# COMMSTACK<FLEXRAY> 1.8

User Manual


Markus Eggenbauer

Version 1.8.0, March, 2006

**Contact Information:**

DECOMSYS - Dependable Computer Systems
Hardware und Software Entwicklung GmbH
Stumpergasse 48/28
A-1060 Wien, Austria

| | |
|---|---|
| +43 (1) 599 83 - 0 | Phone |
| +43 (1) 599 83 – 18 | Fax |
| info@decomsys.com | E-Mail |
| www.decomsys.com | Web |

# Table of Contents

# 1 Introduction

The DECOMSYS::COMMSTACK<FlexRay> is a comprehensible FlexRay controller software driver package. The main intention of a software driver is to provide an abstract software interface to some specific hardware device. The DECOMSYS::COMMSTACK provides a FlexRay specific interface on top of its API while abstracting the peculiarities of specific FlexRay communication controller hardware implementations.

The DECOMSYS::COMMSTACK<FlexRay> is a flexible library that easily extends applications with FlexRay capabilities. To obtain this use case, the DECOMSYS::COMMSTACK<FlexRay> was developed with modularity and flexibility in mind. It is free of any non-FlexRay functionality and free of dependability on any external components. This enables a simple integration of this library into your existing application framework.

## 1.1 Overview

DECOMSYS::COMMSTACK is a software driver that abstracts different FlexRay communication controller implementations and provides a generic FlexRay application programming interface.

It is intended to be used by higher layer software that has no sense about the different FlexRay communication controller implementations available, but is familiar with the standard FlexRay behavior and properties.

Users of DECOMSYS::COMMSTACK are e.g. transport protocols, network management, FlexRay COM layer (FlexCOM) and all kinds of user applications using FlexRay directly.

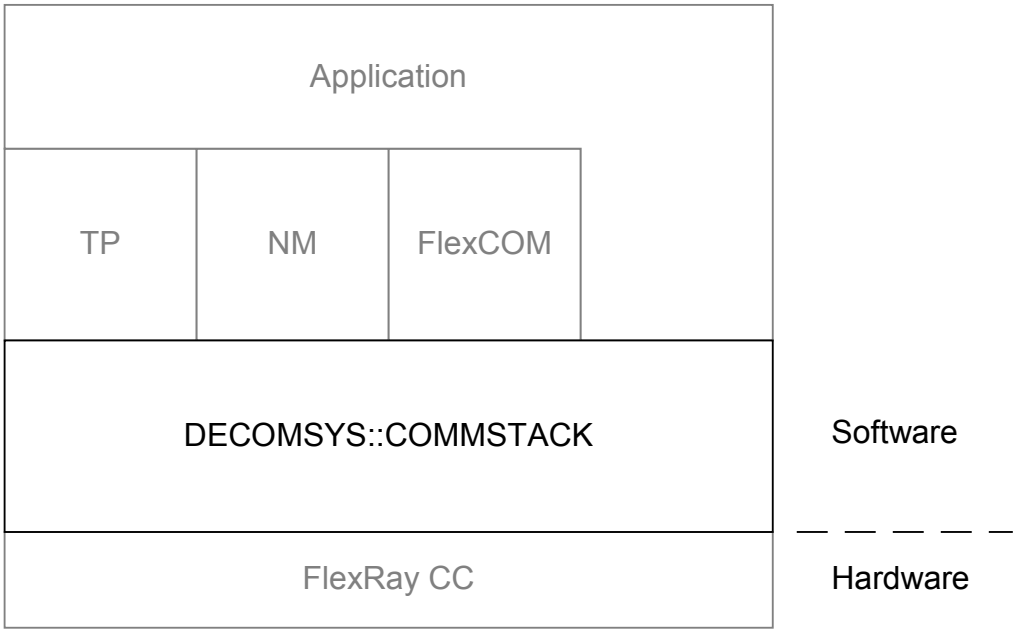


**Figure 1: System Overview**

## 1.2 Glossary

| | |
|---|---|
| API | Application Programming Interface |
| BA | Buffer Assignment |
| BE | Big Endian |
| BOR | Binary Object Repository (DECOMSYS::DESIGNER PRO data exchange format) |
| CC | Communication Controller |
| CPU | Central Processing Unit |
| ECU | Electronic Control Unit |
| ERAY10 | Bosch ERAY-IP FlexRay Communication Controller |
| FIBEX | Field Bus Exchange |
| FIFO | First-In First-Out |
| FlexCOM | DECOMSYS signal communication layer |
| GUI | Graphical User Interface |
| LE | Little Endian |
| LSB | Least Significant Byte |
| lsb | least significant bit |
| MFR4200 | Freescale MFR4200 FlexRay Communication Controller |
| MFR4300 | Freescale MFR4300 FlexRay Communication Controller |
| MSB | Most Significant Byte |
| msb | most significant bit |
| NM | Network Management |
| PHIP1 | Philips ABL FlexRay Communication Controller (e.g. contained in Philips SJA2510). |
| Rx | Reception |
| TDDLL | Time-Driven Data-Link-Layer |
| TP | Transport Protocol |
| Tx | Transmission |
| XCDEF | DECOMSYS::Designer data exchange format |

# 2 Architecture

## 2.1 COMMSTACK Design

The internal structure of DECOMSYS::COMMSTACK 1.8 can be depicted as shown in Figure 2.

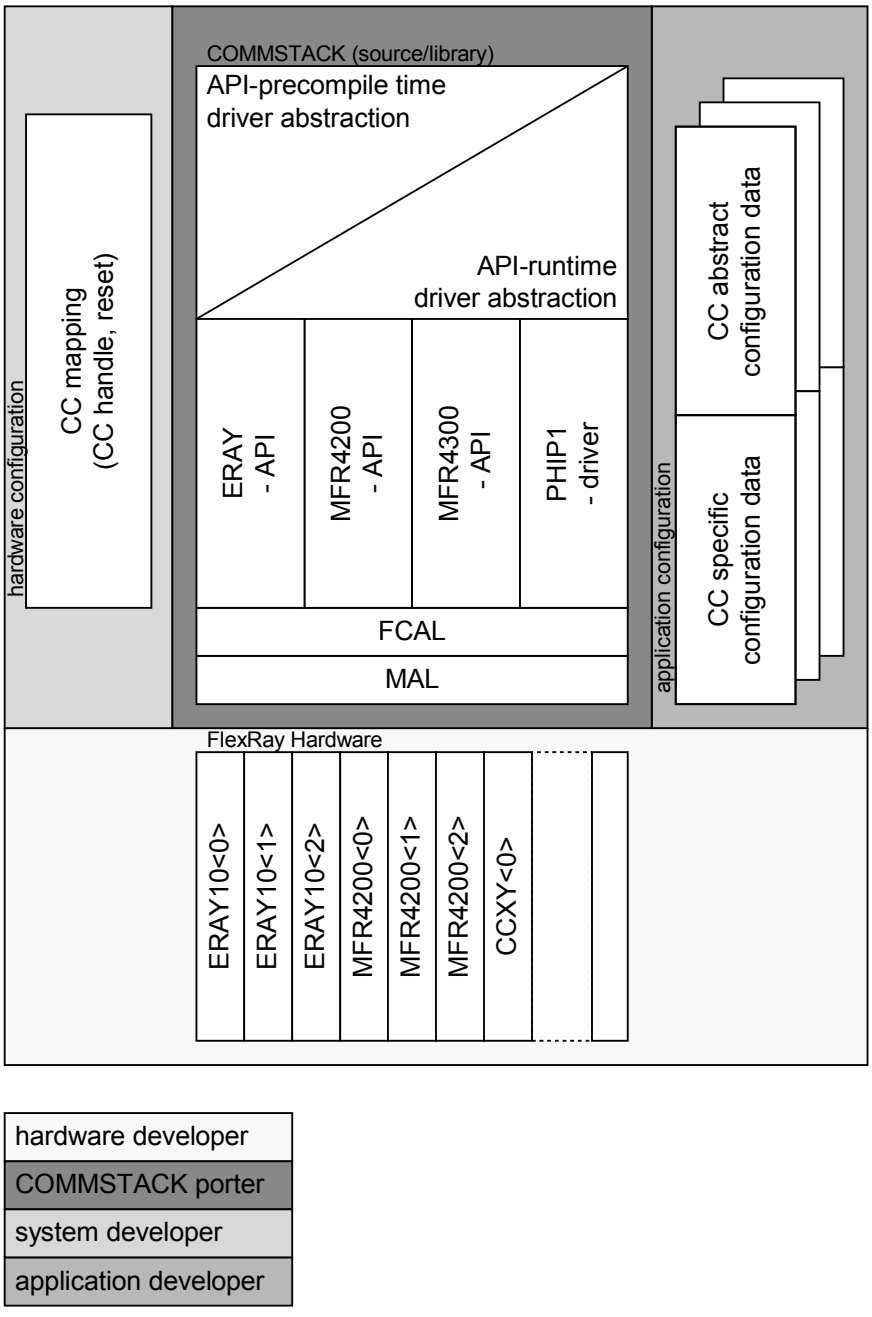


**Figure 2: COMMSTACK<FlexRay> 1.8 System Architecture**

FlexRay Hardware and DECOMSYS::COMMSTACK components:

- **FlexRay Hardware**: The CC hardware consists of one to several FlexRay communication controllers that might be mixed up of different implementations.
- **COMMSTACK**: DECOMSYS::COMMSTACK is ported to a dedicated host-CPU, CC-hardware connection schema and is integrated into a specific development environment. All connection schemas of communication controllers of the same type must be identical. CCs of one implementation may differ in their CC-handle only (e.g. different base addresses).
- **Hardware Configuration**: The hardware configuration contains the device mapping and reset configuration of the FlexRay CC devices. The configuration of CC-handles identifies dedicated connected FlexRay controllers. The CC mapping maps FlexRay CC implementation abstract controller indices to dedicated CC devices. Additionally reset functions (which are strongly hardware specific) have to be supplied within the hardware configuration. The static configuration is a post-compile pre build time configuration.
- **Application Configuration**: The application specific configuration is created by DECOMSYS::DESIGNER PRO. The application configuration is a post-build configuration.

## 2.2 State Model

The COMMSTACK behavior is controlled by a state machine that is administrated for each single FlexRay controller driven by the COMMSTACK. Within the further documentation this state is referred to as COMMSTACK FlexRay controller state. Figure 3 shows the state machine states and transitions that are provided by the COMMSTACK.

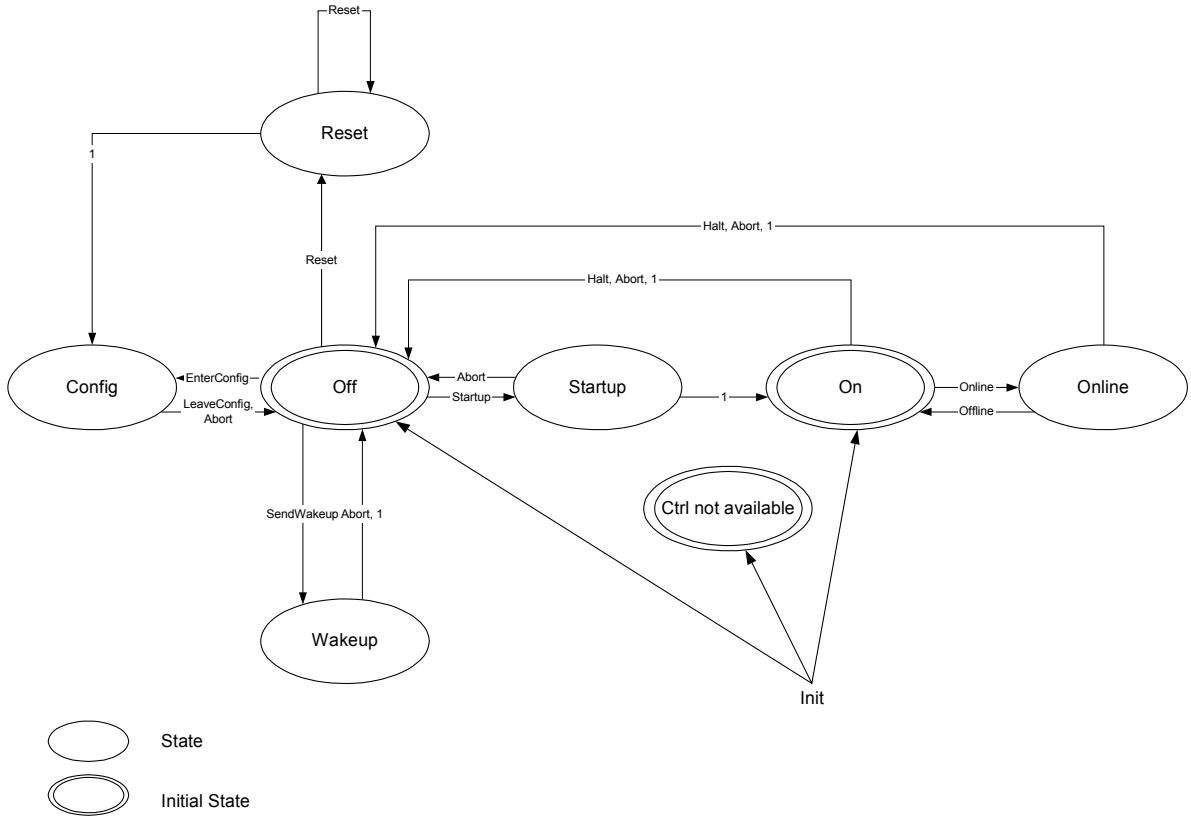


**Figure 3: COMMSTACK FlexRay controller state machine**

The following table describes the states and transitions of the COMMSTACK in detail.

| State | Transition | Description |
|---|---|---|
| `Off` | The FlexRay controller does not perform any access to the network and is not able to be configured. (e.g. FlexRay ready state). This is the start state after COMMSTACK initialization (`TDDLL_Init()`) if the CC is detected successfully and the cluster is not synchronized. | |
| | `Reset` | This transition changes into the state “Reset” and performs a hard reset of the FlexRay CC. |
| | `EnterConfig` | This transition changes into the state “Config” and switches the CC into config mode. |
| | `SendWakeupChA` | This transition changes into the state “Wakeup” und initiates the transmission of a wakeup pattern on FlexRay channel A of the dedicated FlexRay CC. |
| | `SendWakeupChB` | This transition changes into the state “Wakeup” und initiates the transmission of a wakeup pattern on FlexRay channel B of the dedicated FlexRay CC. |
| | `Startup` | This transition changes into the state “Startup” and switches the controller into the startup state. The CC performs an active or passive startup (depending on the configuration). During this transition the global interrupt source is disabled. |
| `ctrl not available` | This is the start state after COMMSTACK initialization (`TDDLL_Init()`) if the CC is not detected. No transition can be invoked from this state. | |
| `Startup` | The FlexRay controller is in state “Startup” and tries to set up network communication (active or passive) depending on the configuration. | |
| | `Abort` | This transition aborts the startup procedure and changes immediately into the “Off” state. During this transition the global interrupt source is disabled. |
| | `1` | If the startup was successfully (the CC is synchronized to the cluster) the state “Startup” is left to state “On” automatically (evaluate the current state using API function `TDDLL_GetCtrlState()`). |

| | | |
|---|---|---|
| **On** | The CC is synchronized to the cluster. The buffer access for communication of all COMMSTACK API functions is turned off. This is the start state after COMMSTACK initialization (**TDDLL_Init()**) if the CC is detected successfully and the cluster is synchronized. | |
| | **Online** | The transition immediately switches the state machine into the state “Online”. |
| | **Halt** | The transition “Halt” immediately changes into the state “Off” and halts the FlexRay communication at the end of the current communication cycle. During this transition the global interrupt source is disabled. |
| | **Abort** | The transition “Abort” immediately changes into the state “Off” and immediately aborts the FlexRay communication. During this transition the global interrupt source is disabled. |
| | **1** | If the synchronization is lost the transition to state “Off” is performed automatically (evaluate the current state using API function **TDDLL_GetCtrlState()**).. During this transition the global interrupt source is disabled. |
| **Online** | The FlexRay controller is synchronized to the network and the software driver is activated to actively access buffers for transmission or reception. | |
| | **Offline** | The transition immediately switches the state machine into the state “On”. |
| | **Halt** | The transition “Halt” immediately changes into the state “Off” and halts the FlexRay communication at the end of the current communication cycle. During this transition the global interrupt source is disabled. |
| | **Abort** | The transition “Abort” immediately changes into the state “Off” and immediately aborts the FlexRay communication. During this transition the global interrupt source is disabled. |
| | **1** | If the synchronization is lost the transition to “Off” is performed automatically (evaluate the current state using API function **TDDLL_GetCtrlState()**).. During this transition the global interrupt source is disabled. |
| **Config** | The FlexRay controller is in configuration mode. All configuration parameters can be changed. | |
| | **LeaveConfig** | The transition “LeaveConfig” leaves the “Config” state of the state machine immediately into the “Off” state. |
| | **Abort** | The transition “Abort” leaves the “Config” state of the state machine immediately into the “Off” state. |
| **Reset** | The FlexRay controller is being reset. No access to the FlexRay controller should be performed. | |
| | **Reset** | The transition “Reset” immediately performs a reset again. |
| | **1** | If the reset is completed the state machine automatically changes into state “Config” (evaluate the current state using API function **TDDLL_GetCtrlState()**).. |

| | | |
|---|---|---|
| **Wakeup** | The FlexRay controller is transmitting a wakeup pattern. | |
| | **Abort** | The transition “Abort” immediately aborts the transmission of a wakeup-pattern and changes into state “Off”. |
| | **1** | If the transmission of the wakeup pattern is completed, the state “Wakeup” is left automatically into state “Off” (evaluate the current state using API function **TDDLL_GetCtrlState()**). |

# 3 API Documentation

## 3.1 Basic Datatypes

Basic datatypes are datatypes every other datatype of the COMMSTACK is built of. The basic datatypes are required by the COMMSTACK. These datatypes will in general be provided by an external source if the COMMSTACK is used in an environment already defining these datatypes. If these datatypes do not exist, the COMMSTACK porter must define them for a proper COMMSTACK operation.

### 3.1.1 uint8

This is an integral unsigned datatype that provides a value range of 8bit (0 to 255 decimal).

### 3.1.2 uint16

This is an integral unsigned datatype that provides a value range of 16bit (0 to 65535 decimal).

### 3.1.3 uint32

This is an integral unsigned datatype that provides a value range of 32bit (0 to 4294967295 decimal).

### 3.1.4 uint8_least

This is an integral unsigned datatype that provides at least a value range of 8bit (0 to 255 decimal) but possibly more if this would improve execution performance on certain hardware.

### 3.1.5 uint16_least

This is an integral unsigned datatype that provides at least a value range of 16bit (0 to 65535 decimal) but possibly more if this would improve execution performance on certain hardware.

### 3.1.6 uint32_least

This is an integral unsigned datatype that provides at least a value range of 32bit (0 to 4294967295 decimal) but possibly more if this would improve execution performance on certain hardware.

### 3.1.7 sint8

This is an integral signed datatype that provides a value range of 8bit (-128 to 127 decimal).

### 3.1.8 sint16

This is an integral signed datatype that provides a value range of 16bit (-32768 to 32767 decimal).

### 3.1.9 sint32

This is an integral signed datatype that provides a value range of 32bit (-2147483648 to 2147483647 decimal).

## 3.2 Specific Datatypes

Specific datatypes are defined by the COMMSTACK and mentioned to be reserved for COMMSTACK specific usage only. The datatypes described in this chapter are single integral datatypes.

### 3.2.1 TDDLL_BooleanType

This datatype is used for variables that have to differ between two states. The states that must be used for **`TDDLL_BooleanType`** are **`TDDLL_TRUE`** and **`TDDLL_FALSE`** or expressions as provided by the compiler.

### 3.2.2 TDDLL_ReturnType

This datatype is an enumerator used as generic function return type. Please consult the API documentation of a specific function for the exact meaning of the return code in the context of the specific function call.

