FlexRay UNIFIED Driver

User Guide

Based on
MFR4300, MC9S12XFR128, MPC5567,
MPC5561, MFR4310, MPC5510, MPC560xP
and MC9S12XF512/384/256/128

UG
Rev. 1.3.8
01/2010

freescale.com

freescale™
semiconductor

FlexRay Freescale UNIFIED Driver

User Guide

by:

David Svrcek, david.svrcek@freescale.com

Roznov Czech System Center
Roznov pod Radhostem, Czech Republic

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.freescale.com

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision History

Date	Revision Level	Description	Page Number(s)
September 2006	1.0	Initial Release	132
October 2006	1.1	Corrected tables in Chapter 3 const modifiers added into Chapter 2, Structure and Configuration of the FlexRay UNIFIED Driver Chapter 2.2.1.1. added, Compiler Modifications in the *Fr_UNIFIED_types.h* file	134
November 2006	1.2	Cross-References modified to support all new FlexRay devices Chapter 2.2.2.1 Fr_HW_config_type modified to support the FlexRay UNIFIED Driver changes Optimization possibility of the RAM memory consumption added - new chapter 2.2.1.2 Memory Consumption Optimization Possibility in the Fr_UNIFIED_cfg.h File New Low Level Access Support functions added in the 2.1.8.2 Low Level Access Support for the FlexRay memory and chapter 3 The definition of the type Fr_FIFO_status_type changed, description in the chapter 2.1.3 Data Transmission and Reception Support modified - Chapter 3.1.3 Fr_buffers_init modified - the function returns the state FR_NOT_SUCCESS - Chapters 2.2.2.3 Fr_buffer_info_type and Fr_index_selector_type, 2.2.2.4 Fr_transmit_buffer_config_type, 2.2.2.5 Fr_receive_buffer_config_type and 2.2.2.6 Fr_FIFO_config_type and Fr_FIFO_range_filters_type modified to support changes in the Fr_buffers_init() function Chapter 2.2.3 UNIFIED Driver Performance Data added	142
March 2007	1.3	Table 2-2 The Fr_HW_config_type Member Description modified - added type Fr_single_channel_mode_type and description for timeout parameter changed Description of new type added into the Chapter 3.9.29 Fr_single_channel_mode_type Target Device chapter changed Table 2-3 The Fr_low_level_config_type Member Description, P_CHANNELS item changed Suggestion reading chapter extended 2.2.1.1 Compiler Modifications in the Fr_UNIFIED_types.h File chapter modified - added the reference to the documentation Chapter 3.9.18 Fr_connected_HW_type changed	142
March 2007	1.3.1	Table 2-2 The Fr_connected_HW_type Member Description modified - added the MPC5514 parameter Suggestion reading chapter extended New type added into the Chapter 3.9.18 Fr_connected_HW_type	142
August 2007	1.3.2	Target Device chapter on page 8 modified	142
October 2008	1.3.3	Target Device chapter on page 8 modified	142

Revision History

Date	Revision Level	Description	Page Number(s)
October 2008	1.3.4	Suggested Reading chapter on page 8 modified Target Device chapter on page 9 modified Table 2-2 The Fr_HW_config_type Member Description on page 27 updated Table 2-1UNIFIED Driver CPU usage on page 57 modified Chapter 3.2.6 Function return value FR_POCSTATE_UNKNOWN added Chapter 3.2.11 Function return value FR_MTS_UNKNOWN added Chapter 3.9.5 FR_POCSTATE_UNKNOWN added to Fr_POC_state_type Chapter 3.9.12 FR_MTS_UNKNOWN added to Fr_MTS_state_type Chapter 3.9.18 FR_MPC551x replaced by FR_MPC5510_0M22M and FR_MPC5510_0M76F in Fr_connected_HW_type	142
December 2008	1.3.5	New target device on page 9 added	142
March 2009	1.3.6	New GreenHills compiler version 5.1.6 added on page 9.	142
April 2009	1.3.7	2.2.1.1 Compiler Modifications in the Fr_UNIFIED_types.h File chapter modified (PTR2FARDATA and PTR2FARREG are now used)	142
January 2010	1.3.8	New supported compiler (CW 2.5) added on page 9.	142

About this Document

This document describes utilization of the FlexRay UNIFIED Driver. The driver package is distributed together with driver source code, two application examples and documentation. This document contains description of the driver functionality and each API function.

Audience

This document targets software developers of X-by-Wire projects based on the FlexRay communication protocol.

Suggested Reading

[1] FlexRay Protocol Specification V2.1

[2] One of following documents according used FlexRay module:
MFR4300 Block Guide, Freescale Semiconductor
MPC5567 Reference Manual, Freescale Semiconductor
MC9S12XFR128 Data Sheet, Freescale Semiconductor
MPC5561 Reference Manual, Freescale Semiconductor
MFR4310 Block Guide, Freescale Semiconductor
MCU9S12XF-Family Reference Manual, Freescale Semiconductor
MPC5516 Reference Manual, Freescale Semiconductor
MPC5514 Reference Manual, Freescale Semiconductor
MPC5517 Reference Manual, Freescale Semiconductor
MPC560XP Microcontroller Reference Manual, Freescale Semiconductor

[3] FlexRay UNIFIED Driver Application Examples User Guide, Freescale Semiconductor

[4] HC(S)12 Compiler Manual, Freescale Semiconductor

Definitions, Acronyms and Abbreviations

The following list defines the acronyms and abbreviations used in this document.

API .. ..Application Programming Interface

MT... .....Macrotick

CC .. ..Communication Controller

POC .....Protocol Operation Control

MTS .....Media Access Test Symbol

MB .. ..Message Buffer

CHI.. .....Controller Host Interface

CW.. .....CodeWarrior™

NIT .. ..Network Idle Time

Target Device

The FlexRay UNIFIED Driver source code were developed and tested on the following platforms:

- Freescale MPC5567DEMO board with embedded FlexRay module (MPC5567 Rev 0 or Rev A), using
  - CW Development Studio for Freescale MPC5500 version 1.5
  - CW Development Studio for Freescale MPC5500 version 2.3
  - GreenHills MULTI Project Builder for Freescale MPC5500 version 4.2.1
  - WindRiver DIAB C Compiler version 5.2.1.0
- Freescale MPC5561DEMO board with embedded FlexRay module, using
  - CW Development Studio for Freescale MPC5500 version 1.5
  - GreenHills MULTI Project Builder for Freescale MPC5500 version 4.2.1
  - WindRiver DIAB C Compiler version 5.2.1.0
- Freescale MPC5554DEMO board with the FlexRay MPC55xx adapter board and the MFR4300, MFR4310 Freescale standalone FlexRay Communication Controllers, using
  - CW Development Studio for Freescale MPC5500 version 1.5
  - GreenHills MULTI Project Builder for Freescale MPC5500 version 4.2.1
  - WindRiver DIAB C Compiler version 5.2.1.0
- SofTec HCS12X Starter Kit with the MC9S12XDP512 microcontroller and the MFR4300, MFR4310 Freescale standalone FlexRay Communication Controllers, using
  - CW Development Studio for Freescale MC9S12 version 4.5
- MC9S12XFR128 EVB with embedded FlexRay module, using
  - CW Development Studio for Freescale MC9S12 version 4.5
- Freescale MPC5567DEMO board with connected MPC5561 microcontroller with embedded FlexRay module, using
  - CW Development Studio for Freescale MPC5500 version 1.5
  - GreenHills MULTI Project Builder for Freescale MPC5500 version 4.2.1
  - WindRiver DIAB C Compiler version 5.2.1.0
- Freescale MPC5516 EVALUATION BOARD board with embedded FlexRay module, using
  - WindRiver DIAB C Compiler version 5.2.1.0
- EVBS12XF512E board with embedded FlexRay module, using
  - CW Development Studio for Freescale MC9S12 version 4.6.
- Freescale MPC5516 EVALUATION BOARD board with connected MPC5517 microcontroller with embedded FlexRay module, using
  - CW Development Studio for Freescale MPC5500 version 2.3
- Freescale XPC56XX EVB Motherboard with XPC560P 144LQFP Mini-module with connected MPC5604P microcontroller with embedded FlexRay module, using
  - GreenHills MULTI Project Builder for Freescale MPC5500 version 5.0.5
  - GreenHills MULTI Project Builder for Freescale PowerPC version 5.1.6
  - CW Development Studio for Freescale for MPC5500 version 2.5

Chapter 1 Introduction

The FlexRay UNIFIED Driver isolates the hardware specific functionality into a set of driver functions with a defined Application Program Interface (API), and provides hardware independency for user applications. The FlexRay UNIFIED Driver covers most of the FlexRay module features by the available functions, and provides a possible method (called Low Level Access Support) of completely covering the FlexRay module functionality.

The FlexRay UNIFIED Driver package is distributed with two application examples to show possible usage of the FlexRay driver.

The FlexRay cluster connectivity based on the FlexRay UNIFIED Driver can be established using various Freescale FlexRay Communication Controllers, like the MFR4300 and MFR4310 standalone FlexRay communication controllers or the MC9S12XFR128, MPC5567 (Rev 0 or Rev A), MPC5561, MPC5514, MPC5516, MPC5517, MPC560xP and MC9S12XF512/384/256/128 microcontrollers with integrated FlexRay modules.

This document is logically divided into two parts. The first part describes usage of the API function, the structure of the FlexRay UNIFIED Driver and the setting of the FlexRay configuration parameters. The FlexRay UNIFIED Driver covers the following FlexRay module functionality

- initialization and configuration functions
- wakeup, startup, the Media Access Test Symbol (MTS) and Change Mode functions
- data transmission and reception support (supports transmit single and double message buffers, receive message buffers and receive FIFO)
- poll driven mode support
- status monitoring support
- interrupt support
- timer support
- Low Level Access support.

The second part contains the specification of each function of the driver and also of the types. It gives a description of each function, its arguments, returned values and shows an example of its use.

Chapter 2 Functionality Description

This chapter describes the use of the FlexRay UNIFIED Driver in the first part, and the structure and configuration of the driver in the second part.

The FlexRay UNIFIED Driver basically implements two approaches to return function parameters/output values.

1. A driver function **returns a standard *Fr_return_type* type** (see section 3.9 Type Definitions for more information about defined types).

   Conditions:

   - the function API contains more output parameters

   - and/or the function should return a value of the type *Fr_return_type* to determine whether the function has been correctly processed or not.

2. A driver function **returns a required value.**

   Conditions:

   - the function has a limited number of output parameters

   - and it is not necessary to return a value of the type *Fr_return_type* to determine whether the function has been correctly processed or not.

In the first case, the required output value is stored in the memory location given by the pointer (to a user defined variable/array). Furthermore, the function returns information on whether the function has been correctly processed or not.

All initialization functions have the input parameters for configuration structure passing. The host has to put the addresses of the relevant configuration structures into these input parameters.

In the second case, one of the required output values is returned directly by the output return parameter of the related function. This approach is typically used by Poll Driven Support functions.

2.1 Basic Operation

2.1.1 Module Initialization and Configuration

To connect to a FlexRay cluster and communicate with others FlexRay module, it is necessary to properly configured the FlexRay module.The host should define necessary instances of the configuration structures in the relevant *Fr_UNIFIED_cfg.c* file for correct operation of the FlexRay UNIFIED Driver. See section 2.2.2 Configuration of the FlexRay UNIFIED Driver for more information regarding driver configuration.

The host application may execute basic configuration of the FlexRay module by calling two following functions:

- the ***Fr_init()*** function with two input parameters. It is the first function which must be called after a hard reset of the FlexRay module, and it stores the base addresses of the FlexRay module and available memories, configures the FlexRay channels and forces the module into the *POC:config* state

NOTE

*The **Fr_init()** function should be called only once during runtime, even if a FlexRay module is reconfigured*

- the ***Fr_set_configuration()*** function. This function initializes the data structures used by the driver, configures low level parameters (e.g. cluster configuration), and disables all FlexRay interrupts.

All necessary message buffers may be configured by calling the ***Fr_buffers_init()*** function according to given configuration structures. Configurations of the receive shadow message buffers and receive FIFO are also included. For each message buffer, the function determines which data segment is used and calculates the Data Field Offsets. The FlexRay UNIFIED Driver support memory model is described in the [FLEXRAY_MODULE], chapter FlexRay Memory Layout. An internal callback function pointers driver structure is also cleared in this function. See section 2.1.6 Interrupt Support for more information about callback functions and interrupts.

The ***Fr_timers_init()*** function initializes the absolute and/or relative timers.

The ***Fr_slot_status_init()*** function may be called to initialize the slot status functionality.

The last function is related to the configuration functions is ***Fr_slot_status_counter_init()***. With the use of this function, the host application can set the slot status counters.

NOTE

The FlexRay module can be easily reconfigured by repeated calling the previous functions (except **Fr_init()**). However, before that, the FlexRay module must be in the *POC:config* state. The host may use the **Fr_enter_configuration_mode()** function to force the FlexRay module into the *POC:config* state.

2.1.2 Startup, the MTS and Change Mode Functions

The FlexRay module still remains in the *POC:config* state after configuration by the previously described routines. It is convenient to call the **Fr_leave_configuration_mode()** function to exit the *POC:config* state. The FlexRay module may proceed to the *POC:ready* state using this function. The FlexRay UNIFIED Driver issues the CONFIG_COMPLETE command and the FlexRay module shall proceeds to the *POC:ready* state.

The host may initiate the cluster *wakeup* by calling the ***Fr_send_wakeup()*** routine. Before that, the host should check the *wakeup* status of the FlexRay module via the ***Fr_get_wakeup_state()*** function, to see if a *wakeup* pattern has been received.

Once the FlexRay module is in the *POC:ready* state, the ***Fr_start_communication()*** function can be used for initialization of a transition from the *POC:ready* to the *POC:startup* state. If the start up is successful, the *POC:normal active* state is automatically reached.

The host may use the ***Fr_stop_communication()*** function to move to the *POC:halt* state. The FlexRay UNIFIED Driver issues the FREEZE or HALT command for either an immediate move or a move at the end of the current communication cycle, to the *POC:halt* state.

For an immediate reset of the FlexRay module, it is also possible to use the ***Fr_reset_protocol_engine()*** routine. The FlexRay UNIFIED Driver initiates a transition into the *POC:default config* state after a RESET command sending.

To support Media Access Test Symbol (MTS) functionality, the FlexRay UNIFIED Driver implements two routines. The ***Fr_send_MTS()*** function for configuration and enabling MTS on a specified channel, and the ***Fr_get_MTS_state()*** routine to retrieve the MTS state on a specified channel.

To the Startup, MTS and Change Mode Functions, there are also two related functions which retrieve the current state of the FlexRay module. The ***Fr_get_POC_state()*** function queries the current value of the POC state of the FlexRay module, and the ***Fr_get_sync_state()*** routine queries whether or not the FlexRay module is synchronous to the rest of the cluster.

