the `IlPut`-function was called. The transmission of the signals may be delayed by the delay timer (see chapter 3.5.6 Reduction of Transmission Bursts and 3.5.7 Delimitation of the Bus Load) to delimit the bus load. This transmission mode, for example, could be used for event triggered signals as the state of a switch.

Figure 3-6 Timing Diagram of the Transmission Mode OnEvent – OnWrite shows the timing diagram of the event triggered transmission mode.

- Writing a signal which is related to the **OnWrite transmission mode** causes the transmission of the message which contains this signal.
- Changing a signal which is related to the **OnChange transmission mode** causes the transmission of the message which contains this signal.

The TxTask checks if the delay time elapsed and decides whether to transmit the message immediately or to delay the transmission until the delay time elapsed. This could cause the loss of data, if the signal was updated two or more times while delay time.

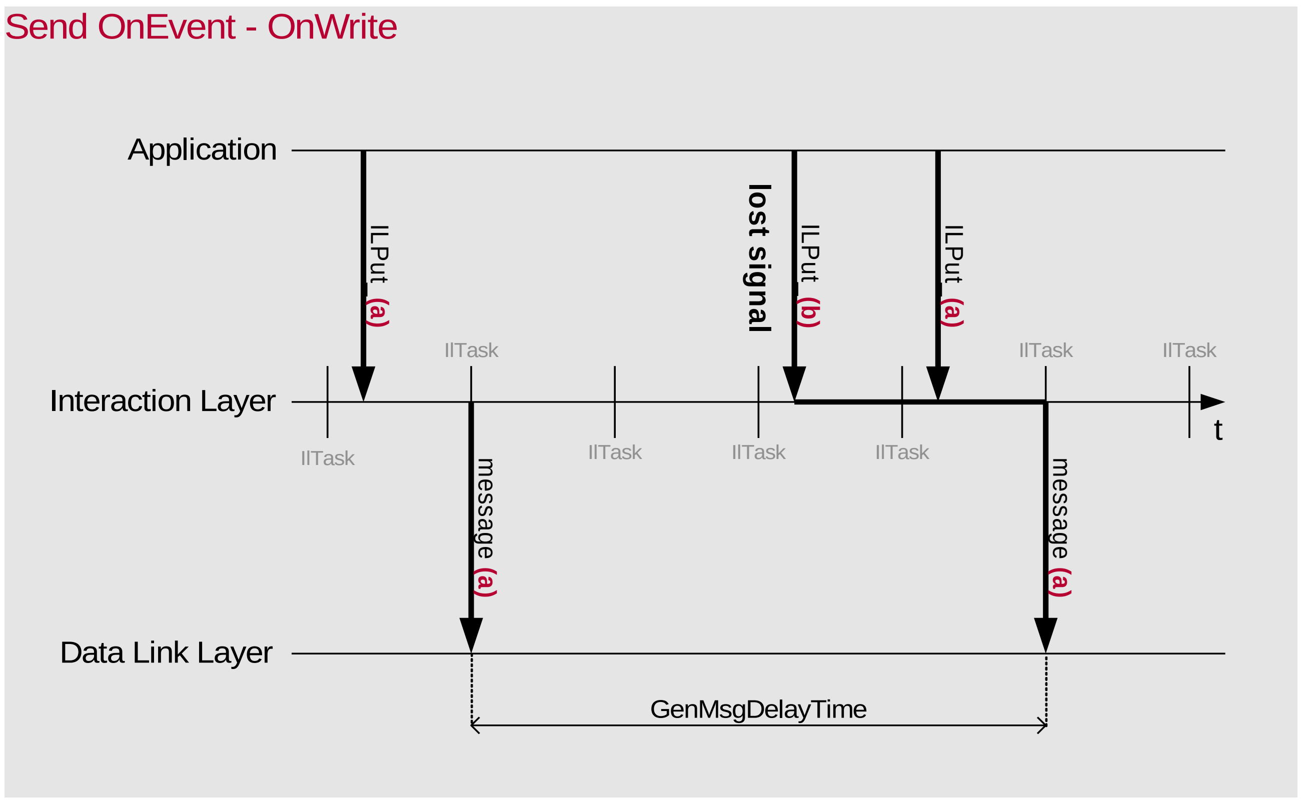


Figure 3-6 Timing Diagram of the Transmission Mode OnEvent – OnWrite

The Diagram for **OnEvent – OnChange** looks like the same way but the decision on whether to send or not is met by a comparison between the old and the new signal value. It will only be sent if the value changes.

3.5.2.3 OnEvent with Repetition (OnWrite, OnChange)

The transmission of the signals using this transmission mode will be repeated n-times after the `ILPut`-function was called once. For example, this mode could be used to transmit important signals which have not to be missed like safety critical information.

Each call of the `ILPut`-function sets the repeat counter (**repeat_counter [GenMsgNrOfRepetitions] = n**). The repeat counter is decremented with each transmission of the signal. The transmission takes place each time the delay timer elapses and the repeat counter is still greater than 0. After the `ILPut`-function the message will be sent **n** times.

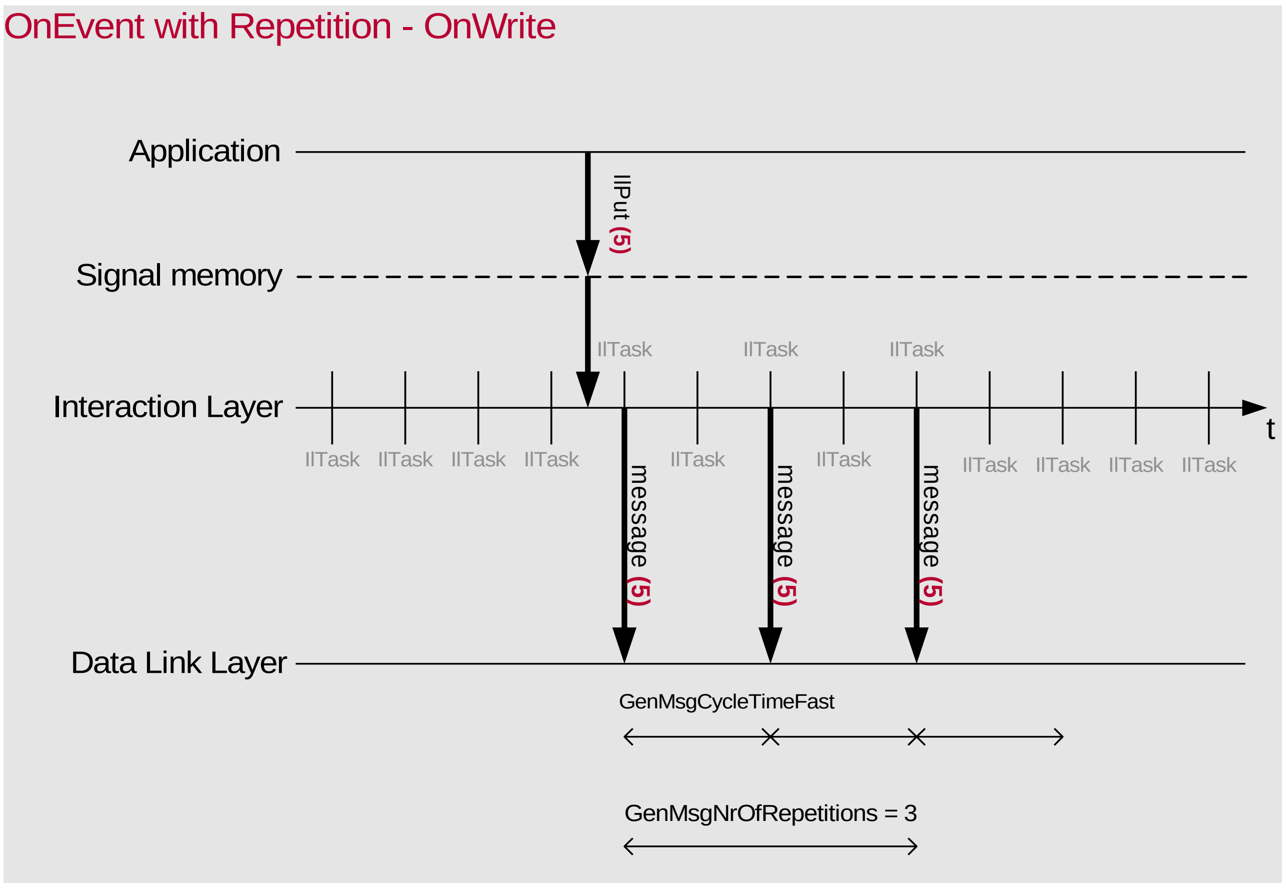


Figure 3-7 Timing Diagram of OnEvent with Repetition - **OnWrite**

The Diagram for **OnEvent – OnChange** looks like the same way but the decision on whether to send or not is met by a comparison between the old and the new signal value. It will only be sent if the value changes.

3.5.2.4 Transmit Fast if Signal is Active

This transmission mode is a Cyclic Transmission Mode with a trigger condition. If the decision is met that the signal is active, the message will be send cyclically with the period `GenMsgCycleTimeFast`.

In the Example in Figure 3-8 Timing Diagram of the Transmit Fast if Signal Active Transmission Mode the condition is defined as `x!=10`. This will cause the transmission mode to transmit the signal with the period `GenMsgCycleTimeFast`. If the signal value is equal to 10 the signal is not sent.

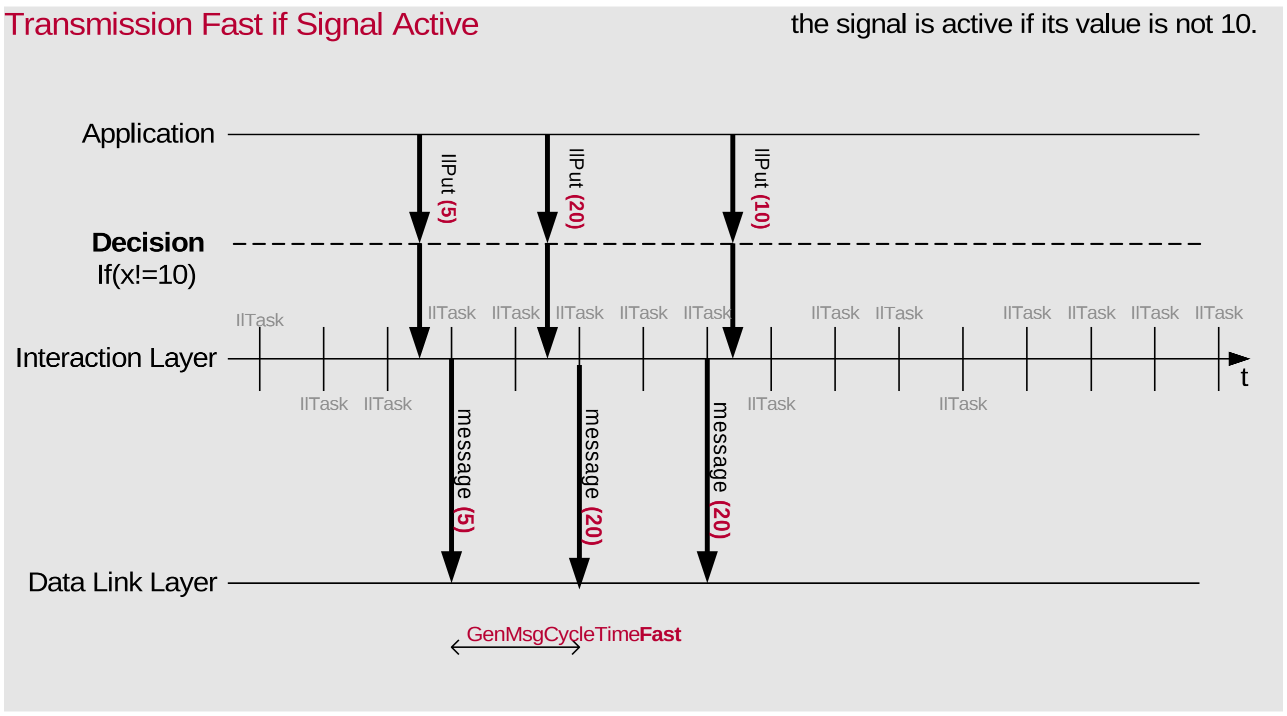


Figure 3-8 Timing Diagram of the Transmit Fast if Signal Active Transmission Mode

If two or more signals using this transmission mode are combined to the same message, a rule is needed to regulate switching between the periods.

Figure 3-9 Example for Combining Signals Related of the Send Fast if Signal Active Mode to the Same Message shows an example where three signals (A, B and C) are combined to the same message. If signal A was written and meet the defined condition, the transmission starts with the fast period. This state is stored in a flag presented by the three squares (grey = set, white = not set). The switch will cause the fast transmission of all signals combined to this message. A second write command for signal A won't cause anything, if the value of A still meets the condition. If signal B was written and meet its condition the flag for signal B will be set. This should cause the transmission mode to switch to the fast period. But this was already done so nothing will happen. The signal value for B which is written next does not meet the condition so the transmission should stop. This won't happen, because the flag for signal A is still set. To switch the transmission off all flags need to be reset. This is shown by setting the flag for signal C, reset the flag for signal A and reset the flag for signal C. After no set flag remains, the transmission stops.

Short: If signals using the Transmit Fast if Signal Active Mode are combined to the same message, the message will be transmitted fast if one ore more of them meets its condition. It will not be transmitted if none of them meet its condition.

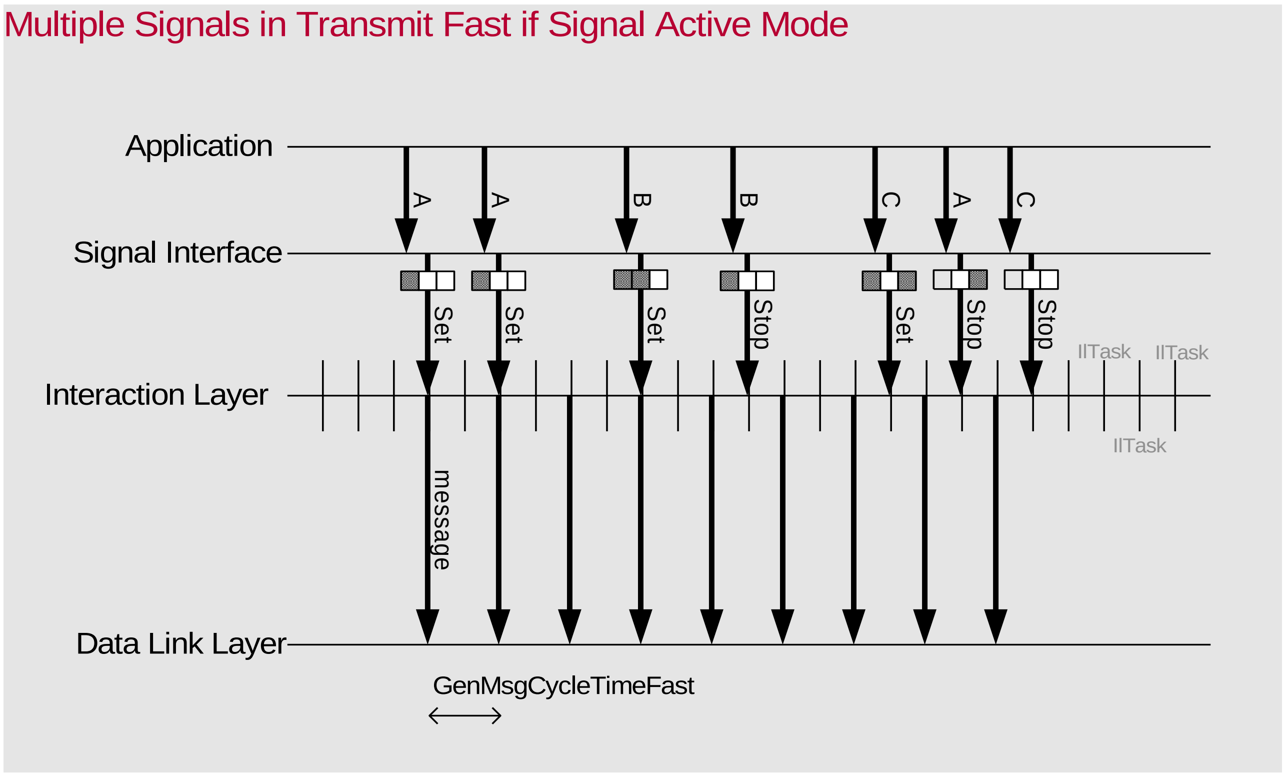


Figure 3-9 Example for Combining Signals Related of the Send Fast if Signal Active Mode to the Same Message

3.5.2.5 Transmit Fast if Signal is Active with Repetition

This is the same mode as above with the exception that the last transmission after the signal became inactive will be sent n times.

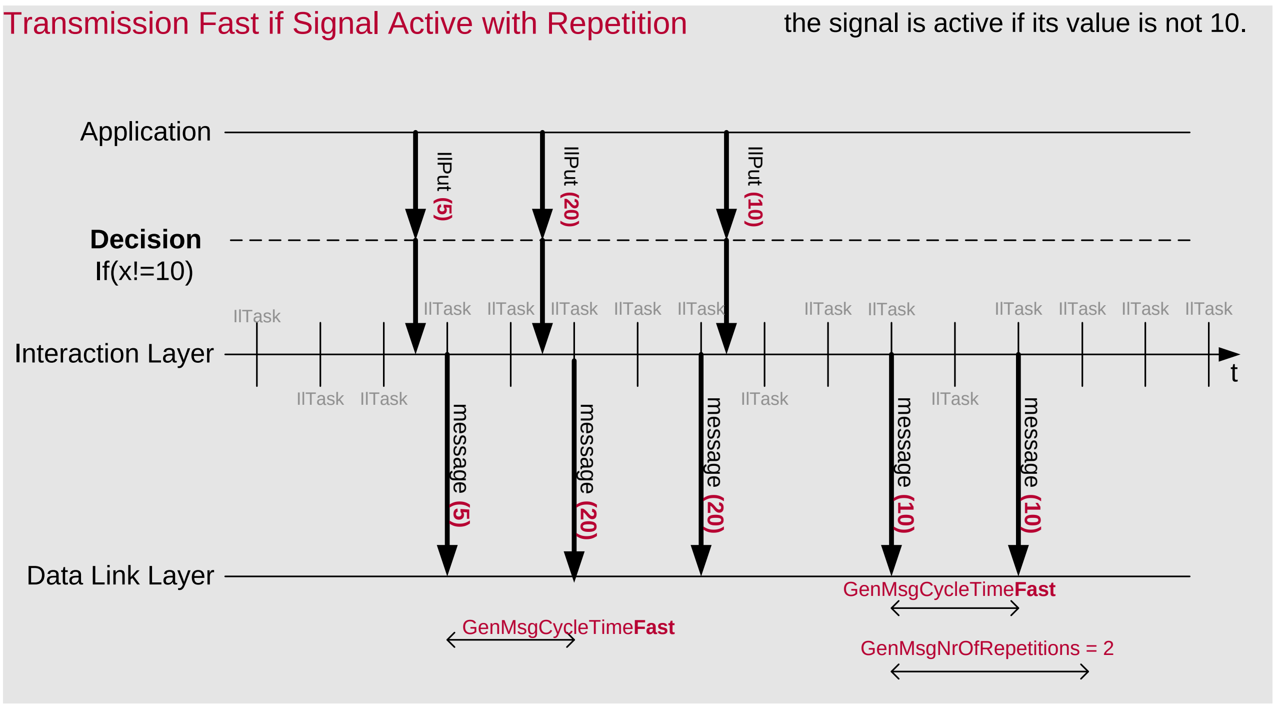


Figure 3-10 Timing Diagram of the Transmit Fast if Signal Active Transmission Mode

3.5.3 Mixed Transmission Mode

The mixed transmission modes are represented of the second column in the table above. This is a combination of the already shown signal oriented transmission modes and the cyclic transmission mode for the message the signals are assigned to.