Supported values for the datatype **`TDDLL_ReturnType`** and the generic meanings of these values are described in the following table.

| Value | Description |
|---|---|
| **`TDDLL_E_OK`** | Function exited successfully. |
| **`TDDLL_E_BAD_CONFIG`** | Some configuration of the FlexRay controller is invalid or bad. |
| **`TDDLL_E_ACCESS`** | Error accessing the FlexRay controller. |
| **`TDDLL_E_OVERFLOW`** | Some date structure or resource is busy. |
| **`TDDLL_E_UNDERFLOW`** | Some data structure or resource is empty. |
| **`TDDLL_E_INVALID_IDX`** | Some argument given is invalid. |
| **`TDDLL_E_OFFLINE`** | The communication controller is in state offline - function execution not permitted. |
| **`TDDLL_E_NOT_SYNC`** | The communication controller is not synchronized to the cluster - function execution not permitted. |
| **`TDDLL_TX_PENDING`** | The transmission request is pending. |
| **`TDDLL_E_INVALID_DATA`** | The data frame received is invalid (implemented for backward compatibility). |

### 3.2.3 TDDLL_CtrlStateType

This datatype is an enumerator used for the COMMSTACK FlexRay controller state of each distinct FlexRay controller. The COMMSTACK FlexRay controller state is an abstract state that is represented in software only. These states do not directly map to vendor-specific FlexRay controller states but map as close as possible to them. Supported COMMSTACK FlexRay controller states by the datatype **`TDDLL_CtrlStateType`** are listed in the following table. Please refer to the COMMSTACK FlexRay controller state model in chapter 2.2 for a detailed description of COMMSTACK FlexRay controller states.

| Value | Description |
|---|---|
| **TDDLL_S_CTRL_NOT_AVAILABLE** | No FlexRay controller could be detected on the configured port. |
| **TDDLL_S_OFF** | The FlexRay controller is halted, no communication nor synchronization mechanism is carried out. |
| **TDDLL_S_RESET** | The FlexRay controller is being reset (hard-reset). |
| **TDDLL_S_CONFIG** | The FlexRay controller is in config mode. |
| **TDDLL_S_WAKEUP** | A wakeup pattern is being transmitted in the FlexRay controller. |
| **TDDLL_S_STARTUP** | The FlexRay controller is performing some start-up procedure to get synchronized to the network (or start-up the network communication). |
| **TDDLL_S_ON** | The FlexRay controller is synchronized to the network. The buffer access for transmission and reception from API is disabled by software. |
| **TDDLL_S_ONLINE** | The FlexRay controller is synchronized to the network. The buffer access for transmission and reception from API is enabled. |

### 3.2.4 TDDLL_CtrlTransitionType

This datatype is an enumerator used for the COMMSTACK FlexRay controller state transitions that can be performed for each distinct FlexRay controllers. These transitions are required to change the COMMSTACK FlexRay controller state of a FlexRay controller.

Supported COMMSTACK FlexRay controller transitions by the datatype **TDDLL_CtrlTransitionType** are listed in the following table. Please refer to the COMMSTACK FlexRay controller state model in chapter 2.2 for a detailed description of COMMSTACK FlexRay controller transitions.

| Value | Description |
|---|---|
| **TDDLL_T_RESET** | Perform a hard-reset on the FlexRay controller. |
| **TDDLL_T_ENTER_CONFIG** | Enter the FlexRay controller config mode (into state **TDDLL_S_CONFIG**). |
| **TDDLL_T_LEAVE_CONFIG** | Leave the FlexRay controller config mode (into state **TDDLL_S_OFF**). |
| **TDDLL_T_STARTUP** | Trigger a “start communication” request on a FlexRay controller. |
| **TDDLL_T_ONLINE** | Set the COMMSTACK FlexRay controller state to **TDDLL_S_ONLINE**. |
| **TDDLL_T_OFFLINE** | Set the COMMSTACK FlexRay controller state from **TDDLL_S_ONLINE** to **TDDLL_S_ON**. |
| **TDDLL_T_HALT** | Halt the FlexRay controller at the end of the current communication cycle. |
| **TDDLL_T_ABORT** | Immediately abort communication and synchronization of the FlexRay controller. |
| **TDDLL_T_WAKEUP_CHA** | Transmit a wakeup pattern on FlexRay channel A. |
| **TDDLL_T_WAKEUP_CHB** | Transmit a wakeup pattern on FlexRay channel B. |

| | |
|---|---|
| **TDDLL_T_CTRL_NOT_AVAILABLE** | Set the FlexRay controller to the state **TDDLL_S_CTRL_NOT_AVAILABLE.** |

### 3.2.5 TDDLL_CtrlTypesType

This datatype is used for variables that represent the different FlexRay vendor-specific controllers. The values allowed for this type are **TDDLL_CTRL_TYPE_ERAY10** for representing a Bosch E-RAY-based FlexRay controller, **TDDLL_CTRL_TYPE_MFR4200** for a Freescale FlexRay controller MFR4200, **TDDLL_CTRL_TYPE_MFR4300** for a Freescale FlexRay controller MFR4300 and **TDDLL_CTRL_TYPE_PHIP1** for a Philips FlexRay Controller Rev1 (SJA2510).

### 3.2.6 TDDLL_FrameDscRefIDXType

This datatype is used for identifying frame to transmit or receive via the COMMSTACK API. Each FlexRay frame is assigned a unique ID within the COMMSTACK configuration. Using this ID identifies the frame when calling API functions operating on FlexRay frames.

### 3.2.7 TDDLL_CtrlIDXType

This datatype is used for variables that represent an index for a FlexRay controller. FlexRay controller indices are used for distinctly identifying a FlexRay controller within the software.

### 3.2.8 TDDLL_FrameIDType

This datatype is used for variables that represent a FlexRay frame identifier of a particular frame.

### 3.2.9 TDDLL_ChannelIDXType

This datatype is used for variables that represent a channel identifier. Possible values for this datatype are provided by the symbolic constants **TDDLL_CHA** for identifying FlexRay channel A, **TDDLL_CHB** for identifying FlexRay channel B and **TDDLL_CHAB** for identifying FlexRay channel A and B.

### 3.2.10TDDLL_CycleType

This datatype is used for variables that are used to carry FlexRay cycle filtering information as the cycle repetition value or the base cycle. The cycle repetition value states the period of a FlexRay frame in units of communication cycles. The COMMSTACK provides cycle repetition values of 1,2,4,8,16,32,64 only.

The base cycle represents the offset in the cycle repetition interval. Allowed values for the base cycle are 0 to (Repetition cycle value – 1).

### 3.2.11TDDLL_LengthType

This datatype is used for variables that represent FlexRay payload length information.

### 3.2.12TDDLL_BufferIDXType

This datatype is used for variables that represent a buffer index that maps to a transmission/reception resource of a FlexRay controller.

### 3.2.13TDDLL_TickType

This type is used for variables that represent time in units of FlexRay macroticks.

### 3.2.14TDDLL_TimeType

This type is used for variables that represent time in units of nanoseconds.

### 3.2.15TDDLL_Interrupt SourceType

This datatype is used for variables that are able to hold interrupt sources supported by the FlexRay controller. The interrupt sources are identified using bitmasks enabling to combine several interrupt sources into a single variable and operation.

Supported values for interrupt sources are described in the following table.

| Value | Description |
|---|---|
| `TDDLL_BM_GLOBAL_INTERRUPT` | This bitmask identifies the global interrupt enable switch for the FlexRay controller. |
| `TDDLL_BM_ABSOLUTE_TIMER_INTERRUPT` | This bitmask identifies the FlexRay absolute timer interrupt. |
| `TDDLL_BM_RELATIVE_TIMER_INTERRUPT` | This bitmask identifies the FlexRay relative timer interrupt. |
| `TDDLL_BM_CYCLE_INTERRUPT` | This bitmask identifies the FlexRay cycle start interrupt. |

### 3.2.16TDDLL_POCStateType

This enumerator is used to represent the FlexRay controller POC state (see FlexRay specification type T_POCState).

| Value | Description |
|---|---|
| `TDDLL_POCSTATE_CONFIG` | The POC state of the FlexRay controller is CONFIG. |
| `TDDLL_POCSTATE_DEFAULT_CONFIG` | The POC state of the FlexRay controller is DEFAULT_CONFIG. |
| `TDDLL_POCSTATE_HALT` | The POC state of the FlexRay controller is HALT. |
| `TDDLL_POCSTATE_NORMAL_ACTIVE` | The POC state of the FlexRay controller is NORMAL_ACTIVE. |
| `TDDLL_POCSTATE_NORMAL_PASSIVE` | The POC state of the FlexRay controller is NORMAL_PASSIVE. |
| `TDDLL_POCSTATE_READY` | The POC state of the FlexRay controller is READY. |
| `TDDLL_POCSTATE_STARTUP` | The POC state of the FlexRay controller is STARTUP. |
| `TDDLL_POCSTATE_WAKEUP` | The POC state of the FlexRay controller is WAKEUP. |

### 3.2.17TDDLL_SlotModeType

This enumerator is used to represent the FlexRay controller POC slot mode (see FlexRay specification type T_SlotMode).

| Value | Description |
|---|---|
| TDDLL_SLOTMODE_SINGLE | The POC slot mode of the FlexRay controller is SINGLE. |
| TDDLL_SLOTMODE_ALL_PENDING | The POC slot mode of the FlexRay controller is ALL_PENDING. |
| TDDLL_SLOTMODE_ALL | The POC slot mode of the FlexRay controller is ALL. |

### 3.2.18TDDLL_ErrorModeType

This enumerator is used to represent the FlexRay controller POC error mode (see FlexRay specification type T_ErrorMode).

| Value | Description |
|---|---|
| TDDLL_ERRORMODE_ACTIVE | The error mode of the FlexRay controller POC is ACTIVE. |
| TDDLL_ERRORMODE_PASSIVE | The error mode of the FlexRay controller POC is PASSIVE. |
| TDDLL_ERRORMODE_COMM_HALT | The error mode of the FlexRay controller POC is COMM_HALT. |

### 3.2.19TDDLL_WakeupStatusType

This enumerator is used to represent the FlexRay controller wakeup status (see FlexRay specification type T_WakeupStatus).

| Value | Description |
|---|---|
| TDDLL_WAKEUP_UNDEFINED | The wakeup status is UNDEFINED. |
| TDDLL_WAKEUP_RECEIVED_HEADER | The wakeup status is RECEIVED_HEADER. |
| TDDLL_WAKEUP_RECEIVED_WUP | The wakeup status is RECEIVED_WUP. |
| TDDLL_WAKEUP_COLLISION_HEADER | The wakeup status is COLLISION_HEADER. |
| TDDLL_WAKEUP_COLLISION_WUP | The wakeup status is COLLISION_WUP. |
| TDDLL_WAKEUP_COLLISION_UNKNOWN | The wakeup status is COLLISION_UNKNOWN. |
| TDDLL_WAKEUP_TRANSMITTED | The wakeup status is TRANSMITTED. |

### 3.2.20TDDLL_StartupStateType

This enumerator is used to represent the FlexRay controller startup substate (see FlexRay specification type T_StartupState).

| Value | Description |
|---|---|
| TDDLL_STARTUP_UNDEFINED | The startup substate is UNDEFINED. |
| TDDLL_STARTUP_COLDSTART_LISTEN | The startup substate is COLDSTART_LISTEN. |
| TDDLL_STARTUP_INTEGRATION_COLDSTART_CHECK | The startup substate is COLDSTART_CHECK. |

| | |
|---|---|
| TDDLL_STARTUP_COLDSTART_JOIN | The startup substate is COLDSTART_JOIN. |
| TDDLL_STARTUP_COLDSTART_COLLISION_RESOLUTION | The startup substate is COLDSTART_COLLISION_RESOLUTION. |
| TDDLL_STARTUP_COLDSTART_CONSISTENCY_CHECK | The startup substate is COLDSTART_CONSISTENCY_CHECK. |
| TDDLL_STARTUP_INTEGRATION_LISTEN | The startup substate is INTEGRATION_LISTEN. |
| TDDLL_STARTUP_INITIALIZE_SCHEDULE | The startup substate is INITIALIZE_SCHEDULE. |
| TDDLL_STARTUP_INTEGRATION_CONSISTENCY_CHECK | The startup substate is INTEGRATION_CONSISTENCY_CHECK. |
| TDDLL_STARTUP_COLDSTART_GAP | The startup substate is COLDSTART_GAP. |

## 3.3 Structure Data Types

### 3.3.1 TDDLL_POCStatusType

This compound datatype represents the FlexRay controller POC status (see FlexRay specification type T_POCStatus).

This datatype is intended to be used at the COMMSTACK user API.

| Member | Description |
|---|---|
| TDDLL_POCStateType POCState | This member holds the FlexRay controller POC state. |
| TDDLL_BooleanType Freeze | This member shows whether the FlexRay POC has entered the halt state due to an error condition. |
| TDDLL_BooleanType CHIHaltRequest | This member shows whether a CHI halt request was performed on the FlexRay controller. |
| TDDLL_BooleanType ColdstartNoise | This member shows whether the FlexRay controller startup mechanism completed under a noisy channel. |
| TDDLL_ErrorSlotModeType SlotMode | This member holds the POC slot mode. |
| TDDLL_WakeupStatusType WakeupStatus | This member holds the FlexRay controller wakeup status. |
| TDDLLStartupStateType StartupState | This member holds the FlexRay controller startup state. |

### 3.3.2 TDDLL_FrameDscType

This compound datatype holds the constant configuration data of a FlexRay frame that can be transmitted/received by the user. Additionally buffers are configured by the same datatype because all required information is stored within this datatype.

This datatype is usually created within the COMMSTACK.

| Member | Description |
|---|---|
| `void *pExtendedConfig` | This member is a pointer to an extended configuration structure if this is necessary for the buffer configuration of some vendor-specific FlexRay controller type. If no extended configuration is required, this member must contain NULL. |
| `const TDDLL_QueueDscType *pQueueDsc` | This member is a pointer to a queue configuration structure for the frame. If no queue is assigned to this frame this member must contain NULL. |
| `const TDDLL_PoolDscType *pPoolDsc` | This member is a pointer to the bufferpool configuration structure of the bufferpool this frame is assigned to. If no bufferpool is assigned to the frame this member must contain NULL. |
| `uint32 nFrameSig` | This member contains an unique FlexRay frame signature that is built up by the following information (bitwise):<br>0-8: FlexRay controller index of the FlexRay frame.<br>9: Tx/Rx FlexRay frame.<br>10-20: FlexRay frame identifier.<br>21: FlexRay channel A usage.<br>22: FlexRay channel B usage.<br>23-25: Cycle repetition value.<br>26-31: Base cycle value. |
| `uint32 nConfigFlags` | This member contains some buffer configuration flags that are used for abstract buffer configuration (bitwise):<br>0-9: FlexRay controller buffer index.<br>10-16: Maximum payload length allowed for this buffer.<br>17: Transmission mode.<br>29-32: Not used. |
| `uint16 nHeaderCRC` | CRC calculated for the header of the FlexRay frame. |

**NOTE:** For obtaining a reference to data structures of this type of a dedicated frame the COMMSTACK user API functions TDDLL_LookupTxFrame()/TDDLL_LookupRxFrame() should be used.

### 3.3.3 TDDLL_QueueDscType

This compound datatype is used for the constant configuration data of software queue assigned to a receive/transmit FlexRay frame. FlexRay frames assigned a queue to them can be received/transmitted using API function **`TDDLL_RxFrameByIDQueued()`** / **`TDDLL_TxFrameByIDQueued()`.**

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| `uint16 nQueueMemoryIndex` | This member is an index into a memory assigned to the |

| | |
|---|---|
| | software queues. The memory area assigned to the queue starts at this offset within the queue memory. |
| **uint16 nPayloadLength** | This member holds the length of the maximum data payload for this queue in units of 16bits. |
| **uint16 nQueueLength** | This member holds the number of entries the queue can store (increased by 1). |

**NOTE:** The memory required by a queue (in units of sizeof(uint16) bytes) is calculated by ( 2 + ( ( nPayloadLength + 1 ) * nQueueLength ) ).

### 3.3.4 TDDLL_PoolDscType

This compound datatype is used for the constant configuration of a transmit bufferpool. Transmit frames of the dynamic FlexRay segment that are assigned to a bufferpool, share buffers among other frames assigned to the same bufferpool. This can be useful for saving buffers when transmitting frames with low frequency.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **const TDDLL_FrameDscType *pPoolStart** | This member points to an array of consecutive buffer configurations assigned to the bufferpool. |
| **uint16 nBuffer** | This member holds the number of buffers assigned to the bufferpool. |
| **uint16 nPoolAdminIndex** | This member is an index into an administrative memory assigned to bufferpooling. The memory assigned to the bufferpool starts at this offset within the pooled administrative memory. |

**NOTE:** The memory required by a pool (in units of sizeof(TDDLL_FrameDscType*) bytes) is equal to the number of buffers assigned to that pool.

### 3.3.5 TDDLL_CtrlDscType

This compound datatype contains the FlexRay controller global configuration data.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **Const TDDLL_FrameDscType *pBufferList** | This member holds a pointer to an array of configuration structures for initial configuration of all buffers of this FlexRay controller. |
| **const void* pCHICommands** | This member holds a pointer to an array of configuration structures for initial generic register initialization of this FlexRay controller. The type of the structure is defined by the vendor-specific type of the FlexRay controller (**TDDLL_CHI_MFR4200_CommandType** /**TDDLL_CHI_ERAY10_CommandType**). |
| **uint32 nFactorTicks** | |

| | |
|---|---|
| **uint32 nFactorTime** | |
| **uint16 nBuffer** | This member holds the number of buffers within the array the member **pBufferList** points to. |
| **uint16 nComands** | This member holds the number of registers within the array the member **pCHICommands** points to. |

### 3.3.6 TDDLL_ConfigType

This compound datatype holds the global COMMSTACK root configuration.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **const TDDLL_FrameDscType *(*pFrameLookupFct)(uint32)** | This member holds a pointer to a lookup function that returns a pointer to a frame description structure by passing a frame signature. |
| **const TDDLL_FrameDscType *const* pFrameList** | This member holds a pointer to an array of pointers to frame description structures. |
| **const TDDLL_CtrlDscType *pCtrlTypeList[TDDLL_NUM_CTRL_TYPES]** | This member is an array of pointers each entry pointing to an array of vendor-specific controller description structures of type **TDDLL_CtrlDscType**. |
| **uint16 nNumFrames** | This member holds the number of frames in array pointed to by member **pFrameList**. |
| **TDDLL_CtrlIDXType nNumCtrlType[TDDLL_NUM_CTRL_TYPES]** | This member is an array that holds the numbers of FlexRay controllers per vendor-specific FlexRay controller type. |

### 3.3.7 TDDLL_APIListType

This compound datatype holds the vendor-specific FlexRay controller API function implementations.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **TDDLL_ReturnType (*pCtrlInit) (TDDLL_CtrlIDXType)** | This member holds a pointer to a vendor-specific implementation of the API function **TDDLL_CtrlInit()**. |
| **TDDLL_ReturnType (*pCheckCtrl) (TDDLL_CtrlIDXType)** | This member holds a pointer to a vendor-specific implementation of the non API function **TDDLL_CheckCtrl()**. |
| **TDDLL_ReturnType (*pConfigBuffer) (TDDLL_CtrlIDXType, TDDLL_BufferIDXType, const TDDLL_FrameDcsType *)** | This member holds a pointer to a vendor-specific implementation of the API function **TDDLL_ConfigBuffer()**. |
| **TDDLL_CtrlStateType (*pDoCtrlTransition) (TDDLL_CtrlIDXType,** | This member holds a pointer to a vendor-specific implementation of the API function |

| | |
|---|---|
| `TDDLL_CtrlTransitionType)` | `TDDLL_DoCtrlTransition()`. |
| `TDDLL_CtrlStateType (*pGetCtrlState) (TDDLL_CtrlIDXType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_GetCtrlState()`. |
| `TDDLL_ReturnType (*pRxFrameByID) (TDDLL_CtrlIDXType, TDDLL_FrameDscRefIDXType, void *, TDDLL_LengthType, TDDLL_LengthType *)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_RxFrameByID()`. |
| `TDDLL_ReturnType (*pTxFrameByID) (TDDLL_CtrlIDXType, TDDLL_FrameDscRefIDXType const void *, TDDLL_LengthType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_TxFrameByID()`. |
| `TDDLL_ReturnType (*pGetTime) (TDDLL_CtrlIDXType, TDDLL_TickType *, TDDLL_CycleType *)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_GetTime()`. |
| `TDDLL_ReturnType (*pGetCycleLength) (TDDLL_CtrlIDXType, TDDLL_TickType *)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_GetCycleLength()`. |
| `void (*pInterruptResetStatus) (TDDLL_CtrlIDXType, TDDLL_InterruptSrcType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_InterruptResetStatus()`. |
| `void (*pInterruptEnable) (TDDLL_CtrlIDXType, TDDLL_InterruptSrcType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_InterruptEnable()`. |
| `void (*pInterruptDisable) (TDDLL_CtrlIDXType, TDDLL_InterruptSrcType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_InterruptDisable()`. |
| `TDDLL_InterruptSrcType (*pInterruptStatus) (TDDLL_CtrlIDXType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_InterruptStatus()`. |
| `TDDLL_ReturnType (*pGetPOCStatus) (TDDLL_CtrlIDXType, TDDLL_POCStatusType *)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_GetPOCStatus()`. |
| `TDDLL_ReturnType (*pCheckTxFrameByID) (TDDLL_CtrlIDXType, TDDLL_FrameDscRefIDXType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_CheckTxFrameByID()`. |
| `TDDLL_ReturnType (*pCheckTxFrameByID) (TDDLL_CtrlIDXType, TDDLL_FrameDscRefIDXType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_CheckTxFrameByIDPtr()`. |
| `TDDLL_ReturnType (*pAbortTxFrameByID) (TDDLL_CtrlIDXType, TDDLL_FrameDscRefIDXType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_AbortTxFrameByID()`. |

| | |
|---|---|
| `TDDLL_ReturnType (*pSetTimerAbs) (TDDLL_CtrlIDXType, TDDLL_TickType, uint8, uint8)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_SetTimerAbs()`. |
| `TDDLL_ReturnType (*pSetTimerRel) (TDDLL_CtrlIDXType, TDDLL_TickType)` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_SetTimerRel()`. |
| `TDDLL_ReturnType (*pRxFIFOFrame) (TDDLL_CtrlIDXType, TDDLL_FrameIDType *, TDDLL_ChannelIDXType *, TDDLL_CycleType *, void *, TDDLL_LengthType,TDDLL_LengthType * )` | This member holds a pointer to a vendor-specific implementation of the API function `TDDLL_RxFIFOFrame()`. |

### 3.3.8 TDDLL_CtrlMappingType

This compound datatype contains a mapping from a generic FlexRay controller to a vendor-specific FlexRay controller implementation.