2.1.3 Data Transmission and Reception Support

The host application can configure the transmit, receive, receive FIFO and shadow message buffers by using the ***Fr_buffers_init()*** function. Using this function, the host application can configure/reconfigure a buffer during the *POC:config* state of the FlexRay module.

For operation in the *POC:normal active* state, the following functions are defined:

- the ***Fr_transmit_data()*** function updates a transmit message buffer with new data and clears any pending interrupt flag. This function tries to lock the appropriate message buffer (the number is passed in the first input parameter), and if the locking is successful, it updates the message buffer Data Area with new data (the second input parameter should contain the address of a host application array with data). If the message buffer has not been successfully locked, the *FR_TXMB_NO_ACCESS* value is returned, otherwise the FR_TXMB_UPDATED value is returned by the function.
  The interrupt flag is cleared even if the message buffer has not been successfully locked
- the ***Fr_receive_data()*** function copies a received data from the receive message buffer Data Area to the referenced data array (in the second input parameter). The FlexRay UNIFIED Driver returns the following values:
  – the *FR_RXMB_RECEIVED* value - if new data has been successfully received into the appropriate message buffer (the number is passed in the first input parameter), and the message buffer has been locked, the data is copied from the message buffer Data Area to the host data array (the second input parameter should contain the address of a host application data array)
  – the *FR_RXMB_NO_ACCESS* value - in the case where the message buffer has not been successfully locked (data has not been copied)
  – the *FR_RXMB_NOT_RECEIVED* value - new data has not been received
  – the *FR_RXMB_NULL_FRAME_RECEIVED* value - a valid null frame has been received.

  The interrupt flag is cleared even if the message buffer has not been successfully locked.
  This function also stores the length and slot status of the received frame to memory locations passed in output parameters

- the ***Fr_receive_fifo_data()*** function copies received data from the receive FIFO buffer Data Area to the referenced data array (in the second input parameter). The FlexRay UNIFIED Driver returns the following values:
  – the *FR_FIFO_RECEIVED* value - if new data has been successfully received into appropriate FIFO entry (the index of the FIFO entry is passed in the first input parameter), the data is copied from the FIFO Data Area to the host data array (the second input parameter should contain the address of a host application data array)
  – the *FR_FIFO_NOT_RECEIVED* value - new data has not been received

  No interrupt flag is cleared by this function. Please, see section 2.1.6 Interrupt Support for more information on FIFO interrupts.
  This function also stores the length, slot status and frame ID of the received frame to memory locations passed in the output parameters.

2.1.4 Poll Driven Mode Support

The host can use some of the created Poll Driven Mode Support functions to check various statuses, errors, global cluster time and also network management vectors. Since some events are indicated by flags in relevant FlexRay module registers, the FlexRay UNIFIED Driver clears these flags.

The host application can use:

- the ***Fr_check_tx_status()*** function to check whether the transmission of the transmit message buffer referenced by the first input parameter has been performed. If a double buffered message buffer is configured, the FlexRay UNIFIED Driver allocates two message buffers (in the ***Fr_buffers_init()*** function), one for the commit side (with an even, lower number) and one for the transmit side (with an odd, higher number). The host must always pass the number of the commit side into the first parameter of the ***Fr_check_tx_status()*** function
- the ***Fr_check_rx_status()*** function to check whether the reception of the frame has been performed. The FlexRay UNIFIED Driver decides which message buffer is checked according to the single input parameter
- the ***Fr_check_CHI_error()*** function to check whether the CHI related error flags have been triggered.
  The function returns the current content of the CHIERFR register (16-bit format). It is up to the host application to deal with possible error flag(s).
  For more information on the CHIERFR register, please see [FLEXRAY_MODULE].
  The function clears any pending flags (in the CHIERFR register)
- the ***Fr_check_cycle_start()*** function to check whether the FlexRay communication cycle has already been started. The function returns a boolean TRUE value if the cycle has already started. However, if the host application calls this function at the end of a communication cycle and the flag has not yet been cleared in the current communication cycle, the function also returns a TRUE value.
  The function clears any pending flag and can be used for synchronisation with the FlexRay global time without using an interrupt (its use is shown in [UNIFIED_APP_EXAMPLES])
- the ***Fr_check_transmission_across_boundary()*** function to check whether the frame transmission across boundary error flag has been set.
  The function clears any pending flag

- the ***Fr_check_violation()*** function to check whether a frame transmission in a dynamic segment has exceeded the dynamic segment boundary.
  The function clears any pending flag
- the ***Fr_check_max_sync_frame()*** function to check whether the number of synchronization frames detected in the current communication cycle has exceeded a configured value.
  The function clears any pending flag
- the ***Fr_check_clock_correction_limit_reached()*** function to check whether the internal calculated values have reached or exceeded the configured thresholds.
  The function clears any pending flag
- the ***Fr_check_missing_offset_correction()*** function to check whether an insufficient number of measurements are available for offset correction.
  The function clears any pending flag
- the ***Fr_check_missing_rate_correction()*** function to check whether an insufficient number of measurements are available for rate correction.
  The function clears any pending flag
- the ***Fr_check_coldstart_abort()*** function to check whether the configured number of allowed cold start attempts has been reached and that none of these attempts were successful.
  This function clears any pending flag
- the ***Fr_check_internal_protocol_error()*** function to check whether the protocol engine has detected an internal protocol error. In this case, the protocol engine goes into the *POC:halt* state immediately. An internal protocol error occurs when the protocol engine has not finished a calculation and a new calculation has been requested.
  For subsequent FlexRay communication, a re-configuration of the FlexRay module is necessary.
  The function clears any pending flag
- the ***Fr_check_fatal_protocol_error()*** function to check whether the protocol engine has detected a fatal protocol error. In this case, the protocol engine goes into the *POC:halt* state immediately. The fatal protocol error is either a *pLatestTx* violation as described in the MAC process of the [FR_PROTOCOL], or a transmission across slot boundary violation as described in the FSP process of the [FR_PROTOCOL].
  For subsequent FlexRay communication, a re-configuration of the FlexRay module is necessary.
  The function clears any pending flag
- the ***Fr_check_protocol_state_changed()*** function to check whether a protocol state has been changed. It is convenient to use this function (also in interrupt driven mode) to detect of a POC state change and the subsequent re-configuration of the Flexray module.
  The function clears any pending flag
- the ***Fr_check_protocol_engine_com_failure()*** function to check whether the protocol engine communication failure has been detected. A protocol engine communication failure occurs when there is a communication failure between the protocol engine and the controller host interface.
  The function clears any pending flag
- the ***Fr_get_global_time()*** function to query the global time of the FlexRay cluster - current cycle and macrotick values. The values are stored in a memory locations referenced by the output parameters
- the ***Fr_get_network_management_vector()*** function to query the network management vectors. The vectors are stored in a host application array referenced the output parameter.

NOTE

It is convenient to use this FlexRay module functionality to detect which node in the FlexRay cluster is connected. See [UNIFIED_APP_EXAMPLES] for more information.

2.1.5 Status Monitoring Support

The FlexRay module provides four types of status monitoring. All four monitor types use the same status vector provided by the protocol engine, for each slot in the static segment, for each slot in the dynamic segment, for the symbol window, and for the NIT, on a per channel basis. The content and the meaning of the status vector is given in [FLEXRAY_MODULE] chapter, Status Monitoring. The protocol engine provides the status vector after the end of the related slot/window/NIT.

The FlexRay UNIFIED Driver implements all four types of status monitoring.

2.1.5.1 Channel Status Error Counter Registers

The FlexRay module provides two channel error counter registers, *Channel A Status Error Counter Register (CASERCR)* and *Channel B Status Error Counter Register (CBSERCR)*.

The host application can use the ***Fr_get_channel_status_error_counter_value()*** function to query the current value of the channel status error counter for required channel.
The protocol engine generates a status vector for each static slot, each dynamic slot, the symbol window, and the NIT. The status vector contains the four protocol related error indicator bits. The FlexRay module increments the error status counter by 1 if, for a slot or segment, at least one error bit is set to 1. The counter wraps around after it has reached the maximum value. See [FLEXRAY_MODULE], the *CASERCR* and *CBSERCR* registers for more information.

To observe the channel status error counter registers, it is not necessary to call any configuration function.

2.1.5.2 Slot Status Registers

The FlexRay module provides eight slot status registers *Slot Status Registers (SSR0 - SSR7)*. Each register can be used to observe a static slot, a dynamic slot, the symbol window or the NIT. The registers provides all the slot status, and received frame / received symbol related slot status information, with the exception of the *first valid* indicator for non-transmission slots.

Although the FlexRay module provides eight slot status registers, they are grouped in four pairs. Each pair can be assigned to only one slot. One register from each pair is assigned to the even communication cycle, the second register to the odd communication cycle. Moreover, each of the eight registers contains slot status information for both channels; slot status information for channel A is stored in the eight lower bits (7:0), for channel B in the eight higher bits (15:8).

The FlexRay UNIFIED Driver simplifies the use of the slot status registers. The host application needs not deal with odd or even communication cycles as, the FlexRay UNIFIED Driver transfers them into a time dimension. This means that there are two possibilities:

- the newest updated slot status information is required from a given slot (the FR_SLOT_STATUS_CURRENT value is passed into the input function parameter). According to which time the service routine (the ***Fr_get_slot_status_reg_value()***) is called within the FlexRay communication cycle (cycle n), two situations can occur:

  - the service routine is called before the time when new slot status information is available for the given slot in the current communication cycle (cycle n). Then, the FlexRay UNIFIED Driver stores the slot status information for the given slot from the previous communication cycle (cycle n - 1)
  - the service routine is called and new slot status information is already available for the current communication cycle (cycle n). Then, the FlexRay UNIFIED Driver stores the slot status information for the given slot from the current communication cycle (cycle n)
- not the historic slot status information is required from the given slot (the FR_SLOT_STATUS_PREVIOUS value is passed into the input function parameter). According to which time the service routine (the ***Fr_get_slot_status_reg_value()***) is called within the FlexRay communication cycle (cycle n), two situations can occur:
  - the service routine is called before the time when new slot status information is available for the given slot in the current communication cycle (cycle n). Then, the FlexRay UNIFIED Driver stores the slot status information for the given slot from communication cycle before the last one (n - 2)
  - the service routine is called and new slot status information is already available for the current communication cycle (cycle n). Then, the FlexRay UNIFIED Driver stores the slot status information for the given slot from the previous communication cycle (cycle n - 1).

The FlexRay UNIFIED Driver also simplifies the operation of the slot status register numbers. The host application uses only the numbers of the required slot (for which the slot status information is configured) to serve the slot status registers, instead of operating with all eight slot status registers.

Before using the slot status registers, the host application must configure these registers by the ***Fr_slot_status_init()*** function.

During application runtime, the host application can call the ***Fr_get_slot_status_reg_value()*** function to get the status vector of the required slot for the static/dynamic segment, for the symbol window or for the NIT, on a per channel base.

The FlexRay UNIFIED Driver implements a slightly different approach to slot status observation for the slots in the static/dynamic segment and for the symbol window and the NIT.

- Slot status observation for slots in the static and dynamic segments -
  the slot status information is queried from the relevant slot status register
- Slot status observation of the symbol window and the NIT -
  for the symbol window or the NIT status vector observation, it is better to use the *PSR2* register (see [FLEXRAY_MODULE] for more information regarding this register)
  In this case, the ***Fr_get_slot_status_reg_value()*** function returns the content of the *PSR2* register instead of the relevant slot status register. The meaning of the bits in the returned slot status structure (in fact, the copied PSR2 register) is a little different to that of the standard slot status register.

NOTE

It is convenient to use the slot status functionality to determine which channel is connected into the FlexRay cluster. It is necessary to configure at least one slot status register to store the slot status information from some received frame (no MB needs to be assigned to this frame)

2.1.5.3 Slot Status Counter Registers

The FlexRay module provides four slot status error counter registers: *Slot Status Counter Registers (SSCR0–SSCR3)*. Each of these slot status counter registers is updated with the value of an internal slot status counter at the start of a communication cycle. The internal slot status counter is incremented if its increment condition, defined by the *Slot Status Counter Condition Register (SSCCR)*, matches the status vector provided by the PE. All static slots, the symbol window, and the NIT status are taken into account. Dynamic slots are excluded.

Before using the slot status counter registers, the host application must configure these by the ***Fr_slot_status_counter_init()*** function.

All slot status counter registers can be easily queried by the ***Fr_get_slot_status_counter_value()*** function, and the related slot status counter value is stored into a memory location given by the output function parameter.

If the counter is in the multicycle mode (i.e. *SSCCRn[MCY]* = 1), the application can reset the counter by calling the ***Fr_reset_slot_status_counter()*** function and waiting for the next cycle start, when the FlexRay module clears the counter. Subsequently, the counter can be set into the multicycle mode again by means of the ***Fr_slot_status_counter_init()*** function.

2.1.5.4 Message Buffer Slot Status Field

Each individual message buffer and each FIFO message buffer provides a slot status field, which provides information for the static/dynamic slot. The update conditions for the slot status field depend on the message buffer type.

The FlexRay UNIFIED Driver implements this status monitoring functionality via the ***Fr_receive_data()*** and the ***Fr_receive_fifo_data()*** functions. During each calling of these functions, the relevant slot status is stored into the memory location given by the output function parameter.

2.1.6 Interrupt Support

The FlexRay UNIFIED Driver provides a full set of functions to allow the host application to work in the interrupt drive and poll driven modes. To handle some events in the interrupt driven mode, the host application calls the corresponding ***Fr_set_xxx_callback()*** function, e.g. ***Fr_set_MB_callback()*** or ***Fr_set_protocol_0_IRQ_callback()***. Each of these functions accepts a pointer to the host application service routine that will handle the occurrence of the corresponding event. A prototype of each application callback function is defined in the *Fr_UNIFIED.h* file.

If the argument passed to the ***Fr_set_xxx_callback()*** function is not NULL, the function stores the passed reference in the internal driver *Fr_callback_functions* structure. When the corresponding event occurs, this pointer will be used to call the required host application service function. If the passed argument is NULL or the function has not been called yet, no service routine will be called when the corresponding interrupt occurs. In this case, it is assumed that the host application works in the poll driven mode for this event.

With the use of the ***Fr_set_xxx_callback()*** functions, the host application may change a handler for any given event. The host application can do this not only in the *POC:config* state, but also in the *POC:normal active* state.

Before using any FlexRay interrupt, the host application has to enable it by calling the ***Fr_enable_interrupts()*** function, which has three input parameters. Interrupt services provided by the

FlexRay UNIFIED Driver are strictly connected to the *Global Interrupt Flag and Enable Register (GIFER)*, the *Protocol Interrupt Enable Register 0 (PIER0),* and the *Protocol Interrupt Enable Register 1(PIER1)*. This means that the first parameter controls the interrupt request lines of the *GIFER* register, the second parameter controls the *PIER0* register, and the third parameter the *PIER1* register. See [FLEXRAY_MODULE] for more information regarding these registers and Interrupt Support descriptions.