3.5.3.1 Cyclic (Message) Transmission OR Cyclic (Signal) Transmission

This is absolutely the same as already described in 3.5.2.1, see there for more information.

3.5.3.2 Cyclic (Message) Transmission OR OnEvent [Write]

The signal is sent cyclically with the period `GenMsgCycleTime` and additionally after an `IlPut`-function call.

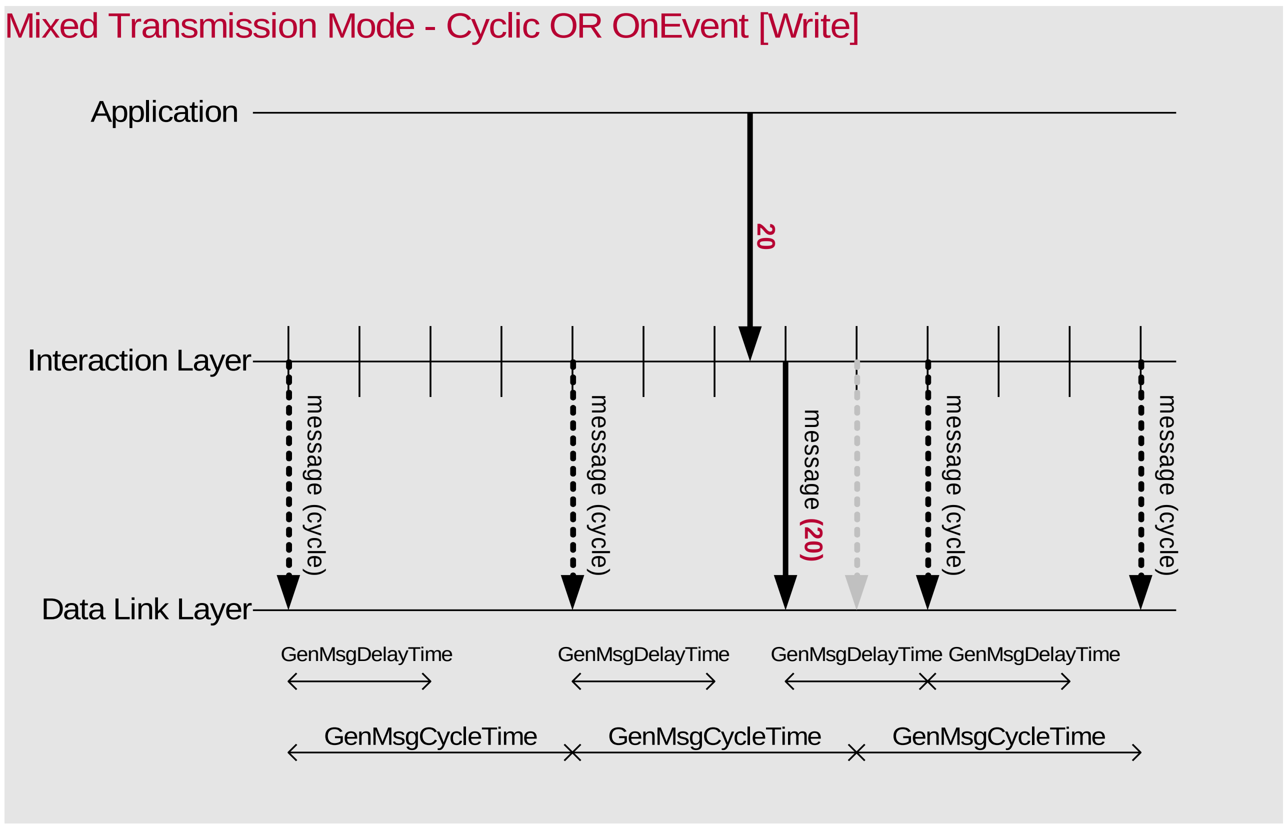


Figure 3-11 Mixed Transmission Mode – Cyclic OR OnEvent [Write]

The cyclic transmission is delayed because of the `GenMsgDelayTime` that has to be waited until the next transmission is possible. As a result two cyclic messages can have a distance that is smaller than `GenMsgCycleTime`.

3.5.3.3 Cyclic (Message) Transmission OR OnEvent [Write] with Repetition

The same behaviour as above but the event triggered message transmission will be performed `GenMsgNrOfRepetitions` times with the period of `GenMsgCycleTimeFast`. The delay times are taken into account.

3.5.3.4 Cyclic (Message) Transmission OR OnEvent [Change]

This is the same transmission mode as shown in 3.5.3.2 with the exception that the trigger for sending the event message is a change of the signal value. In the example below the message value changes from 5 to 20. This change is the trigger for the transmission.

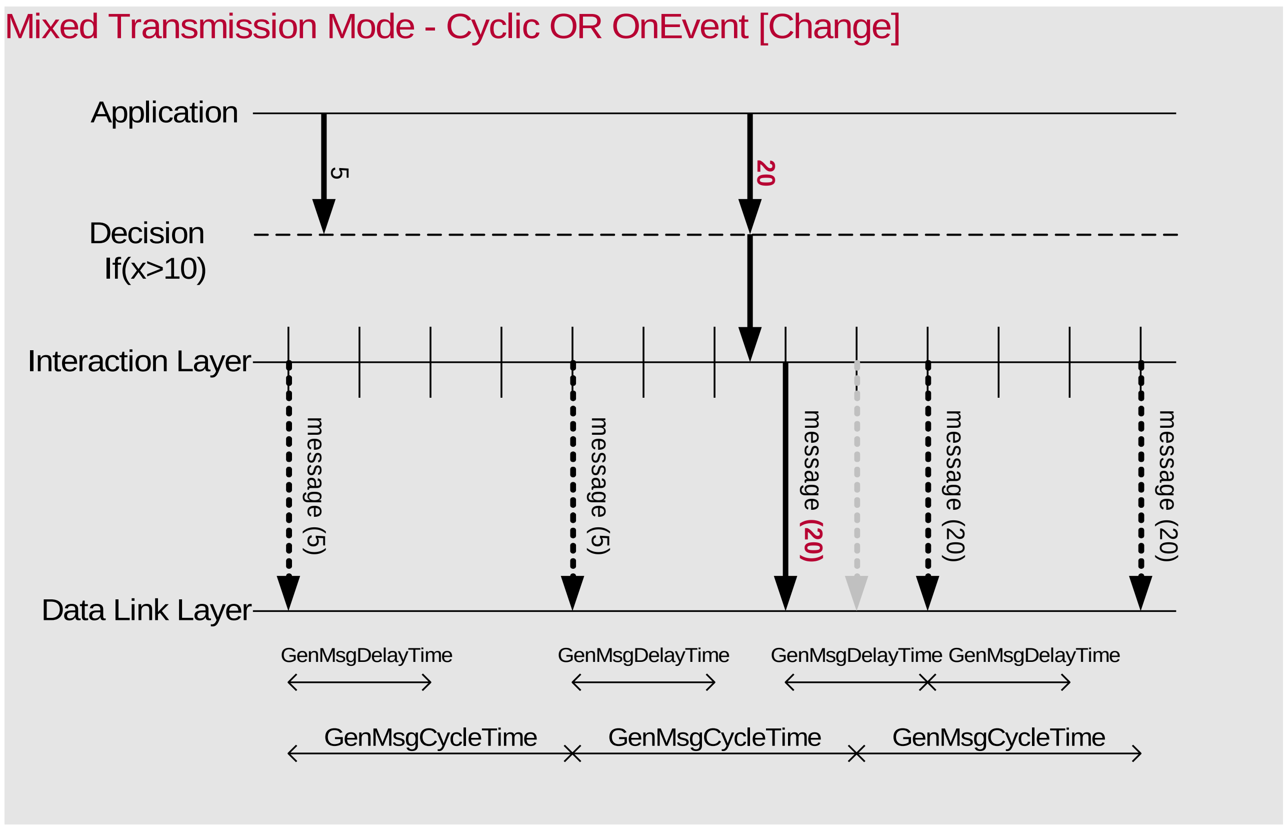


Figure 3-12 Mixed Transmission Mode – Cyclic OR OnEvent [Change]

3.5.3.5 Cyclic (Message) Transmission OR OnEvent [Change] with Repetition

The same behaviour as above but the event triggered message transmission will be performed `GenMsgNrOfRepetitions` times with the period of `GenMsgCycleTimeFast`. The delay times are taken into account.

3.5.3.6 Cyclic (Message) Transmission OR Transmit Fast If Signal is Active

Choosing this combination, the signal is transmitted cyclically with the `GenMsgCycleTime` until the signal becomes active. Then the signal is transmitted with the `GenMsgCycleTimeFast` until the signal becomes inactive. The period changes then back to G`enMsgCycleTime`.

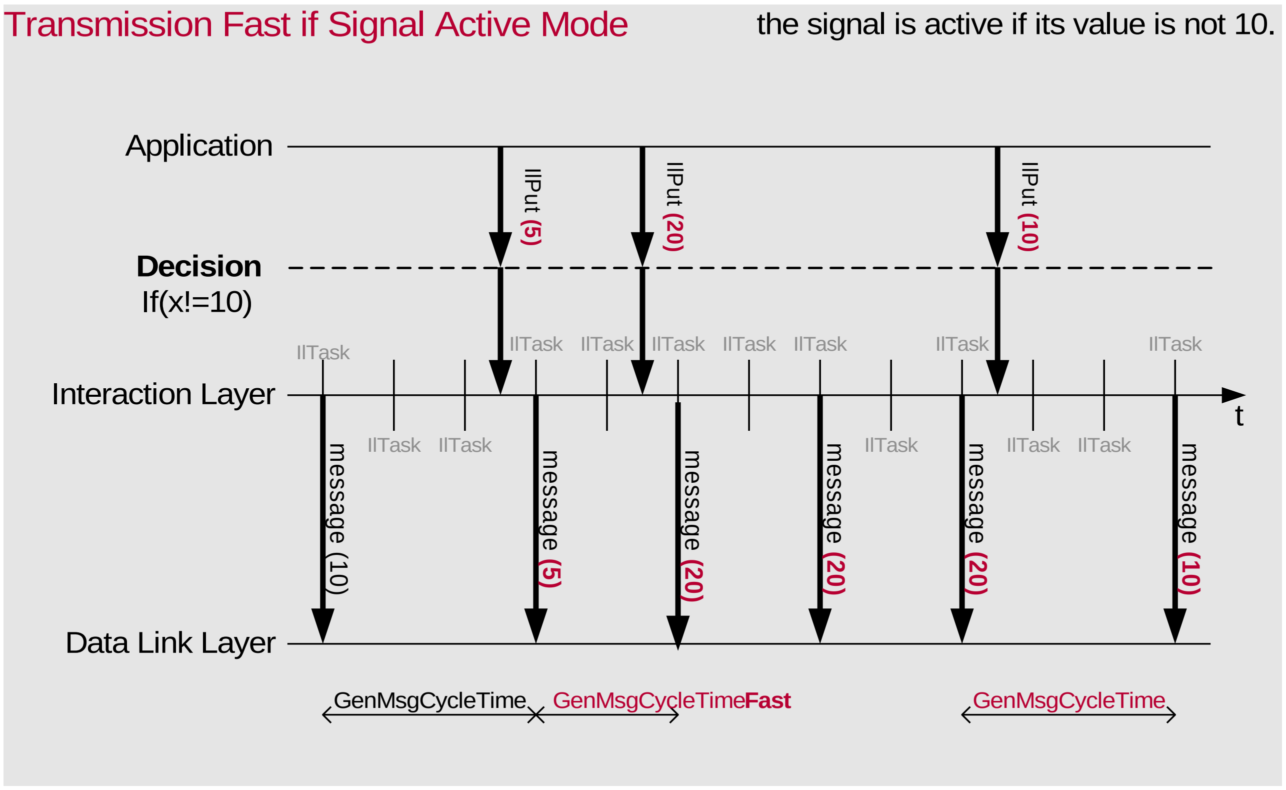


Figure 3-13 Mixed Transmission Mode – Cyclic OR Fast If Signal is Active

3.5.3.7 Cyclic (Message) Transmission OR Transmit Fast If Signal is Active with Repetition

The same behaviour as above but the last message of the fast transmission phase is sent `GenMsgNrOfRepetions` times before switching to the `GenMsgCycleTime` sending period.

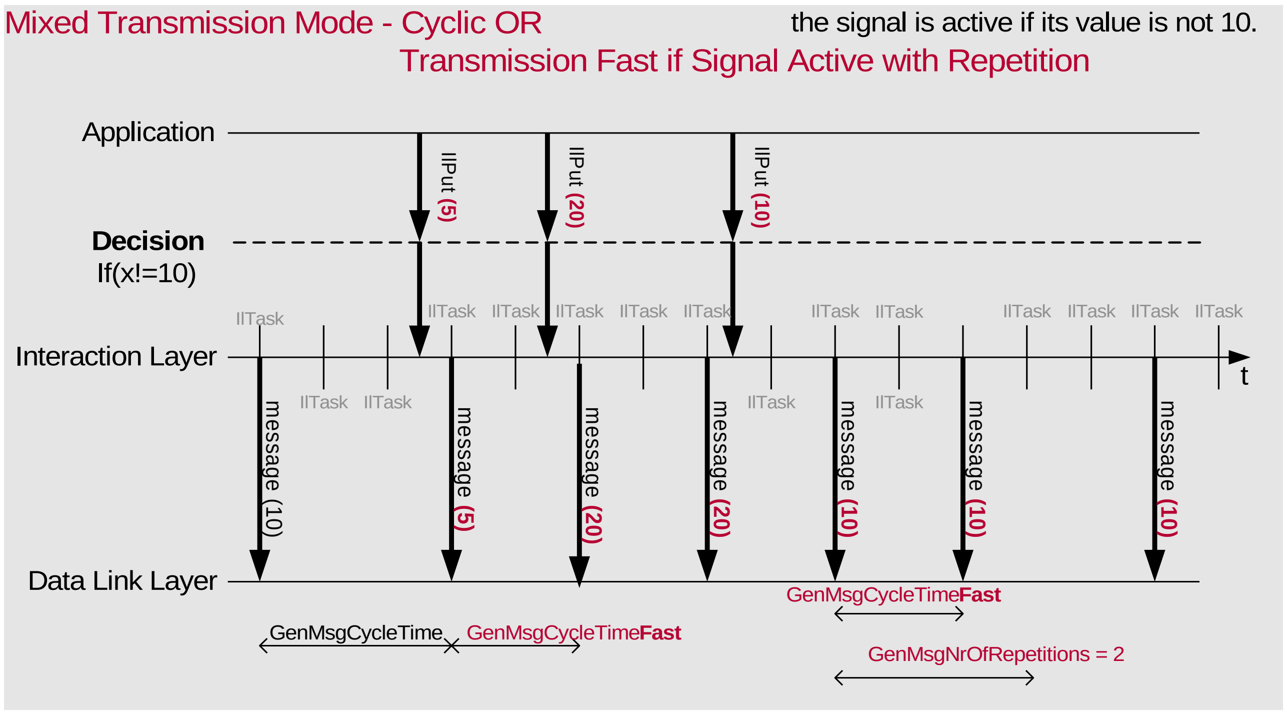


Figure 3-14 Mixed Transmission Mode – Cyclic OR Fast If Signal is Active with Repetition

3.5.3.8 Cyclic (Message) Transmission OR NoSigSendType

The message is sent cyclically with GenMsgCycleTime and with it all signals.

3.5.4 Advanced Transmission Modes

These combinations are only used by a few OEMs and described in the OEM-specific Interaction Layer documentation.

3.5.5 Notification Classes

Two types of notification classes are available which cause the notification of the application by flag or function. The flags are all set to 0 at the start of transmission (Function `IlTxStart()`).

- The Configuration Tool can be used to assign a separate **Confirmation Class** for each signal. This event will be set by the Data Link Layer if the particular message was sent on the bus.
  If a flag is used for notification, the application is responsible for clearing this flag.

	Caution This callback function is called in interrupt context! Reduce the runtime of this function to a minimum

The time between a transmit request and the actual transmission of a signal can be supervised by timeout monitoring. A **Timeout Class** will be set, if the signals were not transmitted within a defined period of time.

3.5.6 Reduction of Transmission Bursts

To prevent transmission bursts caused by interference the start of periodic transmission could be delayed. Therefore, a start delay time related to a message could be defined at compile time. Take e.g. three periodically transmitted messages. If the transmission of the three messages would be started at the same time, the simultaneous transmission of two or three messages will take place at same points in time. To delay the beginning of the periodic transmission of some messages is an appropriate way to reduce these simultaneous transmissions.

3.5.7 Delimitation of the Bus Load

To delimit the bus load a delay time will be inserted after each transmission of data. A defined delay time is related to a message. This means that delay time will be inserted no matter what transmission mode was used or by which signal the transmission of the message was caused.