At the COMMSTACK user API an index into an array based on this datatype is required to access a generic FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| `TDDLL_CtrlTypesType nCtrlType` | This member holds the vendor-specific FlexRay controller implementation. |
| `TDDLL_CtrlIDXType nDeviceListIndex` | This member holds the index within the vendor-specific FlexRay controller list of this generic FlexRay controller. |

**NOTE:** An entry of this type refers to a vendor-specific FlexRay controller list entry of a list of the type TDDLL_ERAY10_CtrlListType or TDDLL_MFR4200_CtrlListType or TDDLL_MFR4300_CtrlListType .

## 3.4 Basic Constants

Basic constants are constants that are used by the COMMSTACK but are usually defined outside the COMMSTACK.

If these datatypes do not exist, the COMMSTACK porter must define them for a proper COMMSTACK operation.

### 3.4.1 NULL

This constant should represent a pointer to an invalid memory location – in most cases 0. The COMMSTACK uses NULL for checking arguments for validity and optional (not used) configuration options.

## 3.5 Initialization & Configuration Services

Functions in this chapter are required for proper COMMSTACK initialization and configuration procedure before invoking any FlexRay operational functionality.

### 3.5.1 TDDLL_Init()

#### 3.5.1.1 Synopsis

```
TDDLL_ReturnType TDDLL_Init(void)
```

#### 3.5.1.2 Semantics

This function initializes the COMMSTACK and checks all connected FlexRay controllers for availability. Initially the COMMSTACK FlexRay controller states are set to the following states depending on the detected state of the FlexRay controllers:

| State | Condition |
|---|---|
| **TDDLL_S_CTRL_NOT_AVAILABLE** | There was an error while accessing the FlexRay controller. The device is not available for further the COMMSTACK operations. |
| **TDDLL_S_OFF** | The FlexRay controller was detected successfully and it is not synchronized to any FlexRay cluster. |
| **TDDLL_S_ON** | The FlexRay controller was detected successfully and it is synchronized to a FlexRay cluster. |

#### 3.5.1.3 Parameters

None.

#### 3.5.1.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The COMMSTACK initialization was successfully performed. |

### 3.5.2 TDDLL_GetNumCtrl()

#### 3.5.2.1 Synopsis

```
TDDLL_CtrlIDXType TDDLL_GetNumCtrl(void)
```

#### 3.5.2.2 Semantics

This function returns the number of FlexRay controllers the COMMSTACK has access to.

#### 3.5.2.3 Parameters

None.

#### 3.5.2.4 Return Values

| Value | Description |
|---|---|
| **value** | Number of FlexRay controller the COMMSTACK is configured to own. |

### 3.5.3 TDDLL_SetConfig()

#### 3.5.3.1 Synopsis

```
const TDDLL_ConfigType* TDDLL_SetConfig(
      const TDDLL_ConfigType* pNewConfig
      )
```

#### 3.5.3.2 Semantics

This function sets a new configuration to be active. The new configuration is set immediately after the function call without any delay.

#### 3.5.3.3 Parameters

| Value | Description |
|---|---|
| `pNewConfig` | Pointer to a COMMSTACK configuration root entry. |

#### 3.5.3.4 Return Values

| Value | Description |
|---|---|
| `value` | Address of the previously active configuration root entry. |

### 3.5.4 TDDLL_CtrlInit()

#### 3.5.4.1 Synopsis

```
TDDLL_ReturnType TDDLL_CtrlInit(
      TDDLL_CtrlIDXType nCtrlIDX
      )
```

#### 3.5.4.2 Semantics

This function initializes the registers and all buffers of a FlexRay controller passed as argument according to the current active COMMSTACK configuration data.

**NOTE:** This function must be called in COMMSTACK controller state TDDLL_S_CONFIG only. Otherwise TDDLL_E_BAD_CONFIG is returned.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_CtrlInit_Static() can be called instead of TDDLL_CtrlInit() which consumes less runtime, by saving one function call overhead.

#### 3.5.4.3 Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index this function should be performed on. |

#### 3.5.4.4 Return Values

| Value | Description |
|---|---|
| `TDDLL_E_OK` | The function was successfully finished. |
| `TDDLL_E_INVALID_IDX` | The given input parameter or some basic configuration is invalid. |
| `TDDLL_E_BAD_CONFIG` | Some precondition for initializing the FlexRay communication |

| | |
|---|---|
| | controller is not met. |
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |

### 3.5.5 TDDLL_ConfigBuffer()

#### 3.5.5.1 Synopsis

```
TDDLL_ReturnType TDDLL_ConfigBuffer(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_BufferIDXType nBufferID,
      const TDDLL_FrameDscType *pBufferDsc
      )
```

#### 3.5.5.2 Semantics

This function initializes a FlexRay buffer with a FlexRay buffer configuration of a dedicated FlexRay controller, with all parameters passed as arguments.

**NOTES:**

- This function must be called in COMMSTACK controller states TDDLL_S_CONFIG, TDDLL_S_OFF, TDDLL_S_ON or TDDLL_S_ONLINE only. Otherwise TDDLL_E_BAD_CONFIG is returned.
- The buffer configuration during COMMSTACK FlexRay controller states TDDLL_S_OFF, TDDLL_S_ON and TDDLL_S_ONLINE strongly depend on hardware facilities. If some reconfiguration option is not possible at the current COMMSTACK state because of some hardware restrictions TDDLL_E_BAD_CONFIG is returned.

**HINT:** If the argument nCtrlIDX and pBufferDsc are ensured to be constant, TDDLLConfigBuffer_Static() can be called instead of TDDLL_ConfigBuffer(), which consumes less runtime, by saving one function call overhead.

#### 3.5.5.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nBufferID** | The FlexRay controller buffer index this function should be performed on. |
| **pBufferDsc** | The FlexRay buffer configuration that should be used by this function. |

#### 3.5.5.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_BAD_CONFIG** | Some precondition for initializing the FlexRay communication buffer is not met. |
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay |

| | |
|---|---|
| | communication controller. |

### 3.5.6 TDDLL_ConfigAllBuffers()

#### 3.5.6.1 Synopsis

```
TDDLL_ReturnType TDDLL_ConfigAllBuffers(
        TDDLL_CtrlIDXType nCtrlIDX
        )
```

#### 3.5.6.2 Semantics

This function initializes all FlexRay buffers according to the current active COMMSTACK configuration.

**NOTES:**

- This function must be called in COMMSTACK controller states TDDLL_S_CONFIG, TDDLL_S_OFF, TDDLL_S_ON or TDDLL_S_ONLINE only. Otherwise TDDLL_E_BAD_CONFIG is returned.
- The buffer configuration during COMMSTACK FlexRay controller states TDDLL_S_OFF, TDDLL_S_ON and TDDLL_S_ONLINE strongly depend on hardware facilities. If some reconfiguration option is not possible at the current COMMSTACK state because of some hardware restrictions TDDLL_E_BAD_CONFIG is returned.

#### 3.5.6.3 Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index of the controller this function should be performed on. |

#### 3.5.6.4 Return Values

| Value | Description |
|---|---|
| `TDDLL_E_OK` | The function was successfully finished. |
| `TDDLL_E_INVALID_IDX` | The given input parameter or some basic configuration is invalid. |
| `TDDLL_E_BAD_CONFIG` | Some precondition for initializing the FlexRay communication buffer is not met. |
| `TDDLL_E_ACCESS` | Some device specific error occurred when accessing the FlexRay communication controller. |

## 3.6 Status Information Services

Functions within this chapter are required for reading and activating the states of the COMMSTACK FlexRay controller states and the FlexRay controller states as provided by the hardware.

### 3.6.1 TDDLL_IsSync()

#### 3.6.1.1 Synopsis

```
TDDLL_BooleanType TDDLL_IsSync(
      TDDLL_CtrlIDXType nCtrlIDX
      )
```

#### 3.6.1.2 Semantics

This function returns whether the given controller is synchronized to a cluster or not.

**NOTES:** In case of any error while accessing the device this function returns TDDLL_FALSE.

#### 3.6.1.3 Parameters

| Value | Description |
|---|---|
| nCtrlIDX | The FlexRay controller index of the controller this function should be performed on. |

#### 3.6.1.4 Return Values

| Value | Description |
|---|---|
| TDDLL_TRUE | The FlexRay controller is synchronized to a FlexRay cluster. |
| TDDLL_FALSE | The FlexRay controller is not synchronized to a FlexRay cluster. |

### 3.6.2 TDDLL_GetCtrlState()

#### 3.6.2.1 Synopsis

```
TDDLL_CtrlStateType TDDLL_GetCtrlState(
      TDDLL_CtrlIDXType nCtrlIDX
      )
```

#### 3.6.2.2 Semantics

This function returns the current active COMMSTACK FlexRay controller state.

**NOTE:** This function actually evaluates the current COMMSTACK state by requesting the FlexRay controller state. So this function is able to perform automatic state transitions as described in the state diagram.

**HINT:** the argument nCtrlIDX is ensured to be constant, TDDLL_GetCtrlState_Static() can be called instead of TDDLL_GetCtrlState() which consumes less runtime, by saving one function call overhead.

#### 3.6.2.3 Parameters

| Value | Description |
|---|---|
| nCtrlIDX | The FlexRay controller index of the controller this function should be performed on. |

#### 3.6.2.4 Return Values

| Value | Description |
|---|---|

| | |
|---|---|
| **state** | The actual COMMSTACK FlexRay controller state. See chapter 3.2.3 for possible values. |

### 3.6.3 TDDLL_DoCtrlTransition()

#### 3.6.3.1 Synopsis

```
TDDLL_CtrlStateType TDDLL_DoCtrlTransition(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_CtrlTransitionType nTransition
      )
```

#### 3.6.3.2 Semantics

This function triggers a COMMSTACK controller state machine transition and returns the current active COMMSTACK FlexRay controller state after triggering the transition.

**NOTES:**

- This function actually evaluates the current COMMSTACK state by requesting the FlexRay controller state. So this function is able to perform automatic state transitions as described in the state diagram before triggering the requested.
- If a requested transition is not allowed in the current COMMSTACK FlexRay controller state, the state machine doesn't leave its current state.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_DoCtrlTranistion_Static() can be called instead of TDDLL_DoCtrlTransition() which consumes less runtime, by saving one function call overhead.

#### 3.6.3.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nTransition** | The transition that should be triggered. See chapter 3.2.4 for possible values. |

#### 3.6.3.4 Return Values

| Value | Description |
|---|---|
| **state** | The actual COMMSTACK FlexRay controller state. See chapter 3.2.3 for possible values. |

### 3.6.4 TDDLL_GetPOCStatus()

#### 3.6.4.1 Synopsis

```
TDDLL_ReturnType TDDLL_GetPOCStatus(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_POCStatusType *pStatus
      )
```

#### 3.6.4.2 Semantics

This function returns the actual POC status as defined by the FlexRay specification of the requested FlexRay controller.

**NOTES:**

- It is always ensured that every element of the status structure will be written in case of positive return code. In case of a negative return code the status structure is not ensured to be consistent and must not be used therefore.
- FlexRay controllers that are not 100% compatible with the POC-status definition here, will map their internal states to this type with the closest meaning states. In this case not all states will be used.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_GetPOCStatus_Static() can be called instead of TDDLL_GetPOCStatus() which consumes less runtime, by saving one function call overhead.

#### 3.6.4.3 Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index of the controller this function should be performed on. |
| `pStatus` | Address the current POC status is written to. |

#### 3.6.4.4 Return Values

| Value | Description |
|---|---|
| `TDDLL_E_OK` | The function was successfully finished. |
| `TDDLL_E_INVALID_IDX` | The given input parameter or some basic configuration is invalid. |
| `TDDLL_E_ACCESS` | Some device specific error occurred when accessing the FlexRay communication controller. |

## 3.7 Transmission Services

The transmission services mentioned in this chapter are required to perform a transmission on the FlexRay network.

### 3.7.1 TDDLL_LookupTxFame()

#### 3.7.1.1 Synopsis

```
TDDLL_FrameDscRefIDXType TDDLL_LookupTxFrame(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameIDType nFrameID,
      TDDLL_ChannelIDXType nChannel,
      TDDLL_CycleType nRepCycle,
      TDDLL_CycleType nBaseCycle
      )
```

#### 3.7.1.2 Semantics

This function returns a unique ID of the reception frame description structure identified by the arguments given.

**NOTE:** The parameters given to this function must be constant expressions.

#### 3.7.1.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameID** | The FlexRay frame identifier of the frame. |
| **nChannel** | The FlexRay channel of the frame. |
| **nRepCycle** | The cycle repetition value of the frame. |
| **nBaseCycle** | The base cycle of the frame. |

#### 3.7.1.4 Return Values

| Value | Description |
|---|---|
| **value** | Unique identifier of the frame description structure of the transmission frame identified by the given arguments. |

### 3.7.2 TDDLL_TX_FRAME_TRIGGERING ()

#### 3.7.2.1 Synopsis

```
TDDLL_FrameDscRefIDXType TDDLL_TX_FRAME_TRIGGERING (
      FrameTriggeringName
      )
```

#### 3.7.2.2 Semantics

This function returns a unique ID of the reception frame description structure identified by frame triggering name given as argument.

**NOTE:** This is a function-like macro the argument has to be a string that maps exactly a frame-triggering name as specified in the communication model.

#### 3.7.2.3 Parameters

| Value | Description |
|---|---|
| **FrameTriggeringName** | Name of the frame triggering to transmit. |

#### 3.7.2.4 Return Values

| Value | Description |
|---|---|
| **Value** | Unique identifier of the frame description structure of the transmission frame identified by the given arguments. |

### 3.7.3 TDDLL_TxFrameByID()

#### 3.7.3.1 Synopsis

```
TDDLL_ReturnType TDDLL_TxFrameByID(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX,
      const void *pData,
      TDDLL_LengthType nLength
      )
```

#### 3.7.3.2 Semantics

This function transmits the payload data given within a particular FlexRay frame on a distinct FlexRay controller. All parameters are passed as arguments. A transmission is carried out in FlexRay controller state **TDDLL_E_ONLINE** only.

**NOTES:**

- There is always an even number of bytes transmitted on the FlexRay bus. If the payload length nLength is odd, one byte extends the payload transmitted on the FlexRay bus. This byte stuffed is set to TDDLL_PAYLOAD_PADDING_PATTERN.
- If a frame belongs to the static FlexRay segment, the payload length must be smaller than or equal to the static payload length of the cluster configuration. In case the payload length passed at the API is smaller than the static payload length of the cluster, the frame is filled with TDDLL_PAYLOAD_PADDING_PATTERN up to the static payload length in case TDDLL_STATIC_PAYLOAD_PADDING_PATTERN_SWITCH is defined. If TDDLL_STATIC_PAYLOAD_PADDING_PATTERN_SWITCH is not defined, the frame is filled with random data up to the static payload length. If a frame belongs to the dynamic FlexRay segment, the payload length must not exceed the maximum payload length this distinct frame was configured to.
- The data bytes are transmitted on the bus in the same order, as the bytes are located in memory. The byte at address pData is transmitted first on the bus, the byte at pData + 1 second, and so on.
- There is no alignment restriction required for the data location pointed to by pData.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_TxFrameByID_Static() can be called instead of TDDLL_TxFrameByID() which consumes less runtime, by saving one function call overhead.

#### 3.7.3.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | Unique ID that identifies the frame to be transmitted. |
| **PData** | Pointer to the payload data to be transmitted within the frame. |
| **nLength** | Length of the payload data in units of bytes to be transmitted.<br>In the static segment the number of bytes to be transmitted must match the FlexRay static segment payload length of the cluster.<br>In the dynamic segment the payload length must not exceed the maximum payload length the FlexRay controller is able to |

| | |
|---|---|
| | transmit with the buffer assigned to this frame.<br>Otherwise **TDDLL_E_INVALID_IDX** will be returned on both cases. |

#### 3.7.3.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |
| **TDDLL_E_OVERFLOW** | No resource for frame transmission (buffer) within the FlexRay controller is available. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |

### 3.7.4 TDDLL_CheckTxFrameByID()

#### 3.7.4.1 Synopsis

```
TDDLL_ReturnType TDDLL_CheckTxFrameByID(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX
      )
```

#### 3.7.4.2 Semantics

This function checks whether a previously performed transmission request (see chapter 3.7.2) has been already executed by the FlexRay controller or not. This operation is carried out in FlexRay controller state **TDDLL_E_ONLINE** only.

**NOTE:** In case there was no frame requested for transmission before, **TDDLL_E_OK** is returned.

**HINTS:**

- In case of buffer pooling the buffer reserved for a frame transmission is released with this function call if the frame was already delivered to the network.
- If the argument nCtrlIDX is ensured to be constant, TDDLL_CheckTxFrameByID_Static() can be called instead of TDDLL_CheckTxFrameByID() which consumes less runtime, by saving one function call overhead.

#### 3.7.4.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | Unique ID that identifies the frame to be checked for transmission. |

#### 3.7.4.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The frame has been transmitted. |
| **TDDLL_E_TX_PENDING** | The frame is still pending for transmission. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |
| **TDDLL_E_ACCESS** | Accessing the FlexRay CC failed. |

### 3.7.5 TDDLL_AbortTxFrameByID()

#### 3.7.5.1 Synopsis

```
TDDLL_ReturnType TDDLL_AbortTxFrameByID(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX
      )
```

#### 3.7.5.2 Semantics

This function aborts a previously performed transmission request (see chapter 3.7.2). This operation is carried out in FlexRay controller state **TDDLL_E_ONLINE** only.

**NOTES:**

- In case the particular FlexRay controller used doesn't support to abort a transmission request, **TDDLL_E_ACCESS** is returned.
- In case there was no frame pending for transmission, **TDDLL_E_OK** is returned.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_AbortTxFrameByID_Static() can be called instead of TDDLL_AbortTxFrameByID() which consumes less runtime, by saving one function call overhead.

#### 3.7.5.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | Unique ID that identifies the frame the transmission request should be aborted of. |

#### 3.7.5.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The frame has been transmitted. |
| **TDDLL_E_ACCESS** | Error accessing the FlexRay controller (operation not permitted). |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |

### 3.7.6 TDDLL_FreeBufferPool()

#### 3.7.6.1 Synopsis

```
TDDLL_ReturnType TDDLL_FreeBufferPool()
```

#### 3.7.6.2 Semantics

This function releases buffers that were dynamically assigned for transmission but are not in use any more and puts them back into the pool of available buffers. This function must be called periodically if buffer pooling was selected during buffer assignment (alternatively **`TDDLL_CheckTxFrameByID()`**, see chapter 3.7.4 can be called).

**NOTE:** This function should be called periodically in case buffer pooling is used to ensure buffers are released from their temporary reservation (Alternatively TDDLL_CheckTxFrameByID() can be used for this job).

#### 3.7.6.3 Parameters

None.

#### 3.7.6.4 Return Values

| Value | Description |
|---|---|
| **`Value`** | Number of buffers that were freed to be available again. |

## 3.8 Reception Services

The reception services mentioned in this chapter are required to perform a receive operations on the FlexRay network.

### 3.8.1 TDDLL_LookupRxFrame()

#### 3.8.1.1 Synopsis

```
TDDLL_FrameDscRefIDXType TDDLL_LookupRxFrame(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameIDType nFrameID,
      TDDLL_ChannelIDXType nChannel,
      TDDLL_CycleType nRepCycle,
      TDDLL_CycleType nBaseCycle
      )
```

#### 3.8.1.2 Semantics

This function returns a unique ID of the reception frame description structure identified by the arguments given.

**NOTE:** The parameters given to this function must be constant expressions.

#### 3.8.1.3 Parameters

| Value | Description |
|---|---|

| | |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameID** | The FlexRay frame identifier of the frame. |
| **nChannel** | The FlexRay channel of the frame. |
| **nRepCycle** | The cycle repetition value of the frame. |
| **nBaseCycle** | The base cycle of the frame. |

#### 3.8.1.4 Return Values

| Value | Description |
|---|---|
| **Value** | Unique identifier of the frame description structure of the reception frame identified by the given arguments. |

### 3.8.2 TDDLL_RX_FRAME_TRIGGERING ()

#### 3.8.2.1 Synopsis

```
TDDLL_FrameDscRefIDXType TDDLL_RX_FRAME_TRIGGERING (
        FrameTriggeringName
        )
```

#### 3.8.2.2 Semantics

This function returns a unique ID of the reception frame description structure identified by frame triggering name given as argument.

**NOTE:** This is a function-like macro. The argument has to be a string that maps exactly a frame-triggering name as specified in the communication model.