When an interrupt arises, the ***Fr_interrupt_handler()*** routine should be called. If a callback function has been configured for the related interrupt (by the related ***Fr_set_xxx_callback()*** function), this is then called by the ***Fr_interrupt_handler()*** routine.

The host application should not perform any actions to clear any pending interrupt flag inside of the callback service routine. The FlexRay UNIFIED Driver clears any necessary flags itself (in the ***Fr_interrupt_handler()*** function). However, there is one exception. In the case where the host application operates with a double transmit message buffer and the interrupt is enabled at transmit side of double message buffer, it is necessary to call the ***Fr_clear_MB_interrupt_flag()*** function (with the index of the transmit side of the MB) after message buffer updating.

It is also possible to disable an interrupt by calling the ***Fr_disable_interrupts()*** function if necessary.

2.1.7 Timer Support

The FlexRay module provides two timers T1 and T2, which run on the FlexRay time base. Each of these timers generates a maskable interrupt when it reaches a configured point in time. The timer T1 is an absolute timer. The timer T2 can be configured as an absolute or a relative timer. Both timers can be configured to be repetitive or not. In the non-repetitive mode, the timer stops if it expires. In repetitive mode, it restarts itself when it expires.

Both timers run only when the protocol is in the *POC:normal active* or *POC:normal passive* state. If the protocol is not in one of these modes, the timers are stopped by the FlexRay module. The application must restart the timers when the protocol has reached the *POC:normal active* or *POC:normal passive* state.

The host application must initialize the timer(s) before utilisation by the ***Fr_timers_init()*** function. Once the timer is configured, the ***Fr_start_timer()*** function should be called to start the timer. It is also possible to stop the timer via the ***Fr_stop_timer()*** function when necessary.

NOTE

It is convenient to use the absolute timer for generating of the periodic interrupt in arbitrary time of the communication cycle. This interrupt can be used for, e.g. synchronisation with the FlexRay global time, appropriate message buffer operations, or other application code execution

2.1.8 Low Level Access Support

The FlexRay UNIFIED Driver covers most of FlexRay module features via available functions and provides a possible method (called Low Level Access Support) of covering the FlexRay module functionality completely.

The host application may use four functions for reading or writing from/to an arbitrary FlexRay register and FlexRay memory field without knowledge of the actual register address (e.g. base address of the FlexRay module or FlexRay memory).

2.1.8.1 Low Level Access Support for the FlexRay registers

The host application needs not deal with the address, as it is only necessary to use the abbreviation of the FlexRay register, which is defined in the FlexRay UNIFIED Driver.

The ***Fr_low_level_access_read_reg()*** function can be used to read of an arbitrary FlexRay register.

To write to an arbitrary register, the ***Fr_low_level_access_write_reg()*** function can be called with the abbreviation of the register and the required value to be written.

Both Low Level Access Support functions use 16-bits access.

2.1.8.2 Low Level Access Support for the FlexRay memory

To access to an memory field in the FlexRay memory, the ***Fr_low_level_access_read_memory()*** or ***Fr_low_level_access_write_memory()*** functions can be used.

These functions use 16-bit access, however the address of the required memory field has to be put in bytes, moreover the value needs to be even number. The value of the *Fr_memory_address* parameter is any offset of the corresponding FlexRay memory field with respect to the FlexRay memory base address as provided by the *CC_FlexRay_memory_base_address* parameter in the *Fr_HW_config_type* structure.

NOTE

Please, read carefully the Message Buffer Data Field Read Access and Message Buffer Data Field Write Access chapters in the [FLEXRAY_MODULE] before using the Low Level Access functions

2.2 Structure and Configuration of the FlexRay UNIFIED Driver

2.2.1 Structure of the FlexRay UNIFIED Driver

The FlexRay UNIFIED Driver consists of the following files:

- *Fr_UNIFIED.c* - the general source code (the FlexRay UNIFIED Driver routines)
- *Fr_UNIFIED.h* - the general header file with definition of the FlexRay registers, structures and C definitions
- *Fr_UNIFIED_types.h* - the definition of the compiler data types
- *Fr_UNIFIED_cfg.c* - the FlexRay module configuration data
- *Fr_UNIFIED_cfg.h* - the declaration of the variables/structures/constants which should be used by an external file (e.g. in the host application code) and the FlexRay UNIFIED Driver

The last three files can be modified by the host and it is convenient to create a copy of these files for each host application.

2.2.1.1 Compiler Modifications in the Fr_UNIFIED_types.h File

This section describes the changes in the driver code, which user should take care of them in order to access correctly the FlexRay device. Note that these changes are compiler and platform dependent.

For correct driver operation, the *Fr_UNIFIED_types.h* file should be modified.

There are two important defines, which specify whether or not the global instructions should be used for accessing the FlexRay registers and the FlexRay memory on the MC9S12X microcontrollers:

- PTR2FARREG - specifies appropriate access mode for the FlexRay registers
- PTR2FARDATA - specifies appropriate access mode for the FlexRay memory

The FlexRay UNIFIED Driver uses then the *__far* or *@far @gpage* pointer modifiers for the FlexRay registers and memory accessing (global addressing) if necessary. Possible combination are shown in Table 2-1 Combination of the far pointer modifiers.

Table 2-1 Combination of the far pointer modifiers

Define in the *Fr_UNIFIED_types.h* file	Possible combination for		
	MC9S12X Microcontroller + Freescale Standalone FlexRay Communication Controller (e.g. MFR4300)	**MC9S12X Microcontroller with integrated FlexRay module (e.g. MC9S12XFR128)**	**Others (e.g. MPC5567 or MPC55xx Microcontroller + MFR4300)**
PTR2FARREG (CodeWarrior)	* __far	- (empty)	- (empty)
PTR2FARDATA (CodeWarrior)	* __far	* __far	- (empty)
PTR2FARREG (Cosmic compiler)	@far @gpage *	- (empty)	- (empty)

Table 2-1 Combination of the far pointer modifiers

Define in the *Fr_UNIFIED_types.h* file	Possible combination for		
	MC9S12X Microcontroller + Freescale Standalone FlexRay Communication Controller (e.g. MFR4300)	MC9S12X Microcontroller with integrated FlexRay module (e.g. MC9S12XFR128)	Others (e.g. MPC5567 or MPC55xx Microcontroller + MFR4300)
PTR2FARDATA (Cosmic compiler)	@far @gpage *	@far @gpage *	- (empty)

Since the memory access configuration is strictly hardware dependant, see relevant documentation for more information. Mainly the [FLEXRAY_MODULE] (e.g. chapter Memory Mapping Control) and [HC(S)12 COMPILER], chapter HC(S)12 Backend.

2.2.1.2 Memory Consumption Optimization Possibility in the Fr_UNIFIED_cfg.h File

This section describes the possible change of the driver code to minimize the RAM memory consumption of the FlexRay UNIFIED Driver. Note that this change is application dependent.

To minimize the memory consumption of the FlexRay UNIFIED Driver, the host can modify the *Fr_UNIFIED_cfg.h* file. The FR_NUMBER_TXRX_MB parameter can be defined as a value of the highest used transmit or receive Message Buffer Index (the value of any *MBIDXRn* register), which is configured in an instance of the structure of type *Fr_buffer_info_type*.
The FlexRay UNIFIED Driver allocates then only the limited size of an internal message buffer configuration structure.

In case that the FR_NUMBER_TXRX_MB parameter is not defined, the FlexRay UNIFIED Driver allocates the maximum size of the internal message buffer configuration structure.

The value of the FR_NUMBER_TXRX_MB parameter must not be lower than the highest Message Buffer Index of any configured message buffer, otherwise the FlexRay UNIFIED Driver can access to a wrong memory location. However, there a verification is implemented in the driver for this case, then the ***Fr_buffers_init()*** function returns the FR_NOT_SUCCESS parameter.

In case that the highest Message Buffer Index belongs to double-buffered transmit message buffer, the FR_NUMBER_TXRX_MB parameter has to contain the value of transmit side of double message buffer (Message Buffer Index given in a instance of type Fr_buffer_info_type + 1) and it has to be odd number.

Example of configuration:

Please, see section 2.2.2 Configuration of the FlexRay UNIFIED Driver for detailed description of following configuration structures.

```
const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
{
    0, 2, 4, 5, FR_LAST_MB
};
const Fr_index_selector_type Fr_buffer_cfg_set_01[] =
```

```
{
    1, 3, 6, 8, 9, 10, FR_LAST_MB
};
const Fr_index_selector_type Fr_buffer_cfg_set_02[] =
{
    0, 4, 10, FR_LAST_MB
};

const Fr_buffer_info_type Fr_buffer_cfg_00[] =
{   /* Buffer type       Configuration structure ptr    MB index xx = configuration
                                                                    index used by
                                                                    Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_01_cfg,0},          // 00
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_03_cfg,2},          // 01
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_04_cfg,3},          // 02
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_02_cfg,0},          // 03
    {FR_RECEIVE_FIFO,    &Fr_FIFOA_cfg,            19},         // 04
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_01_cfg,     0},          // 05
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_07_cfg,4},          // 06
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_08_cfg,5},          // 07
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_09_cfg,6},          // 08
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_10_cfg,8},          // 09
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_02_cfg,     0},          // 10
};
```

Example of usage 1:

In the first example, only the following buffers will be configured:

- transmit message buffer with Message Buffer Index 0
- receive message buffer with Message Buffer Index 3
- FIFO A storage
- receive shadow buffers

The highest Message Buffer Index of transmit or receive message buffers is 3.

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);
if(return_value == FR_NOT_SUCCESS) // Is the FR_NUMBER_TXRX_MB param. correctly configured?
{
}

#define FR_NUMBER_TXRX_MB 3 // The driver will allocate only limited number of elements in
                               the internal configuration array
```

Example of usage 2:

In the second example, only the following buffers will be configured :

- receive message buffer with Message Buffer Index 2
- transmit message buffer with Message Buffer Index 0

- transmit message buffer with Message Buffer Index 4
- transmit message buffer with Message Buffer Index 6
- transmit message buffer with Message Buffer Index 8
- receive shadow buffers

The highest Message Buffer Index of transmit or receive message buffers is 8.

In case that the transmit message buffer with Message Buffer Index 8 is configured as double-buffered (it is not evident from given configuration structures), the value of the FR_NUMBER_TXRX_MB parameter has to be equal to 9.

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_01[0]);
if(return_value == FR_NOT_SUCCESS) // Is the FR_NUMBER_TXRX_MB param. correctly configured?
{
}

#define FR_NUMBER_TXRX_MB 8 // The highest Message Buffer Index belongs to single transmit
                               message buffer (or receive message buffer)
#define FR_NUMBER_TXRX_MB 9 // The highest Message Buffer Index belongs to double transmit
                               message buffer
```

Example of usage 3:

In the third example, only one transmit message buffer will be configured :

- transmit message buffer with Message Buffer Index 0 (e.g. single-buffered message buffer)
- FIFO A storage
- receive shadow buffers

The highest Message Buffer Index of transmit or receive message buffers is 0.

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_02[0]);
if(return_value == FR_NOT_SUCCESS) // Is the FR_NUMBER_TXRX_MB param. correctly configured?
{
}

#define FR_NUMBER_TXRX_MB 0 // The driver will allocate only limited number of elements in
                               the internal configuration array
```

2.2.2 Configuration of the FlexRay UNIFIED Driver

All configuration data for the FlexRay module should be stored in the instances of the configuration structures in the *Fr_UNIFIED_cfg.c* file. Each individual configuration structure is described below. The tables depicting the configuration structure members also show possible values in the *Example* column, and the registers which are directly influenced by the relevant configuration items in the *Direct Impact*

columns.

The concept of the configuration is that it is possible to have arbitrary number of instances of the configuration structures, however, the host application should contain the essential number of the instances which could be used during the host application runtime.
For example, three instances of the timer configuration structure are defined in case where the host application will not use more than three timer configurations during the application runtime.

2.2.2.1 Fr_HW_config_type

The type *Fr_HW_config_type* contains the FlexRay module specific parameters and also specifies which type of the FlexRay module is configured.

The instance of the *Fr_HW_config_type* structure is used by the ***Fr_init()*** and the ***Fr_set_configuration()*** functions.

Table 2-2 The Fr_HW_config_type Member Description

Structure item	Description	Range	Impact
CC_base_address	Base address of the FlexRay module. The value depends on the FlexRay module used	E.g. 0xFFFE0000 for the MPC5567, 0x000400 for the S12XFR128, 0x00140000 for the MFR4300/S12X	All
CC_FlexRay_memory_base_address	The base address of the FlexRay memory within the system memory	E.g. 0x40000000 for the MPC5567, 0x0FF000 for the S12XFR128, 0x00140800 for the MFR4300/S12X	SYMBADLR and SYMBADHR
connected_HW	The type of the Freescale FlexRay module	Fr_connected_HW_type (FR_MFR4300, FR_MC9S12XFR128, FR_MPC5567, FR_MFR4310, FR_MPC5561, FR_MPC5510_0M76F, FR_MC9S12XF, FR_MPC5567_REVA, FR_MPC5510_0M22M, FR_MPC560xP)	-
synchronization_filtering_enable	This item controls the filtering for received synchronisation frames - synchronisation frame filter enabled	boolean (TRUE or FALSE)	MCR: SFFE

Table 2-2 The Fr_HW_config_type Member Description

Structure item	Description	Range	Impact
clock_source	Protocol engine clock source select	Fr_clock_source_type (FR_INTERNAL_SYSTEM_ BUS_CLOCK or FR_EXTERNAL_ OSCILLATOR)	MCR: CLKSEL
prescaler_value	For the MPC5567, MC9S12XFR128, MPC5561: Protocol engine clock prescaler For the MC9S12XF, MFR4310, MPC5516, MPC5514, MPC5517, MPC560xP: FlexRay Channel Bit Rate Selection For the MFR4300: Not used	For the MPC5567, MC9S12XFR128, MPC5561: 000 up to 111 For the MC9S12XF, MFR310, MPC5517, MPC5516, MPC5514 or MPC560xP: 000 up to 011	MCR: PRESCALE or MCR: BITRATE
MB_segment_1_data_size	Message buffer segment 1 data size	0x00 up to 0x7F	MBDSR: MBSEG1DS
MB_segment_2_data_size	Message buffer segment 2 data size	0x00 up to 0x7F	MBDSR: MBSEG2DS
last_MB_seg_1	The message buffer number of the last individual message buffer that is assigned to the first message buffer segment. (The number of MB's in the first segment - 1)	0x00 up to 0x7F	MBSSUTR: LAST_MB_ SEG1
last_MB_util	The message buffer number of the last utilised individual message buffer (except FIFO). (The number of MB's in the first segment + the number of MB's in the second segment - 1)	0x00 up to 0x7F	MBSSUTR: LAST_MB_ UTIL
total_MB_number	Total number of used message buffers including FIFO. (last_MB_util + 1 + the number of FIFO)	0x01 up to 0x0400	-
allow_cold_start_enable	Capability of node to cold start cluster will be activated	boolean (TRUE or FALSE)	POCR: POCCMD

Table 2-2 The Fr_HW_config_type Member Description

Structure item	Description	Range	Impact
timeout	Not used for the MFR4300, MPC5567 (rev 0), MC9S12XFR128 and MFR4310 This value defines the maximum number of wait states on the system memory bus interface. This value must never exceeded in order to ensure no data are lost even under internal worst case conditions. The SYMATOR register is updated only if the value is not equal to 0	0 - register is not updated (value after reset is 0x0004) or 0x01 up to 0x1F	SYMATOR: TIMEOUT
sync_frame_table_offset	Definition of the FlexRay memory partition-related offset for Sync Frame Tables The SFTOR register is updated only if the value is not equal to 0	0 - register is not updated or 0x01 up to 0xFFFF	SFTOR: SFT_ OFFSET
single_channel_mode	Definition of channel device operation mode. Together with the P_CHANNELS parameter (in the structure of the type Fr_low_level_config_type) configure the assigment of the FlexRay ports and internal channels. For more information see a relevant documentation for the MCR register	Fr_single_channel_mode_type (FR_DUAL_CHANNEL_MODE or FR_SINGLE_CHANNEL_MODE)	MCR: SCM

Example of configuration:

The number of message buffers in segment 1 is 11, the number of message buffers in segment 2 is 8, the FIFO depth is 10