The usage of a delay time may cause the delay of the transmission. For example the combination of a periodically transmitted message and a directly transmitted message could cause a delay as shown in Figure 3-15 Delay Time to Delimit Bus Load. The dashed arrows stand for periodically transmitted messages. If the direct transmission of a signal was requested by calling ILPut while the timer GenMsgDelayTime was not elapsed yet, the transmission of the signal will be delayed till the timer GenMsgDelayTime is elapsed.

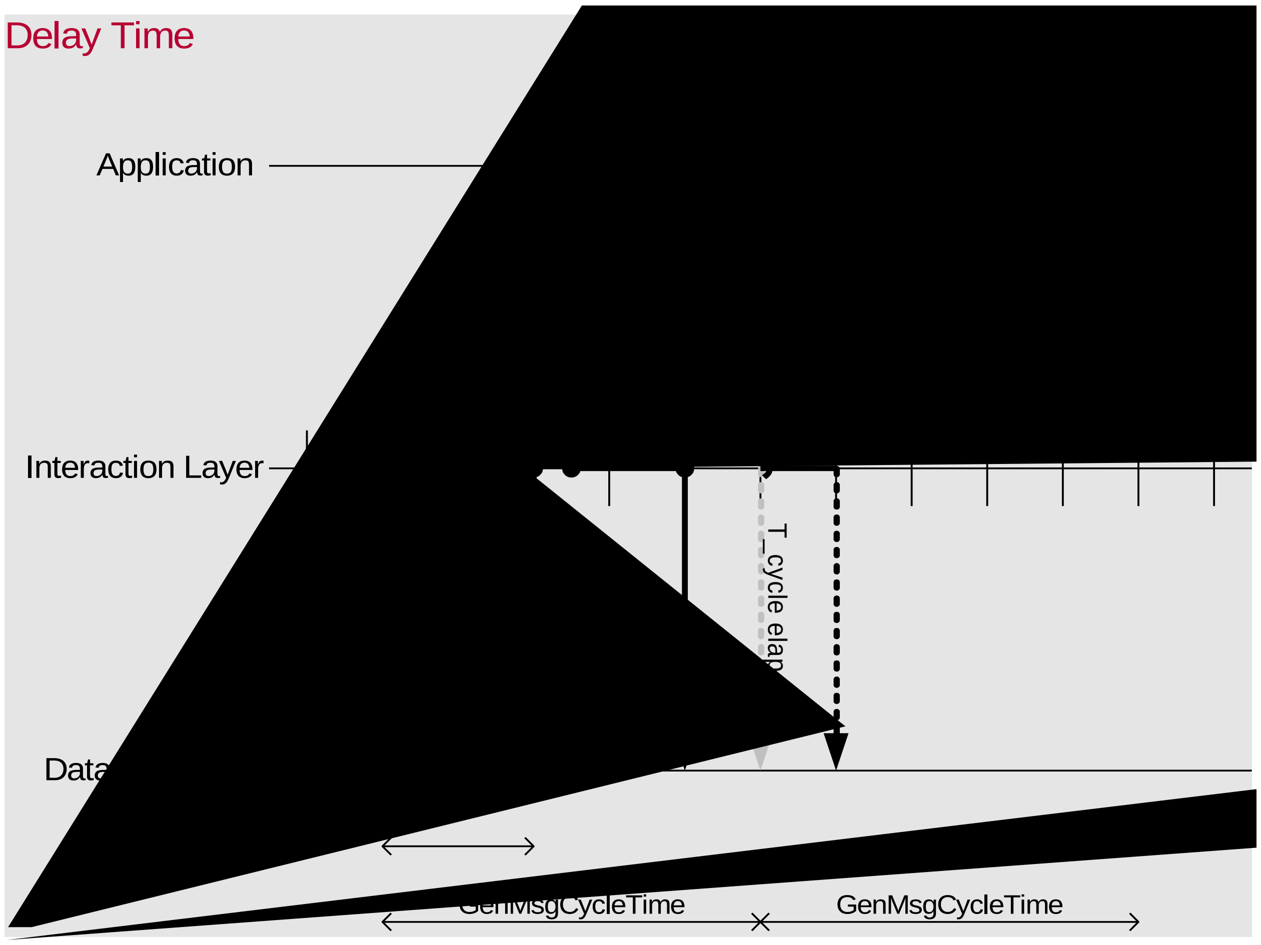


Figure 3-15 Delay Time to Delimit Bus Load

3.5.8 Transmission Timeout Monitoring

The Interaction Layer provides a feature to monitor the transmission of signals. The timeout monitoring for transmitted signals was intended to supervise the time between a transmit request and the actual transmission of the signal. Therefore, a timer will be activated, after a transmission was requested. If the successful transmission was notified (confirmation), the timer will be deactivated. If the timer elapsed before the confirmation, the application will be notified about the timeout.

To configure timeout monitoring for a signal the flag or function for the notification needs to be chosen in the Configuration Tool. If no notification was chosen, the timeout monitoring will be deactivated for this signal.

3.5.9 Transmission of Initialization Messages

The Interaction Layer provides a function to transmit a set of messages, called `IlSendOnInitMsg()`. This function is intended to be used at initialization time. It will transmit each of the messages in the set only once. The transmission modes of the signals and messages will be ignored. The messages belonging to the set can be configured at compile time by using the Configuration Tool.

3.6 Data Reception

3.6.1 Reception Concept

To receive new data the Interaction Layer has to process the messages immediately. Therefore all tasks will be interrupted to copy the new data to the RAM. On the other hand the time of interrupt should be as short as possible. By that reason all the jobs which are not time critical like for example the notification of the application are done by a periodic task. Therefore the Interaction Layer provides several functions and an interrupt handler. Further, the Interaction Layer provides several notification classes for example to notify the application about updated values. The usual response of the application is to read the updated signal values by calling functions or macros provided by the Signal Interface (`ILGetRx`**`signalname`**). Everything else will be done by the Signal Interface and the underlying Message Manager.

3.6.2 Notification Classes

Four types of notification classes are available which cause the notification of the application by flag or function. The flags are all set to 0 at the start of receiving (Function `IlRxStart()`). All classes are optional and in order to save code and run time they can be removed by a configuration switch at compile time, set in the Configuration Tool.

1) If a message was received an **Indication Class** will be set by the Interaction Layer. By this event the application can determine whether a new message has arrived.
   If a flag is used for notification, the application can always check the message contents for changes and perform tasks accordingly, if the flag is set. The application is responsible to clear the flag which was set by the Interaction Layer.
   If needed, multiple indication flags for a single signal can be configured. By this feature several parts of the application can be notified independently.
2) For periodic receive messages the Interaction Layer takes care of timeout monitoring. A **Timeout Class** is set, if a message which should be received periodically arrives too late. Too late means that a new message with the same ID was not received after timeout time (Example: timeout time = 2.5x cycle time) elapsed. The timeout time of a CAN-message is defined at compile time. All timeout functions are running in the same context as the `IlRxTask()` does.
   If a flag is used for notification, the flag will be set and cleared by the Interaction Layer. The application is also allowed to clear the flag, but the Interaction Layer will always overwrite it, if an event occurred.
3) The **First-Value Class** notifies the application about the first reception of a signal. This is done by setting an event each time a signal has been received. Actually, the First-Value Class and the Indication Class are working the same way.
   Only a flag could be used as notification mechanism for this class. The flag will be set by the Interaction Layer. The application is responsible to clear the flag. The flag will be reset by `IlRxStart` to recognize for example the first received message after power-on or bus-off.
4) A **Data-Changed Class** is provided for ECUs with processors of a higher performance class. This event is set if the message contents of an incoming CAN message differs from the contents of the memory in the controller's CAN buffer. For Full-CAN objects this requires a copy of the message in RAM (increased RAM requirement in the controller). Using a mask, portions of the message can be masked out for the comparison. Since the mask is applied to the message bit wise, changes involving partial information of a signal can be used (e.g. only the upper 12 bits of a 16 bit signal).
   Only a flag can be used as notification mechanism for this class. The flag will be set by the Interaction Layer. The application is responsible to clear the flag.

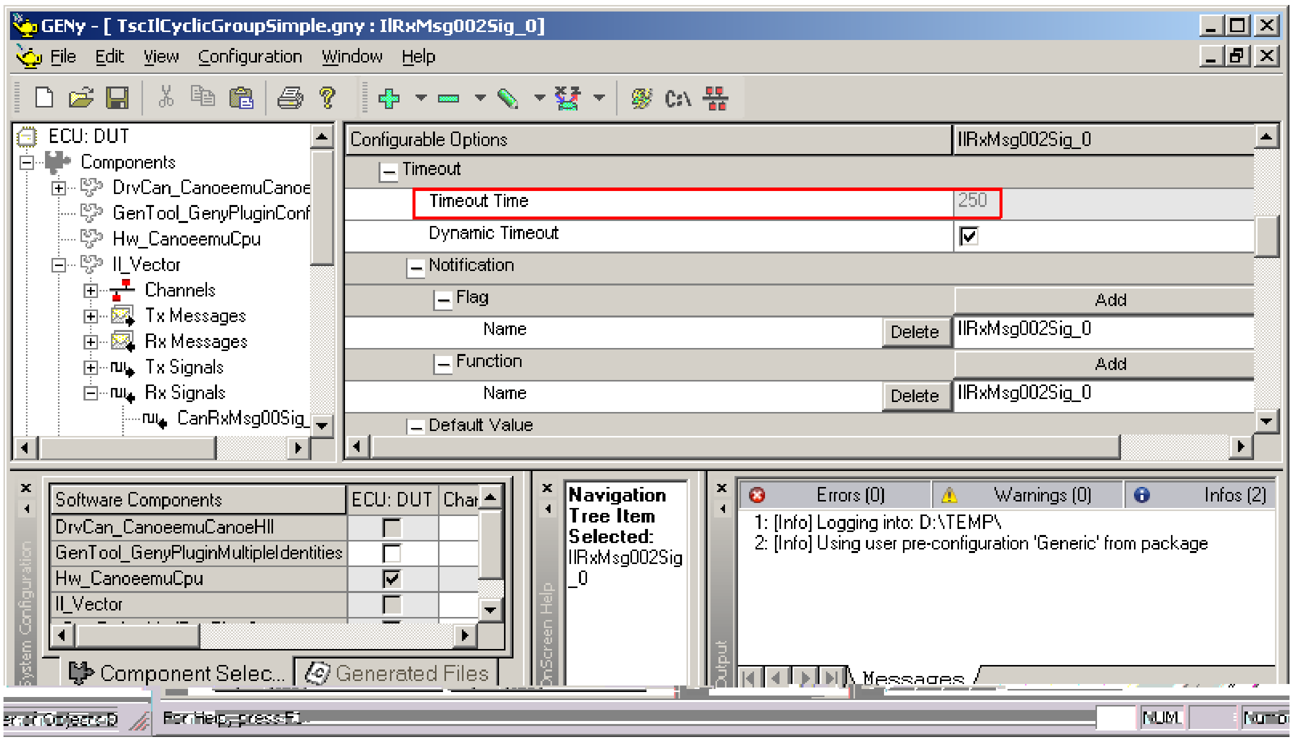
3.6.3 Timeout Monitoring

The periodic receipt of messages which are periodically transmitted by another network node could be overseen by timeout monitoring. For this purpose a time-out timer is provided by the Interaction Layer. This timer will be restarted each time the related message was received. If the timeout timer elapses before the next related message was received, the application will be notified (see chapter 3.6.2 Notification Classes). When a timeout occurs the current value of the message is not valid anymore. In this case the message's memory area in the controller can be pre-filled with 0 (zero) or with a default value (if defined) in order to preserve emergency operation of the application.

The attribute `GenSigTimeoutTime_<ECU>` in the network database needs to be set to activate the timeout monitoring. The attribute GenSigTimeoutMsg_<ECU> may be set to the default value. Then the message, which contains the current signal, will be monitored.

In the following example the configuration of the attributes in the network database will be shown. A network node A transmits a signal to network node B by the Periodic Transmission Mode. Network node B as the receiver of the signal wants to monitor the periodic reception. Therefore, the attributes GenSigTimeoutMsg_<ECU> and GenSigTimeoutTime_<ECU> needs to be adapted for this signal. For the timeout monitoring by the network node B the name of the two attributes must be changed to GenSigTimeoutMsg_**B** and GenSigTimeoutTime_**B**. If another network node, for example network node XY, wants to monitor the reception of this signal, too, the attributes GenSigTimeoutMsg_**XY** and GenSigTimeoutTime_**XY** have to be added. Further the values need to be defined for the attributes. For the attribute GenSigTimeoutMsg_B we need to fill in the message ID of the message which includes the signal to be monitored. The attribute GenSigTimeoutTime_B contains the timeout time. If the signal was not received within this time, the application will be notified.

The value of the dbc attribute GenSigTimeoutTime_<ECU> is displayed on each rx signal in the GENy GUI.



Configurable Options	IlRxMsg002Sig_0
Timeout	
Timeout Time	250
Dynamic Timeout	✓
Notification	
Flag	Add
Name (Delete)	IlRxMsg002Sig_0
Function	Add
Name (Delete)	IlRxMsg002Sig_0
Default Value	

Software Components	ECU: DUT	Char
DrvCan_CanoeemuCanoeHll	☐	
GenTool_GenyPluginMultipleIdentities	☐	
Hw_CanoeemuCpu	✓	
Il_Vector	☐	

```
1: [Info] Logging into: D:\TEMP\
2: [Info] Using user pre-configuration 'Generic' from package
```

	Caution If the Timeout Time is editable the value is **not** derived from the dbc attribute GenSigTimeoutTime_<ECU>, in this case the timeout time can only be edited in the GENy GUI.

For diagnosis efforts the Interaction Layer provides an advanced timeout monitoring for messages. This includes the possibility to reload the timeout timer and the message related notification of the application when this timer elapsed again. I.e. after the first timeout occurred the Interaction Layer will notify the application about the timeout of each signal included in the message. Then the application can reload the timeout timer. After this timer elapsed again, the Interaction Layer will notify the application about the timeout of the whole message. After the reload of the timer the application won't be notified about the signal's timeout again.

3.6.4 Dynamic Timeout Monitoring

If it is required to assign different timeout values to a timer or to start and stop at timer at run-time, a special API can be activated for each signal. The change of one signal of a message influences all signal timers of a message. The dynamic timeout counters are treated by the state machine in the same way as normal timeout events. The timeout defined in the database is the initial timer, which is set on IlInitPowerOn.

Example

You have to supervise a signal, which is received every 200 ms. If a first timeout after 500 ms is detected, another timeout is started. After the following timeout of 4 s, a fault memory entry has to be logged.

```
/* check timeout flag */
if (IlGetRxGwDataTimeout())
{
  /* clear timeout flag */
  IlClrRxGwDataTimeout();
  /* read current timeout value */
  if (IlGetRxGwDataDynRxTimeout() == 500)
  {
    /* First Timeout Level */
    IlSetRxGwDataDynRxTimeout(4000); /* assign new timeout value */
    IlStartRxGwDataDynRxTimeout();/* start timer */
    ToggleEcuState();
  }else
  {
    /* Second Timeout Level */
    IlStopRxGwDataDynRxTimeout();/* stop timer */
    SetErrorMemoryEvent();
    /* Reset the timeout start,
    if the signal is received the next time */
    IlSetRxGwDataDynRxTimeout(500);
  }
}
```

	Caution The timeout value access is implemented as macro. The parameter IlSetRx<SignalName>DynRxTimeout and the return value of

	IlGetRx<SignalName>DynRxTimeout is always IltRxTimeoutCounter which is defined to vuint16. Due to this, the maximum timeout counter is 65535.

3.7 Signal status information (UpdateBits)

The UpdateBit Support is used to indicate whether the application has updated the Signal value.[8] Only Signals and Signal Groups can have UpdateBits. A partial signal cannot have an UpdateBit. Multiplexed Signals can have UpdateBits if the UpdateBit is multiplexed with the same multiplexor value.

Info
For detailed information see AUTOSAR Specification of Module COM [8]

3.7.1 Configuration

An UpdateBit has no configurable attributes in GENy. There is **no** special switch in GENy to enable UpdateBit support.

Caution

- UpdateBits cannot be used in combination with Multiplex API Raw.
- UpdateBit Support needs the CanCopyToCan Driver API.[1]
- UpdateBit cannot be used in combination with dynamic DLC.
- GroupSignals (partial signals) cannot have UpdateBits.
- UpdateBit cannot be used if common buffer is used without identity manager.

3.7.1.1 DBC File

In the DBC file an UpdateBit has the size of one Bit and the postfix "_UB". The UpdateBit name is a combination between the UpdateBit Signal name and the postfix "_UB" e.g. the Signal <SignalName> has the UpdateBit <SignalName>_UB.

UpdateBit dbc file attributes:

GenSigSendType	Fix: NoSigSendType
GenSigTimeoutTime_<ECU>	Message specific timeout time

Signal with UpdateBit attributes:

GenSigTimeoutTime_<ECU>	UpdateBit specific timeout time

3.7.2 UpdateBit Transmission

If the application writes via a Put Macro a Signal with an UpdateBit, the UpdateBit will be set to one. The UpdateBit is reset to zero after the message is transmitted once.

Info

- If a message has signals with UpdateBits the PreTransmitt function is used by the Il_Vector.
- RDS macros can't be used in combination with UpdateBits.

	■ It is possible to use the AUTOSAR Signal Interface in combination with UpdateBits.

3.7.3 UpdateBit Reception

The Indication Flag/Function of a Signal with UpdateBit will only be set/called if the UpdateBit of the Signal is set.

3.7.3.1 Timeout

A Signal with an UpdateBit can have a Signal specific timeout. The Signal specific timeout monitors the time between two UpdateBits equal 1. The value of the DBC Attribute “GenSigTimeoutTime” on the Signal with the UpdateBit is the UpdateBit specific timeout time.

The value of the DBC Attribute “GenSigTimeoutTime” is displayed as timeout time on each RxSignal in the GENy GUI. If the timeout time is editable in the GENy GUI the value is **not** derived from the DBC Attribute “GenSigTimeoutTime” and must be configured in the GUI.

	Info The UpdateBit specific timeout can be used in combination with Dynamic Timeout Monitoring.

3.8 Multiple Channel Support

3.8.1 Overview

Sometimes two or more CAN Controllers are used on the same CAN bus. Therefore, the CAN Driver and the Interaction Layer have to be adapted to multi channel support. This could be done in two ways. First, the whole source code could be doubled. We will refer to this as Crx (Code replicated) Interaction Layer. Second, a single source code could work with doubled data buffers. We will refer to this as Idx (Indexed) Interaction Layer because the access to the data buffers will be controlled by an index. The following two sections will describe these two possibilities.