#### 3.8.2.3 Parameters

| Value | Description |
|---|---|
| **FrameTriggeringName** | Name of the frame triggering to receive. |

#### 3.8.2.4 Return Values

| Value | Description |
|---|---|
| **value** | Unique identifier of the frame description structure of the transmission frame identified by the given arguments. |

### 3.8.3 TDDLL_RxFrameByID()

#### 3.8.3.1 Synopsis

```
TDDLL_ReturnType TDDLL_RxFrameByID(
        TDDLL_CtrlIDXType nCtrlIDX,
        TDDLL_FrameDscRefIDXType nFrameDscRefIDX,
        const void *pData,
        TDDLL_LengthType nRxBufferLength,
        TDDLL_LengthType *pnReceivedLength
```

```
)
```

#### 3.8.3.2 Semantics

This function receives a particular FlexRay frame on a distinct FlexRay controller. A reception is carried out in FlexRay controller state **TDDLL_E_ONLINE** only.

**NOTES:**

- Only syntactically valid frames containing data are received with this API function.
- The data bytes are stored in memory in the same order the bytes were received on the FlexRay bus. The byte at address pData was received first on the bus, the byte at pData + 1 second, and so on.
- There is no alignment restriction required for the data location pointed to by pData.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_RxFrameByID_Static() can be called instead of TDDLL_RxFrameByID() which consumes less runtime, by saving one function call overhead.

#### 3.8.3.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | Unique ID that identifies the frame to be received. |
| **pData** | Pointer the payload data received is written to. |
| **nRxBufferLength** | Length of the buffer **pData** points to in units of bytes. There is never more payload written to **pData** than specified by nRxBufferLength. |
| **pnReceivedLength** | Length of the payload actually received in units of bytes. If there are more bytes received than the buffer is able to store (nRxBufferLength), the payload data is truncated and the truncated part is lost. |

#### 3.8.3.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |
| **TDDLL_E_UNDERFLOW** | No valid frame was received. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |

### 3.8.4 TDDLL_RxFIFOFrameByID()

#### 3.8.4.1 Synopsis

```
TDDLL_ReturnType TDDLL_RxFIFOFrameByID(
      TDDLL_CtrlIDXType nCtrlIDX,
```

```
TDDLL_FrameIDType *pnFrameID,
TDDLL_ChannelIDXType *pnChannel,
TDDLL_CycleType *pnCycle,
void *pData,
TDDLL_LengthType * nRxBufferLength,
TDDLL_LengthType * pnReceivedLength
)
```

#### 3.8.4.2 Semantics

This function fetches a frame from the FlexRay controller receive FIFO. A reception is carried out in FlexRay controller state **TDDLL_E_ONLINE** only.

**NOTES:**

- Only syntactically valid frames containing data are received with this API function.
- The data bytes are stored in memory in the same order the bytes were received on the FlexRay bus.The byte at address pData was received first on the bus, the byte at pData + 1 second, and so on.
- There is no alignment restriction required for the data location pointed to by pData.

**HINT:** If the arguments nCtrlIDX and pFrameDsc are ensured to be constant, TDDLL_RxFIFOFrame_Static() can be called instead of TDDLL_RxFIFOFrame() which consumes less runtime, by saving one function call overhead.

#### 3.8.4.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **pnFrameID** | Pointer to a storage location the frame identifier of the received frame is written to. |
| **pnChannel** | Pointer to a storage location the channel the frame was received on is written to. |
| **pnCycle** | Pointer to a storage location the cycle value the frame was received in is written to. |
| **pData** | Pointer the payload data received is written to. |
| **nRxBufferLength** | Length of the buffer **pData** points to in units of bytes. There is never more payload written to **pData** than specified by nRxBufferLength. |
| **pnReceivedLength** | Length of the payload actually received in units of bytes. If there are more bytes received than the buffer is able to store (nRxBufferLength), the payload data is truncated and the truncated part lost. |

#### 3.8.4.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |

| | |
|---|---|
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |
| **TDDLL_E_UNDERFLOW** | No valid frame was received by the FIFO. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |

## 3.9 Queue Services

The queuing services can be used to decouple the strict synchronous access to the FlexRay controllers from the applications by introducing software queues. The access functions to the FlexRay controller via these software queues are described within this chapter.

### 3.9.1 TDDLL_FlushTxQueue()

#### 3.9.1.1 Synopsis

```
TDDLL_ReturnType TDDLL_TxFrameByIDQueued(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX
      )
```

#### 3.9.1.2 Semantics

This function takes a frame out of a software queue and transmits it on a FlexRay controller. The frame is put into the software queue with the function **TDDLL_TxFrameByIDQueued()** (see chapter 3.9.3). This operation is carried out in FlexRay controller state **TDDLL_E_ONLINE** only.

#### 3.9.1.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | unique ID that identifies the frame to be transmitted. |

#### 3.9.1.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |
| **TDDLL_E_UNDERFLOW** | The queue is empty. |
| **TDDLL_E_OVERFLOW** | The communication controller transmission resource (buffer) is busy. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |

### 3.9.2 TDDLL_FillRxQueue()

#### 3.9.2.1 Synopsis

```
TDDLL_ReturnType TDDLL_FillRxQueue (
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX
      )
```

#### 3.9.2.2 Semantics

This function received a frame from a FlexRay controller and puts it into a software queue. The frame is fetched from the software queue with the function **`TDDLL_RxFrameByIDQueued()`** (see chapter 3.9.4). This operation is carried out in FlexRay controller state **`TDDLL_E_ONLINE`** only.

#### 3.9.2.3 Parameters

| Value | Description |
|---|---|
| **`nCtrlIDX`** | The FlexRay controller index of the controller this function should be performed on. |
| **`nFrameDscRefIDX`** | Unique ID that identifies the frame to be received. |

#### 3.9.2.4 Return Values

| Value | Description |
|---|---|
| **`TDDLL_E_OK`** | The function was successfully finished. |
| **`TDDLL_E_INVALID_IDX`** | The given input parameter or some basic configuration is invalid. |
| **`TDDLL_E_ACCESS`** | Some device specific error occurred when accessing the FlexRay communication controller. |
| **`TDDLL_E_UNDERFLOW`** | The FlexRay controller received no frame. |
| **`TDDLL_E_OVERFLOW`** | The software queue is already filled. |
| **`TDDLL_E_OFFLINE`** | The COMMSTACK FlexRay controller state is not **`TDDLL_E_ONLINE`**. |

### 3.9.3 TDDLL_TxFrameByIDQueued()

#### 3.9.3.1 Synopsis

```
TDDLL_ReturnType TDDLL_TxFrameByIDQueued(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX,
      const void *pData,
      TDDLL_LengthType nLength
      )
```

#### 3.9.3.2 Semantics

This function puts payload data into a distinct frames software queue for transmission. The frame is transmitted out of the software queue with the function **`TDDLL_FlushTxQueue()`** (see chapter 3.9.1). This operation is carried out in FlexRay controller state **`TDDLL_E_ONLINE`** only.

**NOTES:**

- There is always an even number of bytes transmitted on the FlexRay bus. If the payload length nLength is odd, one byte extends the payload transmitted on the FlexRay bus. This byte stuffed is set to TDDLL_PAYLOAD_PADDING_PATTERN.
- If a frame belongs to the static FlexRay segment, the payload length must be smaller than or equal to the static payload length of the cluster configuration. In case the payload length passed at the API is smaller than the static payload length of the cluster, the frame is filled with TDDLL_PAYLOAD_PADDING_PATTERN up to the static payload length in case TDDLL_STATIC_PAYLOAD_PADDING_PATTERN_SWITCH is defined. If TDDLL_STATIC_PAYLOAD_PADDING_PATTERN_SWITCH is not defined, the frame is filled with random data up to the static payload length. If a frame belongs to the dynamic FlexRay segment, the payload length must not exceed the maximum payload length this distinct frame was configured to.
- The data bytes are transmitted on the bus in the same order, as the bytes are located in memory. The byte at address pData is transmitted first on the bus, the byte at pData + 1 second, and so on.
- There is no alignment restriction required for the data location pointed to by pData.

#### 3.9.3.3 Parameters

| Value | Description |
|---|---|
| **`nCtrlIDX`** | The FlexRay controller index of the controller this function should be performed on. |
| **`nFrameDscRefIDX`** | Unique ID that identifies the frame to be transmitted. |
| **`pData`** | Pointer to the payload data to be transmitted within the frame. |
| **`nLength`** | Length of the payload data in units of bytes to be transmitted. |

#### 3.9.3.4 Return Values

| Value | Description |
|---|---|
| **`TDDLL_E_OK`** | The function was successfully finished. |
| **`TDDLL_E_INVALID_IDX`** | The given input parameter or some basic configuration is invalid. |
| **`TDDLL_E_OVERFLOW`** | The queue is full. |
| **`TDDLL_E_OFFLINE`** | The COMMSTACK FlexRay controller state is not **`TDDLL_E_ONLINE`**. |

### 3.9.4 TDDLL_RxFrameByIDQueued()

#### 3.9.4.1 Synopsis

```
TDDLL_ReturnType TDDLL_RxFrameByIDQueued(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX,
      const void *pData,
```

```
        TDDLL_LengthType nRxBufferLength,
        TDDLL_LengthType *pnReceivedLength
        )
```

#### 3.9.4.2 Semantics

This function receives a particular FlexRay frame out of a software queue. A reception is carried out in FlexRay controller state **TDDLL_E_ONLINE** only. The frame is put into the software queue by **TDDLL_FillRxQueue()**.

**NOTES:**

- Only syntactically valid frames containing data are received with this API function.
- The data bytes are stored in memory in the same order the bytes were received on the FlexRay bus.The byte at address pData was received first on the bus, the byte at pData + 1 second, and so on.
- There is no alignment restriction required for the data location pointed to by pData.

#### 3.9.4.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | Unique ID that identifies the frame to be received. |
| **PData** | Pointer the payload data received is written to. |
| **nRxBufferLength** | Length of the buffer **pData** points to in units of bytes. There is never more payload written to **pData** than specified by nRxBufferLength. |
| **pnReceivedLength** | Length of the payload actually received in units of bytes. If there are more bytes received than the buffer is able to store (nRxBufferLength), the payload data is truncated and the truncated part is lost. |

#### 3.9.4.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_UNDERFLOW** | The software queue is empty. |
| **TDDLL_E_OFFLINE** | The COMMSTACK FlexRay controller state is not **TDDLL_E_ONLINE**. |

### 3.9.5 TDDLL_EmptyQueue()

#### 3.9.5.1 Synopsis

```
TDDLL_ReturnType TDDLL_EmptyQueue (
        TDDLL_CtrlIDXType nCtrlIDX,
        TDDLL_FrameDscRefIDXType nFrameDscRefIDX
        )
```

#### 3.9.5.2 Semantics

This function discards all entries from a software queue.

#### 3.9.5.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nFrameDscRefIDX** | Unique ID that identifies the frame, whose queue entries should be discarded. |

#### 3.9.5.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |

## 3.10Time Services

Time services are used for reading time information out of the FlexRay controller configuration and actual state.

### 3.10.1TDDLL_GetTime()

#### 3.10.1.1Synopsis

```
TDDLL_ReturnType TDDLL_GetTime (
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_TickType *pClusterTime,
      TDDLL_CycleType *pCommCycle
      )
```

#### 3.10.1.2Semantics

This function reads the actual time of the cluster a dedicated FlexRay controller is synchronized to.

**NOTE:** If some value different from TDDLL_E_OK is retuned, the output parameters pClusterTime and pCommCycle are not written but left in their previous state.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_GetTime_Static() can be called instead of TDDLL_GetTime() which consumes less runtime, by saving one function call overhead.

#### 3.10.1.3Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **pClusterTime** | Pointer to a storage location the current communication cycle offset in units of macroticks is written to. |
| **pCommCycle** | Pointer to a storage location the current communication cycle value is written to. |

#### 3.10.1.4 Return Values

| Value | Description |
|---|---|
| `TDDLL_E_OK` | The function was successfully finished. |
| `TDDLL_E_INVALID_IDX` | The given input parameter or some basic configuration is invalid. |
| `TDDLL_E_NOT_SYNC` | The FlexRay controller is not synchronized to the cluster. |

### 3.10.2 TDDLL_GetCycleLength()

#### 3.10.2.1 Synopsis

```
TDDLL_ReturnType TDDLL_GetCycleLength (
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_TickType *pCommCycleLength
      )
```

#### 3.10.2.2 Semantics

This function returns the actual communication cycle length a FlexRay controller has configured.

**NOTE:** If some value different from TDDLL_E_OK is retuned, the output parameter pCommCycleLength is not written but left in its previous state.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_GetCycleLength_Static() can be called instead of TDDLL_GetCycleLength() which consumes less runtime, by saving one function call overhead.

#### 3.10.2.3 Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index of the controller this function should be performed on. |
| `pCommCycleLength` | Pointer to a storage location the current communication cycle length in units of macroticks is written to. |

#### 3.10.2.4 Return Values

| Value | Description |
|---|---|
| `TDDLL_E_OK` | The function was successfully finished. |
| `TDDLL_E_INVALID_IDX` | The given input parameter or some basic configuration is invalid. |
| `TDDLL_E_ACCESS` | Some device specific error occurred when accessing the FlexRay communication controller. |

### 3.10.3 TDDLL_NSToMacroticks()

#### 3.10.3.1 Synopsis

```
TDDLL_TickType TDDLL_NSToMacroticks (
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_TimeType nTime
      )
```

#### 3.10.3.2 Semantics

This function calculates time in units of nanoseconds into time in units of macroticks (as configured for a dedicated FlexRay controller).

**NOTE:** Please be aware that the value range of both units – macroticks and nanoseconds are limited by the ranges of their datatypes. This function doesn't check for occurring calculation overflows!

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_NSToMacroticks_Static() can be called instead of TDDLL_NSToMacroticks() which consumes less runtime, by saving one function call overhead.

#### 3.10.3.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nTime** | Time in units of nanoseconds. |

#### 3.10.3.4 Return Values

| Value | Description |
|---|---|
| **Value** | Time in units of macroticks. |

### 3.10.4 TDDLL_MacroticksToNS()

#### 3.10.4.1 Synopsis

```
TDDLL_TimeType TDDLL_MacroticksToNS (
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_TickType nTick
      )
```

#### 3.10.4.2 Semantics

This function calculates time in units of macroticks into time in units of nanoseconds (as configured for a dedicated FlexRay controller).

**NOTE:** Please be aware that the value range of both units – macroticks and nanoseconds are limited by the ranges of their datatypes. This function doesn't check for occurring calculation overflows!

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_MacroticksToNS_Static() can be called instead of TDDLL_MacroticksToNS() which consumes less runtime, by saving one function call overhead.

#### 3.10.4.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nTick** | Time in units of macroticks. |

#### 3.10.4.4 Return Values

| Value | Description |
|---|---|
| `value` | Time in units of nanoseconds. |

## 3.11 Timer & Interrupt Services

The timer and interrupts services are required for configuring and administrating the FlexRay controller's interrupt resources.

### 3.11.1 TDDLL_InterruptEnable()

#### 3.11.1.1 Synopsis

```
void TDDLL_InterruptEnable(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_InterruptSrcType nIntSrc
      )
```

#### 3.11.1.2 Semantics

This function enables the interrupts selected by the sources passed as arguments.

**NOTES:**

- The global interrupt source TDDLL_BM_GLOBAL_INTERRUPT is a global interrupt enable/disable switch, that is in serial connection to all the other interrupt lines (which are parallel to each other). To work with interrupts the global interrupt source must be enabled.
- In case an interrupt source not supported on a particular FlexRay controller is selected (e.g. TDDLL_BM_RELATIVE_TIMER_INTERRUPT on MFR4200) no operation is carried out for this interrupt source.

**HINTS:**

- The interrupt status of each interrupt source is independent from the enable/disable state of the corresponding interrupt source. The status might return an pending interrupt even if the interrupt source is disabled. It is recommended to always clear the interrupt status before enabling an interrupt source (to start operating in a consistent interrupt environment).
- If the argument nCtrlIDX is ensured to be constant, TDDLL_InterruptEnable_Static() can be called instead of TDDLL_InterruptEnable() which consumes less runtime, by saving one function call overhead.

#### 3.11.1.3 Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index of the controller this function should be performed on. |
| `nIntSrc` | Interrupt sources that should be enabled. |

#### 3.11.1.4 Return Values

None.

### 3.11.2TDDLL_InterruptDisable()

#### 3.11.2.1Synopsis

```
void TDDLL_InterruptDisable(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_InterruptSrcType nIntSrc
      )
```

#### 3.11.2.2Semantics

This function disables the interrupts selected by the sources passed as arguments.

**NOTES:**

- The global interrupt source TDDLL_BM_GLOBAL_INTERRUPT is a global interrupt enable/disable switch, that is in serial connection to all the other interrupt lines (which are parallel to each other).
- In case an interrupt source not supported on a particular FlexRay controller is selected (e.g. TDDLL_BM_RELATIVE_TIMER_INTERRUPT on MFR4200) no operation is carried out for this interrupt source.

**HINTS:**

- The interrupt status of each interrupt source is independent from the enable/disable state of the corresponding interrupt source. The status might return an pending interrupt even if the interrupt source is disabled.
- If the argument nCtrlIDX is ensured to be constant, TDDLL_InterruptDisable_Static() can be called instead of TDDLL_InterruptDisable() which consumes less runtime, by saving one function call overhead.

#### 3.11.2.3Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index of the controller this function should be performed on. |
| `nIntSrc` | Interrupt sources that should be disabled. |

#### 3.11.2.4Return Values

None.

### 3.11.3TDDLL_InterruptStatus()

#### 3.11.3.1Synopsis

```
TDDLL_InterruptSrcType TDDLL_InterruptStatus(
      TDDLL_CtrlIDXType nCtrlIDX
      )
```

#### 3.11.3.2Semantics

This function returns the status of all supported interrupt sources.

**NOTES:**

- The global interrupt source TDDLL_BM_GLOBAL_INTERRUPT is implemented as an global interrupt enable/disable switch only and hence doesn't own an interrupt status. A 0-bit will be returned always at the global interrupt status flag.
- In case an interrupt source not supported on a particular FlexRay controller is selected (e.g. TDDLL_BM_RELATIVE_TIMER_INTERRUPT on MFR4200) no operation is carried out for this interrupt source.

**HINTS:**

- The interrupt status of each interrupt source is independent from the enable/disable state of the corresponding interrupt source. The status might return an pending interrupt even if the interrupt source is disabled.
- If the argument nCtrlIDX is ensured to be constant, TDDLL_InterruptStatus_Static() can be called instead of TDDLL_InterruptStatus() which consumes less runtime, by saving one function call overhead.

#### 3.11.3.3Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |

#### 3.11.3.4Return Values

| Value | Description |
|---|---|
| **Value** | Status of all supported interrupts as a bitmask. A bit "1" indicates that an interrupt is pending, a bit "0" indicates that no interrupt occurred. |

### 3.11.4TDDLL_InterruptResetStatus()

#### 3.11.4.1Synopsis

```
void TDDLL_InterruptResetStatus(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_InterruptSrcType nIntSrc
      )
```

#### 3.11.4.2Semantics

This function resets the pending status of the interrupts selected by the sources passed as arguments.

**NOTES:**

- The global interrupt source TDDLL_BM_GLOBAL_INTERRUPT is implemented as an global interrupt enable/disable switch only and hence doesn't own an interrupt status.
- In case an interrupt source not supported on a particular FlexRay controller is selected (e.g. TDDLL_BM_RELATIVE_TIMER_INTERRUPT on MFR4200) no operation is carried out for this interrupt source.

**HINTS:**

- The interrupt sources are selected by a bitmask that can be combined by a bitwise OR to operate on several interrupt sources at a single function call.

- If the argument nCtrlIDX is ensured to be constant, TDDLL_InterruptResetStatus_Static() can be called instead of TDDLL_InterruptResetStatus() which consumes less runtime, by saving one function call overhead.

#### 3.11.4.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nIntSrc** | Pending interrupts stati that should be reset. |

#### 3.11.4.4 Return Values

None.

### 3.11.5 TDDLL_SetTimerAbs()

#### 3.11.5.1 Synopsis

```
TDDLL_ReturnType TDDLL_SetTimerAbs(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_TickType nMacroTick,
      TDDLL_CycleType nRepCycle,
      TDDLL_CycleType nBaseCycle
      )
```

#### 3.11.5.2 Semantics

This function configures the absolute timer of a FlexRay controller.

**NOTE:** The absolute timer always works in continuous mode.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_SetTimerAbs_Static() can be called instead of TDDLLSetTimerAbs() which consumes less runtime, by saving one function call overhead.

#### 3.11.5.3 Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nMacroTick** | Communication cycle offset the timer should elapse in units of macroticks. |
| **nRepCycle** | Communication cycle repetition value the timer should elapse. |
| **nBaseCycle** | Base communication cycle the absolute timer should elapse. |

#### 3.11.5.4 Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |

| | |
|---|---|
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |

## 3.11.6TDDLL_SetTimerRel()

### 3.11.6.1Synopsis

```
TDDLL_ReturnType TDDLL_SetTimerRel(
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_TickType nMacroTick
      )
```

### 3.11.6.2Semantics

This function configures the relative timer of a FlexRay controller.