```
const Fr_HW_config_type Fr_HW_cfg_00 =
{
    0xFFFE0000,      // FlexRay module base address
    0x40000000,      // FlexRay memory base address (MB headers start at this address)
    FR_MPC5567,      // Type of Freescale FlexRay module
    FALSE,           // Synchronization filtering
    FR_INTERNAL_SYSTEM_BUS_CLOCK,
    0,               // Value of the PRESCALE or BITRATE bit field in the MCR register
    16,              // Data size - segment 1
    8,               // Data size - segment 2
    10,              // Last MB in segment 1 (Number of MB in Segment1 - 1)
```

```
    18,             // Last individual MB (except FIFO)
    29,               // Total number of used MB (Last_individual_MB + 1 + FIFO)
    TRUE,           // Allow coldstart
    0,              // The value of the TIMEOUT bit field in the SYMATOR register -
                        not implemented for all FlexRay modules
    0,              // Offset of the Sync Frame Table in the FlexRay memory
    FR_DUAL_CHANNEL_MODE     // Single channel mode disabled
};
```

Example of usage:

```
Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
```

2.2.2.2 Fr_low_level_config_type

The type *Fr_low_level_config_type* contains configuration information on the single configuration set of the FlexRay protocol specific parameters.
The configuration information corresponds strictly to the FlexRay Protocol Specification.

The FlexRay UNIFIED Driver transforms this configuration information and stores it into related configuration registers in the FlexRay module. For more details on the FlexRay related configuration parameters and the allowed parameter ranges, see [FR_PROTOCOL]. The *Description* column in Table 2-3 The Fr_low_level_config_type Member Description refers to the relevant parameter name used by the [FR_PROTOCOL].

The instance of the *Fr_low_level_config_type* structure is used by the ***Fr_init()*** and the ***Fr_set_configuration()*** functions.

Table 2-3 The Fr_low_level_config_type Member Description

Structure item	Description	Example	Unit	Impact
G_COLD_START_ATTEMPTS	gColdstartAttempts	10	number	PCR3
GD_ACTION_POINT_OFFSET	gdActionPointOffset	3	MT	PCR0, PCR6, PCR13
GD_CAS_RX_LOW_MAX	gdCASRxLowMax	83	*gdBit*	PCR4
GD_DYNAMIC_SLOT_IDLE_PHASE	gdDynamicSlotIdlePhase	0	minislot	PCR28
GD_MINISLOT	gdMinislot	40	MT	PCR2
GD_MINI_SLOT_ACTION_POINT_OFFSET	gdMinislotActionPointOffset	3	MT	PCR2, PCR3, PCR13

Table 2-3 The Fr_low_level_config_type Member Description

Structure item	Description	Example	Unit	Impact
GD_STATIC_SLOT	gdStaticSlot	50	MT	PCR0, PCR1, PCR13
GD_SYMBOL_WINDOW	gdSymbolWindow	13	MT	PCR6, PCR9
GD_TSS_TRANSMITTER	gdTSSTransmitter	11	*gdBit*	PCR5
GD_WAKEUP_SYMBOL_RX_IDLE	gdWakeupSymbolRxIdle	59	*gdBit*	PCR5
GD_WAKEUP_SYMBOL_RX_LOW	gdWakeupSymbolRxLow	50	*gdBit*	PCR3
GD_WAKEUP_SYMBOL_RX_ WINDOW	gdWakeupSymbolRxWindow	301	*gdBit*	PCR4
GD_WAKEUP_SYMBOL_TX_IDLE	gdWakeupSymbolTxIdle	180	*gdBit*	PCR8
GD_WAKEUP_SYMBOL_TX_LOW	gdWakeupSymbolTxLow	60	*gdBit*	PCR5
G_LISTEN_NOISE	gListenNoise	2	µT	PCR16, PCR17
G_MACRO_PER_CYCLE	gMacroPerCycle	5000	MT	PCR1, PCR10, PCR28
G_MAX_WITHOUT_CLOCK_ CORRECTION_PASSIVE	gMaxWithoutClockCorrectionPassive	10	cyclepairs	PCR8
G_MAX_WITHOUT_CLOCK_ CORRECTION_FATAL	gMaxWithoutClockCorrectionFatal	14	cyclepairs	PCR8
G_NUMBER_OF_MINISLOTS	gNumberOfMinislots	22	minislot	PCR9, PCR21, PCR29
G_NUMBER_OF_STATIC_SLOTS	gNumberOfStaticSlots	60	static slot	PCR2
G_OFFSET_CORRECTION_START	gOffsetCorrectionStart	4920	MT	PCR11, PCR28
G_PAYLOAD_LENGTH_STATIC	gPayloadLengthStatic	16	2-bytes	PCR19
G_SYNC_NODE_MAX	gSyncNodeMax	5	number	PCR30
G_NETWORK_MANAGEMENT_ VECTOR_LENGTH	gNetworkManagementVectorLength	2	byte	NMVLR
P_ALLOW_HALT_DUE_TO_CLOCK	pAllowHaltDueToClock	FALSE	bool	PCR26
P_ALLOW_PASSIVE_TO_ACTIVE	pAllowPassiveToActive	20	cyclepairs	PCR12
P_CHANNELS	pChannels	FR_CHANNEL_AB	A, B, A&B	MCR: CHA, CHB
PD_ACCEPTED_STARTUP_RANGE	pdAcceptedStartupRange	300	µT	PCR22, PCR26

Table 2-3 The Fr_low_level_config_type Member Description

Structure item	Description	Example	Unit	Impact
P_CLUSTER_DRIFT_DAMPING	pClusterDriftDamping	1	µT	PCR24
P_DECODING_CORRECTION	pDecodingCorrection	56	µT	PCR7, PCR19
P_DELAY_COMPENSATION_CHA	pDelayCompensation[A]	1	µT	PCR19, PCR22
P_DELAY_COMPENSATION_CHB	pDelayCompensation[B]	1	µT	PCR7, PCR26
PD_LISTEN_TIMEOUT	pdListenTimeout	401202	µT	PCR14, PCR15, PCR16, PCR17
PD_MAX_DRIFT	pdMaxDrift	601	µT	PCR24, PCR25, PCR26, PCR27
P_EXTERN_OFFSET_ CORRECTION	pExternOffsetCorrection	0	µT	PCR29
P_EXTERN_RATE_CORRECTION	pExternRateCorrection	0	µT	PCR21
P_KEY_SLOT_ID	pKeySlotId	4	number	PCR18
P_KEY_SLOT_USED_FOR_ STARTUP	pKeySlotUsedForStartup	TRUE	bool	PCR11
P_KEY_SLOT_USED_FOR_SYNC	pKeySlotUsedForSync	TRUE	bool	PCR11
P_KEY_SLOT_HEADER_CRC	header CRC for key slot	1747	number	PCR12
P_LATEST_TX	pLatestTx	21	minislot	PCR21
P_MACRO_INITIAL_OFFSET_A	pMacroInitialOffset[A]	5	MT	PCR6
P_MACRO_INITIAL_OFFSET_B	pMacroInitialOffset[B]	5	MT	PCR16
P_MICRO_INITIAL_OFFSET_A	pMicroInitialOffset[A]	23	µT	PCR20
P_MICRO_INITIAL_OFFSET_B	pMicroInitialOffset[B]	23	µT	PCR20
P_MICRO_PER_CYCLE	pMicroPerCycle	200000	µT	PCR22, PCR23, PCR24, PCR25, PCR26, PCR27
P_OFFSET_CORRECTION_OUT	pOffsetCorrectionOut	1201	µT	PCR9
P_RATE_CORRECTION_OUT	pRateCorrectionOut	600	µT	PCR14
P_SINGLE_SLOT_ENABLED	pSingleSlotEnabled	FALSE	bool	PCR10

Table 2-3 The Fr_low_level_config_type Member Description

Structure item	Description	Example	Unit	Impact
P_WAKEUP_CHANNEL	pWakeupChannel	FR_CHANNEL_A	A or B	PCR10
P_WAKEUP_PATTERN	pWakeupPattern	16	number	PCR18
P_MICRO_PER_MACRO_NOM	pMicroPerMacroNom	40	µT	PCR7
P_PAYLOAD_LENGTH_DYN_MAX	pPayloadLengthDynMax	8	2-bytes	PCR24

Example of configuration:

```
const Fr_low_level_config_type Fr_low_level_cfg_set_00 =
{
    10,          /* G_COLD_START_ATTEMPTS */
    3,           /* GD_ACTION_POINT_OFFSET */
    83,          /* GD_CAS_RX_LOW_MAX */
    0,           /* GD_DYNAMIC_SLOT_IDLE_PHASE */
    40,          /* GD_MINISLOT */
    3,           /* GD_MINI_SLOT_ACTION_POINT_OFFSET */
    50,          /* GD_STATIC_SLOT */
    13,          /* GD_SYMBOL_WINDOW */
    11,          /* GD_TSS_TRANSMITTER */
    59,          /* GD_WAKEUP_SYMBOL_RX_IDLE */
    50,          /* GD_WAKEUP_SYMBOL_RX_LOW */
    301,         /* GD_WAKEUP_SYMBOL_RX_WINDOW */
    180,         /* GD_WAKEUP_SYMBOL_TX_IDLE */
    60,          /* GD_WAKEUP_SYMBOL_TX_LOW */
    2,           /* G_LISTEN_NOISE */
    5000,        /* G_MACRO_PER_CYCLE */
    10,          /* G_MAX_WITHOUT_CLOCK_CORRECTION_PASSIVE */
    14,          /* G_MAX_WITHOUT_CLOCK_CORRECTION_FATAL */
    22,          /* G_NUMBER_OF_MINISLOTS */
    60,          /* G_NUMBER_OF_STATIC_SLOTS */
    4920,        /* G_OFFSET_CORRECTION_START */
    16,          /* G_PAYLOAD_LENGTH_STATIC */
    5,           /* G_SYNC_NODE_MAX */
    2,           /* G_NETWORK_MANAGEMENT_VECTOR_LENGTH */
    FALSE,       /* G_ALLOW_HALT_DUE_TO_CLOCK */
    20,          /* G_ALLOW_PASSIVE_TO_ACTIVE */
    FR_CHANNEL_AB,  /* P_CHANNELS */
    300,         /* PD_ACCEPTED_STARTUP_RANGE */
    1,           /* P_CLUSTER_DRIFT_DAMPING */
    56,          /* P_DECODING_CORRECTION */
    1,           /* P_DELAY_COMPENSATION_A */
    1,           /* P_DELAY_COMPENSATION_B */
    401202,      /* PD_LISTEN_TIMEOUT */
    601,         /* PD_MAX_DRIFT */
    0,           /* P_EXTERN_OFFSET_CORRECTION */
    0,           /* P_EXTERN_RATE_CORRECTION */
    1,           /* P_KEY_SLOT_ID */
    TRUE,        /* P_KEY_SLOT_USED_FOR_STARTUP */
    TRUE,        /* P_KEY_SLOT_USED_FOR_SYNC */
    242,         /* P_KEY_SLOT_HEADER_CRC */
    21,          /* P_LATEST_TX */
    5,           /* P_MACRO_INITIAL_OFFSET_A */
    5,           /* P_MACRO_INITIAL_OFFSET_B */
    23,          /* P_MICRO_INITIAL_OFFSET_A */
    23,          /* P_MICRO_INITIAL_OFFSET_B */
    200000,      /* P_MICRO_PER_CYCLE */
    1201,        /* P_OFFSET_CORRECTION_OUT */
    600,         /* P_RATE_CORRECTION_OUT */
    FALSE,       /* P_SINGLE_SLOT_ENABLED */
```

```
    FR_CHANNEL_A,    /* P_WAKEUP_CHANNEL */
    16,           /* P_WAKEUP_PATTERN */
    40,           /* P_MICRO_PER_MACRO_NOM */
    8           /* P_PAYLOAD_LENGTH_DYN_MAX */
};
```

Example of usage:

```
Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
```

2.2.2.3 Fr_buffer_info_type and Fr_index_selector_type

The host application can easily configure all buffers (including receive shadow buffers and receive FIFOs) by calling the ***Fr_buffers_init()*** function. This routine has two input parameters. The first refers to the buffer configuration structures (an instance of the type *Fr_buffer_info_type*) and the second refers to the index selector (an instance of the type *Fr_index_selector_type*).

The instance of the type *Fr_buffer_info_type* is an array of configuration structures. Each item in this array describes one buffer and defines:

- the type of the buffer (either transmit message buffer, receive message buffer, receive shadow buffer or receive FIFO)
- the address of the relevant configuration structure for each buffer (there should be a separate configuration structure defined with detailed buffer settings - see section 2.2.2.4 Fr_transmit_buffer_config_type, section 2.2.2.5 Fr_receive_buffer_config_type, section 2.2.2.6 Fr_FIFO_config_type and Fr_FIFO_range_filters_type or section 2.2.2.7 Fr_receive_shadow_buffers_config_type)
- buffer index - the representation of this value depends on the type of buffer:
  - for individual message buffers (transmit and receive) it indicates the index of the initially associated message buffer header field (Message Buffer Index)
  - for receive FIFO's it indicates the start index of the receive FIFO (the first FIFO buffer header field number)
  - for receive shadow buffers, it is not used (the host application can pass, e.g. a null value)

The instance of the type *Fr_index_selector_type* is an array of numbers. These define which buffer described in the instance of the type *Fr_buffer_info_type* will be configured. Each number corresponds to a relevant item of the *Fr_buffer_info_type* type array (individual items are counting from null value). This means that the number 0 corresponds to the first item of the array, number 1 to the second array item, and so on.

In the case where the host application defines one instance of the type *Fr_buffer_info_type* and several instances of the type *Fr_index_selector_type*, this concept allows an easy reconfiguration of the FlexRay module during the host application runtime by calling the ***Fr_buffers_init()*** function (with different instances of the type *Fr_index_selector_type*).

In summary, the instance of the type *Fr_buffer_info_type* should describe all the buffers which could be configured during the host application runtime. According to the *Fr_index_selector_type* type array, the FlexRay UNIFIED Driver selects which buffer will be configured.

NOTE

For correct operation, the instance of type Fr_index_selector_type must have the FR_LAST_MB (0xFF) value stored in the last item.

The structure of the *Fr_buffer_info_type* type is depicted in Figure 2-1. Description of the Fr_buffer_info_type type.

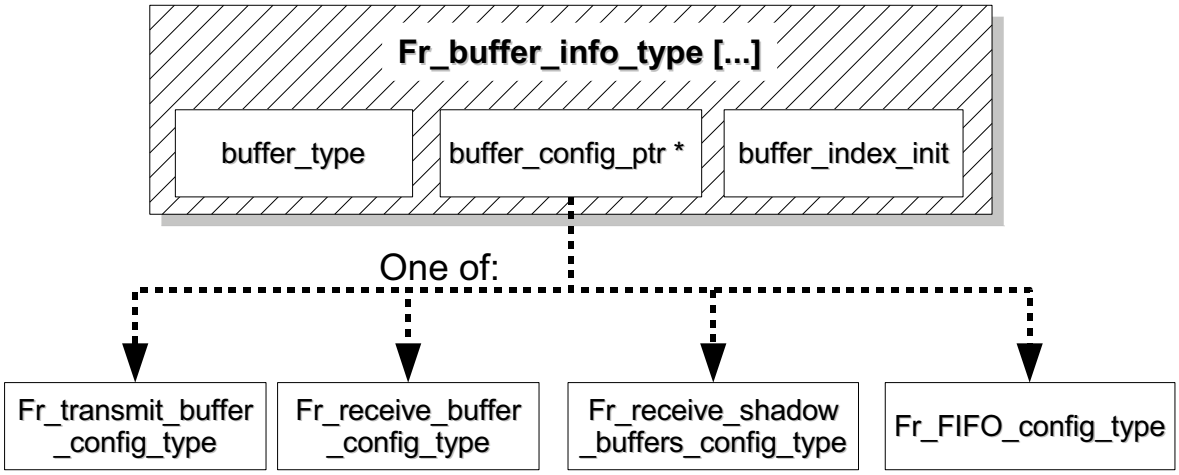

Figure 2-1. Description of the Fr_buffer_info_type type

Table 2-4 The Fr_buffer_info_type Member Description

Structure item	Description	Example	Type	Inpact
buffer_type	Determine which type of buffer is configured (transmit MB, receive MB, receive shadow buffers or receive FIFO)	FR_TRANSMIT_ BUFFER	Fr_buffer_type (FR_TRANSMIT_BUFFER, FR_RECEIVE_BUFFER, FR_RECEIVE_SHADOW, FR_RECEIVE_FIFO)	MBCCSRn: MTD *for individual MB*
buffer_config_ptr	Reference to relevant configuration structure, see Figure 2-1. Description of the Fr_buffer_info_type type	&Fr_rx_ shadow_cfg	void *	-

Table 2-4 The Fr_buffer_info_type Member Description

Structure item	Description	Example	Type	Inpact
buffer_index_init	- for individual message buffers: initial message buffer index - for receive FIFO: start index - for receive shadow buffers: not used	1	number (uint16)	MBIDXR *for individual MB or* RFSIR *for FIFO*

Example of configuration:

```
const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
{
    0, 2, 4, 5, FR_LAST_MB
};
const Fr_index_selector_type Fr_buffer_cfg_set_01[] =
{
    1, 3, 6, 8, 9, 10, FR_LAST_MB
};
const Fr_buffer_info_type Fr_buffer_cfg_00[] =
{   /* Buffer type       Configuration structure ptr    MB index xx = configuration
                                                        index used by
                                                        Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_01_cfg,0},    // 00
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_03_cfg,2},    // 01
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_04_cfg,3},    // 02
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_02_cfg,0},    // 03
    {FR_RECEIVE_FIFO,    &Fr_FIFOA_cfg,           19},    // 04
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_01_cfg,    0},    // 05
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_07_cfg,4},    // 06
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_08_cfg,5},    // 07
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_09_cfg,6},    // 08
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_10_cfg,7},    // 09
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_02_cfg,    0},    // 10
};
```

Example of usage 1:

In the first example, only the following buffers will be configured:

- transmit message buffer with initial buffer index 0 - *Fr_tx_buffer_slot_01_cfg* structure
- receive message buffer with initial buffer index 3 - *Fr_rx_buffer_slot_04_cfg* structure
- FIFO A storage with start index 19 - *Fr_FIFOA_cfg* structure
- receive shadow buffers - *Fr_rx_shadow_01_cfg* structure

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);
```

Example of usage 2:

In the second example, only the following buffers will be configured (e.g. after the FlexRay module reconfiguration):

- receive message buffer with initial buffer index 2 - *Fr_rx_buffer_slot_03_cfg* structure
- transmit message buffer with initial buffer index 0 - *Fr_tx_buffer_slot_02_cfg* structure
- transmit message buffer with initial buffer index 4 - *Fr_tx_buffer_slot_07_cfg* structure
- transmit message buffer with initial buffer index 6 - *Fr_tx_buffer_slot_09_cfg* structure
- transmit message buffer with initial buffer index 7- *Fr_tx_buffer_slot_10_cfg* structure
- receive shadow buffers - *Fr_rx_shadow_02_cfg* structure

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_01[0]);
```

2.2.2.4 Fr_transmit_buffer_config_type

The configuration information for each individual transmit message buffer (single or double buffered) is stored in a separate instance of the type *Fr_transmit_buffer_config_type*.

The address of each separate structure of the type *Fr_transmit_buffer_config_type* should be passed into the relevant *buffer_config_ptr* parameter of the configuration structure of the type *Fr_buffer_info_type*.

Table 2-5 The Fr_transmit_buffer_config_type Member Description

Structure item	Description	Range	Direct Impact
transmit_frame_ID	The slot in which the message buffer will be transmitted	0 up to 2047	MBFIDRn and Frame Header: FID
header_CRC	Header CRC	0 up to 0x03FF	Frame Header: HDCRC
payload_length	For a transmit message buffer assigned to the dynamic segment, this value is used to set the value of the *Payload length* field (16-bit units) in the transmitted frame	0 up to 127	Frame Header: PLDLEN
transmit_MB_buffering	Defines the buffering type	Fr_transmit_MB_type (FR_SINGLE_ TRANSMIT_BUFFER or FR_DOUBLE_ TRANSMIT_BUFFER)	MBCCSRn: MBT, Frame Header: FID

Table 2-5 The Fr_transmit_buffer_config_type Member Description

Structure item	Description	Range	Direct Impact
transmission_mode	Defines the transmission mode	Fr_transmission_type (FR_EVENT_ TRANSMISSION_ MODE or FR_STATE_ TRANSMISSION_ MODE)	MBCCFRn: MTM
transmission_commit_mode	Defines the commit mode. Applied only to a double buffered transmit receive buffer	Fr_transmission_commit_type (FR_STREAMING_ COMMIT_MODE or FR_IMMEDIATE_ COMMIT_MODE)	MBCCSRn: MCM
transmit_channel_enable	Defines the channel assignment and controls the transmit behaviour of the message buffer	Fr_channel_type (FR_CHANNEL_A or FR_CHANNEL_B or FR_CHANNEL_AB)	MBCCFRn: CHA, MBCCFRn: CHB
payload_preamble	The payload preamble indicator FALSE No network management vector or message ID in the frame payload data TRUE: **- Static Segment**: Frame payload data contains network management vector - **Dynamic Segment**: Frame payload data contains message ID	boolean (TRUE or FALSE)	Frame Header: PPI
tx_cycle_counter_filter_enable	FALSE: - Cycle counter filtering is disabled TRUE: - Cycle counter filtering is enabled	boolean (TRUE or FALSE)	MBCCFRn: CCFE
tx_cycle_counter_filter_value	Defines the filter value for the cycle counter filtering	0 up to 0x3F	MBCCFRn: CCFVAL
tx_cycle_counter_filter_mask	Defines the filter mask for the cycle counter filtering	0 up to 0x3F	MBCCFRn: CCFMSK

Table 2-5 The Fr_transmit_buffer_config_type Member Description

Structure item	Description	Range	Direct Impact
tx_MB_interrupt_enable	Defines whether the message buffer will generate an interrupt FALSE: - Interrupt request generation disabled TRUE: - Interrupt request generation enabled	boolean (TRUE or FALSE)	MBCCSRn: MBIE
tx_MB_interrupt_transmit_side_ enable	Applied only to a double buffered transmit receive buffer. Defines whether interrupt generation will be enabled at commit or transmit side FALSE: - Interrupt request generation enabled at commit side TRUE: - Interrupt request generation enabled at transmit side	boolean (TRUE or FALSE)	-

Example of configuration:

The transmit message buffer is configured as double buffered, in the state transmission mode, in streaming commit mode. No network management vector is enabled, frames will be transmitted over both channels in the first slot. Cycle counter filter is disabled. The interrupt generation is enabled on the transmit side of the double message buffer.

```
const Fr_transmit_buffer_config_type Fr_tx_buffer_slot_01_cfg =
{
   1,                            // Transmit frame ID
   242,                          // Header CRC
   16,                           // Payload length
   FR_DOUBLE_TRANSMIT_BUFFER,    // Transmit MB buffering
   FR_STATE_TRANSMISSION_MODE,   // Transmission mode
   FR_STREAMING_COMMIT_MODE,     // Transmission commit mode
   FR_CHANNEL_AB,                // Transmit channels
   FALSE,                        // Payload preamble
   FALSE,                        // Transmit cycle counter filter enable
   0,                            // Transmit cycle counter filter value
   0,                            // Transmit cycle counter filter mask
   TRUE,                         // Transmit MB interrupt enable
   TRUE                          // TRUE - interrupt is enabled at transmit side
};
```

```
const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
{
    0, FR_LAST_MB
};
const Fr_buffer_info_type Fr_buffer_cfg_00[] =
{   /* Buffer type       Configuration structure ptr    MB index xx = configuration
                                                                    index used by
                                                                    Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_01_cfg,     0},          // 00
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_cfg,             0},          // 01
};
```

Example of usage:

Only the transmit message buffer 0 will be configured (not the receive shadow buffers).

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);
```

2.2.2.5 Fr_receive_buffer_config_type

The configuration information for each individual receive message buffer is stored in a separate instance of the type *Fr_receive_buffer_config_type*.

The address of each separate structure of the type *Fr_receive_buffer_config_type* should be passed into the relevant *buffer_config_ptr* parameter of the configuration structure of the type *Fr_buffer_info_type*.

Table 2-6 The Fr_receive_buffer_config_type Member Description

Structure item	Description	Range	Direct Impact
receive_frame_ID	A filter value to determine whether or not the message buffer is used for reception of a message received in a slot whose slot ID equals the Frame ID (FID)	0 up to 2047	Frame Header: FID
receive_channel_enable	Defines the channel assignment and controls the receive behaviour of the message buffer	Fr_channel_type (FR_CHANNEL_A or FR_CHANNEL_B or FR_CHANNEL_AB)	MBCCFRn: CHA, MBCCFRn: CHB

Table 2-6 The Fr_receive_buffer_config_type Member Description

Structure item	Description	Range	Direct Impact
rx_cycle_counter_filter_enable	FALSE: - Cycle counter filtering is disabled TRUE: - Cycle counter filtering is enabled	boolean (TRUE or FALSE)	MBCCFRn: CCFE
rx_cycle_counter_filter_value	Defines the filter value for the cycle counter filtering	0 up to 0x3F	MBCCFRn: CCFVAL
rx_cycle_counter_filter_mask	Defines the filter mask for the cycle counter filtering	0 up to 0x3F	MBCCFRn: CCFMSK
rx_MB_interrupt_enable	Defines whether the message buffer will generate an interrupt FALSE: - Interrupt request generation disabled TRUE: - Interrupt request generation enabled	boolean (TRUE or FALSE)	MBCCSRn: MBIE

NOTE

For correct FlexRay module operation, the receive shadow buffers have to be configured together with the receive message buffers. See section 2.2.2.7 Fr_receive_shadow_buffers_config_type for more information regarding a receive shadow buffer.

Example of configuration:

The receive message buffer is configured to receive messages in slot 4 on channel A. Cycle counter filter is disabled. The interrupt generation is enabled.

```
const Fr_receive_buffer_config_type  Fr_rx_buffer_slot_04_cfg =
{
   4,                              // Receive frame ID
   FR_CHANNEL_A,                   // Receive channel enable
   FALSE,                          // Receive cycle counter filter enable
   0,                              // Receive cycle counter filter value
   0,                              // Receive cycle counter filter mask
   TRUE                            // Receive MB interrupt enable
};


const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
```

```
{
    1, 2, FR_LAST_MB
};
const Fr_buffer_info_type Fr_buffer_cfg_00[] =
{   /* Buffer type       Configuration structure ptr    MB index xx = configuration
                                                                    index used by
                                                                    Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_01_cfg,     0},         // 00
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_04_cfg,     3},         // 01
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_cfg,             0},         // 02
};
```

Example of usage:

The receive message buffer 3 will be configured together with the receive shadow buffers (the configuration structure for the receive shadow buffers is not presented in this example). See section 2.2.2.7 Fr_receive_shadow_buffers_config_type for more information regarding receive shadow buffers.