Note that only API services which relate to one specific channel, i.e. one physical medium have to distinguish between different channels. Signal access services are not channel related.

3.8.2 Idx (Indexed) Interaction Layer

An Idx Interaction Layer will work on two or more CAN busses without doubling of code. It will work with multiple data buffers which can be accessed by an index. This results in a kind of array. And even the access by the application will be similar to an array. Function names won't get a suffix as for the Crx Interaction Layer. The access to the different buffers will be done by a parameter. We will refer to this parameter as index.

For example the function call `IlTxTask()` of a single channel Interaction Layer will result in `IlTxTask(` **`0`** `)` for channel number 0 and `IlTxTask(` **`1`** `)` for channel number 1 of an Idx Interaction Layer. However, the initialization of all the channels will be handled by the single function `IlInitPowerOn()`.

3.9 Advanced Communication Features

3.9.1 Physical Multiple and Multiple Configuration ECU

Please see in [2].

3.9.2 Multiplexed Signals

To save message IDs the Interaction Layer supports the use of several message layouts for a single message. The signals combined in this several layouts are called multiplexed signals. The current message layout will be indicated by a multiplexor signal (mode signal).

3.9.2.1 Standard API

The Interaction Layer will provide data access and notification mechanisms for all multiplexed signals. I.e. multiplexed signals will be encapsulated by the Interaction Layer. However, here are some restrictions and some helpful information for the use of multiplexed signals:

The several layouts are defined in the network database

The current layout will be chosen by the use of a multiplexor signal (mode signal)

With multiplexed signals only the Cyclic transmission modes can be used

The actual cycle time of a multiplexed signal can be calculated by the following formula:
Message Cycle Time * Number of Message Layouts = Multiplexed Signal Cycle Time

A delay time has to be specified for messages with multiplexed signals by means of the attribute GenMsgDelayTime.

Data changed flags are not supported for multiplexed signals

Note that multiplexed signals are provided by the Interaction Layer in version 3.27 and higher.

3.9.2.2 Raw API

The Interaction Layer provides a raw interface for multiplex signals. The advantage is runtime improvement, reduction of Ram and Rom requirements.

Example
The following code example demonstrates the implementation of a reception of a multiplexed signal.

- Configure a PreCopy function for the multiplex message in the CAN Driver.
- Configure RDS access to the signals multiplexor signal and multiplexed signals, of the message MultiplexMessage. MultiplexedSignal is for example valid, if the MultiplexorSignal is 0x10

```
vuint16 MyMultiplexedSignal;
/* Implementation of the reserved indication function */
void ApplRawApiMultiplexMessagePrecopy(CanReceiveHandle
rxObject)
{
   /* Get the multiplexor value */
   vuint8 MyMultiplexorSignal;
   MyMultiplexorSignal = IlGetRxCANMultiplexorSignal();

   /* Check for the correct multiplexor */
   if ( MyMultiplexorSignal == 0x10 )
```

```
    {
      /* Get the multiplexed signal */
      MyMultiplexedSignal = IlGetRxCANMultiplexedSignal();
    }
    if ( MyMultiplexorSignal == 0xA3 )
    {
      /*
      Implement the reception of other Multiplexed Signals
      for the Multiplexorvalue 0xA3 here
      */
    }
    ............
    /* The returnvalue kCanNoCopyData is mandatory */
    return kCanNoCopyData;
}
```

Example

The following code example shows the implementation of a transmission of a multiplexed signal.

- Configure a Pretransmit function for the multiplex message in the Can Driver.
- Configure Rds access to write the Multiplexor signal and multiplexed signals, of the message multiplex message. MultiplexedSignal for example is valid, if the MultiplexorSignal is 0x10

```
vuint16 MyMultiplexedSignal;
vuint16 MyMultiplexorValue;

void
ApplRawApiMultiplexMessagePretransmit(CanTransmitHandle
txObject)
{
   /* Implementation of the Multiplexor toggle mechanism */
  if ( MyMultiplexorValue == 0x43 )
  {
    IlPutTxCANMultiplexorSignal(0x10);
    IlPutTxCANMultiplexedSignal(MyMultiplexedSignal);
    /*
    Implement here the transmission of other signals for the
    multiplexor 0x10
    */
    MyMultiplexorValue = 0x10;
  }
  else if ( MyMultiplexorValue == 0x10 )
  {
     IlPutTxCANMultiplexorSignal(0x43);
     /*
    Implement here the transmission of other signals for the
    multiplexor 0x43
    */
    MyMultiplexorValue = 0x43;
  }
  …………
  /* The returnvalue kCanNoCopyData is mandatory */
  return kCanNoCopyData;
}
```

3.9.3 Manipulation of the Notification Frequency

The periodic tasks for transmission and reception (See 3.2 Main Functions) are responsible for all the cyclic tasks the Interaction Layer has to do. Both need runtime and for this reason the application developer wants to set the cycle time to invoke the tasks as high as possible. A high cycle time has the major drawback that the notification about urgent events will slow down, too. This is because the events will be checked within these tasks. I.e. they are checked in the same cycle time.

To solve this problem the Interaction Layer provides a function called `IL<Tx/Rx>StateTask`. This function is responsible for the notification of the application. It will check, if there was any event the application wants to be notified about and notifies the application. The function will be invoked by the IL<Tx/Rx>Task in the defined cycle time. Further this function can be invoked by the application any time it is necessary. So the developer can shorten the time between checking for new events and for notification.

To reduce the time between an event and the notification even more, the application can be notified in interrupt context. This feature can be configured by the Configuration Tool for each message and will have effect on each signal the message contains. Checking for events in interrupt context means checking for events each time a message was transmitted or received, respectively. This is the fastest way to be notified by the Interaction Layer.

It’s recommended to think about the consequences of using the StateTask or the notification in interrupt context, respectively. Both ways to speed up the notification has pros and cons.

4 Integration

This chapter gives necessary information for the integration of the Interaction Layer into an application environment of an ECU.

4.1 Include structure

To use the Vector Interaction Layer, only the file il_inc.h must be included in all application components that want to use Interaction Layer functionality. The file can_inc.h (which provides the CAN Driver interface and data buffers) must not be included separately, it is automatically included by il_inc.h.

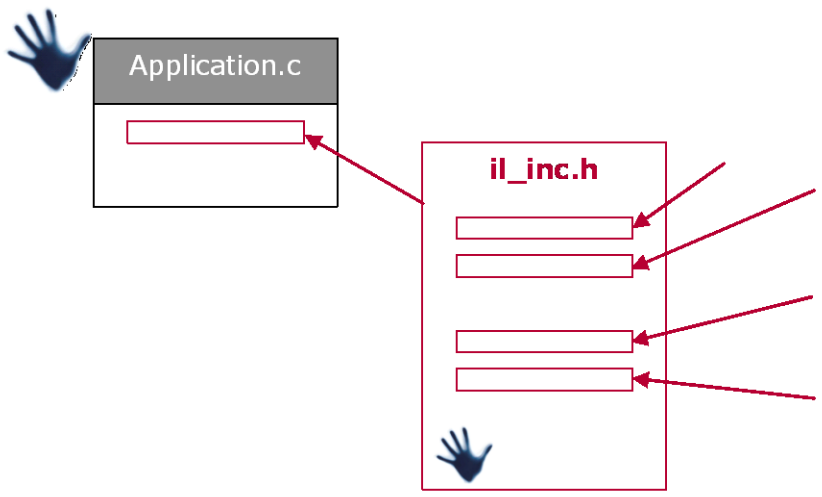


Figure 4-1 Including Interaction Layer

4.2 Scope of Delivery

The delivery of the Interaction Layer contains the files which are described in the chapters 4.2.1 and 4.2.2:

4.2.1 Static Files

File Name	Description
il_inc.h	This is the header file to be included by other components to use the Interaction Layer.
il_def.h	This is the header file of the Interaction Layer.
il.c	This is the source file of the Interaction Layer.

Table 4-1 Static files

4.2.2 Dynamic Files

The dynamic files are generated by the configuration tool GENy.

File Name	Description
il_cfg.h	This is the generated header file containing pre-compile switches.
il_par.h	This is the generated header file providing symbolic defines, macros and prototypes.
il_par.c	This is the generated source file containing generated parameters and functions.

Table 4-2 Generated files

4.3 Operating Systems Requirements

The CAN communication components are designed and programmed to work with or without operating systems. Since the components have to work without an operating system, resource locking mechanisms are not handled. To lock critical resources, interrupts will be disabled and restored. The CAN driver (Data Link Layer) provides functions to fulfil this task.

Each component has one or two functions (tasks) which have to be called periodically. For operating systems it is advisable to create one task and call all the Interaction Layer component functions subsequently. To implement different periods of time, the OS task could have a counter to implement this.

Data consistency issues:

Cyclic IL tasks are not allowed to interrupt signal accesses. This has the following consequences:

> No cyclic IL task shall be called on Interrupt level e.g. directly in a timer ISR.

> In a priority driven multitasking operating system with preemptive scheduling such as OSEK-OS cyclic IL tasks should have a lower priority than the tasks performing signal accesses.

To ensure data consistency on pre-emptive multi-tasking operating systems or when using IL signal access services on interrupt level, there are two things to keep in mind.

> The Interaction Layer provides mechanisms to keep data consistency on multi-byte signals. That means, reading multi-byte data is always done while interrupts are locked. In that case, no task switch can occur. The disadvantage to that mechanism is a longer interrupt latency time. If your system is critical to long latency times, ensure that your system works properly in all cases.

> Bit field manipulation is done by macros. Some compilers and processors realize bit field manipulation by read-modify-write accesses. If data access to bit fields in the same byte is used in pre-emptive tasks or on interrupt level, a problem could be caused. Try to avoid this or make resource locking to such accesses.

These issues can be circumvented by using the Interaction Layer API only in non-preemptive tasks.

5 Configuration

In the Interaction Layer the attributes can be configured with the following methods:

> Configuration in GENy for a detailed description see 5.2 Configuration with GENy

> Configuration in Database, for a detailed description see chapter 5.1 Configuration in Data Base

5.1 Configuration in Data Base

The following attributes can be used to configure the Interaction Layer in the DBC file.

Info
Bold Value Types should be Used as default. Value Types marked with * are available for CANgen compatibility reasons.

Caution
Don't mix up the order of enumeration values. Not the value of the attribute is interpreted, the position of the selected value.

Caution
The "Type of Object" can be configured in the dbc file for some attributes as "Signal" or "Node – Mapped Rx Signal". Use only one "Type of Object" in a single dbc file. If the "Type of Object" is "Signal", the attribute must be defined in the dbc file for each Ecu. Due to this, replace <ECU> in the attribute name by the Node name.

Name	ILUsed
Description	This attribute must be defined, to use the Vector Interaction Layer with this node. 0 : The node is cannot be used with the Interaction Layer 1 : The node is can be used with the Interaction Layer
Type Of Object	Node
Value Type	Enumeration
Default	No
Values	No, Yes

Name	GenMsgILSupport
Description	Configurate the usage of the Interaction Layer for this message 0 : The message is not used by the Interaction Layer 1 : The message is used by the Interaction Layer
Type Of Object	Message
Value Type	Enumeration
Default	Yes
Values	No, Yes

5.1.1 Send Type

	Info The strings used for the GenMsgSendType is often OEM-specific and can differ from here.

Name	**GenMsgSendType**
Description	Message related transmission mode. Use only Cyclic for messages with multiplexed signals.
Type Of Object	Message
Value Type	Enumeration
Default	NoMsgSendType (Use only signal related transmission modes.)
Values	Cyclic, NotUsed, NotUsed, NotUsed, NotUsed, NotUsed, NotUsed, IfActive, NoMsgSendType

Name	**GenSigSendType**
Description	Signal related transmission mode. Use only NoSigSendType for messages with multiplexed signals.
Type Of Object	Signal
Value Type	Enumeration
Default	NoSigSendType
Values	Cyclic, OnWrite, OnWriteWithRepetition, OnChange, OnChangeWithRepetition, IfActive, IfActiveWithRepetition, NoSigSendType, OnChangeAndIfActive, OnChangeAndIfActiveWithRepetition

5.1.2 Send Type Dependent

Name	GenMsgCycleTime
Description	Time in ms between each cyclic transmission of a message.
Type Of Object	Message
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Name	GenMsgCycleTimeFast
Description	Value of the second cycle time, if the GenMsgSendType/GenMsgSendType IfActive or GenMsgFastOnStart is configured.
Type Of Object	Message
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Name	GenMsgNrOfRepetition
Description	Number of repetitions used if the GenSigSendType is OnChangeWithRepetition, OnWriteWithRepetition, IfActiveWithRepetition or OnChangeAndIfActiveWithRepetition is defined. The number of repetitions can be configured separately for each message. To reduce Rom and Ram requirements configure the same number for each message. This value defines how often a message is sent before its transmission is stopped. See e.g. Figure 3-7 Timing Diagram of OnEvent with Repetition - **OnWrite**.
Type Of Object	Message
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Caution
Please note, the attribute GenSigStartValue sets the Default value at initialization time, not if IlRxStart or IlTxStart is called. Due to historical and compatibility reasons, this confusing definition cannot be changed any more.

Name	GenSigStartValue
Description	This Value is the default value for the signal, if IlInitPowerOn is called. The string value type can represent hexadecimal and integer values.
Type Of Object	Signal
Value Type	**String**, Integer*, Float*
Default	0x0
Minimum	0x0
Maximum	0xffffffffffffffff

Name	GenSigInactiveValue
Description	Value for which Transmit Fast If Active will be set inactive. It is not recommended to use Hex values, due to the range restriction. The string value type can represent hexadecimal and integer values. The usage of the hex value is not recommended, because it cannot represent values greater than 0x7fffffff.
Type Of Object	Signal
Value Type	**String**, Hex*
Default	0x0
Minimum	0x0
Maximum	0xffffffffffffffff Hex : 0x7fffffff

Name	GenSigTimeoutValue
Description	This Value is the timeout default value for the signal, if a timeout occurs. The integer value allows the definition of timeout values for signals with a maximum Length of 4 Bytes.
Type Of Object	Signal
Value Type	Integer
Default	0x0
Minimum	0x0
Maximum	4294967296

5.1.3 Advanced Attributes

Name	GenMsgDelayTime
Description	This is the minimum time in ms between the transmissions of messages with the same identifier.
Type Of Object	Message
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Name	GenMsgStartDelayTime
Description	This is the time in ms after IlTxStart has been called, when the cyclic transmission event starts. If a transmission is triggered by OnEvent, OnEventWithRepetition, IfActive, IfActiveWithRepetition within the MsgStartDelayTime, the transmission is not performed within the GenMsgStartDelayTime.
Type Of Object	Message
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Name	GenMsgFastOnStart
Description	This is the time in ms after IlTxStart has been called, where the message is transmitted cyclic with GenMsgCycleTimeFast. The value has to be an integer multiple of GenMsgCycleTimeFast.
Type Of Object	Message
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

5.1.4 Timeout Supervision Attributes

Name	ILTxTimeout
Description	Value of the timeout time in ms for Tx used for all messages within the channel. To use the timeout monitoring it must be activated for each signal in the Configuration Tool.
Type Of Object	Network
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Name	GenSigTimeoutMsg_<ECU>
Description	Message ID to enable the timeout monitoring for signals which are not transmitted periodically by the receiver <ECU>. If this attribute is set to default the message will chosen, which contains the current signal. If you must reference extended IDs, use the following representation format, where the CAN identifier is combined with 0x80000000 by a logical or. Example: ID 0x208 is used for the standard ID and ID 0x80000208 for the extended ID.
Type Of Object	Signal
Value Type	Hex
Default	0
Minimum	0x80000000
Maximum	0xfffffff

Name	GenSigTimeoutMsg
Description	Message ID to enable the timeout monitoring for signals which are not transmitted periodically by the receiver Ecu. If this attribute is set to default the message will chosen, which contains the current signal. If you must reference extended IDs, use the following representation format, where the CAN identifier is combined with 0x80000000 by a logical or. Example: ID 0x208 is used for the standard ID and ID 0x80000208 for the extended ID.

Name	GenSigTimeoutTime
Description	Timeout time in ms used for this signal received by Ecu. If different GenSigTimeoutTime values are configured for a message, the lowest timeout time (strongest definition) is used for timeout monitoring.
Type Of Object	Node – Mapped Rx Signal
Value Type	Integer
Default	0
Minimum	0
Maximum	65535

Name	GenSigSuprvResp
Description	This value preconfigurates the timeout flag and timeout default value. 0 : Preconfigurate nothing 1 : A timeout flag is configured for the signal 2 : A timeout default value is configured for the signal 3 : A timeout flag and timeout default value is configured for the signal
Type Of Object	Node – Mapped Rx Signal
Value Type	Enumeration
Default	None
Values	None, TimeoutFlag, TimeoutDefaultValue, TimeoutFlag and TimeoutDefaultValue

Name	GenSigSuprvRespSubValue
Description	This Value is the timeout default value for the signal, if a timeout occurs. The integer value allows the definition of timeout values for signals with a maximum Length of 4 Bytes.
Type Of Object	Node – Mapped Rx Signal
Value Type	Integer
Default	0x0
Minimum	0x0
Maximum	4294967296

5.1.5 Former Attributes

The following attributes are not supported any more.

Name	GenMsgNoIalSupport
Replaced by	GenMsgILSupport
Description	GenMsgILSupport is the inverted view of the attribute GenMsgNoIalSupport.

5.1.6 Example

A signal A and a signal B are included in the message XY. Signal A should be transmitted periodically by the cycle time of 50ms. Signal B should be transmitted each time an event occurs.

For signal A we chose the Periodic Transmission Mode (Transmission Modes see chapter 3.5.2). For the event driven transmission of signal B, the Direct Transmission Mode will fit best.

Next step is to choose the attributes for this constellation. To chose the Periodic Transmission Mode for signal A we need to set the GenSigSendType to Cyclic and the GenMsgCycleTime to 200 [ms]. To choose the Direct Transmission Mode for signal B we need to set the GenSigSendType to OnWrite. This will result to the attributes listed in the table below.

Node Attribute	Value	Comment
ILUsed	Yes	Yes, use the Interaction Layer within this network node.