**NOTES:**

- In case the particular FlexRay controller doesn't support a relative timer, TDDLL_E_ACCESS is returned.
- The relative timer always works in continuous mode.

**HINT:** If the argument nCtrlIDX is ensured to be constant, TDDLL_SetTimerRel_Static() can be called instead of TDDLL_SetTimerRel() which consumes less runtime, by saving one function call overhead.

### 3.11.6.3Parameters

| Value | Description |
|---|---|
| **nCtrlIDX** | The FlexRay controller index of the controller this function should be performed on. |
| **nMacroTick** | Communication cycle offset the timer should elapse in units of macroticks. |

### 3.11.6.4Return Values

| Value | Description |
|---|---|
| **TDDLL_E_OK** | The function was successfully finished. |
| **TDDLL_E_INVALID_IDX** | The given input parameter or some basic configuration is invalid. |
| **TDDLL_E_ACCESS** | Some device specific error occurred when accessing the FlexRay communication controller. |

# 4 CC Type Specific Extensions

This chapter describes FlexRay CC implementation specific extensions that are supported by special purpose API functions that are available in a generic style. Additionally CC implementation specific datatypes are described in this chapter.

## 4.1 Freescale MFR4200

### 4.1.1 Datatypes

#### 4.1.1.1 TDDLL_MFR4200_CtrlStateInfoType

This compound datatype holds the FlexRay controller COMMSTACK state administrative information required per Freescale MFR4200 FlexRay controller.

This datatype is required for static volatile memory allocation within the COMMSTACK configuration and COMMSTACK internal functionality but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **TDDLL_CtrlStateType nCtrlState** | This member holds the COMMSTACK FlexRay controller state. |
| **uint16 sISRsave** | This member saves the enable/disable state for all interrupt sources when emulating the global interrupt enable/disable switch. |
| **uint8 sGlobalInterruptEnable** | This member saves the state of the global interrupt enable/disable switch. |

#### 4.1.1.2 TDDLL_MFR4200_CtrlListType

This compound datatype holds the FlexRay controller hardware mapping information for a Freescale MFR4200 FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **FCAL_MFR4200_CtrlHandleType nCtrlHandle** | This member holds the hardware access port information to the Freescale MFR4200 FlexRay controller. |
| **void (*PctrlReset)(void)** | This member holds pointer to a routine that performs a hardware platform dependent hard-reset on the Freescale MFR4200 FlexRay controller. If there is no way for performing the hardware reset by software this field must contain NULL. |

#### 4.1.1.3 TDDLL_CHI_MFR4200_CommandType

This compound datatype holds a register configuration value for a Freescale MFR4200 FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint16 nCHIOffset** | This member holds a FlexRay controller register address offset. The offset is calculated from the base CHI address of a Freescale MFR4200 FlexRay controller. |
| **uint16 nCHIValue** | This member holds the value to be written in the Freescale MFR4200 FlexRay controller register identified by the member **nCHIOffset**. |

### 4.1.2 Extensions

There are no extensions specific to MFR4200 COMMSTACK implementation.

## 4.2 Bosch ERAY

### 4.2.1 Datatypes

#### 4.2.1.1 TDDLL_ERAY10_CtrlStateInfoType

This compound datatype holds the FlexRay controller COMMSTACK state administrative information required per Bosch ERAY-based FlexRay controller.

This datatype is required for static volatile memory allocation within the COMMSTACK configuration and COMMSTACK internal functionality but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **TDDLL_CtrlStateType nCtrlState** | This member holds the COMMSTACK FlexRay controller state. |
| **uint16 sISRsave** | This member saves the enable/disable state for all interrupt sources when emulating the global interrupt enable/disable switch. |
| **uint8 sGlobalInterruptEnable** | This member saves the state of the global interrupt enable/disable switch. |

#### 4.2.1.2 TDDLL_ERAY10_CtrlListType

This compound datatype holds the FlexRay controller hardware mapping information for a Bosch ERAY based FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **FCAL_ERAY10_CtrlHandleType nCtrlHandle** | This member holds the hardware access port information to the Bosch ERAY-based FlexRay controller. |
| **void (*PctrlReset)(void)** | This member holds pointer to a routine that performs a hardware platform dependent hard-reset on the Bosch ERAY-based FlexRay controller. If there is no way for performing the hardware reset by software this field must contain NULL. |

#### 4.2.1.3 TDDLL_CHI_ERAY10_CommandType

This compound datatype holds a register configuration value for a Bosch ERAY-based FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint16 nCHIOffset** | This member holds a FlexRay controller register address offset. The offset is calculated from the base CHI address of a Bosch ERAY-based FlexRay controller. |
| **uint32 nCHIValue** | This member holds the value to be written in the Bosch ERAY-based FlexRay controller register identified by the member **nCHIOffset**. |

#### 4.2.1.4 TDDLL_ERAY10_FrameDsc_ExtendedConfigType

This compound datatype contains extended information that is required to configure the buffer of an ERAY-based FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint32 nWRHS3** | This member holds the WRHS3 register value of an ERAY-based FlexRay controller buffer. |

### 4.2.2 Extensions

There are no extensions specific to ERAY COMMSTACK implementation.

## 4.3 Freescale MFR4300

### 4.3.1 Datatypes

#### 4.3.1.1 TDDLL_MFR4300_CtrlStateInfoType

This compound datatype holds the FlexRay controller COMMSTACK state administrative information required per Freescale MFR4300 FlexRay controller.

This datatype is required for static volatile memory allocation within the COMMSTACK configuration and COMMSTACK internal functionality but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **TDDLL_CtrlStateType nCtrlState** | This member holds the COMMSTACK FlexRay controller state. |
| **uint16 sISRsave** | This member saves the enable/disable state for all interrupt sources when emulating the global interrupt enable/disable switch. |
| **uint8 sGlobalInterruptEnable** | This member saves the state of the global interrupt enable/disable switch. |

#### 4.3.1.2 TDDLL_MFR4300_CtrlListType

This compound datatype holds the FlexRay controller hardware mapping information for a Freescale MFR4300 FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **FCAL_MFR4300_CtrlHandleType nCtrlHandle** | This member holds the hardware access port information to the Freescale MFR4300 FlexRay controller (access to the controller's internal registers). |
| **FCAL_MFR4300_CtrlHandleType nCtrlHandle_DMA** | This member holds the hardware access port information to the Freescale MFR4300 FlexRay controller (access to the data stored in message buffers). |
| **void (*PctrlReset)(void)** | This member holds pointer to a routine that performs a hardware platform dependent hard-reset on the Freescale MFR4300 FlexRay controller. If there is no way for performing the hardware reset by software this field must contain NULL. |

#### 4.3.1.3 TDDLL_CHI_MFR4300_CommandType

This compound datatype holds a register configuration value for a Freescale MFR4300 FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint32 nCHIOffset** | This member holds a FlexRay controller register address offset. The offset is calculated from the base CHI address of a Freescale MFR4300 FlexRay controller. |
| **uint16 nCHIValue** | This member holds the value to be written in the Freescale MFR4300 FlexRay controller register identified by the member **nCHIOffset**. |

#### 1.1.1 TDDLL_MFR4300_FrameDsc_ExtendedConfigType

This compound datatype contains extended information that is required to configure the buffer of a MFR4300 FlexRay controller.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint32 nMessageBufferConfiguration** | This member holds some information on a message buffer configuration (message buffer index on the System Memory side and offset to the message buffer data) |

### 4.3.2 Extensions

There are no extensions specific to MFR4300 COMMSTACK implementation.

# 4.4 Philips Rev1 FlexRay CC (SJA2510)

## 4.4.1 Datatypes

### 4.4.1.1 TDDLL_PHIP1_SlotControlType

This enumerator is used to represent the Philips FlexRay controller Rev 1 (as used within the SJA2510 MCU) SCL mode.

| Value | Description |
|---|---|
| **TDDLL_SLOTMODE_RX** | The slot is used to receive data. |
| **TDDLL_SLOTMODE_TX** | The slot is used to transmit data. |
| **TDDLL_SLOTMODE_IGNORE** | The slot is not used anymore. |

### 4.4.1.2 TDDLL_PHIP1_CtrlStateInfoType

This compound datatype holds the FlexRay controller COMMSTACK state administrative information required per Philips Rev1 FlexRay controller as contained in the SJA2510 MCU.

This datatype is required for static volatile memory allocation within the COMMSTACK configuration and COMMSTACK internal functionality but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **TDDLL_CtrlStateType nCtrlState** | This member holds the COMMSTACK FlexRay controller state. |
| **uint16 SCLAIdx** | Index into memory allocated for SCL channel A where the SCL channel A for the dedicated FlexRay Controller starts. As long as a single FlexRay CC is used, this index will be 0. |
| **uint16 SCLBIdx** | Index into memory allocated for SCL channel B where the SCL channel B for the dedicated FlexRay Controller starts. As long as a single FlexRay CC is used, this index will be 0. |
| **uint16 BCLIdx** | Index into memory allocated for BCL where the BCL for the dedicated FlexRay Controller starts. As long as a single FlexRay CC is used, this index will be 0. |
| **uint16 BDLIdx** | Index into memory allocated for BDL where the BDL for the dedicated FlexRay Controller starts. As long as a single FlexRay CC is used, this index will be 0. |
| **uint32 sISRsave** | This member saves the enable/disable state for all interrupt sources when emulating the global interrupt enable/disable switch. |
| **uint8 sGlobalInterruptEnable** | This member saves the state of the global interrupt enable/disable switch. |

### 4.4.1.3 TDDLL_PHIP1_CtrlListType

This compound datatype holds the FlexRay controller hardware mapping information for a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|

| | |
|---|---|
| **FCAL_PHIP1_CtrlHandleType nCtrlHandle** | This member holds the hardware access port information to the Philips Rev1 FlexRay controller as contained in the SJA2510 MCU (access to the controller's internal registers). |

#### 4.4.1.4 TDDLL_CHI_PHIP1_CommandType

This compound datatype holds a register configuration value for a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint16 nCHIOffset** | This member holds a FlexRay controller register address offset. The offset is calculated from the base CHI address of a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU. |
| **uint16 nCHIValue** | This member holds the value to be written in the Philips Rev1 FlexRay controller (as contained in the SJA2510 MCU) register identified by the member **nCHIOffset**. |

### 1.1.2 TDDLL_PHIP1_FrameDsc_ExtendedConfigType

This compound datatype contains extended information that is required to configure the buffer of a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint16 SCLEIdx** | This is the index into the SCL used for the particular FlexRay frame the frame is assigned to. |
| **uint16 BDLEIdx** | This is the index into the BDL used for the particular FlexRay frame the frame is assigned to. |

### 1.1.3 TDDLL_PHIP1_RegisterType

This compound datatype contains extended information that is required to configure a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **const TDDLL_CHI_PHIP1_CommandType *pCHICommands** | Pointer to an array of CHI initialization values. |
| **const TDDLL_PHIP1_VirtualRegisterType *pCHC_SCL_A** | This member holds the pointer for SCL channel A initialization values. In case global configuration switch TDDLL_PHIP1_CONSTANT_SCL is enabled this table will be used immediately. In case it is NULL no channel A SCL is available. |
| **const TDDLL_PHIP1_VirtualRegisterType** | This member holds the pointer for SCL channel B initialization values. In case global configuration switch TDDLL_PHIP1_CONSTANT_SCL is enabled this |

| | |
|---|---|
| ***pCHC_SCL_B** | table will be used immediately. In case it is NULL no channel B SCL is available. |
| **const TDDLL_PHIP1_VirtualRegisterType *pCHC_BCL** | This member holds the pointer for BCL initialization values. This table is copied into RAM during CC initialization. In case it is NULL no BCL is available. |
| **const TDDLL_PHIP1_VirtualRegisterType *pCHC_BDL** | This member holds the pointer for BDL initialization values. This table is copied into RAM during CC initialization. In case it is NULL no BDL is available. |
| **const uint32 *pFrameBuffer_A** | This member holds the pointer for FrameBuffer channel A space. If it is NULL no initialization table is available - buffer space will be initialized with 0 in this case. |
| **const uint32 *pFrameBuffer_B** | This member holds the pointer for FrameBuffer channel B space. If it is NULL no initialization table is available - buffer space will be initialized with 0 in this case. |
| **uint32 NumCHC_FrameBuffer_A** | This member holds the size of FrameBuffer channel A space in units of 32bit words. |
| **uint32 NumCHC_FrameBuffer_B** | This member holds the size of FrameBuffer channel B space in units of 32bit words. |
| **uint16 NumCHICommands** | This member holds the number of entries within the CHI register initialization list. |
| **uint16 NumCHC_SCL_A** | This member holds the number of SCL elements for Channel A. |
| **uint16 NumCHC_SCL_B** | This member holds the number of SCL elements for Channel B. |
| **uint16 NumCHC_BCL** | This member holds the number of BCL elements. |
| **uint16 NumCHC_BDL** | This member holds the number of BDL elements. |

#### 4.4.1.5 TDDLL_PHIP1_VirtualRegisterType

This compound datatype holds a virtual register list element as used in lists SCL channel A, SCL channel B, BCL and BDL for a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| **uint32 Register1** | This is the first register within an virtual register list element. Since this datatype is used for different virtual registers, there was an abstract member name choosen. |
| **uint32 Register2** | This is the second register within an virtual register list element. Since this datatype is used for different virtual registers, there was an abstract member name choosen. |

#### 4.4.1.6 TDDLL_PHIP1_FrameBufferType

This compound datatype holds the framebuffer header part used in a Philips Rev1 FlexRay controller as contained in the SJA2510 MCU.

This datatype is required within the COMMSTACK configuration and COMMSTACK internal functionality only but it is not required at the COMMSTACK user API.

| Member | Description |
|---|---|
| `uint32 Status` | This member holds the message buffer status bit virtual register. |
| `uint32 HeaderPart1` | This member holds the message buffer header part1 virtual register. |
| `uint32 HeaderPart2` | This member holds the message buffer header part2 virtual register |
| `uint32 aPayload[1]` | This member holds the first payload virtual register word. |

### 4.4.2 Extensions

COMMSTACK implementation for PHIP1 FlexRay controller provides one additional API function, which is available for this FlexRay CC COMMSTACK implementation only.

#### 4.4.2.1 TDDLL_SetSlotControl_PHIP1()

##### 4.4.2.1.1 Synopsis

```
TDDLL_ReturnType TDDLL_SetSlotControl_PHIP1 (
      TDDLL_CtrlIDXType nCtrlIDX,
      TDDLL_FrameDscRefIDXType nFrameDscRefIDX,
      TDDLL_PHIP1_SlotControlType nSlotType
      )
```

##### 4.4.2.1.2 Semantics

This function changes the operation mode of a FlexRay buffer identified by **`nFrameDscRefIDX`** into mode specified by **`nSlotType`** on FlexRay Controller **`nCtrlIDX`**. This Function operates immediately on the SCL element bitfield "*Control*" used by the dedicated FlexRay frame.

##### 4.4.2.1.3 Parameters

| Value | Description |
|---|---|
| `nCtrlIDX` | The FlexRay controller index of the controller this function should be performed on. |
| `nFrameDscRefIDX` | Unique ID that identifies the frame to be controlled. |
| `nSlotType` | Type (Ignore, Rx, Tx) the FlexRay slot should be set to. |

##### 4.4.2.1.4 Return Values

| Value | Description |
|---|---|
| `TDDLL_E_OK` | The function was successfully finished. |
| `TDDLL_E_INVALID_IDX` | The given input parameter or some basic configuration is invalid. |

# 5 DESIGNER PRO Integration

The COMMSTACK API is based on application dependent configuration data, which must exist for any application. The configuration data are represented by C source code and mainly consists of data structures used for initializing the FlexRay hardware and on the other hand for supporting the message exchange over the FlexRay communication system. As in general the manually creation of these configuration data is quite a complex task, there is convenient code generator support for this purpose. This tool support is implemented with DECOMSYS::DESIGNER PRO.

The following sections give instructions on how DESIGNER PRO can be used to generate the COMMSTACK configuration for an application and how this data is utilized by the application later on.

## 5.1 Use Case Assumption

As a basis for the documentation the following use case is assumed: There is an application consisting of 2 ECUs interconnected via a FlexRay network. The FlexRay network consists of two communication channels. For fault tolerant reasons all nodes are connected to both FlexRay channels.

Figure 4 illustrates the previous described hardware setup.

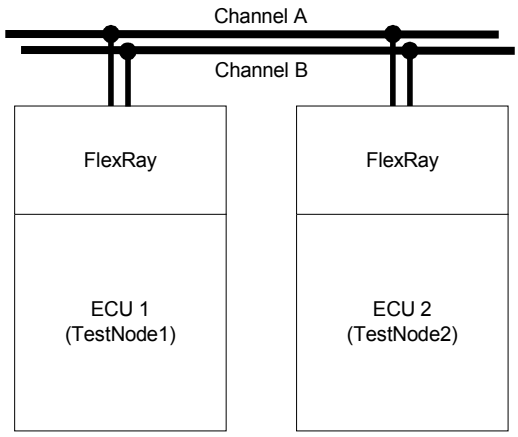


**Figure 4: Assuemd Hardware Setup**

ECU 1 sends in slot 39 and slot 111 on both channels, ECU 2 receives all frames sent by ECU 1. ECU 2 sends in slot 64 and slot 120 on both channels. ECU 1 receives all frames sent by ECU 2.

Figure 5 illustrates the previous described communication setup.

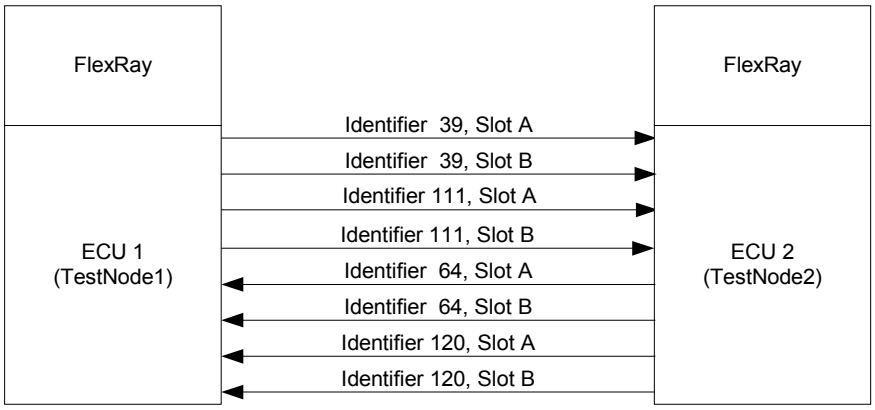


**Figure 5: Assumed Communication Setup**

## 5.2 Generating the COMMSTACK Configuration

### 5.2.1 Assumption

The description assumes that a Binary Object Repository file (BOR file) is available, containing a valid FlexRay cluster definition for the above depicted application setup (that has either been created with DESIGNER PRO, or imported via FIBEX or XCDEF importer into the DESIGNER PRO) and that all prior configuration steps to the COMMSTACK configuration generation have been applied.

### 5.2.2 Invocation of the Driver Configuration Plug-In

The required tool for the COMMSTACK configuration is the Driver Configuration Plug-in of DESIGNER PRO. The GUI window of this tool can be invoked by opening the Driver Config section in the OPERATION menu of DESIGNER PRO and by clicking the COMMSTACK item in it. Figure 6 shows the opened Driver Config operation menu of DESIGNER PRO, Figure 7 depicts the initial appearance of the Driver Configuration GUI window.