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);
```

2.2.2.6 Fr_FIFO_config_type and Fr_FIFO_range_filters_type

The configuration information for each individual receive FIFO (either FIFO A or FIFO B) is stored in a separate instance of the type *Fr_FIFO_config_type*.

The address of each separate structure of the type *Fr_FIFO_config_type* should be passed into the relevant *buffer_config_ptr* parameter of the configuration structure of the type *Fr_buffer_info_type*.

Table 2-7 The Fr_FIFO_config_type Member Description

Structure item	Description	Range	Direct Impact
FIFO_channel	Defines for which channel the receiver FIFO is configured	Fr_channel_type (FR_CHANNEL_A or FR_CHANNEL_B)	RFSR: SEL
FIFO_depth	Defines the depth of the selected receive FIFO, i.e. the number of entries	0 up to 255	RFDSR: FIFO_DEPTH
FIFO_entry_size	Defines the size of the frame data sections for the selected receive FIFO in 2 bytes entities	0 up to 127	RFDSR: ENTRY_SIZE
FIFO_message_ID_acceptance_filter_value	Defines the filter value for the message ID acceptance filter of the selected receive FIFO	0 up to 0xFFFF	RFMIDAFVR

Table 2-7 The Fr_FIFO_config_type Member Description

Structure item	Description	Range	Direct Impact
FIFO_message_ID_acceptance_ filter_mask	Defines the filter mask for the message ID acceptance filter of the selected receive FIFO	0 up to 0xFFFF	RFMIDAFMR
FIFO_frame_ID_rejection_filter_value	Defines the filter value for the message ID rejection filter of the selected receive FIFO	0 up to 0x07FF	RFFIDRFVR
FIFO_frame_ID_rejection_filter_mask	Defines the filter mask for the message ID rejection filter of the selected receive FIFO	0 up to 0x07FF	RFFIDRFMR
FIFO_range_filters_config[4]	An array with four elements (of type *Fr_FIFO_range_filters_type*). Each element describes one range filter configuration See Table 2-8 The Fr_FIFO_range_filters_type Member Description	See Table 2-8 The Fr_FIFO_range_filters_type Member Description	See Table 2-8 The Fr_FIFO_range_filters_type Member Description
FIFO_interrupt_enable	Defines whether the FIFO will generate an interrupt FALSE: - Interrupt request generation disabled TRUE: - Interrupt request generation enabled The FIFO interrupt can also be enabled by the ***Fr_enable_interrupts()*** function	boolean (TRUE or FALSE)	GIFER: FNEAIE, GIFER: FNEBIE

The *FIFO_range_filters_config[4]* array in the *Fr_FIFO_config_type* type structure describes the configuration ot the four range FIFO filters. This array is of type *Fr_FIFO_range_filters_type* and the structure is depicted in Table 2-8 The Fr_FIFO_range_filters_type Member Description.

Table 2-8 The Fr_FIFO_range_filters_type Member Description

Structure item	Description	Range	Direct Impact
range_filter_enable	Defines whether or not the selected range filter is enabled	boolean (TRUE or FALSE)	RFRFCTR: F0EN or, RFRFCTR: F1EN or, RFRFCTR: F2EN or, RFRFCTR: F3EN

Table 2-8 The Fr_FIFO_range_filters_type Member Description

Structure item	Description	Range	Direct Impact
range_filter_mode	Defines the filter mode of the selected frame ID range filter	Fr_FIFO_range_filter_mode_ type (FR_ACCEPTANCE, FR_REJECTION)	RFRFCTR: F0MD or, RFRFCTR: F1MD or, RFRFCTR: F2MD or, RFRFCTR: F3MD
range_filter_lower_interval	Defines the lower interval boundary frame ID value for the selected frame ID range filter	0 up to 2047	RFRFCFR: SID, RFRFCFR: IBD
range_filter_upper_interval	Defines the upper interval boundary frame ID value for the selected frame ID range filter	0 up to 2047	RFRFCFR: SID, RFRFCFR: IBD

NOTE

*The FIFO interrupt can also be enabled by the **Fr_enable_interrupts()** function.*

Example of configuration:

The receive FIFO is configured to receive frames in the frame ID range of 60 to 64. The depth of the FIFO is 10, entry size is 8 words (2-bytes entities). Receive FIFO interrupt is enabled. Only the first range filter is enabled.

```
const Fr_FIFO_config_type Fr_FIFOA_cfg =
{
    FR_CHANNEL_A,                          // FIFO channel
    10,                                    // FIFO depth
    8,                                     // FIFO entry size
    0,                                     // FIFO message ID acceptance filter value
    0,                                     // FIFO message ID acceptance filter mask
    0,                                     // FIFO frame ID rejection filter value
    0x07FF,                                // FIFO frame ID rejection filter mask
    {                                      // Range filters (RF) configuration
       // RF enable, RF mode, RF lower interval, RF upper interval
       {TRUE, FR_ACCEPTANCE, 60, 64},   // 1st range filter
       {FALSE, FR_ACCEPTANCE, 0, 0},    // 2nd range filter
        {FALSE, FR_ACCEPTANCE, 0, 0},    // 3rd range filter
        {FALSE, FR_ACCEPTANCE, 0, 0},    // 4th range filter
    },
    TRUE                                   // FIFO interrupt enable
};


const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
{
    2, FR_LAST_MB
};
const Fr_buffer_info_type Fr_buffer_cfg_00[] =
```

```
{   /* Buffer type       Configuration structure ptr    MB index xx = configuration
                                                                    index used by
                                                                    Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_01_cfg,      0},          // 00
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_04_cfg,      3},          // 01
    {FR_RECEIVE_FIFO,    &Fr_FIFOA_cfg,                  15},         // 02
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_cfg,              0},          // 03
};
```

Example of usage:

Only the receive FIFO A will be configured (no other buffers). The number of the message buffer header field for the first message buffer of the receive FIFO (the start index) is 15.

```
Fr_return_type return_value;

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);
```

2.2.2.7 Fr_receive_shadow_buffers_config_type

The receive shadow buffers are required for the frame reception process for individual message buffers. The FlexRay module provides four receive shadow buffers, one receive shadow buffer per channel and per message buffer segment.

The FlexRay UNIFIED Driver executes all the necessary operations with the receive shadow buffer, and the host application only has to configure the relevant receive shadow buffers.

The host application has to calculate with these buffers and it is necessary to configure one receive shadow buffer per channel used, and one buffer per message buffer segment used. During the message buffer layout design, the designer should allocate the necessary amount of message buffers for the receive shadow buffers and assign the initial message buffer header indexes.

For example, in the case where the host application uses only segment 1 and receives frames on both channels, two receive shadow buffers have to be configured, one for channel A, the second for channel B.

NOTE

The initial message buffer header index of the relevant receive shadow buffer should be located in the same segment for which it is configured.

The structure of the *Fr_receive_shadow_buffers_config_type* type is depicted in Table 2-9 The Fr_receive_shadow_buffers_config_type Member Description.

Table 2-9 The Fr_receive_shadow_buffers_config_type Member Description

Structure item	Description	Range	Direct Impact
RSBIR_A1_enable	Defines whether or not the receive shadow buffer for channel A and segment 1 is enabled	boolean (TRUE or FALSE)	RSBIR: SEL
RSBIR_A2_enable	Defines whether or not the receive shadow buffer for channel A and segment 2 is enabled	boolean (TRUE or FALSE)	RSBIR: SEL
RSBIR_B1_enable	Defines whether or not the receive shadow buffer for channel B and segment 1 is enabled	boolean (TRUE or FALSE)	RSBIR: SEL
RSBIR_B2_enable	Defines whether or not the receive shadow buffer for channel B and segment 2 is enabled	boolean (TRUE or FALSE)	RSBIR: SEL
RSBIR_A1_buffer_number_init	Receive shadow buffer index Defines initial message buffer header index of receive shadow buffer for channel A and segment 1	0 up to 255	RSBIR: RSBIDX
RSBIR_A2_buffer_number_init	Receive shadow buffer index Defines initial message buffer header index of receive shadow buffer for channel A and segment 2	0 up to 255	RSBIR: RSBIDX
RSBIR_B1_buffer_number_init	Receive shadow buffer index Defines initial message buffer header index of receive shadow buffer for channel B and segment 1	0 up to 255	RSBIR: RSBIDX
RSBIR_B2_buffer_number_init	Receive shadow buffer index Defines initial message buffer header index of receive shadow buffer for channel B and segment 2	0 up to 255	RSBIR: RSBIDX

Example of configuration:

In this example, all receive shadow buffers are enabled. The configuration of the receive message buffers is not depicted below, however, they are configured for both segments and both channels.

The number of the static slots in the FlexRay communication cycle is 60 (from 1 to 60), the maximum number of minislots is 22. All message buffers configured for transmission or reception in the static part of the communication cycle belong to the first segment (segment 1), transmit message buffers configured for transmission and reception in the dynamic part of the communication cycle belong to the second segment (segment 2).

The layout of the segments and message buffer is as follows:

- Segment 1 is located in the message buffer number range of 0 to 10 (the number of message numbers in segment 1 is 11)
- Segment 2 from 11 to 18 (the number of message numbers in segment 2 is 8).

The initial message buffer header index allocation is as follows:

- The first shadow buffer for segment 1 and channel A - initial header index is 8
- The second shadow buffer for segment 1 and channel B - initial header index is 9
- The third shadow buffer for segment 2 and channel A - initial header index is 17
- The fourth shadow buffer for segment 2 and channel B - initial header index is 18

```
const Fr_receive_shadow_buffers_config_type Fr_rx_shadow_cfg =
{
    TRUE,        // Rx shadow buffer for channel A, seg 1 - enabled
    TRUE,        // Rx shadow buffer for channel A, seg 2 - enabled
    TRUE,        // Rx shadow buffer for channel B, seg 1 - enabled
    TRUE,        // Rx shadow buffer for channel B, seg 2 - enabled
    8,           // Ch A, seg 1 - the current index of the MB header field
    17,          // Ch A, seg 2 - the current index of the MB header field
    9,           // Ch B, seg 1 - the current index of the MB header field
    18           // Ch B, seg 2 - the current index of the MB header field
};


const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
{
    1, 2, 5, 6, 7, FR_LAST_MB
};
const Fr_buffer_info_type Fr_buffer_cfg_00[] =
{   /* Buffer type       Configuration structure ptr    MB index xx = configuration
                                                          index used by
                                                          Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_01_cfg,0},        // 00
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_03_cfg,2},        // 01
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_04_cfg,3},        // 02
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_02_cfg,0},        // 03
    {FR_RECEIVE_FIFO,    &Fr_FIFOA_cfg,            19},       // 04
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_cfg,        0},        // 05
    {FR_TRANSMIT_BUFFER, &Fr_rx_buffer_slot_62_cfg,12},       // 06
    {FR_TRANSMIT_BUFFER, &Fr_rx_buffer_slot_64_cfg,13},       // 07
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_09_cfg,6},        // 08
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_10_cfg,7},        // 09
};
```

Example of usage:

The configuration of four receive message buffers is not presented in this example. Two of them are assigned to the static segment (and segment 1) and to receive in slots 2 and 3. The rest of the receive buffers are assigned to the dynamic segment (and segment 2) and receive in slots 62 and 64.

```
Fr_return_type return_value;
```

```
return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);
```

2.2.2.8 Fr_timer_config_type

The FlexRay module provides two timers, T1 and T2, which run on the FlexRay time base. Each of these timers generates a maskable interrupt when it reaches a configured point in time. The timer T1 is an absolute timer. The timer T2 can be configured as an absolute or a relative timer. Both timers can be configured as repetitive or not. In non-repetitive mode, the timer stops if it expires. In repetitive mode, it restarts itself when it expires.

The host application can easily configure both timers by calling the ***Fr_timers_init()*** function. This routine has one input parameter which refers to the timers configuration array (the instance of the *Fr_timer_config_type* type).

This timer configuration array (type of the *Fr_timer_config_type*) refers to separate configuration structures for each timer (for each timer a separate configuration structure should be defined with detailed timer settings, which is also of type *Fr_timer_config_type*). This concept is shown in Figure 2-2. Timers Configuration Concept.

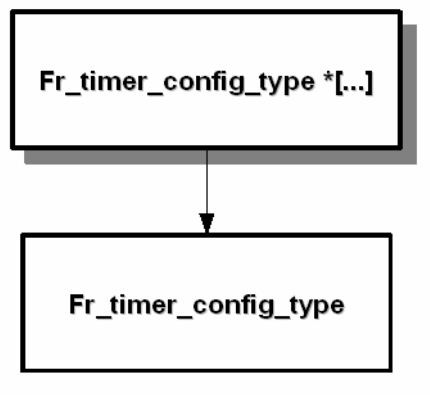

Figure 2-2. Timers Configuration Concept

NOTE

For correct operation, the timer configuration array must have the NULL value stored in the last item.

The structure of each separate configuration structure is described in Table 2-10 The Fr_timer_config_type Member Description.

Table 2-10 The Fr_timer_config_type Member Description

Structure item	Description	Range	Direct Impact
timer_ID	Defines which timer is configured	Fr_timer_ID_type (FR_TIMER_T1 or FR_TIMER_T2)	TICCR
timer_type	Defines the timebase mode of the selected timer	Fr_timer_timebase_type (FR_ABSOLUTE or FR_RELATIVE)	TICCR: T2_CFG
timer_repetition	Defines the repetition mode of the selected timer	Fr_timer_repetition_type (FR_NON_REPETITIVE or FR_REPETITIVE)	TICCR: T1_REP or TICCR: T2_REP
timer_macrotick_offset	Defines the macrotick offset value for the selected timer	0 up to 0x3FFF for absolute timer or 0 up to 0xFFFFFFFF for relative timer	TI1MTOR or TI2CR1 only or TI2CR0 and TI2CR1
timer_cycle_filter_mask	Defines the cycle filter mask for the selected timer	0 up to 0x3F	TI1CYSR or TI2CR0
timer_cycle_filter_value	Defines the cycle filter value for the selected timer	0 up to 0x3F	TI1CYSR or TI2CR0

Example of configuration:

In this example, both timers are configured, timer T1 as absolute and timer T2 as relative.
The timer T1 interrupt occurs (in case where the timer interrupt is enabled) each communication cycle when the macrotick counter is equal to 2050 MT.
The timer T2 interrupt occurs (in case where the timer interrupt is enabled) when the programmed amount of macroticks (1000000 MT) has expired.

```
// Configuration data for absolute timer T1
const Fr_timer_config_type Fr_timer_1_cfg =
{
    FR_TIMER_T1,      // Timer number T1
    FR_ABSOLUTE,      // Timer timebase
    FR_REPETITIVE,    // Timer repetition mode
    2050,             // Timer macrotick offset
    0,                // Timer cycle filter mask, only for absolute timer
    0                 // Timer cycle filter value, only for absolute timer
};

// Configuration data for relative timer T2
const Fr_timer_config_type Fr_timer_2_cfg =
{
```

```
    FR_TIMER_T2,       // Timer number (T2)
    FR_RELATIVE,       // Timer timebase
    FR_REPETITIVE,     // Timer repetition mode
    1000000,           // Timer macrotick offset
    0,                 // Timer cycle filter mask, not applied for relative timer
    0                  // Timer cycle filter value, not applied for relative timer
};

// Array with timers configuration information
const Fr_timer_config_type * Fr_timers_cfg_00_ptr[] =
{
    &Fr_timer_1_cfg,                    // Pointer to configuration structure for timer T1
    &Fr_timer_2_cfg,                    // Pointer to configuration structure for timer T2
    NULL
};
```

Example of usage:

```
Fr_timers_init(&Fr_timers_cfg_00_ptr[0]);   // Initialization of the timers

Fr_enable_interrupts(FR_PROTOCOL_IRQ,(FR_TIMER_1_EXPIRED_IRQ |
FR_TIMER_2_EXPIRED_IRQ), 0);                // Enable appropriate interrupts
Fr_start_timer(FR_TIMER_T1);                // Start Timer T1
Fr_start_timer(FR_TIMER_T2);                // Start Timer T2
```

2.2.2.9 Fr_MTS_config_type

The FlexRay module provides a flexible means to request the transmission of the Media Access Test Symbol MTS in the symbol window on channel A or channel B.

The host application enables or disables the generation of the MTS on the required channel by calling the ***Fr_send_MTS()*** function and passing the reference to the MTS configuration structure (of the type *Fr_MTS_config_type*).

The *Fr_MTS_config_type* configuration structure is described in Table 2-11 The Fr_MTS_config_type Member Description.

Table 2-11 The Fr_MTS_config_type Member Description

Structure item	Description	Range	Direct Impact
cycle_counter_value	Defines the filter value for the MTS cycle count filter	0 up to 0x3F	MTSACFR or MTSBCFR
cycle_counter_mask	Defines the filter mask for the MTS cycle count filter	0 up to 0x3F	MTSACFR or MTSBCFR

Example of configuration:

The FlexRay module is configured to transmit the MTS in each cycle.