Attribute for Message XY	Value	Comment
GenMsgDelayTime	Yes	not necessary for this example
GenMsgCycleTime	50 [ms]	Value of the cycle time of the message. There can’t be signals with different cycle times combined in one message. Therefore this is the cycle time of signal A.
GenMsgCycleTimeFast		not necessary for this example
GenMsgStartDelayTime		not necessary for this example
GenMsgILSupport	Yes	Enable the usage of the Interaction Layer for this message (for messages used by diagnosis, network management ...).
GenMsgNrOfRepetition		not necessary for this example
GenMsgSendType	NoMsgSendType	We use only signal related transmission modes within this example. Therefore, see GenSigSendType.

Attribute for Signal A	Value	Comment
GenSigSendType	Cyclic	Use the Periodic Transmission Mode for signal A.
GenSigInactiveValue		not necessary for this example
GenSigTimeoutMsg_<ECU>		not necessary for this example

GenSigTimeoutTime_<ECU>		not necessary for this example
Attribute for Signal B	**Value**	**Comment**
GenSigSendType	OnWrite	Use the Direct Transmission Mode for signal B.
GenSigInactiveValue		not necessary for this example
GenSigTimeoutMsg_<ECU>		not necessary for this example
GenSigTimeoutTime_<ECU>		not necessary for this example

The result of combining signal A and signal B with different Transmission Modes to one message XY will be shown in the timing diagram below. The dashed lines are used in consideration of signal A. The solid lines are used in consideration of signal B.

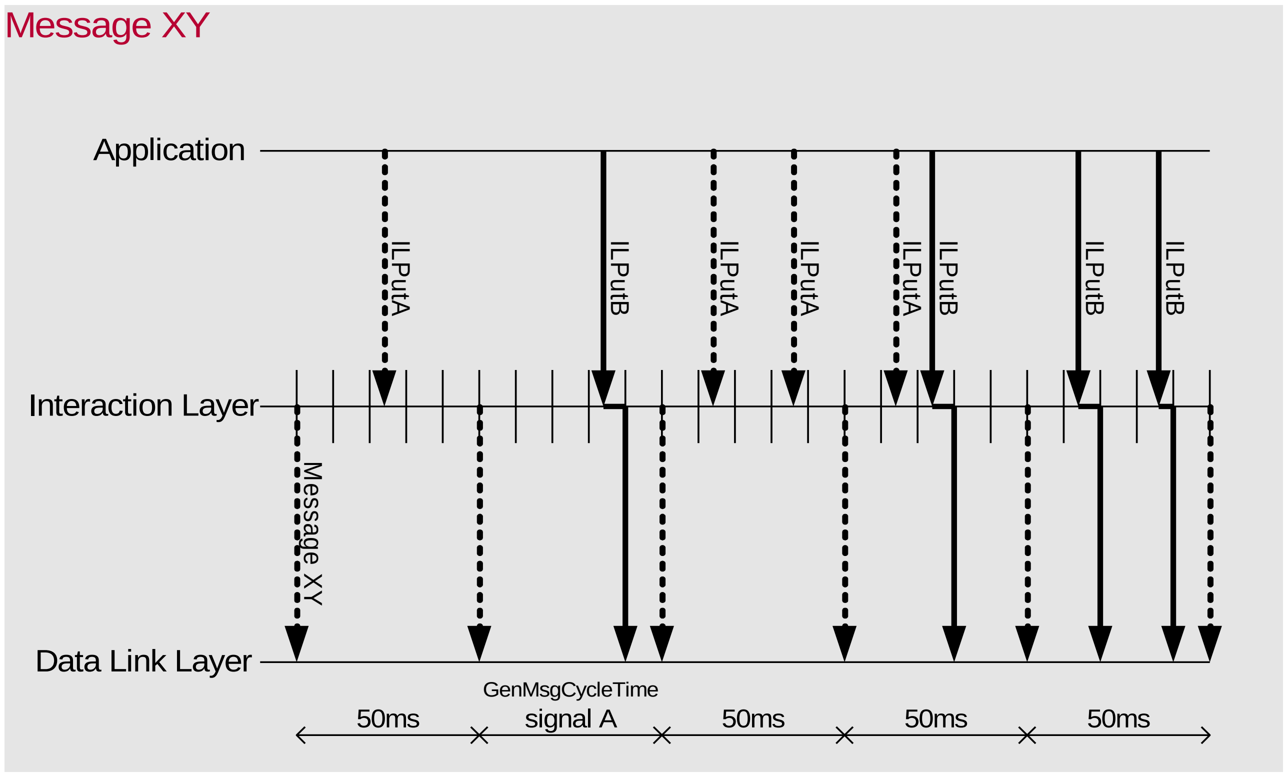


Figure 5-1 A Signal w ith the Periodic Transmission Mode and one w ith the Direct Transmission Mode Combined to a message.

5.2 Configuration with GENy

	Info The detailed information for all checkboxes and settings is given in the so-called OnScreen Help view of GENy just by clicking the checkbox or the name of the switch. This is activated via **View | OnScreen Help**. Information about how to work with GENy can be found in the OnlineHelp of GENy opened via **Help | Help topics**.

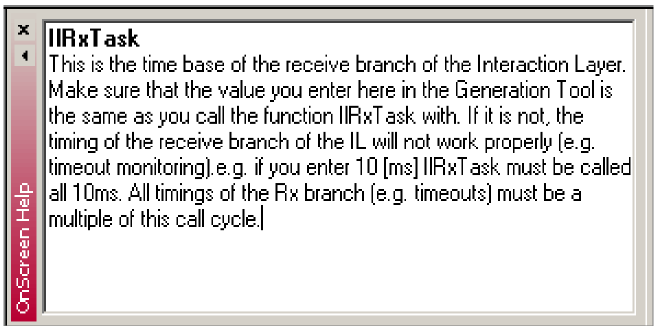
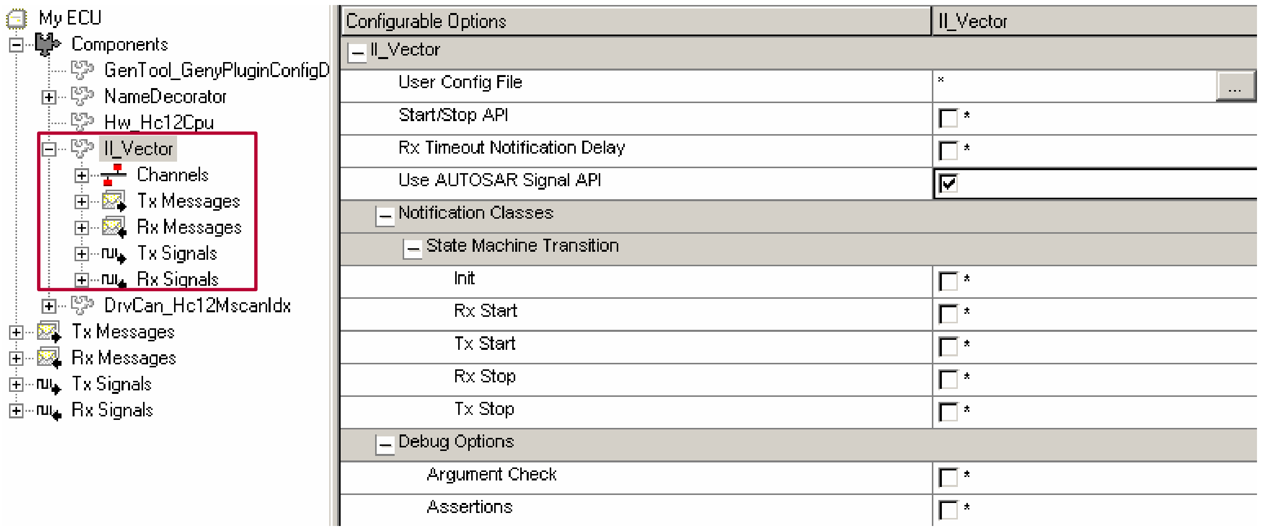

IlRxTask
This is the time base of the receive branch of the Interaction Layer. Make sure that the value you enter here in the Generation Tool is the same as you call the function IlRxTask with. If it is not, the timing of the receive branch of the IL will not work properly (e.g. timeout monitoring).e.g. if you enter 10 [ms] IlRxTask must be called all 10ms. All timings of the Rx branch (e.g. timeouts) must be a multiple of this call cycle.

Figure 5-2 OnScreen Help View for fast information

In the Configuration Tool GENy all settings for the Vector Interaction Layer are done via the marked tree items.



Configurable Options	Il_Vector
Il_Vector	
User Config File	*
Start/Stop API	☐ *
Rx Timeout Notification Delay	☐ *
Use AUTOSAR Signal API	☑
Notification Classes	
State Machine Transition	
Init	☐ *
Rx Start	☐ *
Tx Start	☐ *
Rx Stop	☐ *
Tx Stop	☐ *
Debug Options	
Argument Check	☐ *
Assertions	☐ *

Figure 5-3 Overview of Vector Interaction Layer Configuration in GENy

IL Vector

Configure the basic settings for the Vector Interaction Layer. Use the OnScreen Help View of GENy to get more information about each option.

	Support for AUTOSAR The Vector Interaction Layer can be configured to support a signal API according to the AUTOSAR Specification of Module COM [8]. This has effects on the signal API, see more there…

Channels

Define the task call cycle times.

Tx/Rx Messages

Most of those entries are fixed and already set in the DBC file and explained for information only.

Tx/Rx Signals

This is where to configure the settings for the signals, its access macros, the way of notification, etc.

Attribute Name	Value Type	Default Value	Description
ModuleInstance > Il_Vector			
User Config File	String	N.a.	The Interaction Layer configuration file (il_cfg.h) is generated by GENy. If you want to overwrite settings in the generated Il_Vector configuration file (il_cfg.h), you can specify a path to a user defined configuration file. The user defined configuration file will be included at the end of the generated file il_cfg.h. Therefore definitions in the user defined configuration file can overwrite definitions in the generated configuration file.
Start/Stop API	Boolean	false	The two API functions 'IlStartCycle' and 'IlStopCycle' are enabled which enable or disable the transmission of one single cyclic message. This option is necessary for software tests or special use cases. It is recommended not to use this option.
Timeout Monitoring on first Reception	Boolean	false	If this option is enabled timeout monitoring of a message is started when this message is received for the first time. Normally timeout monitoring starts after the Rx part of the Interaction Layer is started or resumed.
ModuleInstance > Il_Vector > Notification Classes > State Machine Transition			
Init	Boolean	false	This callback function is called after the Interaction Layer has been initialized. The callback function itself (ApplIlInit()) must be provided by the application.

Attribute Name	Value Type	Default Value	Description
Rx Start	Boolean	false	This callback function is called if the Rx branch of the Interaction layer is started. The callback function (ApplIlRxStart()) itself must be provided by the application.
Tx Start	Boolean	false	This callback function is called if the Tx branch of the Interaction layer is started. The callback function itself (ApplIlTxStart()) must be provided by the application.
Rx Stop	Boolean	false	This callback function is called if the Rx branch of the Interaction layer is stopped. The callback function itself (ApplIlRxStop()) must be provided by the application.
Tx Stop	Boolean	false	This callback function is called if the Tx branch of the Interaction layer is stopped. The callback function itself (ApplIlTxStop()) must be provided by the application.
ModuleInstance > Il_Vector > Debug Support			
Argument Check	Boolean	false	Used to enable extended checks of the arguments passed to functions of the Interaction Layer. If an error was detected, the return value of the functions will contain an error code. This option should be used for debugging purposes only and not in production code.
Assertion Handling	Boolean	false	The SW component provides built-in debug support (assertion) to ease up the integration and test into the user's project. Please see the technical reference for detailed information on the available options and how to use them. In general, the usage of assertions is recommended during the integration and pre-test phases. It is not recommended to enable the assertions in production code due to increased runtime and ROM needs. The assertion checks the correctness of the assigned condition and calls an error handler in case this fails. The error handler is called with an error number. Information about the defined error numbers is given in the technical reference.
ModuleInstance > Il_Vector			
AUTOSAR Signal API	Boolean	false	Configure the Vector Interaction Layer to support the signal access as defined in the AUTOSAR Specification of Module COM Version 1.0.0. Efficient macros are additionally generated if Put/Get signal access is configured. A signal is written using: Com_ReturnType Com_SendSignal(<SigName>, &<data>);

Attribute Name	Value Type	Default Value	Description
			A signal is read using Com_ReturnType Com_ReceiveSignal(<SigName>, &<data>); Please see the documentation chapter 'AUTOSAR Signal Interface'.
AUTOSAR Specification Version	Enum	N.a.	Select the AUTOSAR COM interface that shall be provided by the Il_Vector.
Detect Active Repetitions API	Boolean	false	If this option is enabled an API is activated, to detect, if messages are transmitted with repetitions and the repetitions are active. API: 1 Channel :Il_Boolean IlTxRepetitionsAreActive() 2..N Channels :Il_Boolean IlTxRepetitionsAreActive(<channel>)
Detect Active Signals API	Boolean	false	If this option is enabled an API is activated, to detect, if signals are active and transmitted with the fast cycle time. API: 1 Channel :Il_Boolean IlTxSignalsAreActive() 2..N Channels :Il_Boolean IlTxSignalsAreActive(<channel>)
Reset Rx Timeout Flags On IlRxRelease	Boolean	false	If this option is enabled and the transition IlRxRelease is performed all rx timeout flags on the channel are cleared.
Enable UpdateBit Support	Boolean	false	This option enables UpdateBit Support
Channel > Il_Vector > Task Call Cycle Time			
IlRxTask [ms]	Integer	10	This is the time base of the receive branch of the Interaction Layer. Make sure that the value you enter here in the Generation Tool is the same as the cycle time for calling the function 'IlRxTask'. If it is not, the timing of the receive branch in the IL will not work properly (e.g. for timeout monitoring). All timings of the Rx branch (e.g. timeouts) must be a multiple of this cycle time. If you enter e.g. 10 [ms] in the Generation Tool the function 'IlRxTask' must be called every 10ms.
IlTxTask [ms]	Integer	10	This is the time base of the transmit branch of the Interaction Layer.

Attribute Name	Value Type	Default Value	Description
			Make sure that the value you enter here in the Generation Tool is the same as the cycle time for calling the function 'IlTxTask'. If it is not, the timing of the transmit branch in the IL will not work properly (e.g. wrong timing of cylic send messages). All timings of the Tx branch (e.g. cycle times of transmit messages) must be a multiple of this cycle time. If you enter e.g. 10 [ms] in the Generation Tool the function 'IlTxTask' must be called every 10ms.
Message > Il_Vector > Database Attributes			
CycleTime [ms]	Integer	0	The CANdb attribute 'GenMsgCycleTime' is displayed as defined in the dbc file. This value is only relevant for messages with a cyclic transmission mode. For other messages '0' is displayed.
CycleTimeFast [ms]	Integer	0	The CANdb attribute 'GenMsgCycleTimeFast' is displayed as defined in the dbc file. This value is only relevant for messages with a transmission mode including a fast cyclic rate. For other messages '0' is displayed.
DelayTime [ms]	Integer	0	The CANdb attribute 'GenMsgDelayTime' is displayed as defined in the dbc file. This attribute defines the minimum transmit delay between two subsequent messages of the same ID.
FastOnStart [ms]	Integer	0	The CANdb attribute 'GenMsgFastOnStart' is displayed as defined in the dbc file. 'GenMsgFastOnStart' is the duration in ms of a startup phase in which the message is transmitted with a higher rate (GenMsgCycleTimeFast).
NrOfRepetition	Integer	0	The CANdb attribute 'GenMsgNrOfRepetition' is displayed as defined in the dbc file. This attribute defines the number of repetitions of a message caused by signal updates of signals which have transmission modes with repetition.
TxMessage > Il_Vector			
Polling	Boolean	true	The Interaction Layer confirmation of the sent message is handled in the CAN driver confirmation context. This context depends on the CAN driver Tx polling configuration. For details please see the CAN driver documentation. CAN driver Tx polling is activated ========================== If IL polling is activated, the message confirmation is handled via the CAN driver confirmation flag that is polled in the 'IlTxTask'. The Interaction Layer notification is separated from the initial event. If IL polling is deactivated, the message confirmation is handled via the CAN driver confirmation function that is called directly by the CAN driver in its own polling context.