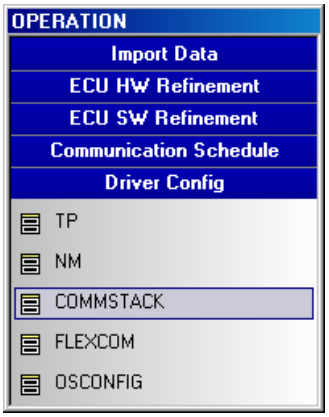


**Figure 6: DESIGNER PRO Operation Menu**

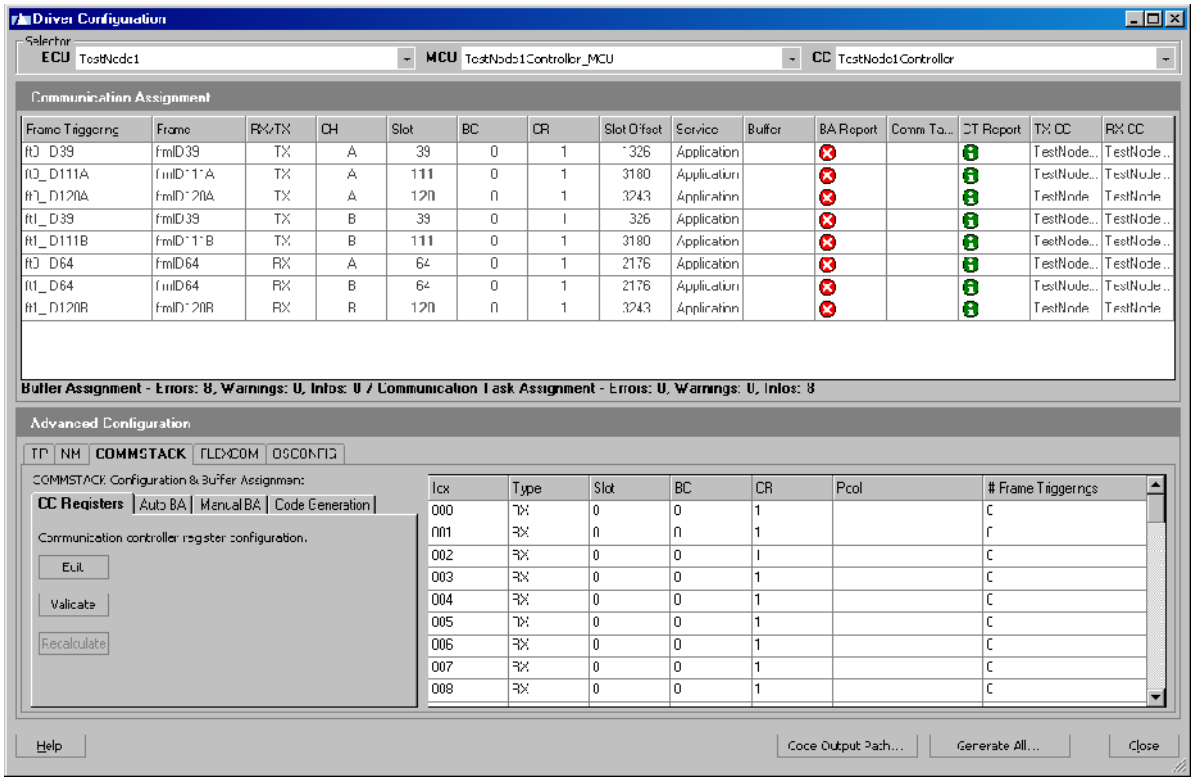


**Figure 7: Driver Configuration Window**

### 5.2.3 Selector Panel

The popup menus ECU, MCU and CC are used to select a distinct communication controller in order to display and to manipulate its configuration data.

### 5.2.4 Communication Assignment Grid

The Communication Assignment Grid seen in Figure 7 reflects the initial configuration corresponding to the assumed setup introduced in section 5.1 (and from which the COMMSTACK configuration generation starts off).

The Communication Assignment Grid shows information on the exchanged frames among the ECUs of the cluster from the point of view of TestNode1Controller (TestNode1Controller is the name for the FlexRay controller hosted by TestNode1 ECU). The grid shows all message frames, sent and received by the controller.

Each line of the Communication Assignment Grid refers to a distinct frame and summarizes several frame attributes. The following table gives some explanation on the seen frame attributes.

| Column | Description |
|---|---|
| **Frame Triggering** | A frame triggering is a FlexRay frame which is uniquely identified by the channel (CH), slot, base cycle (BC) and cycle repetition (CR) |
| **Frame** | Short name of the frame. |
| **RX/TX (Type)** | Specifies whether the frame is a receive (RX) or a transmit (TX) frame. |
| **CH (Channel)** | Specifies on which physical FlexRay channel the frame is sent. |

| | |
|---|---|
| **Slot** | Specifies the FlexRay slot used for the frame transmission. |
| **BC (Base Cycle)** | Specifies the base communication cycle (0..63), in which the frame is sent the first time. |
| **CR (Cycle Repetition)** | Specifies the sending repetition of the frame in units of communication cycles (1..64). (Counting starts at BC + 1. It must be satisfied that BC + CR <= 64.) |
| **Slot Offset** | Specifies the offset of frame in the communication cycle in units of µs. |
| **Service** | Denotes the "owner" of the frame. For example it is possible that the frame is owned by the NM, TP, FlexCOM, or seen in Figure 7, by the application itself. |
| **Buffer** | Shows the assigned physical FlexRay message buffer, or the assigned buffer pool respective. |
| **BA Report (Buffer Assignment Report)** | Reports information on the buffer assignment configuration. To see a report the mouse must be dragged over the appropriate grid cell. If the configuration contains errors, it displays a red sign, if everything is ok, a green sign. |
| **Communication Task** | In case the frame is owned by the FlexCOM service, this column shows the assigned communication task. |
| **CT Report (Communicaiton Task Report)** | Reports information on the communication task assignment configuration. To see a report the mouse must be dragged over the appropriate cell. If the configuration contains errors, it displays a red sign, if everything is ok, a green sign. |
| **TX CC** | Shows the sending communication controller of the frame. |
| **RX CC** | Shows the receiving communication controller of the frame. |

**Figure 8: Frame Attributes**

**NOTES:**

- For the use of the COMMSTACK message API (in order to uniquely identify a frame), the controller interface must be considered additionally. This is the cause why the COMMSTACK message API takes the controller interface in the parameter list (see later in this chapter).
- The red signs in the BA Report column indicate that there is no buffer assignment done yet.

### 5.2.5 Assigning Queues

Additionally, each frame configured for reception or transmission can be equipped with a software queue, which enables the usage of COMMSTACK queuing API functions (chapter 3.9).

The depth of each queue can be configured for each distinct frame. A queue depth with size 0 disables queuing for the dedicated Frame-triggering. No memory will be required in this case.

For using the feature one has to go deep into the DESIGNER Pro internal database view. Open the Entity-View and open the Type Selector "Frame Triggering Receiver Channel-Port" or "Frame Triggering Transmitter Channel-Port" depending whether you want to apply the queue to a receive- or transmission frame triggering. Select the frame triggering you want to apply the queue to, and enter the appropriate queue depth into field "Rx Queue Depth" / "Tx Queue Depth" as shown in Figure 9.

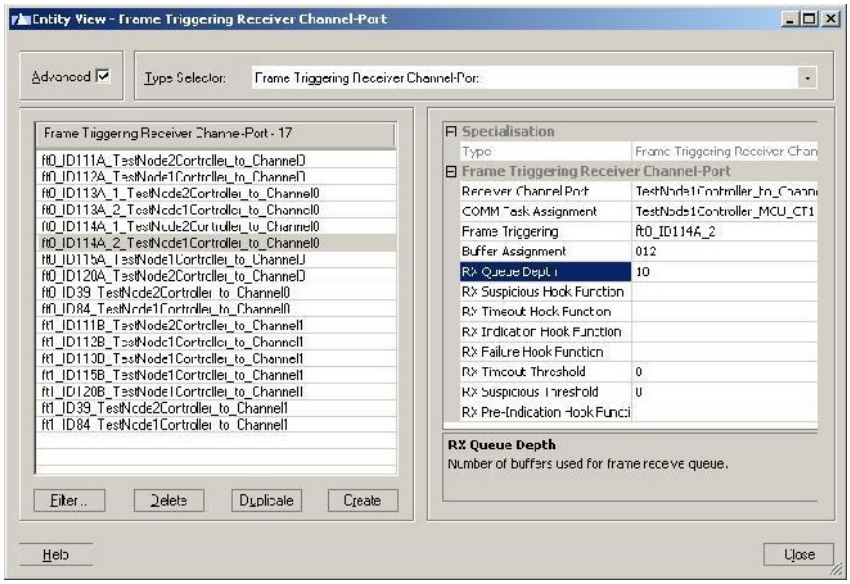


**Figure 9: Queue Depth Configuration**

### 5.2.6 Advanced Configuration Panel

In the Advanced Configuration panel of the Driver Configuration window the user has the ability for the advanced setup of the considered target service. (As this documentation is focused on the COMMSTACK itself and not on the services that are based on it, the configuration of NM, TP and FlexCOM are not discussed closer.)

#### 5.2.6.1 Communication Controller Registers

As already described earlier in this chapter, the Driver Configuration Plug-In of DESIGNER PRO generates initialization code for the FlexRay controllers. In particular, this initialization code consists of register settings for the communication controllers.

Usually the DESIGNER PRO calculates the register values (independent of the cluster settings, the message schedule and the used controller derivate). With the CC Registers feature the user has the ability to influence the calculated register values. Figure 10 shows the appearance of the selected CC Registers tab.

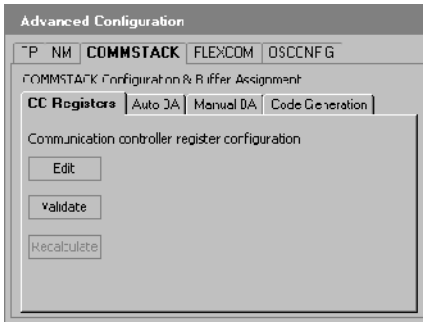


**Figure 10: CC Registers Tab**

When pressing the Edit button, the Entity View opens in edit mode with communication controller selected in the Selector Panel of the Driver Configuration window. After changing any register values with the Entity View the Validate button can be pressed to check the modified parameters.

**NOTE:** This feature is useful in particular for the advanced configuration of the communication controllers.

### 5.2.6.2 Automatic Buffer Assignment

To enable the controller to receive and transmit the defined frames, the physical FlexRay message buffers must be configured accordingly. This developing step is also named as Buffer Assignment. There is the possibility to let the DESIGNER PRO to do the buffer assignment automatically by using the Automatic Buffer Assignment feature. This function can be executed by clicking on the Generate button in the Auto BA tab (Advanced Configuration section) of the Driver Configuration dialog. Figure 11 shows a screenshot of the focused Auto BA tab.

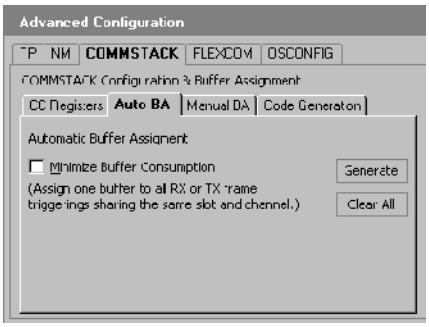


**Figure 11: Auto BA Tab**

The above step must be repeated for each communication controller of an ECU and for all the ECUs to be configured. On success all defined frames get a physical FlexRay buffer assigned. Thereafter, the Communication Assignment grid of the considered controller has the appearance depicted in Figure 12. It can be seen, that the Buffer column is filled up with indices (which map to the physical FlexRay message buffers). The success of the operation can bee seen additionally on the deleted error report signs in the BA Report column (please compare Figure 7).

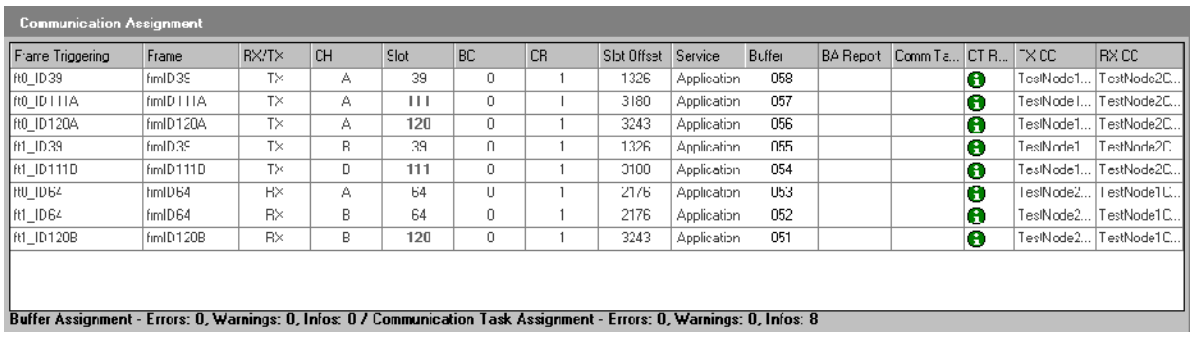


**Figure 12: Communication Assignment – assigend Buffers**

### 5.2.6.3 Minimize Buffer Consumption

If this option is checked, DECOMSYS::DESINGER PRO attempts to assign the same buffer to multiple frame triggerings. By that way, the consumption of the buffer use is reduced.

For this optimization it must be satisfied that the frame triggerings, sharing the same buffer have the same values for the type (RX/TX), slot and the channel assigned, but different values for the base cycle and cycle repetition, so that conflicts in accessing the different frames via this single buffer are avoided.

### 5.2.6.4 Manual Buffer Assignment

Beside the Automatic Buffer Assignment feature, DECOMSYS::DESINGER PRO offers the possibility for the manual buffer assignment. This function includes the assignment of buffer pools.

For this feature, first it is necessary to define the buffer pool (which consists of a single buffer in the simplest case) in the Manual BA tab of the Driver Configuration dialog.

The definition of the buffer pool can be done by:

- Adding a name for the pool: Enter the pool name in the Name field and pressing Create.
- Adding the pool members: Click on the Pool cell in the buffer information grid beneath the Manual BA tab of the buffer you want to add and select the target pool (name) from the appearing combobox.

Figure 13 shows an example for the definition of a buffer pool (MyBufferPool).

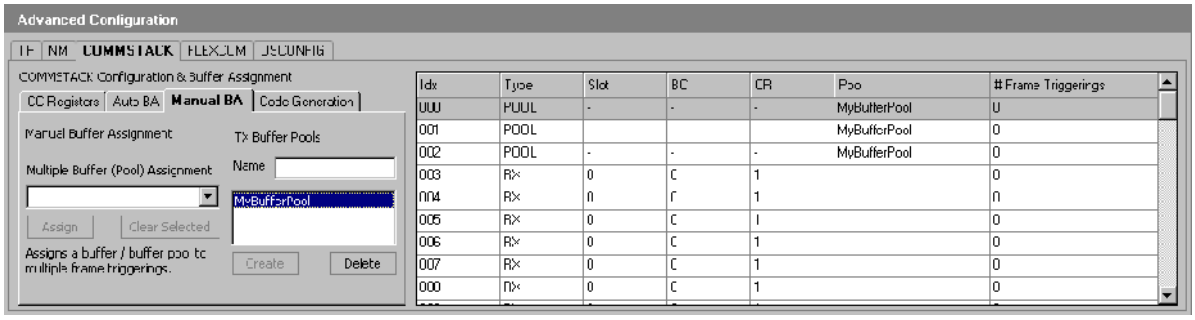


**Figure 13: Manual BA Tab**

The next step consist of assigning the defined buffer pools to frame triggerings. Hereby the name for the buffer pool must be entered in the Multiple Buffer (Pool) Assignment combobox, then the multiple frame triggerings selected in the Communication Assignment grid (by pressing the CTRL botton on the keyboard and selecting the appropriate lines in the Communication Assignment grid) and last, the button Assign be pressed.

### 5.2.6.5 Code Generation

When the buffer assignment step is finished, it can be proceeded with generating the configuration files for an ECU. This operation can also be done in the Advanced Configuration section of the Driver Config window, in tab Code Generation. First, the output path for the files must be set. The setting can be performed by calling the output directory selection dialog by clicking the button Code Output Path… in the Advanced Configuration section. After clicking this button, a standard windows file open dialog appaers, with which the user can set the output directory. When the directory is set the files can be generated into it. This function can be executed by clicking the Generate Code button in the Code Generation tab. Figure 14 shows the Advanced Configuration section of the Driver Config window. (**NOTE:** The Code Output Path... button can be found right on the bottom of the Driver Config dialog and can bee seen in Figure 7.)

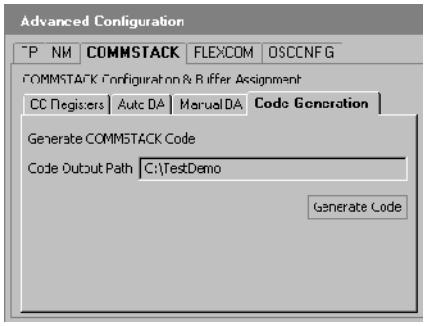


**Figure 14: Code Generation Tab**

The code generation procedure should produce the following files in the output diectory:

| File Name | Description |
|---|---|
| **dcsCstFr_<MCU_Name>[1]_Cfg.h** | Contains pre-processor definitions, which maps generic COMMSTACK API functions to FlexRay controller specific ones. |
| **dcsCstFr_<MCU_Name>_Cfg.c** | Contains initialization code for the FlexRay controller(s) and data structures describing the used FlexRay frames (i.e., holding the assigned physical message buffer index). |
| **dcsCstFr_<MCU_Name>_Memory_Cfg.c** | Contains data structures providing memory for the queued reception and transmission of messages and code for administrating the queues. |

The above depicted source code files must be included with the application build process for the owning ECU, which normally compiles the remaining COMMSTACK files, the ECU application source files, and the generated ECU configuration files to objects that are in turn linked to the executable for the target.

#### 5.2.6.6 Generate All Function

With this feature the user has the ability to generate all COMMSTACK configuration files for all ECUs in a single step (provided the output directories for all ECUs have been set).

The function can be activated by clicking the “Generate All…” button, which can be seen in the Driver Configuration window depicted in Figure 7.

## 5.3 Applying the COMMSTACK Configuration

This section gives some examples on the implementation of the message functions on basis of the generated COMMSTACK configuration and from the point of view of ECU 1 (TestNode1).

### 5.3.1 Example for sending a frame using the TDDLL_TxFrameByID() and TDDLL_LookupTxFrame() COMMSTACK API functions

The following code illustrates the use of TDDLL_TxFrameByID() and TDDLL_LookupTxFrame() COMMSTACK API functions to transmit the frame, which is configured by the first line of the Communication Assignment grid in Figure 12. For calling the function the configuration parameters Slot (39), Channel (A), Cycle Repetition (1), and Base Cycle (0) are required.

In summary, the code for sending the considered frame has the following appearance:

```
{
      TDDLL_ReturnType nStatus;
      const char aTxMsg[ PAYLOADLENGTH ] = "Example 1";

      nStatus = TDDLL_TxFrameByID(
            0,
            TDDLL_LookupTxFrame( 0, 39, TDDLL_CHA, 1, 0 ),
```

[1] <MCU_Name> is the placeholder for the actual MCU name and is expanded on code generation.

```
            aTxMsg,
            PAYLOAD_LENGTH
      );
}
```

The first parameter of TDDLL_TxFrameByID() and TDDLL_LookupTxFrame() gives the interface number the FlexRay controller is mapped to.

aTxMsg is the temporary message buffer containing the data for transmission. In the shown example this is “Example 1”.

PAYLOAD_LENGTH specifies the frame length.

On success the function returns TDDLL_E_OK.

### 5.3.2 Example for sending a frame using the TDDLL_TxFrameByID() and TDDLL_TX_FRAME_TRIGGERING() COMMSTACK API functions

The following code illustrates the use of TDDLL_TxFrameByID() and TDDLL_TX_FRAME_TRIGGERING() COMMSTACK API functions to transmit the frame, which is configured by the first line of the Communication Assignment grid in Figure 12. For calling the function the configuration parameter Frame Triggering (ft0_ID39) is required.

In summary, the code for sending the considered frame has the following appearance:

```
{
      TDDLL_ReturnType nStatus;
      const char aTxMsg[ PAYLOADLENGTH ] = "Example 1";

      nStatus = TDDLL_TxFrameByID(
            0,
            TDDLL_TX_FRAME_TRIGGERING( ft0_ID39 ),
            aTxMsg,
            PAYLOAD_LENGTH
      );
}
```

The first parameter of TDDLL_TxFrameByID() gives the interface number the FlexRay controller is mapped to.

aTxMsg is the temporary message buffer containing the data for transmission. In the shown example this is “Example 1”.

PAYLOAD_LENGTH specifies the frame length.

On success the function returns TDDLL_E_OK.

### 5.3.3 Example for receiving a frame using TDDLL_RxFrameByID() and TDDLL_LookupRxFrame() COMMSTACK API functions

The following code illustrates the use of TDDLL_RxFrameByID() and TDDLL_LookupRxFrame() COMMSTACK API functions to receive the frame, which is configured by the sixth line of the Communication Assignment grid in Figure 12. For calling the function the configuration parameters Slot (64), Channel (A), Cycle Repetition (1), and Base Cycle (0) are required.

In summary, the code for receiving the considered frame has the following appearance:

```
{
      TDDLL_ReturnType nStatus;
      TDDLL_LengthType nLength;
      const char aRxMsg[ PAYLOAD_LENGTH ];

      nStatus = TDDLL_RxFrameByID(
            0,
            TDDLL_LookupRxFrame( 0, 64, TDDLL_CHA, 1, 0 ),
            &aRxMsg[ 0 ],
            PAYLOAD_LENGTH,
            &nLength
      );
}
```

The first parameter of TDDLL_RxFrameByID() and TDDLL_LookupRxFrame() gives the interface the FlexRay controller is mapped to.

aRxMsg is the temporary message buffer for receiving the frame data.

nLength is the length of the frame actually received

PAYLOAD_LENGTH gives the buffer size the frame is written to.

On success the function returns TDDLL_E_OK.

### 5.3.4 Example for receiving a frame using the TDDLL_RxFrameByID() and TDDLL_RX_FRAME_TRIGGERING() COMMSTACK API functions

The following code illustrates the use of TDDLL_RxFrameByID() and TDDLL_RX_FRAME_TRIGGERING() COMMSTACK API functions to receive the frame, which is configured by the sixth line of the Communication Assignment grid in Figure 12. For calling the function the configuration parameter Frame Triggering (ft0_ID64) is required.

In summary, the code for sending the considered frame has the following appearance:

```
{
      TDDLL_ReturnType nStatus;
      TDDLL_LengthType nLength;
      const char aRxMsg[ PAYLOAD_LENGTH ];

      nStatus = TDDLL_RxFrameByID(
            0,
            TDDLL_RX_FRAME_TRIGGERING( ft0_ID64 ),
            & aRxMsg [ 0 ],
            PAYLOAD_LENGTH,
            &nLength
      );
}
```

The first parameter of TDDLL_RxFrameByID() gives the interface number to which the FlexRay controller is mapped to.

aRxMsg is the temporary message buffer for receiving the frame data.

nLength is the length of the frame actually received

PAYLOAD_LENGTH gives the buffer size the frame is written to.

On success the function returns TDDLL_E_OK.

# 6 COMMSTACK Configuration

The COMMSTACK design is focused on flexibility. The COMMSTACK philosophy is to provide mechanisms that don't limit the user to any specific use case, but offers the full range of applicability in the COMMSTACK function domain. To enable this feature the COMMSTACK provides a high degree of configurability, which has to be provided for operation. The binding method of the configuration options is carefully selected to meet the performance and application requirements.

There are three different binding methods used for the COMMSTACK configuration:

- Pre-Compile configuration
- Post-Compile configuration
- Runtime configuration

Pre-Compile configuration has impacts on the COMMSTACK compilation process. Using compiler switches this kind of configuration influences the COMMSTACK object code. This kind of configuration is used for enabling/disabling specific COMMSTACK features and setting the main hardware access properties of the COMMSTACK.

In Post-Compile configuration, the configuration doesn't affect the COMMSTACK binary data, but the configuration data must be present at link-time. Certain constant tables that affect the runtime behavior of the COMMSTACK must be provided during linking process. This kind of configuration is used for small hardware adaptations (memory mapping, reset configuration), which enables a COMMSTACK library to be used flexible on a certain hardware design without recompilation.

Runtime configuration must be available at latest before COMMSTACK usage at runtime. The application configuration, which affects the runtime communication behavior of the COMMSTACK, is configured this way. This means that also any other process than the build-process like RAM-download or flashprogramming can be enabled to load the COMMSTACK application configuration.

In this chapter the COMMSTACK file structure is described as well as the configuration files to edit for target customization.

## 6.1 COMMSTACK File Structure

The COMMSTACK consists of a root directory that contains all source files and header files. For structural reasons the COMMSTACK contains two sub-modules that are each implemented as macros or inline functions (depending on your compiler's capabilities). These sub-modules improve the ease of COMMSTACK porting by providing COMMSTACK internal abstraction, while obtaining as much performance as possible.

The following directories are contained the root directory:

- "*include*" – containing public interface header files
  Public interface header files offer an interface to other software layers or the application.

- "*src*" – containing source files and private header files
  Private header files are included only in the respective software layer where they reside. They are primary necessary for a clean software structure and consistency reasons.

- "*MAL*" – containing private Memory Access Library.
  The MAL COMMSTACK private library provides generic mechanisms for accessing memory and swapping bytes on a dedicated hardware configuration.

- "*FCAL*" – containing private FlexRay Controller Access Layer.
  The FCAL COMMSTACK private layer provides mechanisms for accessing memory mapped FlexRay controllers.

Each of these subdirectories ("*include*" and "*src*") may contain one subdirectory per supported hardware target architecture, however they may contain additional subdirectories. The architecture dependent subdirectories are prefixed with "*arch"*. Additionall subdirectories for FlexRay CC specific implementations might be present, named by the COMMSTACK FlexRay CC name in uppercase letters (e.g. ERAY10, MFR4200, MFR4300).

*When building the COMMSTACK ensure that you pass all include directories and their respective subdirectories (include, include/arch-…, include/MFR4200, include/ERAY10, include/MFR4300…) mentioned before to the compilers include path. Also the source path containing COMMSTACK private header-files (src, src/ERAY10, src/MFR4200, src/MFR4300 …) should be added to the compilers include path.*

Below, the COMMSTACK source code product file-structure as delivered for an example target architecture called dcsnodearm-generic is shown.

| | |
|---|---|
| dcsCstFr | COMMSTACK root directory |
| └doc | COMMSTACK documentation |
| MAL | |
| ├include | MAL macro library main header-files |
| │ └generic | MAL macro library CPU dependent implementations. |
| │ | |
| FCAL | |
| ├include | FCAL macro library main header-files |
| │ ├ERAY10 | FCAL macro library ERAY access implementations |
| │ ├MFR4200 | FCAL macro library MFR4200 access implementations |
| │ └MFR4300 | FCAL macro library MFR4300 access implementations |
| │ | |
| include | COMMSTACK interface header-files |
| ├arch-dcsnodearm-generic | COMMSTACK configuration files for example architecture |
| │ | |
| src | COMMSTACK CC independent source code |
| ├Cfg | |
| │ └arch-dcsnodearm-generic | COMMSTACK configuration files for example architecture |
| ├ERAY10 | COMMSTACK ERAY implementation source code |
| ├MFR4200 | COMMSTACK MFR4200 implementation source code |
| └MFR4300 | COMMSTACK MFR4300 implementation source code |

## 6.2 COMMSTACK Feature Configuration

Certain COMMSTACK features can be enabled/disabled using compiler switches. These switches have immediate impact on the COMMSTACK compilation process. Disabled features are removed from the compilation process, which optimizes memory and execution time consumption.

*The COMMSTACK feature configuration is of type Pre-Compile configuration. Modifying these switches requires having a source code COMMSTACK distribution.*

### 6.2.1 COMMSTACK Feature Switches

All COMMSTACK feature switches are implemented in the header-file **`dcsCstFr_Cfg.h`**.

All switches operate on a switch defined/undefined basis. This means that a feature is enables as soon as a key macro mapping to that feature is defined.

Enabling feature TDDLL_FEATURE_1_SWITCH would be implemented by setting the following source code line into file **`dcsCstFr_Cfg.h`**:

```
#define TDDLL_FEATURE_1_SWITCH
```

Disabling feature TDDLL_FEATURE_1_SWITCH is implemented by either leaving this line completely out of the file or setting the following source code line into file **`dcsCstFr_Cfg.h`**:

```
#undef TDDLL_FEATURE_1_SWITCH
```

The following table describes all available features that can be activated/deceived using pre-compile switches:

| Switch Name | Description |
|---|---|
| **`TDDLL_STATE_CHECK_SWITCH`** | This switch enables the synchronization of the COMMSTACK state at each API function invocation with the FlexRay CC. This switch should be always enabled if you do don't have very special knowledge about the COMMSTACK implementation. |
| **`TDDLL_ARGUMENT_CHECK_SWITCH`** | This switch enables an argument check of the validity of arguments given to COMMSTACK API functions. Enabling this switch could help debugging during the development phase. |
| **`TDDLL_QUEUEING_SWITCH`** | This switch enables the build process of the queuing API functions listed below:<br>**`TDDLL_RxFrameByIDQueued()`**<br>**`TDDLL_TxFrameByIDQueued()`**<br>**`TDDLL_FlushTxQueue()`**<br>**`TDDLL_FillRxQueue()`**<br>**`TDDLL_EmptyQueue()`** |
| **`TDDLL_BUFFERPOOLING_SWITCH`** | This switch enables the buffer pooling mechanism that enables the dynamic buffer assignment over a range of dynamic FlexRay frames. If this switch is disabled the application configuration must not use this feature. |
| **`TDDLL_FIFO_SWITCH`** | This switch includes the FIFO reception API function (**`TDDLL_RxFrameByFIFO()`**) into the COMMSTACK build process. |
| **`TDDLL_INTERRUPTS_SWITCH`** | This switch includes the following interrupt handling API functions into the COMMSTACK build process:<br>**`TDDLL_InterruptStatus()`**<br>**`TDDLL_InterruptResetStatus()`**<br>**`TDDLL_InterruptEnable()`**<br>**`TDDLL_InterruptDisable()`** |
| **`TDDLL_ABSOLUTE_TIMER_SWITCH`** | This switch includes the absolute timer API function |

| | |
|---|---|
| | (**TDDLL_SetTimerAbs()**) into the COMMSTACK build process. |
| **TDDLL_RELATIVE_TIMER_SWITCH** | This switch includes the relative timer API function (**TDDLL_SetTimerRel()**) into the COMMSTACK build process. |
| **TDDLL_POC_STATUS_SWITCH** | This switch includes the POC status API function (**TDDLL_GetPOCStatus()**) into the COMMSTACK build process. |
| **TDDLL_STATIC_PAYLOAD_PADDING_PATTERN_SWITCH** | In case a frame shorter than the application schedule configured static payload length should be transmitted from a COMMSTACK transmission API function, this switch enables to fill the remaining bytes of the frame with a fill-pattern defined in macro **TDDLL_PAYLOAD_PADDING_PATTERN** within the COMMSTACK configuration file **dcsCstFr_Cfg.h**. |
| **TDDLL_SINGLE_CC_TYPE_OPTIMIZATION** | |

*Since COMMSTACK transmission API functions operate on the per-byte frame length but FlexRay requires 16-bit words (resulting in an even number of bytes) to be transmitted, the value defined in COMMSTACK macro **TDDLL_PAYLOAD_PADDING_PATTERN** is used for filling the last byte in case an odd number of bytes should be transmitted.*

## 6.3 COMMSTACK Target Hardware Configuration

*Most COMMSTACK target hardware configuration options can be modified in COMMSTACK source code distributions only since these settings affect the COMMSTACK compilation process. Address mapping and reset configuration of the COMMSTACK is done in a post-compile configuration file (**dcsCstFr_CtrlHW_Cfg.c**) that can be modified in COMMSTACK library distributions also.*

The COMMSTACK is designed to enable porting to a different hardware by modifying some very low-level hardware specific files. The objective of this chapter is to give a quick introduction on how to adapt the COMMSTACK to your target hardware.

Additionally the provided example adaptations (for memory mapped FlexRay controller access only) are described. This chapter gives a comprehensive description on the all hardware related settings and explains how they must be adopted for the new target hardware.

### 6.3.1 Compiler specific function implementations

The COMMSTACK sub-modules MAL and FCAL provide C-syntax API implementations that provide some required functionality and work on most C-compilers. For different compiler capabilities there are several alternative implementations provided, which can be identified by their different filename postfixes “inline”, “noinline” and “macro” (e.g. **dcsMAL_generic_swap_inline.h**, **dcsMAL_generic_swap_macro.h**, **dcsMAL_generic_swap_noinline.h**).

In general the proposed solution is the “inline” implementation variant. If your compiler doesn’t support the “inline” keyword consult the manual whether inline functions are supported by some other compiler-specific mechanism or not. If function inlining is supported by some different compiler

syntax of you compiler “customCompiler” use the provided ”inline” implementation as a template, modify it according to your compiler’s inline syntax and provide a compiler specific implementation (e.g. copy **`dcsMAL_generic_swap_inline.h`** to **`dcsMAL_customCompiler_swap_inline.h`**) matching your compiler’s syntax.

Some compilers that automatically perform inlining on static functions and remove not used static functions during optimization from the target binary but don’t support any syntactical function inlining might also accept the “noinline” implementation variant.

**Finally a pure macro implementation “macro” is provided. According to poor performance and potential data consistency problems this implementation should not be used in production code.**

*In case several alternative MAL/FCAL implementations are provided (“inline”,”noinline”,”macro”) the DECOMSYS recommended implementation is “inline”.*

*Don’t use the function implementation variant “macro” in production code.*

### 6.3.2 CPU Memory Access Configuration

The MAL layer contains a library of macros that provide C-syntax capabilities for the COMMSTACK implementation.

These capabilities are:

- *aligned*/*unaligned* [volatile] *32*/*16*/*8*-bit *read*/*write* access
- byte-swapping for *16*/*32*-bit words

*If your hardware architecture provides distinct assembler operations for byte swapping or memory access the compiler doesn’t support, the generic implementations might be replaced by inline assembler operations for performance reasons.*

*The COMMSTACK memory access configuration is of type pre-compile configuration.*

In the COMMSTACK main configuration file **`dcsCstFr_Cfg.h`** two macros that select the correct COMMSTACK behavior for the selected target hardware architecture have to be defined.

Macro **`_TDDLL_HEADER_FILE_DCSMAL_SWAP_`** selects the byte-swapping implementation. The following options are provided by the default COMMSTACK product:

| swap implementation | Description |
|---|---|
| **`dcsMAL_generic_swap_inline.h`** | “inline” implementation variant (see chapter 6.3.1) of byte-swapping functionality. |
| **`dcsMAL_generic_swap_noinline.h`** | “noinline” implementation variant (see chapter 6.3.1) of byte-swapping functionality. |
| **`dcsMAL_generic_swap_macro.h`** | “macro” implementation variant (see chapter 6.3.1) of byte-swapping functionality. |

Macro **`_TDDLL_HEADER_FILE_DCSMAL_ALIGNMENT_`** selects the alignment restrictions of the target hardware architecture the COMMSTACK is executed on. An alignment restriction on memory accesses requires (in general) that the address an memory access occurs must be divisible by the access size with no remainder (**`addr%sizeof(access) == 0`**). In case of an access restriction the COMMSTACK memory access function option “forbid_unaligned” is provided. In case the target

architecture requires no alignment restrictions (`addr%sizeof(access) != 0`) the access function provided by option “allow_unaligned” shall be used.

The following options are provided by the default COMMSTACK product:

| alignment restriction implementation | Description |
|---|---|
| `<dcsMAL_generic_rw_allow_unaligned.h>` | Implementation variant for target architectures that allow unaligned memory accesses. |
| `<dcsMAL_generic_rw_forbid_unaligned_be_inline.h>` | “inline” implementation variant (see chapter 6.3.1) for big endian target architectures that require aligned memory access. |
| `<dcsMAL_generic_rw_forbid_unaligned_be_noinline.h>` | “noinline” implementation variant (see chapter 6.3.1) for big endian target architectures that require aligned memory access. |
| `<dcsMAL_generic_rw_forbid_unaligned_be_macro.h>` | “macro” implementation variant (see chapter 6.3.1) for big endian target architectures that require aligned memory access. |
| `<dcsMAL_generic_rw_forbid_unaligned_le_inline.h>` | “inline” implementation variant (see chapter 6.3.1) for little endian target architectures that require aligned memory access. |
| `<dcsMAL_generic_rw_forbid_unaligned_le_noinline.h>` | “noinline” implementation variant (see chapter 6.3.1) for little endian target architectures that require aligned memory access. |
| `<dcsMAL_generic_rw_forbid_unaligned_le_macro.h>` | “macro” implementation variant (see chapter 6.3.1) for little endian target architectures that require aligned memory access. |

For practical usage of the memory access configuration see the following example taken from the file `dcsCstFr_Cfg.h` of the COMMSTACK product target architecture configuration for target dcsnodearm_generic:

…

```
/* SWAP macros selection configuration */
#define _TDDLL_HEADER_FILE_DCSMAL_SWAP_ \
    <dcsMAL_generic_swap_inline.h>

/* memory access restriction configuration */
#define _TDDLL_HEADER_FILE_DCSMAL_ALIGNMENT_ \
    <dcsMAL_generic_rw_forbid_unaligned_le_inline.h>
```

…

### 6.3.3 FlexRay CC Access Configuration

The COMMSTACK product is prepared to connect different FlexRay CC via a parallel interface (memory mapped I/O) to a wide range of CPUs. There are three important properties that have to be considered when connecting a CPU memory mapped to a FlexRay CC:

- Endianess of the CPU
- Databus connection (bit width, byte crossing)

- Endianess of the FlexRay CC

According to these properties several system configuration combinations might be possible. The connection type to be used has to be configured in the COMMSTACK main configuration file **dcsCstFr_Cfg.h** using switches **_TDDLL_HEADER_FILE_<CC_Type>_ACCESS_** where **<CC_Type>** is one of the supported FlexRay Controllers (see the following chapters).

*If only a single FlexRay CC type is required, the access switches as described above should be removed completely for not required FlexRay CC types. This allows enabling the compiler switch **TDDLL_SINGLE_CC_TYPE_OPTIMIZATION** in file **dcsCstFr_Cfg.h** that completely removes the overhead of CC type evaluation at runtime. This saves code memory as well as execution time.*

#### 6.3.3.1 MFR4200 access configuration

MFR4200 FlexRay CC access configuration has to be done setting the compiler switch **_TDDLL_HEADER_FILE_MFR4200_ACCESS_** in file **dcsCstFr_Cfg.h** to one of the following values:

| MFR4200 access option | Description |
|---|---|
| **<dcsCstFr_FCAL_mfr4200_mmap_access_be.h>** | See Figure 15 for a detailed description. |
| **<dcsCstFr_FCAL_mfr4200_mmap_access_le_straight.h>** | See Figure 16 for a detailed description. |
| **<dcsCstFr_FCAL_mfr4200_mmap_access_le_swapped.h>** | See Figure 17 for a detailed description. |

This configuration item takes into account, that the data representation of CPU and the FlexRay communication controller might be different. There are three common methods for connecting a 16-bit communication controller, depending on the data bus connection and the data representation of the host CPU. The following examples show the supported connections to a memory-mapped 16-bit MFR4200 communication controller (CC) with big endian data representation. In the following figures we assume the host CPU and the CC having the same notation regarding the bit numbering (lsb=D0, msb=D15).

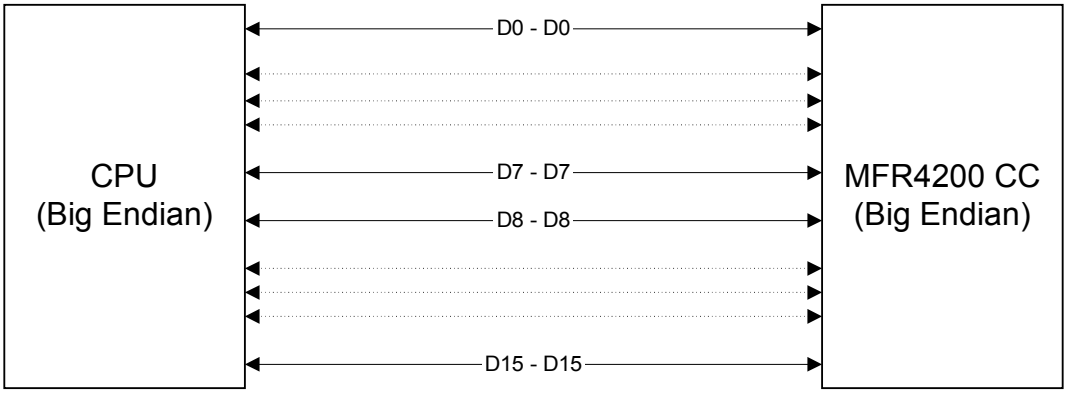


**Figure 15: CPU (BE) – straight connection- CC (BE)**

If both data bus participants have the same data representation, obviously there should be a 1:1 connection. The more complex case occurs if the data representation differs. Let's assume the CPU is a little endian (LE) device.

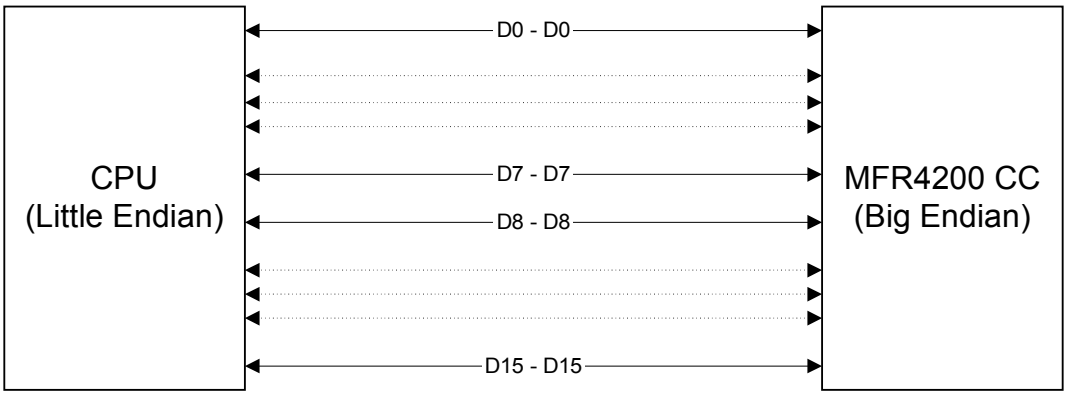


**Figure 16: CPU (LE) – straight connection – CC (BE)**

One solution to the data bus interface would be a bitwise 1:1 connection, which offers transparency for 16-bit data accesses. But if 16-bit data will be read into the CPU memory and interpreted as 8-bit data array (e.g. a string) the byte-access will end up in a mess.