Attribute Name	Value Type	Default Value	Description
			CAN driver Tx polling is deactivated ============================== If IL polling is activated, the message confirmation is handled via the CAN driver confirmation flag that is polled in the 'IlTxTask'. This is to minimize the interrupt load. If IL polling is deactivated, the message confirmation is handled via the CAN driver confirmation function that is called directly by the CAN driver in the interrupt context. Disabling the polling results in longer ISR run times. PLEASE NOTE: The implemented application code in this context must be programmed interrupt context secure. It's recommended to use the default.
Send on Init	Boolean	false	If you enable the attribute 'Send on Init' the selected Tx message is added to the set of messages which will be transmitted when 'IlSendOnInitMsg()' is called. If the DBC attribute "NetworkInitialization" is available at a message, the value of "Send on Init" is derived of this dbc attribute.
TxMessage > Il_Vector > Database Attributes			
StartDelayTime [ms]	Integer	0	The CANdb attribute 'GenMsgStartDelayTime' is displayed as defined in the dbc file. This attribute defines the time between 'IlTxStart()' and the begin of the cyclic transmission of a message.
RxMessage > Il_Vector			
Polling	Boolean	false	The Interaction Layer indication of the receive message is handled in the CAN driver indication context. This context depends on the CAN driver Rx polling configuration. For details please see the CAN driver documentation. CAN driver Rx polling is activated =========================== If IL polling is activated, the message indication is handled via the CAN driver indication flag that is polled in the 'IlRxTask'. The Interaction Layer notification is separated from the initial event. If IL polling is deactivated, the message indication is

Attribute Name	Value Type	Default Value	Description
			handled via the CAN driver indication function that is called directly by the CAN driver in its own polling context. CAN driver Rx polling is deactivated ============================= If IL polling is activated, the message indication is handled via the CAN driver indication flag that is polled in the 'IlRxTask'. This is to minimize the interrupt load. If IL polling is deactivated, the message indication is handled via the CAN driver indication function that is called directly by the CAN driver in the interrupt context. PLEASE NOTE: The implemented application code in this context must be programmed interrupt context secure. It's recommended to use the default.
RxMessage > Il_Vector > Notification Classes			
Timeout Function	String		If a valid function name is defined in the IL 'Timeout Function' field, this function is called by the Interaction Layer when a timeout of this receive message occurs. The timeout time for this message is defined in the data base file (dbc). API: void Appl<MsgName>MsgTimeout(void)
MsgSignal > Il_Vector > Database Attributes			
InactiveValue	String	0	The CANdb attribute 'GenSigInactiveValue' is displayed as defined in the dbc file. This attribute is relevant for signals with the transmission mode 'IfActive'.
RxSignal > Il_Vector > Signal Access			
Put	Boolean	false	The generation of signal value write access macros and -functions for receive signals can be enabled or disabled. If enabled, the Generation Tool will generate macros and functions for signal access using the signal names in the network data base (macros or functions, depending on the type of the signal). API: Length <=1Byte void IlPutRx<SigName>(vuint8 data) 1Byte< Length <=2Byte void IlPutRx<SigName>(vuint16 data) 2Byte< Length <=4Byte void IlPutRx<SigName>(vuint32

Attribute Name	Value Type	Default Value	Description
			data) 4Byte< Length void IlPutRx<SigName>(vuint8* pData) If the AUTOSAR signal API is enabled, set the signal value via Com_ReturnType Com_SendSignal(<SigName>, &<data>); Please see the documentation of Il_Vector, chapter 'AUTOSAR Signal Interface'.
Get	Boolean	true	The generation of signal value read access macros and -functions for receive signals can be enabled or disabled. If enabled, the Generation Tool will generate macros and functions for signal access using the signal names in the network data base (macros or functions, depending on the type of the signal). API: Length <=1Byte vuint8 IlGetRx<SigName>(void) 1Byte< Length <=2Byte vuint16 IlGetRx<SigName>(void) 2Byte< Length <=4Byte vuint32 IlGetRx<SigName>(void) 4Byte< Length void IlGetRx<SigName>(vuint8* pData) If the AUTOSAR signal API is enabled, read the signal value via Com_ReturnType Com_ReceiveSignal(<SigName>, &<data>) Please see the documentation of Il_Vector, chapter 'AUTOSAR Signal Interface'.
RDS	Boolean	false	Enable macros to read from the Rx register of the CAN controller. This switch will be used for example by the 'DataChanged' flag. CAUTION: Use these macros in a PreCopy function only! API: Length <=1Byte vuint8 IlGetRxCAN<SigName>(void)

Attribute Name	Value Type	Default Value	Description
			1Byte< Length <=2Byte vuint16 IlGetRxCAN<SigName>(void) 2Byte< Length <=4Byte vuint32 IlGetRxCAN<SigName>(void) 4Byte< Length void IlGetRxCAN<SigName>(vuint8* pData)
RxSignal > Il_Vector > Notification Classes > Indication			
Flag	Pool	N.a.	If this signal is received, one or more flags with the same names plus pre- and postfix from the name decorator configuration view will be set. CAUTION: The flag(s) must be reset by the application. It is recommended to use the macros for flag manipulation. API: vuint8 IlGetRx<SigName>SigIndication(void) void IlSetRx<SigName>SigIndication(void) void IlClrRx<SigName>SigIndication(void) vuint8 IlGetClrRx<SigName>SigIndication(void)
Function	Pool	N.a.	If this signal is received, one or more functions with these names plus pre- and postfix from the name decorator configuration view will be called. CAUTION: - The function(s) may run in interrupt context (if polling is not activated), so keep action short within. - If you use more than one indication function they are called in the order they are displayed in GENy. API: void ApplIl<SigName>SigIndication(void)
RxSignal > Il_Vector > Notification Classes > Firstvalue			
Flag	Pool	N.a.	If this signal is received for the first time after the system startup the 'FirstValue' flag will be set. This flag can not and should not be reset. The only reset is a restart of the Interaction Layer. API: vuint8 IlGetRx<SigName>SigFirstvalue(void) void IlSetRx<SigName>SigFirstvalue(void) void IlClrRx<SigName>SigFirstvalue(void)

Attribute Name	Value Type	Default Value	Description
RxSignal > Il_Vector > Notification Classes > DataChanged			
Flag	Pool	N.a.	If this signal is received and the new signal value differs from the current signal value, the 'DataChanged' flag(s) are set. The name of the flags are built-up using this name plus pre- and postfixes from the name decorator configuration view. CAUTION: The flag(s) must be reset by the application. It is recommended to use the macros for flag manipulation. API: vuint8 IlGetRx<SigName>SigDataChanged(void) void IlSetRx<SigName>SigDataChanged(void) void IlClrRx<SigName>SigDataChanged(void)
RxSignal > Il_Vector > Notification Classes > Timeout			
Flag	Pool	N.a.	The interaction layer is able to supervise receive signals. If the message containing the signal is not received within a (in data base file dbc) predefined time one ore more timeout flags are set. The name of the flags are built-up using this name plus pre- and postfixes from the name decorator configuration view. CAUTION: The flag(s) must be reset by the application. It is recommended to use the macros for flag manipulation. API: vuint8 IlGetRx<SigName>SigTimeout(void) void IlSetRx<SigName>SigTimeout(void) void IlClrRx<SigName>SigTimeout(void)
Function	Pool	N.a.	The interaction layer is able to supervise receive signals. If the message containing the signal is not received within a (in data base file dbc) predefined time one ore more timeout functions are called. The name of the functions are built-up using these names plus pre- and postfixes from the name decorator configuration view. CAUTION: - The function(s) may run in interrupt context (if polling is not activated), so keep action short within. - If you use more than one indication function they are called in the order they are displayed in GENy.

Attribute Name	Value Type	Default Value	Description
			API: void ApplIl<SigName>SigTimeout(void)
RxSignal > Il_Vector > Notification Classes > State Machine Transition			
Init	String		If you enter a function name you determine for this receive signal that an init function shall be called after the Interaction Layer was initialized. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>RxInit(void)
Start	String		If you enter a function name you determine for this receive signal that this start function shall be called after the Interaction Layer was switched to the running state. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>RxStart(void)
Stop	String		If you enter a function name you determine for this receive signal that this stop function shall be called after the Interaction Layer was suspended. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>RxStop(void)
RxSignal > Il_Vector > Default Value			
Init	Boolean	false	Use the default values below at initialization time to initialize the signal buffer. If you enable this feature, the fields - Start default - Stop default - Default value will be activated and can be configured, too.
Start	Boolean	false	Use the default value to initialize the signal buffer within the callcontext of 'IlRxStart()'. To use this option the default of 'Init' must be used, too.
Stop	Boolean	false	Use the default value, which is set in the callcontext 'IlRxStop()' to initialize the signal buffer. To use this option the default at 'IlInit()' must be used, too.
Value	String	0	Use the default value within 'IlInit()', 'IlRxStart()' and

Attribute Name	Value Type	Default Value	Description
			'IlRxStop()' to initialize the signal buffer. The data type of the default value depends on the data type of the related signal.
RxSignal > Il_Vector > Default Value > Timeout			
Enable	Boolean	false	Use the default value to replace the signal value in case of a timeout.
Value	String	0x0	Default value to replace the signal value in case of timeout. The data type of the default value depends on the data type of the related signal. The attribute in the data base file must be set to be able to set the wanted default value.
RxSignal > Il_Vector > API			
Dynamic Timeout	Boolean	false	The timeout of the receive signals can be set dynamically. If enabled an additional API to access the timer is provided. This functionality is required for special use cases and should normally be disabled. API: * IlGetRx<SigName>DynRxTimeout (void) void IlSetRx<SigName>DynRxTimeout(*) void IlStartRx<SigName>DynRxTimeout(void) void IlStopRx<SigName>DynRxTimeout(void) * This type depends on the configuration.
TxSignal > Il_Vector > Signal Access			
Put	Boolean	true	The generation of signal value write access macros and functions for send signals can be enabled or disabled. If enabled, the Generation Tool will generate macros and functions for signal access using the signal names in the network data base (macros or functions, depending on the type of the signal). API: Length <=1Byte void IlPutTx<SigName>(vuint8 data) 1Byte< Length <=2Byte void IlPutTx<SigName>(vuint16 data) 2Byte< Length <=4Byte void IlPutTx<SigName>(vuint32 data) 4Byte< Length void IlPutTx<SigName>(vuint8* pData) If the AUTOSAR signal API is enabled, set the signal value via Com_ReturnType Com_SendSignal(<SigName>, &<data>);

Attribute Name	Value Type	Default Value	Description
			For details please see the documentation of Il_Vector, chapter 'AUTOSAR Signal Interface'.
Get	Boolean	false	The generation of signal value read access macros and functions for send signals can be enabled or disabled. If enabled, the Generation Tool will generate macros and functions for signal access using the signal names in the network data base (macros or functions, depending on the type of the signal). API: Length <=1Byte vuint8 IlGetTx<SigName>(void) 1Byte< Length <=2Byte vuint16 IlGetTx<SigName>(void) 2Byte< Length <=4Byte vuint32 IlGetTx<SigName>(void) 4Byte< Length void IlGetTx<SigName>(vuint8* pData) If the AUTOSAR signal API is enabled, read the signal value via Com_ReturnType Com_ReceiveSignal(<SigName>, &<data>) For details please see the documentation of Il_Vector, chapter 'AUTOSAR Signal Interface'.
RDS	Boolean	false	Macros to write and read the Tx register of the CAN controller. These macros can only be used in the context of the PreTransmit function. Set API: Length <=1Byte void IlPutTxCAN<SigName>(vuint8 data) 1Byte< Length <=2Byte void IlPutTxCAN<SigName>(vuint16 data) 2Byte< Length <=4Byte void IlPutTxCAN<SigName>(vuint32 data) 4Byte< Length void IlPutTxCAN<SigName>(vuint8* pData)
TxSignal > Il_Vector > Notification Classes > Confirmation			
Flag	Boolean	false	After successful transmission of a signal, the Interaction Layer sets the confirmation flag. API: vuint8 IlGetIlTx<SigName>SigConfirmation()

Attribute Name	Value Type	Default Value	Description
			void IlSetIlTx<SigName>SigConfirmation() void IlClrIlTx<SigName>SigConfirmation()
Function	String		If you enter a function name you determine for this transmit signal that a confirmation function shall be called after transmission of this signal. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>SigConfirmation(void)
TxSignal > Il_Vector > Notification Classes > Timeout			
Flag	Boolean	false	To supervise the actual transmission of signals, a timeout flag can be configured. If a signal is sent and the confirmation is not received until timeout, the application will be notified by this timeout flag. API: vuint8 IlGetIlTx<SigName>SigTimeout() void IlSetIlTx<SigName>SigTimeout() void IlClrIlTx<SigName>SigTimeout()
Function	String		If you enter a function name you determine for this transmit signal that a timeout function shall be called if the confirmation will not occur in time. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>SigTimeout(void)
TxSignal > Il_Vector > Notification Classes > State Machine Transition			
Init	String		If you enter a function name you determine for this transmit signal that an init function shall be called after the Interaction Layer was initialized. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>TxInit(void)
Start	String		If you enter a function name you determine for this transmit signal that a start function shall be called after the Interaction Layer was switched to the running state. The default is the name of the signal, but you can change this name. The final name of the function will be determined

Attribute Name	Value Type	Default Value	Description
			out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>TxStart(void)
Stop	String		If you enter a function name you determine for this transmit signal that a stop function shall be called after the Interaction Layer was suspended. The default is the name of the signal, but you can change this name. The final name of the function will be determined out of the defined pre- and postfixes on the name decorator configuration view and the specified name. API: void ApplIl<SigName>TxStop(void)
TxSignal > Il_Vector > Default Value			
Init	Boolean	false	Use the default value at initialization time to initialize the signal buffer. If you activate the initialization default value handling, the following fields (Start, Stop and Value) will be enabled and can be configured, too.
Start	Boolean	false	Use the default value to initialize the signal buffer within the callcontext of 'IlTxStart()'. To use this option 'Init' in 'Default Value' must be activated, too.
Stop	Boolean	false	Use the default value with 'IlTxStop' to initialize the signal buffer. To use this option 'Init' in 'Default Value' must be activated, too.
Value	String	0	Use the default value within 'IlInit()', 'IlTxStart()' and 'IlTxStop()' to initialize the signal buffer. The data type of the default value depends on the data type of the related signal. To use this option 'Init' in 'Default Value' must be activated, too.
MsgSignalContainer > Il_Vector > Signal Access			
Shadow Buffer	Boolean	false	If this option is enabled, the appplication has to provide the shadow buffer for signal groups. Due to this setting, the signal group API changes. For detailed information please see the technical reference of IL_Vector.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > Confirmation			
Prefix	String	Appl	Specify the prefix for the signal confirmation function.
Postfix	String	SigConfirmation	Specify the postfix for the signal confirmation function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > Indication			
Prefix	String	Appl	Specify the prefix for the signal indication function.
Postfix	String	SigIndication	Specify the postfix for the signal indication function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > Tx Timeout			

Attribute Name	Value Type	Default Value	Description
Signal Prefix	String	Appl	Specify the prefix for the signal based timeout function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > Rx Timeout			
Signal Prefix	String	Appl	Specify the prefix for the signal based timeout function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > Tx Timeout			
Signal Postfix	String	TxSigTimeout	Specify the postfix for the signal based timeout function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > Rx Timeout			
Signal Postfix	String	RxSigTimeout	Specify the postfix for the signal based timeout function.
Message Prefix	String	Appl	Specify the prefix for the message based timeout function.
Message Postfix	String	MsgTimeout	Specify the postfix for the message based timeout function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > State Machine Transition > Rx Init			
Prefix	String	Appl	Specify the prefix for the Rx signal init callback function.
Postfix	String	RxInit	Specify the postfix for the Rx signal init callback function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > State Machine Transition > Tx Init			
Prefix	String	Appl	Specify the prefix for the Tx signal init callback function.
Postfix	String	TxInit	Specify the postfix for the Tx signal init callback function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > State Machine Transition > Rx Start			
Prefix	String	Appl	Specify the prefix for the Rx signal start callback function.
Postfix	String	RxStart	Specify the postfix for the Rx signal start callback function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > State Machine Transition > Tx Start			
Prefix	String	Appl	Specify the prefix for the Tx signal start callback function.
Postfix	String	TxStart	Specify the postfix for the Tx signal start callback function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > State Machine Transition > Rx Stop			
Prefix	String	Appl	Specify the prefix for the Rx signal stop callback function.
Postfix	String	RxStop	Specify the postfix for the Rx signal stop callback function.
ModuleInstance > Il_Vector > Notification Mechanism > Functions > State Machine Transition > Tx Stop			
Prefix	String	Appl	Specify the prefix for the Tx signal stop callback function.
Postfix	String	TxStop	Specify the postfix for the Tx signal stop callback function.
ModuleInstance > Il_Vector > Signal Access > Rx			
Get Prefix	String	IlGetRx	Specify the prefix for the Rx signal read access functions and macros.
Put Prefix	String	IlPutRx	Specify the prefix for the Rx signal write access functions and macros.
ModuleInstance > Il_Vector > Signal Access > Tx			
Get Prefix	String	IlGetTx	Specify the prefix for the Tx signal read access functions and macros.

Attribute Name	Value Type	Default Value	Description
Put Prefix	String	IlPutTx	Specify the prefix for the Tx signal write access functions and macros.
ModuleInstance > Il_Vector > RDS Signal Access			
Rx Get Prefix	String	IlGetRxCAN	Specify the prefix for the Rx signal read access functions and macros.
Tx Put Prefix	String	IlPutTxCAN	Specify the prefix for the Tx signal write access functions and macros.
ModuleInstance > Il_Vector > Signal Access > Rx			
SignalGroup Get Prefix	String	IlGetRx	Specify the prefix for the Rx signal group read access functions and macros.
SignalGroup Put Prefix	String	IlPutRx	Specify the prefix for the Rx signal group write access functions and macros.
ModuleInstance > Il_Vector > Signal Access > Tx			
SignalGroup Get Prefix	String	IlGetTx	Specify the prefix for the Tx signal group read access functions and macros.
SignalGroup Put Prefix	String	IlPutTx	Specify the prefix for the Tx signal group write access functions and macros.
ModuleInstance > Il_Vector > Dynamic Rx Timeout API			
Get Prefix	String	IlGetRx	Specify a prefix for the user defined IlGetRxDynamicTimeout function.
Set Prefix	String	IlSetRx	Specify a prefix for the user defined IlSetRxDynamicTimeout function.
Start Prefix	String	IlStartRx	Specify a prefix for the user defined IlStartRxDynamicTimeout function.
Stop Prefix	String	IlStopRx	Specify a prefix for the user defined IlStopRxDynamicTimeout function.
Postfix	String	DynRxTimeout	Specify a postfix for the generated DynamicApi macros/functions.
ModuleInstance > Il_Vector > Notification Mechanism > Flags > Prefixes			
Get	String	IlGet	Specify the prefix for macros to read flags.
Set	String	IlSet	Specify the prefix for macros to write flags.
Clear	String	IlClr	Specify the prefix for macros to clear flags.
Get + Clear	String	IlGetClr	Specify the prefix for macros to read and clear flags.
ModuleInstance > Il_Vector > Notification Mechanism > Flags > Postfixes			
Indication	String	Indication	Specify the postfix for indication flag macros.
Confirmation	String	Confirmation	Specify the postfix for confirmation flag macros.
FirstValue	String	Firstvalue	Specify the postfix for FirstValue flag macros.
Tx Timeout	String	TxTimeo	Specify the postfix for Tx timeout flag macros.

Attribute Name	Value Type	Default Value	Description
		ut	
Rx Timeout	String	RxTimeout	Specify the postfix for Rx timeout flag macros.
DataChanged	String	DataChanged	Specify the postfix for DataChanged flag macros.
MsgSignal > Il_Vector > Database Attributes			
MsgSendType	Object	N.a.	CANdb attribute 'GenMsgSendType'
SigSendType	Object	N.a.	CANdb attribute 'GenSigSendType'
Message > Il_Vector > Database Attributes			
SendType	Object	N.a.	CANdb attribute 'GenMsgSendType'

Table 5-1 GENy attributes

6.1.2 Services provided by Interaction Layer

	Caution The APIs of the IL do not support reentrant calls and must not interrupt each other. Exceptions are described in the API description. See also chapter 4.3 Operating Systems Requirements.