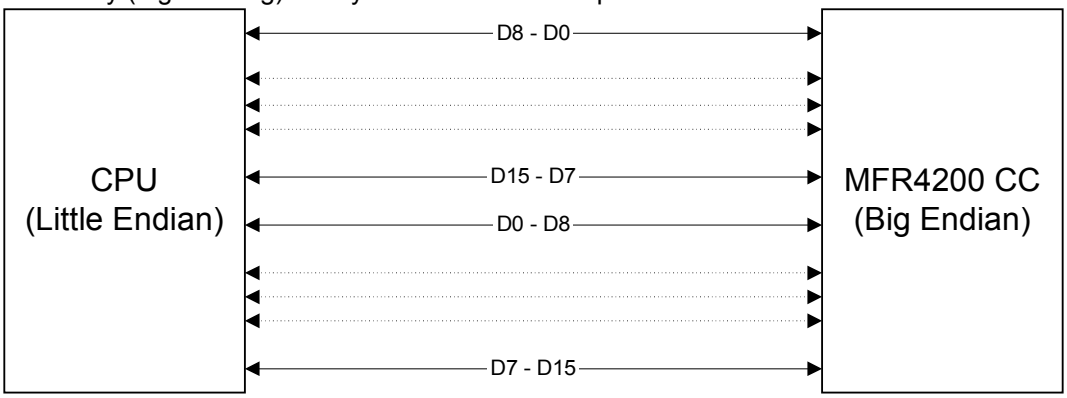


**Figure 17: CPU (LE) – swapped connection – CC (BE)**

Another solution would be to do the byte swapping by the data bus connection, which offers transparency to 8-bit array accesses but 16-bit units have to be swapped back again.
No matter what hardware connection chosen, the COMMSTACK offers the solution for all three types of connections between a CPU and a MFR4200 FlexRay CC.

*If no MFR4200 CC support is required within the COMMSTACK the switch* ***`_TDDLL_HEADER_FILE_MFR4200_ACCESS_`*** *should be completely removed from file* ***`dcsCstFr_Cfg.h`****.*

### 6.3.3.2 ERAY10 access configuration

*Please note that the ERAY10 FlexRay CC IP doesn't specify any endianess on the databus interface. Instead customer specifc interfaces must be provided which can implement as well a little endian databus-interface semantic as well as a big endian datrabus-interface semantic. The COMMSTACK is prepared to cover most ERAY10 access variants for little endian ERAY10 databus interface implementations. Please consult DECOMSYS if you need a solution provided for ERAY10 implementations offering a big endian databus-interface.*

ERAY10 FlexRay CC access configuration has to be done setting the compiler switch **`_TDDLL_HEADER_FILE_ERAY10_ACCESS_`** in file **`dcsCstFr_Cfg.h`** to one of the following values:

| ERAY10 access option | Description |
|---|---|
| **`<dcsCstFr_FCAL_eray10_mmap_access32_le_straight.h>`** | See Figure 18 for a detailed description. |
| **`<dcsCstFr_FCAL_eray10_mmap_access32_be_straight32.h>`** | See Figure 19 for a detailed description. |
| **`<dcsCstFr_FCAL_eray10_mmap_access32_be_straight16.h>`** | See Figure 20 for a detailed description. |

This configuration item takes into account, that the data representation of CPU and the FlexRay communication controller might be different. There are several common methods for connecting a 32-bit communication controller, depending on the data bus width, the data bus connection and the data representation of the host CPU. The following examples show the supported connections to a memory-mapped 32-bit ERAY10 communication controller (CC) with little endian data representation at the databus interface. In the following figures we assume the host CPU and the CC having the same notation regarding the bit numbering (lsb=D0, msb=D31). The following figures present the supported data bus connection patterns.

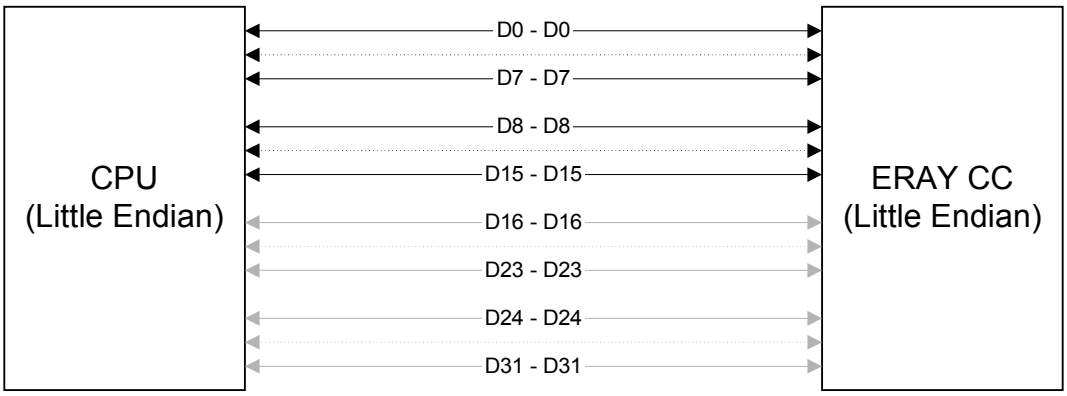


**Figure 18: CPU (LE) - 32/16-bit straight connection - CC (LE)**

Figure 18 shows the connection of a little endian CPU to an ERAY CC with a 16-bit or 32-bit data bus 1:1 connection.

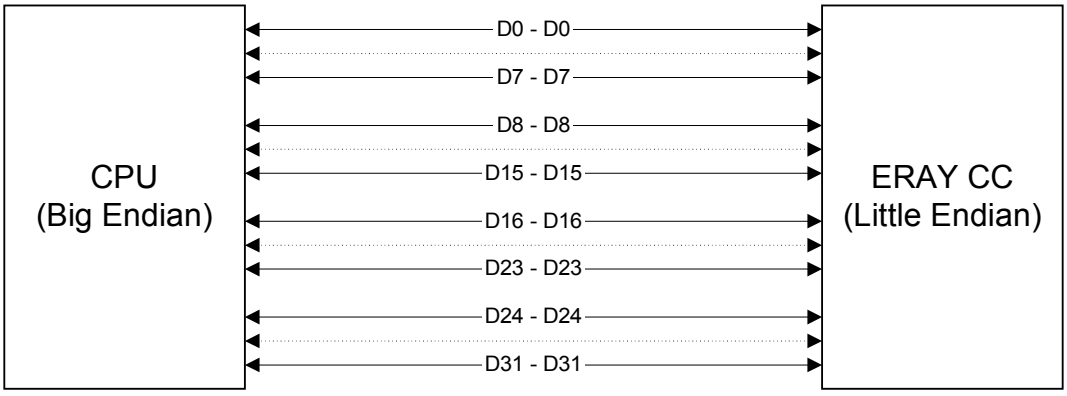


**Figure 19: CPU (BE) - 32-bit straight connection - CC(LE)**

Figure 19 shows the connection of a big endian CPU to an ERAY CC with a 32-bit data bus 1:1 connection.

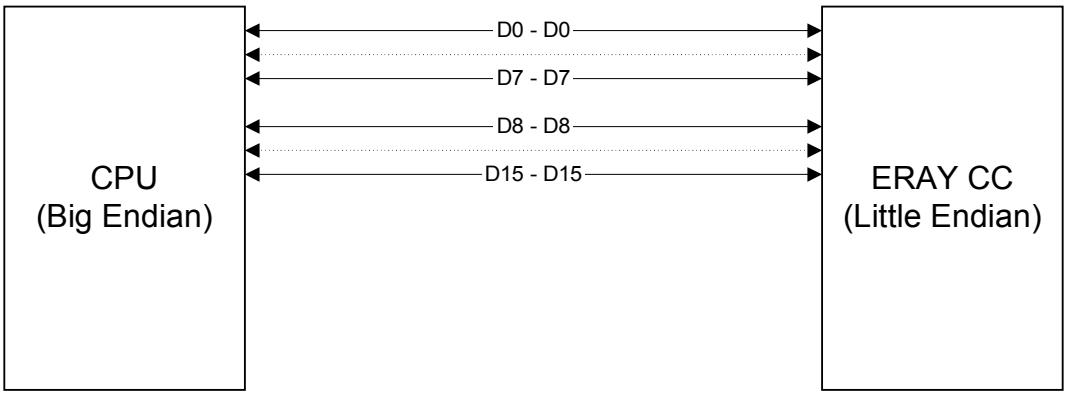


**Figure 20: CPU (BE) - 16-bit straight connection - CC (LE)**

Figure 20 shows the connection of a big endian CPU to an ERAY CC with a 16-bit data bus 1:1 connection.

*If no ERAY10 CC support is required within the COMMSTACK the switch* ***_TDDLL_HEADER_FILE_ERAY10_ACCESS_*** *should be completely removed from file* ***dcsCstFr_Cfg.h****.*

#### 6.3.3.3 MFR4300 access configuration

MFR4300 FlexRay CC access configuration has to be done setting the compiler switch **_TDDLL_HEADER_FILE_MFR4300_ACCESS_** in file **dcsCstFr_Cfg.h** to one of the following values:

| MFR4300 access option | Description |
|---|---|
| **<dcsCstFr_FCAL_mfr4300_mmap_access_be.h>** | See Figure 21 for a detailed description. |
| **<dcsCstFr_FCAL_mfr4300_mmap_access_le_straight.h>** | See Figure 22 for a detailed description. |
| **<dcsCstFr_FCAL_mfr4300_mmap_access_le_swapped.h>** | See Figure 23 for a detailed description. |

This configuration item takes into account, that the data representation of CPU and the FlexRay communication controller might be different. There are three common methods for connecting a 16-bit communication controller, depending on the data bus connection and the data representation of the host CPU. The following examples show the supported connections to a memory-mapped 16-bit MFR4300 communication controller (CC) with big endian data representation. In the following figures we assume the host CPU and the CC having the same notation regarding the bit numbering (lsb=D0, msb=D15).

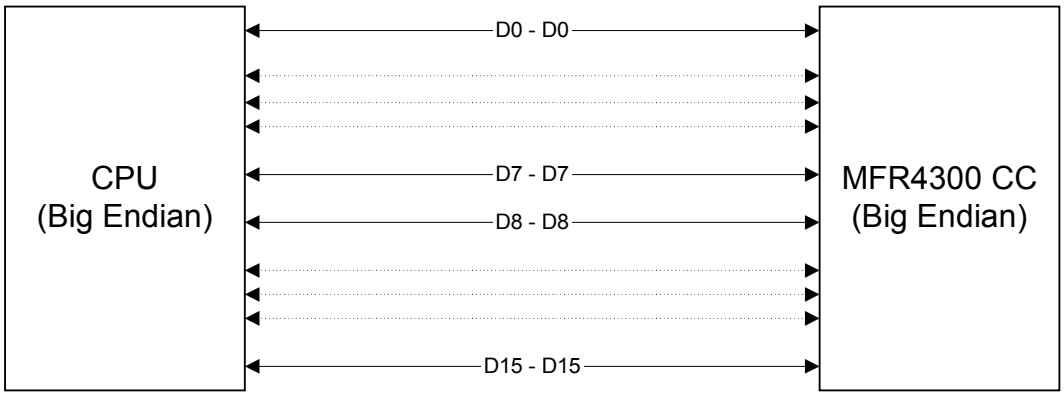


**Figure 21: CPU (BE) – straight connection- CC (BE)**

If both data bus participants have the same data representation, obviously there should be a 1:1 connection. The more complex case occurs if the data representation differs. Let's assume the CPU is a little endian (LE) device.

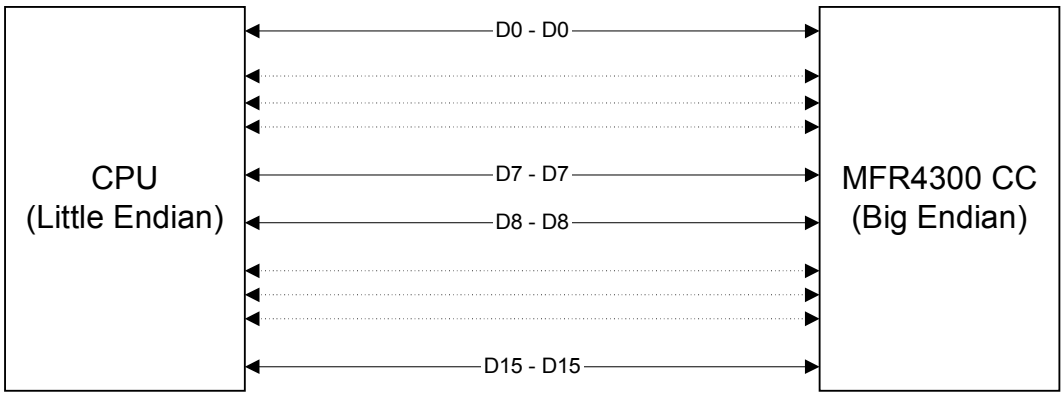


**Figure 22: CPU (LE) – straight connection – CC (BE)**

One solution to the data bus interface would be a bitwise 1:1 connection, which offers transparency for 16-bit data accesses. But if 16-bit data will be read into the CPU memory and interpreted as 8-bit data array (e.g. a string) the byte-access will end up in a mess.

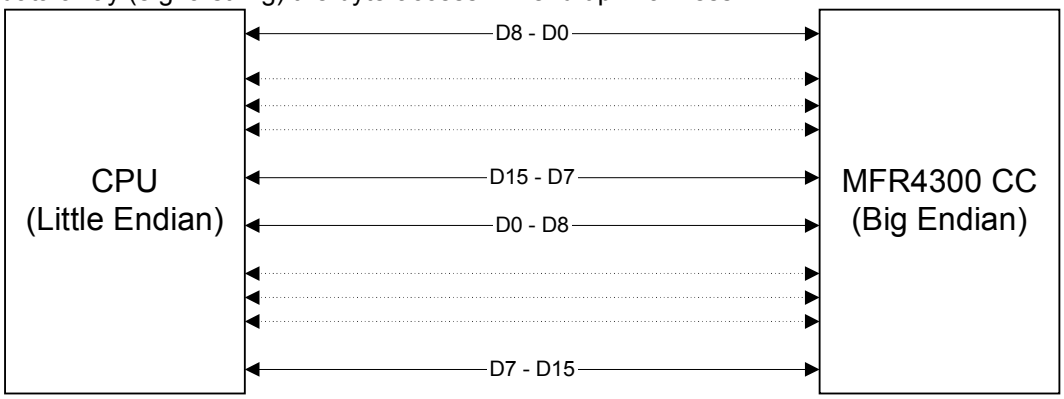


**Figure 23: CPU (LE) – swapped connection – CC (BE)**

Another solution would be to do the byte swapping by the data bus connection, which offers transparency to 8-bit array accesses but 16-bit units have to be swapped back again.

No matter what hardware connection chosen, the COMMSTACK offers the solution for all three types of connections between a CPU and a MFR4200 FlexRay CC.

*If no MFR4300 CC support is required within the COMMSTACK the switch **_TDDLL_HEADER_FILE_MFR4300_ACCESS_** should be completely removed from file **dcsCstFr_Cfg.h**.*

#### 6.3.3.4 PHIP1 access configuration

MFR4300 FlexRay CC access configuration has to be done setting the compiler switch **_TDDLL_HEADER_FILE_PHIP1_ACCESS_** in file **dcsCstFr_Cfg.h**.

Since the Philips FlexRay Controller Rev1 is supported in combination with a single MCU right now (SJA2510) there is also a single configuration option that describes the connection schema.:

| PHIP1 access option | Description |
|---|---|
| **<dcsCstFr_FCAL_phip1_mmap_access32_le_straight.h>** | Only PHIP1 FlexRay access configuration available. |

### 6.3.4 FlexRay CC Mapping Configuration

The mapping from FlexRay CC indices given at the COMMSTACK API to real hardware devices is done in a Post-Compile configuration style. This enables to configure a COMMSTACK delivered as library to be adapted to a wide range of hardware adaptations on a specific hardware platform.

These configuration modifications have to be done in the configuration file **dcsCstFr_CtrlHW_Cfg.c**. In general this file has to be adapted to a hardware a COMMSTACK is used on and must be linked to every application using the COMMSTACK.

The following configuration options can be modified in this file:

- number of FlexRay CC provided by the hardware
- mapping from API CC index to CC hardware type
- mapping a dedicated FlexRay CC to a memory base address
- supplying a dedicated FlexRay CC with a hardware reset mechanism

First of all an array has to be created that maps FlexRay CC indices as used at the COMMSTACK API to FlexRay controller types. Please note that the FlexRay CC indices should follow a closed index range from 0 to the number of FlexRay CCs –1 for optimal resource utilization:

```
const TDDLL_CtrlMappingType TDDLL_CtrlMapping[2] =
{
    {
        TDDLL_CTRL_TYPE_MFR4200, /* FlexRay CC type of CC index 0 */
         0,                       /* first MFR4200 in type specific list */
    },
    {
        TDDLL_CTRL_TYPE_ERAY10, /* FlexRay CC type of CC index 1 */
         0,                     /* first ERAY10 in type specific list */
    },

};

const TDDLL_CtrlIDXType TDDLL_NumCtrl = 2; /* number of FlexRay CCs */
```

In the example above the FlexRay CC with index 0 maps to an MFR4200 FlexRay CC type and FlexRay CC with index 1 to an ERAY10 FlexRay CC type implementation. The number of FlexRay controllers as connected to the target hardware must be stored in the constant variable

**TDDLL_NumCtrl.** As we will see soon each CC type has an own list, mapping the devices to hardware resources. The second parameter in the example above is the index into the device specific hardware resource list, which is described next.

```
extern void CC0_ResetFunction(void);

const TDDLL_MFR4200_Ctrl_List[1] =
{
    {
        (FCAL_MFR4200_CtrlHandleType) 0x80001800, /* CC base address   */
        CC0_ResetFunction                         /* CC reset function */
    }
};
const TDDLL_CtrlIDXType TDDLL_MFR4200_NumCtrl = 1;
TDDLL_MFR4200_CtrlStateInfoType TDDLL_MFR4200_CtrlStateList[1];
```

With this example the first (and only) MFR4200 FlexRay CC is mapped to the memory base address 0x80001800. The hardware reset is provided by an external function (**CC0_ResetFunction()**), which must be registered for this device. The absolute number of MFR4200 FlexRay CCs is defined in the constant variable **TDDLL_MFR4200_NumCtrl.**

```
const TDDLL_ERAY10_Ctrl_List[1] =
{
    {
        (FCAL_ERAY10_CtrlHandleType) 0x80002800, /* CC base address   */
        NULL                                     /* CC reset function */
    }
};
const TDDLL_CtrlIDXType TDDLL_ERAY10_NumCtrl = 1;
TDDLL_ERAY10_CtrlStateInfoType TDDLL_ERAY10_CtrlStateList[1];
```

The one (and only) ERAY10 FlexRay CC is mapped to the memory base address 0x80002800. There is no hardware reset provided for the ERAY10 FlexRay controller. Therefore the reset function pointer has to be set to **NULL**. The absolute number of ERAY10 FlexRay CCs is defined in the constant variable **TDDLL_ERAY10_NumCtrl.**

Please be aware that there is also some RAM required to store the actual COMMSTACK state for each FlexRay CC used. Therefore an array (the array-size equals to the number of FlexRay CCs of the corresponding type) owning the state variable has to be allocated for each FlexRay CC type. The symbols allocating that memory are named **TDDLL_MFR4200_CtrlStateList** and **TDDLL_ERAY10_CtrlStateList** respectively.

# Document Information

| Date | Version | Author | Changes |
|---|---|---|---|
| 30.05.2005 | 0.1.0 | Markus Eggenbauer | Initial version, created based on doxygen output |
| 22.06.2005 | 0.2.0 | Markus Eggenbauer | Updated according to final source code version |
| 11.07.2005 | 0.9.0 | Michael Ziehensack | Finalized for preliminary release version (format, spell check, …) |
| 18.08.2005 | 0.9.1 | Markus Eggenbauer | Fixed minor mistypings.<br>Added API function TDDLL_IsSync.<br>Changed FrameID-based API functions.<br>Changed version information from 1.4.0 to 1.6 |
| 18.08.2005 | 0.9.2 | Markus Eggenbauer | Added some chapter headings for structural design. |
| 06.09.2005 | 0.9.3 | Gerald Freiberger | Added description on COMMSTACK application configuration with DESIGNER PRO.<br>Added index.<br>General updates. |
| 24.10.2005 | 1.6.0 | Markus Eggenbauer | Added section COMMSTACK configuration.<br>Changed versioning according to COMMSTACk versioning system. |
| 6.12.2005 | 1.6.1 | Markus Eggenbauer | Added notes about MFR4300 implementation and configuration. Implemented review remarks from FHOL. |
| 31.12.2006 | 1.6.2 | Markus Eggenbauer | Added notes about PHIP1 implementation and configuration. |
| 27.2.2006 | 1.6.4 | Markus Eggenbauer | -) Moved document source to OpenOffice. |
| 28.2.2006 | 1.6.5 | Markus Eggenbauer | -) Added “Abort” transition from state “Config” to state “Off”. |
| 13.3.2006 | 1.6.6 | Markus Eggenbauer | -) Added return code TDDLL_E_ACCESS to API function TDDLL_CheckTxFraemByID(). |
| 24.3.2006 | 1.8.0 | Markus Eggenbauer | Document Update to COMMSTACK 1.8. |

# 7 Index