6.1.2.1 IlInitPowerOn

IlInitPowerOn

Prototype	
`void` **`IlInitPowerOn`** `(void)`	
Parameter	
void	none
Return code	

Return code	
void	none
Functional Description	
This method initializes the Il_Vector on a channel. Rx and Tx data buffers and flags are set to the initial state. If no default value for a message is defined, the data buffer is set to 0x00.	
Particularities and Limitations	
The function is called by the Application, Ccl (Communication Control Layer) or IlInitPowerOn.	
Call context	
The function must be called with disabled interrupts. The function must not interrupt IlRxTask, IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlRxStart, IlTxStart, IlRxStop, IlTxStop.	

6.1.2.3 IlRxStart

IlRxStart

Prototype	
Single Channel	
Single Receive Channel	`void IlRxStart (void)`
Multi Channel	
Indexed (MRC)	`void IlRxStart (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method enables the reception of messages. The transition "start" of the Rx state machine is performed.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function must be called on task level. The function must not interrupt IlRxTask, IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlTxStart, IlRxStop, IlTxStop.	

6.1.2.4 IlTxStart

IlTxStart

Prototype
Single Channel

Single Receive Channel	`void IlTxStart (void)`
Multi Channel	
Indexed (MRC)	`void IlTxStart (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method enables the transmission of messages and starts the transmission of periodic messages. The transition "start" of the Tx state machine is performed.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function must be called on task level. The function must not interrupt IlRxTask, IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlTxStart, IlRxStop, IlTxStop.	

6.1.2.5 IlRxStop

IlRxStop

Prototype	
Single Channel	
Single Receive Channel	`void IlRxStop (void)`
Multi Channel	
Indexed (MRC)	`void IlRxStop (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method disables the reception of messages. The transition "stop" of the Rx state machine is performed. The method is used for example to enter the Sleep Mode of an ECU.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function must be called on task level. The function must not interrupt IlRxTask, IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlTxStart, IlRxStop, IlTxStop.	

6.1.2.6 IlTxStop

IlTxStop

Prototype	
Single Channel	
Single Receive Channel	`void IlTxStop (void)`
Multi Channel	
Indexed (MRC)	`void IlTxStop (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method disables the transmission of messages (Sleep Mode). The transition "stop" of the Tx state machine is performed. The method is used for example to enter the Sleep Mode of an ECU.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function must be called on task level. The function must not interrupt IlInitPowerOn, IlInit, IlRxTask, IlRxStateTask, IlRxTimerTask, IlTxTask, IlTxStateTask, IlTxTimerTask, IlRxStart, IlTxStart, IlRxStop	

6.1.2.7 IlRxWait

IlRxWait

Prototype	
Single Channel	
Single Receive Channel	`void IlRxWait (void)`
Multi Channel	
Indexed (MRC)	`void IlRxWait (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method halts the timeout monitoring of reception messages. The transition "wait" of the Rx state machine is performed. The method is used for example when the bus-off mode of an ECU was entered.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	

The function can be called on task and interrupt level.

6.1.2.8 IlTxWait

IlTxWait

Prototype	
Single Channel	
Single Receive Channel	`void` **`IlTxWait`** `(void)`
Multi Channel	
Indexed (MRC)	`void` **`IlTxWait`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method halts the transmission of messages. The transition "wait" of the Tx state machine is performed. The method is used for example when the bus-off mode of an ECU was entered.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function can be called on task and interrupt level.	

6.1.2.9 IlRxRelease

IlRxRelease

Prototype	
Single Channel	
Single Receive Channel	`void` **`IlRxRelease`** `(void)`
Multi Channel	
Indexed (MRC)	`void` **`IlRxRelease`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
The transition "release" of the Rx state machine is performed. -Restart the Rx Timeout Monitoring -Clear the timeout flags if IL_ENABLE_SYS_RX_RESET_TIMEOUT_FLAGS_ON_ILRXRELEASE is defined.	

Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function can be called on task and interrupt level.	

6.1.2.10 IlTxRelease

IlTxRelease

Prototype	
Single Channel	
Single Receive Channel	`void` **`IlTxRelease`** `(void)`
Multi Channel	
Indexed (MRC)	`void` **`IlTxRelease`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method resumes the transmission of messages from the "Waiting" state. The transition "release" of the Tx state machine is performed.	
Particularities and Limitations	
The function is called by the Application or Nm (Network Management).	
Call context	
The function can be called on task and interrupt level.	

6.1.2.11 IlRxTask

IlRxTask

Prototype	
Single Channel	
Single Receive Channel	`void` **`IlRxTask`** `(void)`
Multi Channel	
Indexed (MRC)	`void` **`IlRxTask`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none

Functional Description
This method must be called periodically in the Rx task cycle time configured in the generation tool. The IlRxTimerTask and IlRxStateTask are called by this method.
Particularities and Limitations
The function is called by the Application or Ccl (Communication Control Layer).
Call context
The function must be called on task level. The function must not interrupt IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlRxStart, IlTxStart, IlRxStop, IlTxStop

6.1.2.12 IlTxTask

IlTxTask

Prototype	
Single Channel	
Single Receive Channel	`void IlTxTask (void)`
Multi Channel	
Indexed (MRC)	`void IlTxTask (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method must be called periodically in the Tx task cycle time configured in the generation tool. The IlTxTimerTask and IlTxStateTask are called by this method.	
Particularities and Limitations	
The function is called by the Application or Ccl (Communication Control Layer).	
Call context	
The function must be called on task level. The function must not interrupt IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlRxStart, IlTxStart, IlRxStop, IlTxStop	

6.1.2.13 IlRxStateTask

IlRxStateTask

Prototype	
Single Channel	
Single Receive Channel	`void IlRxStateTask (void)`
Multi Channel	
Indexed (MRC)	`void IlRxStateTask (CanChannelHandle channel)`

Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called periodically by the IlRxTask. The function can be called in a faster rate than the IlRxTask to check additionally for polled indication events. The usage of the IlRxTask shall be preferred.	
Particularities and Limitations	
The function is called by the Application or IlRxTask.	
Call context	
The function must be called on task level. The function must not interrupt IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlRxStart, IlTxStart, IlRxStop, IlTxStop	

6.1.2.14 IlTxStateTask

IlTxStateTask

Prototype	
Single Channel	
Single Receive Channel	`void IlTxStateTask (void)`
Multi Channel	
Indexed (MRC)	`void IlTxStateTask (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called periodically by the IlTxTask. The function can be called in a faster rate, than the IlTxTask, to check additionally for polled confirmation events. The usage of the IlTxTask shall be preferred.	
Particularities and Limitations	
The function is called by the Application or IlTxTask.	
Call context	
The function must be called on task level. The function must not interrupt IlRxStateTask, IlTxTask, IlTxStateTask, IlInitPowerOn, IlInit, IlRxStart, IlTxStart, IlRxStop, IlTxStop	

6.1.2.15 IlSendOnInitMsg

IlSendOnInitMsg

Prototype	
Single Channel	
Single Receive Channel	`void IlSendOnInitMsg (void)`
Multi Channel	
Indexed (MRC)	`void IlSendOnInitMsg (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method serves to set a transmission request flag for all messages configured as SendOnInit messages.	
Particularities and Limitations	
The function is called by the Application.	
Call context	
The function must be called on task level.	

6.1.2.16 IlGetStatus

IlGetStatus

Prototype	
Single Channel	
Single Receive Channel	`Il_Status IlGetStatus (void)`
Multi Channel	
Indexed (MRC)	`Il_Status IlGetStatus (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
Il_Status	Il_Status : the value must be decoded with the following set of macros. The macros will return 0 (false) or 1 (true). - IlIsTxRun(state) : Tx is running - IlIsTxWait(state) : Tx is waiting - IlIsTxSuspend(state) : Tx is suspended - IlIsRxRun(state) : Rx is running - IlIsRxWait(state) : Rx is waiting - IlIsRxSuspend(state) : Rx is suspended

Functional Description	
Gets the current state of the Interaction Layer state machine.	
Particularities and Limitations	
The function is called by the Application.	
Call context	
The function can be called on task and interrupt level.	

6.1.2.17 IlTxRepetitionsAreActive

IlTxRepetitionsAreActive

Prototype	
Single Channel	
Single Receive Channel	`Il_Boolean` **`IlTxRepetitionsAreActive`** `(void)`
Multi Channel	
Indexed (MRC)	`Il_Boolean` **`IlTxRepetitionsAreActive`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
Il_Boolean	IL_TRUE : Messages with repetitions are queued for transmission. IL_FALSE : No message with repetitions is queued for transmission.
Functional Description	
This method can be used to detect if messages with repetitions are queued for transmission on a channel.	
Particularities and Limitations	
The function is called by the Application.	
⚠	**Caution** The function does not support Virtual Networks.
Call context	
The function can be called on task and interrupt level.	

6.1.2.18 IlTxSignalsAreActive

IlTxSignalsAreActive

Prototype	
Single Channel	
Single Receive Channel	`Il_Boolean` **`IlTxSignalsAreActive`** `(void)`
Multi Channel	
Indexed (MRC)	`Il_Boolean` **`IlTxSignalsAreActive`** `(CanChannelHandle channel)`

Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
Il_Boolean	IL_TRUE : Signals are in the active state. IL_FALSE : No signal is in the active state.
Functional Description	
This method can be used to detect if signals are active on a channel.	
Particularities and Limitations	
The function is called by the Application.	
⚠	**Caution** The function does not support Virtual Networks.
Call context	
The function can be called on task and interrupt level.	

6.1.3 Generated Services provided by the Interaction Layer

ℹ	**Info** The generated service declarators in this chapter are depending on the configuration.

6.1.3.1 Read and Write Signals and Signal Groups

WriteSignalByValue

Prototype	
`void WriteSignalByValue (vuintx sigData)`	
Parameter	
sigData	new value of the signal. (vuint8) : The length of the network signal is between 1 and 8 bits. (vuint16) : The length of the network signal is between 9 and 16 bits. (vuint32) : The length of the network signal is between 17 and 32 bits.
Return code	
void	none
Functional Description	
Write a signal value to the message buffer and evaluate the transmission mode for Tx signals. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlPutTx><NetworkSignalName>.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function can be called on task and interrupt level.	

WriteSignalByReference

Prototype	
`void WriteSignalByReference (vuint8 *pData)`	
Parameter	
pData	pointer to a vuint8 array with the new value of the signal. The length of the network signal is between 33 and 64 bits.
Return code	
void	none

Functional Description
Write a signal value to the message buffer and evaluate transmission mode for Tx signals. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlPutTx><NetworkSignalName>.
Particularities and Limitations
The function is called by the application.
Call context
The function can be called on task and interrupt level.

ReadSignalByValue

Prototype	
`vuintx` **`ReadSignalByValue`** `(void)`	
Parameter	
void	none
Return code	
vuintx	(vuint8) : The length of the network signal is between 1 and 8 bits. (vuint16) : The length of the network signal is between 9 and 16 bits. (vuint32) : The length of the network signal is between 17 and 32 bits.
Functional Description	
Read a signal value from the message buffer. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlGetRx><NetworkSignalName>.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function can be called on task and interrupt level.	

ReadSignalByReference

Prototype
`void` **`ReadSignalByReference`** `(vuint8 *pData)`

Parameter	
pData	pointer to a vuint8 array where the value of the signal shall be stored. The length of the network signal is between 33 and 64 bits.
Return code	
void	none
Functional Description	
Read a signal value from the message buffer. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlGetRx><NetworkSignalName>.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function can be called on task and interrupt level.	

WriteGroupedSignalByValue

Prototype	
`void` **`WriteGroupedSignalByValue`** `(SignalGroupBufferType pBuffer, vuintx sigData)`	
Parameter	
pBuffer	pointer to the signal group buffer. (SignalGroupBufferType) : the generated data type for each signal group. ->"Shadow Buffer" is enabled in the configuration for the signal group: The application has to provide a shadow buffer with the generated type. The generated data type name can be identified by the prefix "_c_" and the network signal group name and the postfix "_buf". ->"Shadow Buffer" is disabled in the configuration for the signal group: The parameter is omitted and the IL provides the shadow buffer.
sigData	new value of the signal. (vuint8) : The length of the network signal is between 1 and 8 bits. (vuint16) : The length of the network signal is between 9 and 16 bits. (vuint32) : The length of the network signal is between 17 and 32 bits.
Return code	
void	none

Functional Description
Write a grouped signal value to the signal group buffer. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name and - "ShadowBuffer" e.g. <IlPutTx><NetworkSignalName>ShadowBuffer.
Particularities and Limitations
The function is called by the application.
Call context
The function can be called on task and interrupt level.

WriteGroupedSignalByReference

Prototype	
`void` **`WriteGroupedSignalByReference`** `(SignalGroupBufferType pBuffer, vuint8 *pData)`	
Parameter	
pBuffer	pointer to the signal group buffer. (SignalGroupBufferType) : the generated data type for each signal group. ->"Shadow Buffer" is enabled in the configuration for the signal group: The application has to provide a shadow buffer with the generated type. The generated data type name can be identified by the prefix "_c_" and the network signal group name and the postfix "_buf". ->"Shadow Buffer" is disabled in the configuration for the signal group: The parameter is omitted and the IL provides the shadow buffer.
pData	pointer to a vuint8 array with the new value of the signal. The length of the network signal is between 33 and 64 bits.
Return code	
void	none
Functional Description	
Write a grouped signal value to the signal group buffer. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name and - "ShadowBuffer" e.g. <IlPutTx><NetworkSignalName>ShadowBuffer.	
Particularities and Limitations	
The function is called by the application.	

Call context
The function can be called on task and interrupt level.

ReadGroupedSignalByValue

Prototype	
`vuintx` **`ReadGroupedSignalByValue`** `(SignalGroupBufferType pBuffer)`	
Parameter	
pBuffer	pointer to the signal group buffer. (SignalGroupBufferType) : the generated data type for each signal group. ->"Shadow Buffer" is enabled in the configuration for the signal group: The application has to provide a shadow buffer with the generated type. The generated data type name can be identified by the prefix "_c_" and the network signal group name and the postfix "_buf". ->"Shadow Buffer" is disabled in the configuration for the signal group: The parameter is omitted and the IL provides the shadow buffer.
Return code	
vuintx	(vuint8) : The length of the network signal is between 1 and 8 bits. (vuint16) : The length of the network signal is between 9 and 16 bits. (vuint32) : The length of the network signal is between 17 and 32 bits.
Functional Description	
Read a signal value from the message buffer. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name and - "ShadowBuffer" e.g. <IlGetRx><NetworkSignalName>ShadowBuffer.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function can be called on task and interrupt level.	

ReadGroupedSignalByReference

Prototype
`void` **`ReadGroupedSignalByReference`** `(SignalGroupBufferType pBuffer, vuint8 *pData)`

Parameter	
pBuffer	pointer to the signal group buffer. (SignalGroupBufferType) : the generated data type for each signal group. ->"Shadow Buffer" is enabled in the configuration for the signal group: The application has to provide a shadow buffer with the generated type. The generated data type name can be identified by the prefix "_c_" and the network signal group name and the postfix "_buf". ->"Shadow Buffer" is disabled in the configuration for the signal group: The parameter is omitted and the IL provides the shadow buffer.
pData	pointer to a vuint8 array where the value of the signal shall be stored. The length of the network signal is between 33 and 64 bits.
Return code	
void	none
Functional Description	
Read a signal value from the message buffer. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name and - "ShadowBuffer" e.g. <IlGetRx><NetworkSignalName>ShadowBuffer.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function can be called on task and interrupt level.	

WriteSignalGroup

Prototype	
void **WriteSignalGroup** (SignalGroupBufferType pBuffer)	
Parameter	
pBuffer	pointer to the signal group buffer. (SignalGroupBufferType) : the generated data type for each signal group. ->"Shadow Buffer" is enabled in the configuration for the signal group: The application has to provide a shadow buffer with the generated type. The generated data type name can be identified by the prefix "_c_" and the network signal group name and the postfix "_buf". ->"Shadow Buffer" is disabled in the configuration for the signal group: The parameter is omitted and the IL provides the shadow buffer.

Return code	
void	none
Functional Description	
Write a signal group from the signal group buffer to the message buffer and evaluate the transmission mode for Tx signals. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal group name and - "ShadowBuffer" e.g. <IlPutTx><NetworkSignalGroupName>ShadowBuffer.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function can be called on task and interrupt level.	

ReadSignalGroup

Prototype	
`void` **`ReadSignalGroup`** `(SignalGroupBufferType pBuffer)`	
Parameter	
pBuffer	pointer to the signal group buffer. (SignalGroupBufferType) : the generated data type for each signal group. ->"Shadow Buffer" is enabled in the configuration for the signal group: The application has to provide a shadow buffer with the generated type. The generated data type name can be identified by the prefix "_c_" and the network signal group name and the postfix "_buf". ->"Shadow Buffer" is disabled in the configuration for the signal group: The parameter is omitted and the IL provides the shadow buffer.
Return code	
void	none
Functional Description	
Read a signal group from the message buffer to the signal group buffer. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal group name and - "ShadowBuffer" e.g. <IlGetRx><NetworkSignalGroupName>ShadowBuffer.	
Particularities and Limitations	
The function is called by the application.	
Call context	

The function can be called on task and interrupt level.

6.1.3.2 Read and Write Signals and SignalGroups in the RDS Buffer.

WriteSignalByValue2RDS

Prototype	
`void` **`WriteSignalByValue2RDS`** `(vuintx sigData)`	
Parameter	
sigData	new value of the signal. (vuint8) : The length of the network signal is between 1 and 8 bits. (vuint16) : The length of the network signal is between 9 and 16 bits. (vuint32) : The length of the network signal is between 17 and 32 bits.
Return code	
void	none
Functional Description	
Write a signal or grouped signal value to the register of the CAN controller. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlPutTx><NetworkSignalName>.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function must be called in the context of the signals message specific CAN Driver PreTransmit function.	

WriteSignalByReference2RDS

Prototype	
`void` **`WriteSignalByReference2RDS`** `(vuint8 *pData)`	
Parameter	
pData	pointer to a vuint8 array with the new value of the signal. The length of the network signal is between 33 and 64 bits.
Return code	
void	none

Functional Description
Write a signal or grouped signal value to the register of the CAN controller. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlPutTx><NetworkSignalName>.
Particularities and Limitations
The function is called by the application.
Call context
The function must be called in the context of the signals message specific CAN Driver PreTransmit function.

ReadSignalByValueFromRDS

Prototype	
`vuintx` **`ReadSignalByValueFromRDS`** `(void)`	
Parameter	
void	none
Return code	
vuintx	(vuint8) : The length of the network signal is between 1 and 8 bits. (vuint16) : The length of the network signal is between 9 and 16 bits. (vuint32) : The length of the network signal is between 17 and 32 bits.
Functional Description	
Read a signal or grouped signal value from the register of the CAN controller. The function is generated optimized for the configuration and can be a function like macro or generated function. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlGetRx><NetworkSignalName>.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function must be called within the context of the signals message specific CAN Driver PreCopy function.	

ReadSignalByReferenceFromRDS

Prototype	
`void` **`ReadSignalByReferenceFromRDS`** `(vuint8 *pData)`	
Parameter	
pData	pointer to a vuint8 array where the value of the signal shall be stored. The length of the network signal is between 33 and 64 bits.
Return code	
void	none
Functional Description	
Read a signal or grouped signal value from the register of the CAN controller. The length of the network signal is between 33 and 64 bits. The generated prototype declarator is composed of - a configurable prefix for writing signals and - the network signal name e.g. <IlGetRx><NetworkSignalName>.	
Particularities and Limitations	
The function is called by the application.	
Call context	
The function must be called within the context of the signals message specific CAN Driver PreCopy function.	

6.1.3.3 Notification Flags of Signals, Signal Groups and Grouped Signals

GetNotificationFlag

Prototype	
`vuint8 GetNotificationFlag (void)`	
Parameter	
void	none
Return code	
vuint8	(vuint8) 0 : The notification is NOT set. > 0 : The notification is set.
Functional Description	
This macro is used to detect the notification of a signal, signal group or grouped signal. The generated prototype declarator is composed of - a configurable prefix for the notification flag to get flags and - the network signal, signal group or grouped signal name and - a configurable postfix for the notification flag. e.g. <IlGet><NetworkSignalName><Indication>.	
Particularities and Limitations	
The macro is called by the application.	
Call context	
The macro can be called on task and interrupt level.	

SetNotificationFlag

Prototype	
`void SetNotificationFlag (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This macro sets the notification flag of a signal, signal group or grouped signal. The generated prototype declarator is composed of - a configurable prefix for the notification flag to set flags and - the network signal, signal group or grouped signal name and - a configurable postfix for the notification flag. e.g. <IlSet><NetworkSignalName><Indication>.	

Particularities and Limitations
The macro is called by the application.
Call context
The macro must be called with disabled interrupts or in a non-preemptive task.

ClearNotificationFlag

Prototype	
`void ClearNotificationFlag (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This macro clears the notification flag of a signal, signal group or grouped signal. The generated prototype declarator is composed of - a configurable prefix for the notification flag to clear flags and - the network signal, signal group or grouped signal name and - a configurable postfix for the notification flag. e.g. <IlClr><NetworkSignalName><Indication>.	
Particularities and Limitations	
The macro is called by the application.	
Call context	
The macro must be called with disabled interrupts or in a non-preemptive task.	

GetAndClearNotificationFlag

Prototype	
`vuint8 GetAndClearNotificationFlag (void)`	
Parameter	
void	none
Return code	
vuint8	(vuint8) 0 : The notification is NOT set > 0 : The notification is set.

Functional Description
This macro is used to determine and clear the notification of a signal, signal group or grouped signal. The generated prototype declarator is composed of - a configurable prefix for the notification flag to get and clear flags - and the network signal, signal group or grouped signal name and - a configurable postfix for the notification flag. e.g. <IlGetClr><NetworkSignalName><Indication> The get and clear macro is only provided for the signal, signal group or grouped signal indication flag.
Particularities and Limitations
The macro is called by the application.
Call context
The macro can be called on task and interrupt level.

6.1.3.4 Dynamic Rx Timeout

GetDynamicRxTimeout

Prototype	
`IltRxTimeoutCounter` **`GetDynamicRxTimeout`** `(void)`	
Parameter	
void	none
Return code	
IltRxTimeoutCounter	(IltRxTimeoutCounter) The current Rx timeout counter in milliseconds with the maximum value 65535.
Functional Description	
This method is used to receive the timeout counter in milliseconds for a message. The generated prototype declarator is composed of - a configurable prefix for the reception of the dynamic timeout counter and - the network signal, signal group or grouped signal name and - a configurable postfix for the dynamic timeout. e.g. <IlGetRx><NetworkSignalName><DynRxTimeout>.	
Particularities and Limitations	
The macro is called by the application.	
⚠	**Caution** This API is signal oriented, but the effect will take place for all signals in the message.
Call context	
The macro can be called on task and interrupt level.	

SetDynamicRxTimeout

Prototype	
`void` **`SetDynamicRxTimeout`** `(IltRxTimeoutCounter msTimer)`	
Parameter	
msTimer	The new Rx timeout counter in milliseconds with the maximum value 65535.
Return code	
void	none

Functional Description
This method is used to set the timeout counter in milliseconds for a message. The generated prototype declarator is composed of - a configurable prefix for setting the dynamic timeout counter and - the network signal, signal group or grouped signal name and - a configurable postfix for setting the dynamic timeout. e.g. <IlSetRx><NetworkSignalName><DynRxTimeout>.
Particularities and Limitations
The macro is called by the application.
Caution This API is signal oriented, but the effect will take place for all signals in the message.
Call context
The macro must be called with disabled interrupts or in a non-preemptive task.

StartDynamicRxTimeout

Prototype	
`void` **`StartDynamicRxTimeout`** `(void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This method is used to start a stopped timeout counter for a message. The generated prototype declarator is composed of - a configurable prefix for starting the dynamic timeout counter and - the network signal, signal group or grouped signal name and - a configurable postfix for starting the dynamic timeout. e.g. <IlSetRx><NetworkSignalName><DynRxTimeout>.	
Particularities and Limitations	
The macro is called by the application.	
Caution This API is signal oriented, but the effect will take place for all signals in the message.	
Call context	
The macro can be called on task and interrupt level.	

Prototype	
`void` **`StopDynamicRxTimeout`** `(void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This method is used to stop the timeout counter for a message. The generated prototype declarator is composed of - a configurable prefix for stopping the dynamic timeout counter and - the network signal, signal group or grouped signal name and - a configurable postfix for stopping the dynamic timeout. e.g. <IlSetRx><NetworkSignalName><DynRxTimeout>.	
Particularities and Limitations	
The macro is called by the application.	
⚠	**Caution** This API is signal oriented, but the effect will take place for all signals in the message.
Call context	
The macro can be called on task and interrupt level.	

6.1.4 Callback Functions

All callback functions can be activated or deactivated by a switch in the Configuration Tool. If a callback function is activated by the user, the application has to provide this function.

6.1.4.1 ApplIlInit

ApplIlInit

Prototype	
Single Channel	
Single Receive Channel	`void` **`ApplIlInit`** `(void)`
Multi Channel	
Indexed (MRC)	`void` **`ApplIlInit`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called to indicate the performed initialization.	
Particularities and Limitations	
none	
Call context	
The function is called by the IL in the context of IlInit.	

6.1.4.2 ApplIlRxStart

ApplIlRxStart

Prototype	
Single Channel	
Single Receive Channel	`void` **`ApplIlRxStart`** `(void)`
Multi Channel	
Indexed (MRC)	`void` **`ApplIlRxStart`** `(CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called to indicate the performed transition start for the Rx state machine.	
Particularities and Limitations	
none	

Call context
The function is called by the IL in the context of IlRxStart.

6.1.4.3 ApplIlTxStart

ApplIlTxStart

Prototype	
Single Channel	
Single Receive Channel	`void ApplIlTxStart (void)`
Multi Channel	
Indexed (MRC)	`void ApplIlTxStart (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called to indicate the performed transition start for the Tx state machine.	
Particularities and Limitations	
none	
Call context	
The function is called by the IL in the context of IlTxStart.	

6.1.4.4 ApplIlRxStop

ApplIlRxStop

Prototype	
Single Channel	
Single Receive Channel	`void ApplIlRxStop (void)`
Multi Channel	
Indexed (MRC)	`void ApplIlRxStop (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called to indicate the performed transition stop for the Rx state machine.	
Particularities and Limitations	
none	

Call context
The function is called by the IL in the context of IlRxStop.

6.1.4.5 ApplIlTxStop

ApplIlTxStop

Prototype	
Single Channel	
Single Receive Channel	`void ApplIlTxStop (void)`
Multi Channel	
Indexed (MRC)	`void ApplIlTxStop (CanChannelHandle channel)`
Parameter	
channel (Indexed)	Handle of the logical Can Driver channel.
Return code	
void	none
Functional Description	
This method is called to indicate the performed transition stop for the Tx state machine.	
Particularities and Limitations	
none	
Call context	
The function is called by the IL in the context of IlTxStop.	

6.1.4.6 ApplIlFatalError

ApplIlFatalError

Prototype	
`void ApplIlFatalError (vuint8 errorNumber)`	
Parameter	
errorNumber	numeric error code
Return code	
void	none
Functional Description	
If assertions are configured, this function is called to indicate invalid user conditions (API, reentrance), inconsistent generated data, hardware errors and internal errors.	
Particularities and Limitations	
none	
Call context	
The function is be called by the IL on task and interrupt level.	

6.1.5 Generated Callback Functions

All callback functions can be activated or deactivated by a switch in the Configuration Tool. If a callback function is activated by the user, the application has to provide this function.

	Info The generated callback declarators in this chapter are depending on the configuration.

NotificationFunction

Prototype	
`void` **`NotificationFunction`** `(void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This function is used to determine the notification of a signal, signal group or grouped signal. The generated prototype declarator is composed of a configurable prefix for the notification and the network signal, signal group or grouped signal name and a configurable postfix for the notification. e.g. <ApplIl><NetworkSignalName><SigIndication>.	
Particularities and Limitations	
The function is called by the IL and implemented by the application.	
Call context	

The function is called notification class and configuration dependent: - Indication : ->Il polling is enabled in the configuration for the message: The function is called in the context of the IlRxTask or IlRxStateTask. ->Il polling is disabled in the configuration for the message: The function is called in the context of the CAN Driver message indication function. - Confirmation : ->Il polling is enabled in the configuration for the message: The function is called in the context of the IlTxTask or IlTxStateTask. ->Il polling is disabled in the configuration for the message: The function is called in the context of the CAN Driver message confirmation function. - RxTimeout : ->The function is called in the context of the IlRxTask or IlRxTimerTask. - TxTimeout : ->The function is called in the context of the IlTxTask or IlTxTimerTask.

TransitionChangeNotificationFunction

Prototype	
`void` **`TransitionChangeNotificationFunction`** `(void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This function is used to determine the transition change of a signal, signal group or grouped signal. The generated prototype declarator is composed of - a configurable prefix for the transition notification and - the network signal, signal group or grouped signal name and - a configurable postfix for the transition notification. e.g. <ApplIl><NetworkSignalName><TxStart>.	
Particularities and Limitations	
The function is called by the IL and implemented by the application.	
Call context	
The function is called in the context of IlInitPowerOn, IlInit, IlRxStart, IlTxStart, IlRxStop, IlTxStop.	

RxMessageTimeoutFunction

Prototype	
`void` **`RxMessageTimeoutFunction`** `(void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
This function is used to determine the Rx timeout of a message. The generated prototype declarator is composed of - a configurable prefix for the Rx message timeout and - the network message name and - a configurable postfix for the Rx message timeout. e.g. <ApplIl><NetworkMessageName><MsgTimeout>.	
Particularities and Limitations	
The function is called by the IL and implemented by the application.	
Call context	
The function is called in the context of the IlRxTask or IlRxTimerTask.	

7 Limitations

7.1 CANgen Compatibility

7.1.1 Database attributes

Signal value definitions are defined sometimes as decimal or hexadecimal value. Due to this signal values can be defined as string attribute in the database. Both value representations are interpreted by GENy. GENy is fully compatible to attributes used with CANGen. Deviations are described in chapter 5.1.5 Former Attributes.

7.1.2 Application Code

If you want to use your existing application with generated GENy code, pay attention to:

Rx Rds Write access and Tx Rds Read access is not supported any more.

The IlOldstyleAPI is not supported any more.

A strict Naming concept is introduced with GENy. The default Pre- and Postfixes have changed. (The prefix „_a_" is now „Appl") It is possible for you, to restore the CANGen compatible Pre- and Postfixes in the NameDecorator.

Data Type Prefixes are not supported any more.

The Prefixes of Signal Handles have been changed to „IlTxSigHnd" and „IlRxSigHnd" instead of „ILTx".

The Can Driver Interface is in GENy strictly separated from the Interaction Layer. Due to this, a used Appl message must be adapted.

- The Can Driver message Indication flag postfix is now separately configurable from the IL Indication Flag.
- If Can Driver Signal access macros are used for signals <= 1 Byte, the function style interface is not supported any more. The signal value is assigned like a value to a variable

CANGen supported a configuration switch, to activate the multiple channel interfaces in single channel configurations. This feature is discontinued in GENy.

7.1.3 Generator

- ESCAN00024091

  If "Common Buffer" is configured between Rx or Tx messages which are used in different identities of the ECU, configure both messages in the CAN Driver as Basic CAN message and use the same Basic CAN object.

- ESCAN00017472

  Do not configure Common Buffer for messages containing multiplexed signals.

> ESCAN00023799

  Do not configure Common Buffer for tx messages which are used in the same identitiy containing signals the GenSigSendType OnChange or OnChangeWithRepetition.

8 Glossary and Abbreviations

8.1 Glossary

Term	Description
API	Application Program Interface, for OSEK: The description of the user interface to the operating system, communications and network management functions.
AUTOSAR	Automotive Open System Architecture
bus	Defines what we call internal as channel or connection.
CAN	Controller Area Network protocol originally defined for use as a communication network for control applications in vehicles.
CANGen	Generation tool for CANbedded components
configuration	The communication configuration adapts the communication stack to the specific component requirements by means of the Generation Tool.
DBC	CAN data base format of the Vector company which is used by Vector tools
ESCANXXXXXXXX	Vector PES Clearquest Database ID. Replace XXXXXXXX by the numeric identifier.
generation tool	See CANgen, DBKOMGen and GENy. The generation tool configures the communication stack, Flash Bootloader, etc. based on database attributes (vehicle manufacturer), project settings (module supplier) and license information (Vector).
GENy	Generation tool for CANbedded and MICROSAR components
ID	Identifier (e.g. Identifier of a CAN message)
manufacturer	Vehicle manufacturer
message	A message is responsible for the logical transmission and reception of information depending on the restrictions of the physical layer. The definition of the message contents is part of the data base given by the vehicle manufacturer.
module	A module designates a controller (Identical with ECU).
MRC	Multiple Receive Channel
MSC	Message Sequence Chart
Mutual exclusion	To modify shared data, a task must be able to get exclusive access for a limited time, i.e. all other tasks must be excluded to access this data. All tasks modifying shared data must be able to do this exclusion. Therefore this exclusion is called mutual exclusion.
Node	A network topological entity. Nodes are connected by data links forming the network. Each node is separately addressable on the network.
Online	(Normal) state of the data link layer. Application and Network Management communication are possible.
OSEK	Name of the overall project: Abbreviation of the German term "Offene Systeme und deren Schnittstellen fÃ¼r die Elektronik im Kraftfahrzeug" -

	Open Systems and the Corresponding Interfaces for Automotive Electronics.
Protocol	A formal set of conventions or rules governing the exchange of information between protocol entities. Protocol comprises syntax and semantics of the protocol messages as well as the instructions on how to react to them.
Register	A register is a memory area in the controller, e.g. in the CAN controller. Distinguish register from Buffer.
Release	State transition of a task from waiting to ready. At least one event has occurred which a task has waited on.
Request	A service primitive in compliance with the ISO/OSI Reference Mode (ISO 7498). With the service primitive 'request' a service user requests a service from a service provider.
Resource	Generally, resources are hardware or software components which are managed by the operating system. The OSEK operating system provides resources to support task coordination by mutual exclusion of critical sections. As an example, the scheduler is treated like a resource. A task which holding the scheduler cannot be interrupted by all other tasks. Resource Management. Access control for inseparable operations to jointly used (logic) resources or devices, or for control of a program flow.
Response	A service primitive defined in the ISO/OSI Reference Model (ISO 7498). The service primitive 'response' is used by a service user in order to reply to a preceding indication from service provider.
Running	A task state. In the running state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time. The state is entered by the state transition start and can be exited via the state transitions Wait, Preempt or Terminate.
Safety	A situation in which the risk is equal or lower than the limit risk. Risk is a measure which considers both the probability of an accident and the expected extend of damage in the case of an accident. The limit risk is the highest risk which is still justifiable.
Software specification	A software specification is a set of requirements that can be of different types, as behavior, interfaces, timing constraints, needed resources, safety, etc.
Start	State transition of a task from ready to running. A ready task selected by the scheduler is executed.
SWC	Software Component, application software entity in AUTOSAR
Task	A task provides the framework for the execution of functions. Therefore a task has a context of its own, i.e. a stack, a register retrieval range and a memory of its own. A task can be executed in principle on a processor concurrently with other tasks. A task is executed under the control of the scheduler according to the task priority assigned to it, and to the selected scheduling policy. A distinction is made between basic tasks and extended task.
Task level	Processing level where the actual application software, is executed. Tasks are executed according to the priority assigned to them, and to the selected scheduling policy. Other processing levels are: Interrupt level and operating system level.

Task priority	The priority of a task is a measure for the precedence with which the task is to be executed. In principle, priorities are defined statically. However, in particular cases (see Priority Ceiling Protocol) a task can be processed by the operating system with a defined higher priority. As a capability of the CC, tasks of the same priority are admissible within a system. Tasks of equal priority are started according to the order in which they are called. To this effect, extended tasks which change from the waiting state into the ready state are treated like new tasks.
Validation	Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled. Ensuring the correctness of a specification.
wait	State transition of a task from running to waiting. The running task requires an event to continue operation. It causes its transition into the waiting state by using a system service.
Window	Communication object of the data link layer for sending and receiving NM messages.

8.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CCL	Communication Control Layer
COM	Communication Layer
CPU	Central Processing Unit
DM	Deadline Monitoring
ECU	Electronic Control Unit
IL	Interaction Layer
IRQ	Interrupt Request
ISO	International Standardization Organization
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
NM	Network Management
OEM	Original Equipment Manufacturer
OS	Operating System
OSI	Open Systems Interconnection
RAM	Random Access Memory
RDS	Read Data Segment
ROM	Read-Only Memory
SRS	Software Requirement Specification

SWC	Software Component
SWS	Software specification

9 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector-informatik.com