```
const Fr_MTS_config_type Fr_MTS_B_cfg =
{
    0,              // Define the filter value for the MTS cycle count filter
    0               // Define the filter mask for the MTS cycle count filter
};
```

Example of usage:

```
Fr_send_MTS(FR_CHANNEL_B, &Fr_MTS_B_cfg);// Configure the FlexRay module to sending
                                          the MTS on channel B
```

2.2.2.10 Fr_slot_status_config_type

The FlexRay module provides eight slot status registers. Each slot status register can be used to observe a static slot, a dynamic slot, the symbol window, or the NIT.
Although the FlexRay module provides eight slot status registers, they are grouped in four pairs. Each pair can be assigned to only one slot. One register from each pair is assigned to the even communication cycle, the second register to the odd communication cycle.

The host application can configure the slot status registers by calling the ***Fr_slot_status_init()*** function with the reference to the slot status configuration structure (the instance of the type *Fr_slot_status_config_type*).

The *Fr_slot_status_config_type* configuration structure is described in Table 2-12 The Fr_slot_status_config_type Member Description.

Table 2-12 The Fr_slot_status_config_type Member Description

Structure item	Description	Range	Direct Impact
SSSR0_slot_number	Specifies the number of the slot whose status will be saved in the first pair of slot status registers	0 up to 1023	SSR0 and SSR1
SSSR1_slot_number	Specifies the number of the slot whose status will be saved in the second pair of slot status registers	0 up to 1023	SSR2 and SSR3
SSSR2_slot_number	Specifies the number of the slot whose status will be saved in the third pair of slot status registers	0 up to 1023	SSR4 and SSR5
SSSR3_slot_number	Specifies the number of the slot whose status will be saved in the fourth pair of slot status registers	0 up to 1023	SSR6 and SSR7

NOTE

If the value of the SSSR0_slot_number, SSSR1_slot_number, SSSR2_slot_number or SSSR3_slot_number parameter is set to 0, the related slot status register provides the status of the symbol window and the NIT instead of a slot. See section 2.1.5.2 Slot Status Registers for more details

Example of configuration:

In this example, the host application can observe the status vectors of the first and twelfth slots and also symbol window and the NIT.

```
const Fr_slot_status_config_type Fr_slot_status_cfg_set_00 =
{
    1,           // Static Slot number for the SSR0 and the SSR1
    12,        // Static Slot number for the SSR2 and the SSR3
    0,         // Static Slot number for the SSR4 and the SSR5
    0          // Static Slot number for the SSR6 and the SSR7
};
```

Example of usage:

```
Fr_slot_status_init(&Fr_slot_status_cfg_set_00);
```

2.2.2.11 Fr_slot_status_counter_config_type

The FlexRay module provides four slot status error counter registers. Each of these registers is updated with an internal slot status counter value at the start of a communication cycle. The internal slot status counter is incremented if its condition matches the status vector provided by the PE. All static slots, the symbol window, and the NIT status are taken into account. Dynamic slots are excluded.

The host application can configure the slot status counter registers by calling the ***Fr_slot_status_counter_init()*** function. This routine has one input parameter which refers to the slot status counter configuration array (the instance of the type *Fr_slot_status_counter_config_type*).

This slot status counter configuration array (of type *Fr_slot_status_counter_config_type*) refers to the separate configuration structures for each slot status register (for each slot status register a separate configuration structure should be defined with detailed settings, which is also of type *Fr_slot_status_counter_config_type*). This concept is depicted in Figure 2-3. Slot Status Counter Configuration Concept.

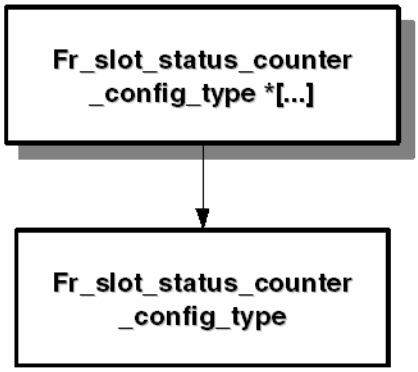

Figure 2-3. Slot Status Counter Configuration Concept

NOTE

For correct operation, the slot status counter configuration array should not contain more than four references to the slot status counter configuration structures, and must have the NULL value stored in the last item.

The structure of each separate configuration structure is described in Table 2-13 The Fr_slot_status_counter_config_type Member Description.

Table 2-13 The Fr_slot_status_counter_config_type Member Description

Structure item	Description	Range	Direct Impact
counter_ID	Selects one of the four slot status register	Fr_slot_status_counter_ID_type (FR_SLOT_STATUS_COUNTER_0 or FR_SLOT_STATUS_COUNTER_1 or FR_SLOT_STATUS_COUNTER_2 or FR_SLOT_STATUS_COUNTER_3)	SSCCR: SEL

Table 2-13 The Fr_slot_status_counter_config_type Member Description

Structure item	Description	Range	Direct Impact
counter_configuration	Defines the channel related to incrementing the selected slot status counter	Fr_slot_status_counter_channel_type (FR_SLOT_STATUS_CHANNEL_A or FR_SLOT_STATUS_CHANNEL_B or FR_SLOT_STATUS_CHANNEL_AB_BY_1 or FR_SLOT_STATUS_CHANNEL_AB_BY_2)	SSCCR: CNTCFG
multi_cycle_selection	Defines whether the selected slot status counter accumulates over multiple communication cycle or provides information for the previous communication cycle only	boolean (TRUE or FALSE)	SSCCR: MCY
valid_frame_restriction	Defines whether the selected slot status counter restricts the valid frames only	boolean (TRUE or FALSE)	SSCCR: VFR
sync_frame_restriction	Defines whether the selected slot status counter restricts the counter to receive frames with the sync frame indicator bit set to 1	boolean (TRUE or FALSE)	SSCCR: SYF
null_frame_restriction	Defines whether the selected slot status counter restricts the counter to receive frames with the null frame indicator bit set to 0	boolean (TRUE or FALSE)	SSCCR: NUF
startup_frame_restriction	Defines whether the selected slot status counter restricts the counter to receive frames with the startup frame indicator bit set to 1	boolean (TRUE or FALSE)	SSCCR: SUF
syntax_error_counting	Defines whether the selected slot status counter counts the slots with the syntax error indicator bit set to 1	boolean (TRUE or FALSE)	SSCCR: STATUSMASK
content_error_counting	Defines whether the selected slot status counter counts the slots with the content error indicator bit set to 1	boolean (TRUE or FALSE)	SSCCR: STATUSMASK
boundary_violation_counting	Defines whether the selected slot status counter counts the slots with the boundary violation indicator bit set to 1	boolean (TRUE or FALSE)	SSCCR: STATUSMASK

Table 2-13 The Fr_slot_status_counter_config_type Member Description

Structure item	Description	Range	Direct Impact
transmission_conflict_ counting	Defines whether the selected slot status counter counts the slots with the transmission conflict indicator bit set to 1	boolean (TRUE or FALSE)	SSCCR: STATUSMASK

Example of configuration:

In this example, three slot status counters are configured.
The first counter increments its value

- by 2 if the condition is fulfilled on both channels
- by 1 if the condition is fulfilled on one channel only

when a valid sync frame is received. It accumulates over multiple communication cycles.

The second counter is configured to increment its value

- by 2 if the condition is fulfilled on both channels
- by 1 if the condition is fulfilled on one channel only

when a content error, boundary violation or transmission conflict indicators are set during the communication cycle. It accumulates over multiple communication cycles.

The third counter is configured to increment its value by 1 when a startup frame is received during the communication cycle. The slot status counter provides information on the previous communication cycle only.

```
// Configuration data for the Slot Status Counter Register 0
const Fr_slot_status_counter_config_type Fr_slot_status_counter_0_cfg =
{
    FR_SLOT_STATUS_COUNTER_0,        // Select slot status counter register
    FR_SLOT_STATUS_CHANNEL_AB_BY_2, // Control the channel related incrementing
    TRUE,                            // Define whether the counter accumulates over
                                         multiple com.cycle
    TRUE,                            // Restrict the counter to receive valid frames
    TRUE,                            // Restrict the counter to receive sync frames
    FALSE,                           // Restrict the counter to receive null frames
    FALSE,                           // Restrict the counter to receive startup frames
    FALSE,                           // Enable the counting for slots with the syntax
                                         error indicator bit set to 1
    FALSE,                           // Enable the counting for slots with set content
                                         error indicator
    FALSE,                           // Enable the counting for slots with set
                                         boundary violation indicator
    FALSE                            // Enable the counting for slots with set
                                         transmission conflict indicator
};

// Configuration data for the Slot Status Counter Register 1
```

```
const Fr_slot_status_counter_config_type Fr_slot_status_counter_1_cfg =
{
    FR_SLOT_STATUS_COUNTER_1,        // Select slot status counter register
    FR_SLOT_STATUS_CHANNEL_AB_BY_2, // Control the channel related incrementing
    TRUE,                            // Define whether the counter accumulates over
                                        multiple com.cycle
    FALSE,                           // Restrict the counter to receive valid frames
    FALSE,                           // Restrict the counter to receive sync frames
    FALSE,                           // Restrict the counter to receive null frames
    FALSE,                           // Restrict the counter to receive startup frames
    FALSE,                           // Enable the counting for slots with the syntax
                                        error indicator bit set to 1
    TRUE,                            // Enable the counting for slots with set content
                                        error indicator
    TRUE,                            // Enable the counting for slots with set
                                        boundary violation indicator
    TRUE                             // Enable the counting for slots with set
                                        transmission conflict indicator
};

// Configuration data for the Slot Status Counter Register 2
const Fr_slot_status_counter_config_type Fr_slot_status_counter_2_cfg =
{
    FR_SLOT_STATUS_COUNTER_2,        // Select slot status counter register
    FR_SLOT_STATUS_CHANNEL_AB_BY_1, // Control the channel related incrementing
    FALSE,                           // Define whether the counter accumulates over
                                        multiple com.cycle
    TRUE,                            // Restrict the counter to receive valid frames
    FALSE,                           // Restrict the counter to receive sync frames
    FALSE,                           // Restrict the counter to receive null frames
    TRUE,                            // Restrict the counter to receive startup frames
    FALSE,                           // Enable the counting for slots with the syntax
                                        error indicator bit set to 1
    FALSE,                           // Enable the counting for slots with set content
                                        error indicator
    FALSE,                           // Enable the counting for slots with set
                                        boundary violation indicator
    FALSE                            // Enable the counting for slots with set
                                        transmission conflict indicator
};

// Array with Slot Status Counters configuration information
const Fr_slot_status_counter_config_type * Fr_slot_status_counter_cfg_00_ptr[] =
{
    &Fr_slot_status_counter_0_cfg,  // Reference to configuration structure for Slot
                                       Status Counter Condition Register 0
    &Fr_slot_status_counter_1_cfg,  // Reference to configuration structure for Slot
                                       Status Counter Condition Register 1
    &Fr_slot_status_counter_2_cfg,  // Reference to configuration structure for Slot
                                       Status Counter Condition Register 2
    NULL
};
```

Example of the usage:

```
Fr_slot_status_counter_init(&Fr_slot_status_counter_cfg_00_ptr[0]);
```

2.2.3 UNIFIED Driver Performance Data

The comparison of the FlexRay UNIFIED Driver CPU usage for the different platforms was done. For this purpose one of the application example provided together with the FlexRay UNIFIED Driver was used. The appllication features are:

- application runs in the interrupt driven mode
- two message buffers are used - data payload length is 32 Bytes
- one FIFO storage is configured to receive 2 frames in one communication cycle - data payload length is 16 Bytes
- other informartion is stored into the application variables (slot status information, payload length and so on).

Please, see [UNIFIED_APP_EXAMPLES] for more information.

The Table 2-1 shows the total CPU time needed to handle the interrupts incoming from the FlexRay module in one communication cycle. The CPU usage represents the time spent in the FlexRay related interrupts according to the following equation:

$$CPU_{Load} = T_{Interrupt} / T_{Cycle} * 100[\%]$$

where:

CPU_{Load} is the CPU load in percent;
$T_{Interrupt}$ is the amount of time spent in all FlexRay interrupts;
T_{Cycle} is the time of FlexRay communication cycle.

Table 2-1 UNIFIED Driver CPU usage

Platform	Conditions	CPU_{Load} time [%]
MPC5561 and MPC5567 (embedded FlexRay module)	f_{system} = 120 [MHz]	0.45[x]
MPC5516 (embedded FlexRay module)	f_{system} = 80 [MHz]	2.99[x]
MPC5554 with standalone MFR4310 CC	f_{system} = 120 [MHz]*	0.67[x]
MC9S12XFR128 (embedded FlexRay module)	f_{bus}= 40 [MHz]	2.30[xx]
MC9S12XDP512 with standalone MFR4300 CC	f_{bus}= 25 [MHz]**	3.96[xx]
MC9S12XF512 (embedded FlexRay module)	f_{bus}= 40 [MHz]	2.30[xx]

* Note, three stretch cycles are used
** Note, five wait states in single cycle were used (see appconfig.h file)
x Note, optimized with the following options “-XO” and “-Xsize-opt” in WindRiver DIAB C Compiler
xx Note, all optimizations are enabled in CodeWarrior Development Studio for Freescale MC9S12 v4.5 or v4.6 for MC9S12XF

Chapter 3 The FlexRay UNIFIED Driver Function Specification

This chapter describes application programming interface (API) for each function of the FlexRay UNIFIED Driver.

3.1 Initialization and Configuration Functions

3.1.1 Fr_init

Function name:	**Fr_init**	
Syntax	Fr_return_type Fr_init (const Fr_HW_config_type *Fr_HW_config_temp_ptr, const Fr_low_level_config_type *Fr_low_level_config_temp_ptr)	
When called:	This function should be called before the others. The FlexRay module is disabled	
Parameters (in):	Fr_HW_config_temp_ptr	Reference to the hardware configuration parameters for the FlexRay driver
	Fr_low_level_config_temp_ptr	Reference to the low level configuration parameters
Parameters (out):	none	
Return value:	FR_SUCCESS	API call has been successful
	FR_NOT_SUCCESS	Error occurred during transition into FR_POCSTATE_CONFIG
Description:	This API call stores the base addresses of the FlexRay module and available memories, configures the FlexRay channels and forces the module into the *POC:config* state. In case where the FlexRay module does not transmit into the *POC:config* state after certain time (defined by FR_MAX_WAIT_CYCLES constant), the FR_NOT_SUCCESS parameter is returned. Note: This function should be called only once during runtime, even if a FlexRay module is reconfigured	

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);`	

3.1.2 Fr_set_configuration

Function name:	**Fr_set_configuration**	
Syntax	void Fr_set_configuration (const Fr_HW_config_type *Fr_HW_config__temp_ptr, const Fr_low_level_config_type *Fr_low_level_config_temp_ptr)	
When called:	*POC:config*	
Parameters (in):	Fr_HW_config_temp_ptr	Reference to the hardware configuration parameters for the FlexRay UNIFIED Driver
	Fr_low_level_config_temp_ptr	Reference to the low level configuration parameters
Parameters (out):	none	
Return value:	none	
Description:	This API call initializes the FlexRay module. The following actions are performed by this function: • initialization of the data structures of the FlexRay driver; the pointers to configuration structures are stored into internal variables (to be used by the FlexRay UNIFIED Driver later on) • configuration of the FlexRay parameters (e.g. cluster configuration) • disabling all FlexRay module interrupts	
Example of usage:	`Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);`	

3.1.3 Fr_buffers_init

Function name:	**Fr_buffer_init**	
Syntax	Fr_return_type Fr_buffers_init (const Fr_buffer_info_type *Fr_buffers_config_temp_ptr, const uint8 *Fr_buffer_config_set_temp_ptr)	
When called:	*POC:config*	
Parameters (in):	Fr_buffers_config_temp_ptr	Pointer to structure with message buffers configuration data
	Fr_buffer_config_set_temp_ptr	Reference to an array with information which message buffers will be used from MB configuration structure (referenced by the Fr_buffers_config_temp_ptr pointer)
Parameters (out):	none	
Return value:	FR_SUCCESS	API call has been successful
	FR_NOT_SUCCESS	The FR_NUMBER_TXRX_MB parameter is not correctly set in the *Fr_UNIFIED_cfg.h* file
Description:	This API call initializes the FlexRay module message buffers. The following actions are performed by this function: • initialization of internal driver pointers • configuration of the transmit, receive, receive shadow and FIFO message buffers. Configuration information is described in the structures referenced by the Fr_buffers_config_temp_ptr and the Fr_buffer_config_set_temp_ptr pointers. For each message buffer, the function determines which data segment is used and calculates the Data Field Offsets • initialization of an internal structure with information about already configured FIFO buffers In case that the configured transmit or receive Message Buffer Index is greater than the value of the FR_NUMBER_TXRX_MB parameter in the *Fr_UNIFIED_cfg.h* file, the FR_NOT_SUCCESS parameter is returned. See section 2.2.1.2 Memory Consumption Optimization Possibility in the Fr_UNIFIED_cfg.h File for more information.	

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0], &Fr_buffer_cfg_set_00[0]);`	

3.1.4 Fr_timers_init

Function name:	**Fr_timers_init**
Syntax	void Fr_timers_init (const Fr_timer_config_type ** Fr_timer_config_temp_ptr_ptr)
When called:	*POC:config*, *POC:normal active*, *POC:normal passive*
Parameters (in):	Fr_timer_config_temp_ptr_ptr Reference to timer configuration structure
Parameters (out):	none
Return value:	none
Description:	This function initializes the absolute and/or relative FlexRay module timers
Example of usage:	`Fr_timers_init(&Fr_timers_cfg_00_ptr[0]);`

3.1.5 Fr_slot_status_init

Function name:	**Fr_slot_status_init**
Syntax	void Fr_slot_status_init (const Fr_slot_status_config_type *Fr_slot_status_config_temp_ptr)
When called:	*POC:config*, *POC:normal active*, *POC:normal passive*
Parameters (in):	Fr_slot_status_config_temp_ptr Pointer to structure with slot status configuration data
Parameters (out):	none
Return value:	none
Description:	This API call initializes the FlexRay module slot status functionality
Example of usage:	`Fr_slot_status_init(&Fr_slot_status_cfg_set_00);`

3.1.6 Fr_slot_status_counter_init

Function name:	**Fr_slot_status_counter_init**
Syntax	void Fr_slot_status_counter_init (const Fr_slot_status_counter_config_type ** Fr_slot_status_counter_config_temp_ptr_ptr)
When called:	*POC:config*, *POC:normal active*, *POC:normal passive*
Parameters (in):	Fr_slot_status_counter_config_temp_ptr_ptr — Reference to slot status counter configuration structure
Parameters (out):	none
Return value:	none
Description:	This API call initializes the FlexRay module slot status counter functionality. The FlexRay module provides four slot status error counter registers: Slot Status Counter Registers (SSCR0–SSCR3). Each of these slot status counter registers is updated with the value of an internal slot status counter at the start of a communication cycle. The internal slot status counter is incremented if its increment condition, defined by the Slot Status Counter Condition Register (SSCCR), matches the status vector provided by the PE. See [FLEXRAY_MODULE] for more information, chapter Status Monitoring
Example of usage:	`Fr_slot_status_counter_init(&Fr_slot_status_counter_cfg_00_ptr[0]);`

3.2 Startup, the MTS and Change Mode Functions

3.2.1 Fr_leave_configuration_mode

Function name:	**Fr_leave_configuration_mode**
Syntax	Fr_return_type Fr_leave_configuration_mode (void)
When called:	*POC:config*
Parameters (in):	none
Parameters (out):	none
Return value:	FR_SUCCESS API call has been successful FR_NOT_SUCCESS Error occurred
Description:	This API call initiates a transition from the *POC:config* state into the *POC:ready* state. If the transition is not successful, this function returns the FR_NOT_SUCCESS

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_leave_configuration_mode();`

3.2.2 Fr_start_communication

Function name:	**Fr_start_communication**
Syntax	Fr_return_type Fr_start_communication (void)
When called:	*POC:ready*
Parameters (in):	none
Parameters (out):	none
Return value:	FR_SUCCESS API call has been successful FR_NOT_SUCCESS Error occurred
Description:	This API call initiates the start of the FlexRay communication. It initiates a transition from the *POC:ready* state into the *POC:startup* state. If the start up of the FlexRay module is successful, the state *POC:normal active* is automatically reached. If the FlexRay module is not in the *POC:ready* state, but e.g. in the *POC:halt* state, this function returns FR_NOT_SUCCESS and remains in its original state

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_start_communication();`

3.2.3 Fr_stop_communication

Function name:	**Fr_stop_communication**		
Syntax	Fr_return_type Fr_stop_communication (Fr_stop_communication_type Fr_stop_option)		
When called:	*POC:normal active*, *POC:normal passive*; Only with the FR_ABORT_COMMUNICATION parameter passed: *POC:config*, *POC:ready*, *POC:startup*, *POC:wakeup*		
Parameters (in):	Fr_stop_option	FR_HALT_COMMUNICATION	The function halts the communication of the FlexRay CC at the end of current FlexRay cycle
		FR_ABORT_COMMUNICATION	The function immediately aborts the communication of the FlexRay module
Parameters (out):	none		
Return value:	FR_SUCCESS		API call has been successful
	FR_NOT_SUCCESS		Error occurred
Description:	This API call halts the communication of the FlexRay module at the end of current FlexRay cycle in case where the FR_HALT_COMMUNICATION parameter is placed. The FR_HALT_COMMUNICATION parameter can be used only in the *POC:normal active* or *POC:normal passive* state. This function aborts the communication of the FlexRay module immediately in case where the FR_ABORT_COMMUNICATION parameter is placed. In case that the FR_ABORT_COMMUNICATION parameter is passed, the ***Fr_stop_communication()*** function can also be used for the transition of the POC process from any POC state to the *POC:halt* state		

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_stop_communication(FR_ABORT_COMMUNICATION);`		

3.2.4 Fr_send_wakeup

Function name:	**Fr_send_wakeup**
Syntax	Fr_return_type Fr_send_wakeup (void)
When called:	*POC:ready*
Parameters (in):	none
Parameters (out):	none
Return value:	FR_SUCCESS API call has been successful FR_NOT_SUCCESS Error occurred
Description:	This API call sends a Wakeup Pattern over configured channel. Before that, the host should check the wakeup status of the FlexRay module via the ***Fr_get_wakeup_state()*** function, to see if a wakeup pattern has been received

Example of usage:	`Fr_return_type return_value;`
	`Fr_wakeup_state_type wakeup_status;`
	
	`wakeup_status = Fr_get_wakeup_state();`
	`if(wakeup_status == FR_WAKEUPSTATE_UNDEFINED)`
	`{`
	`  return_value = Fr_send_wakeup();`
	`}`

3.2.5 Fr_get_wakeup_state

Function name:	**Fr_get_wakeup_state**	
Syntax	Fr_wakeup_state_type Fr_get_wakeup_state (void)	
When called:	anytime	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FR_WAKEUPSTATE_UNDEFINED	Undefined state (the communication controller has not executed the wakeup mechanism yet)
	FR_WAKEUPSTATE_RECEIVED_HEADER	Received_Header
	FR_WAKEUPSTATE_RECEIVED_WUP	Receiver_WUP
	FR_WAKEUPSTATE_COLLOSION_HEADER	Collision_Header
	FR_WAKEUPSTATE_COLLISION_WUP	Collision_WUP
	FR_WAKEUPSTATE_COLLISION_UNKNOWN	Collision_Unknown
	FR_WAKEUPSTATE_TRANSMITTED	Transmitted
Description:	This function retrieves the wakeup state of the FlexRay module, which is part of the startup of the FlexRay module. For more information about possible status, please see [FR_PROTOCOL], chapter Wakeup State Diagram	

Example of usage:	`Fr_return_type return_value;`
	`Fr_wakeup_state_type wakeup_status;`
	
	`wakeup_status = Fr_get_wakeup_state();`
	`if(wakeup_status == FR_WAKEUPSTATE_UNDEFINED)`
	`{`
	`  return_value = Fr_send_wakeup();`
	`}`	

3.2.6 Fr_get_POC_state

Function name:	**Fr_get_POC_state**	
Syntax	Fr_POC_state_type Fr_get_POC_state (void)	
When called:	anytime	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FR_POCSTATE_CONFIG	*POC:config* - configuration state
	FR_POCSTATE_DEFAULT_CONFIG	*POC:default config* - state prior to the *POC:config*, only left with an explicit configuration request
	FR_POCSTATE_HALT	*POC:halt* - error state, can only be left by a reinitialization
	FR_POCSTATE_NORMAL_ACTIVE	*POC:normal active* - Normal operation
	FR_POCSTATE_NORMAL_PASSIVE	*POC:normal passive* - errors detected, no transmission of data, but attempting to return into normal operation
	FR_POCSTATE_READY	*POC:ready* - state reached from the *POC:config* after concluding the configuration
	FR_POCSTATE_STARTUP	*POC:startup* - module transmits only startup frames
	FR_POCSTATE_WAKEUP	*POC:wakeup* - module sends a wakeup pattern if it could not find anything on the bus
	FR_POCSTATE_UNKNOWN	Returned value if an error has occured.
Description:	Query the current value of the POC state of FlexRay module. For more information about possible POC status, please see [FR_PROTOCOL]	

Example of usage:	`Fr_POC_state_type protocol_state;`
	
	`protocol_state = Fr_get_POC_state();`
	`if(protocol_state == FR_POCSTATE_READY)`
	`{`
	`}`	

3.2.7 Fr_get_sync_state

Function name:	**Fr_get_sync_state**	
Syntax	Fr_sync_state_type Fr_get_sync_state (void)	
When called:	anytime	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FR_ASYNC	The local FlexRay module is asynchronous to the FlexRay global time
	FR_SYNC	The local FlexRay module is synchronous to the FlexRay global time
Description:	Query whether or not the FlexRay module is synchronous to the rest of the cluster	

Example of usage:	`Fr_sync_state_type sync_state;`
	
	`sync_state = Fr_get_sync_state();`
	`if(sync_state == FR_ASYNC)`
	`{`
	`}`	

3.2.8 Fr_enter_configuration_mode

Function name:	**Fr_enter_configuration_mode**
Syntax	Fr_return_type Fr_enter_configuration_mode (void)
When called:	*POC:halt*, *POC:default config*, *POC:ready*
Parameters (in):	none
Parameters (out):	none
Return value:	FR_SUCCESS API call has been successful FR_NOT_SUCCESS Error occurred during transmission process
Description:	This function queries the POC state of the FlexRay protocol and performs the following sequence to initiate a transition into the *POC:config* state: - if the state of the protocol is the *POC:halt*, the function sends the DEFAULT_CONFIG command and later on the CONFIG command - if the state of the protocol is the *POC:default config* or *POC:ready*, the function sends the CONFIG command If the transition is not successful (e.g. this function is called and the FlexRay module is not in the *POC:halt*, *POC:default config* or *POC:ready* states) this function returns FR_NOT_SUCCESS

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_enter_configuration_mode();`

3.2.9 Fr_reset_protocol_engine

Function name:	**Fr_reset_protocol_engine**
Syntax	Fr_return_type Fr_reset_protocol_engine (void)
When called:	anytime
Parameters (in):	none
Parameters (out):	none
Return value:	FR_SUCCESS API call has been successful FR_NOT_SUCCESS Error occurred during transmission process
Description:	This function immediately resets the Protocol Engine and initiates a transition into the *POC:default config* state

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_reset_protocol_engine();`

3.2.10 Fr_send_MTS

Function name:	**Fr_send_MTS**		
Syntax	Fr_return_type Fr_send_MTS (Fr_channel_type Fr_channel, const Fr_MTS_config_type *Fr_MTS_config_temp_ptr)		
When called:	*POC:normal active*		
Parameters (in):	Fr_channel	FR_CHANNEL_A	Media test symbol will be transmitted on channel A
		FR_CHANNEL_B	Media test symbol will be transmitted on channel B
	Fr_MTS_config_temp_ptr	Reference to structure with configuration data for Media Test Symbol transmission from given channel	
Parameters (out):	none		
Return value:	FR_SUCCESS	API call has been successful	
	FR_NOT_SUCCESS	Function has been called with incorrect value of the *Fr_channel* parameter	
Description:	This function configures the FlexRay module for sending Media Access Test Symbol (MTS) on a given channel. If an MTS is to be transmitted in a certain communication cycle, this function should be called during the static segment of the preceding communication cycle. For transmitting an MTS on both channels, it is possible to call the ***Fr_send_MTS*** function with *Fr_channel* parameter configured once to the FR_CHANNEL_A and next to the FR_CHANNEL_B		

Example of usage:	`Fr_return_type return_value;`
	
	`return_value = Fr_send_MTS(FR_CHANNEL_B, &Fr_MTS_B_cfg);`		

3.2.11 Fr_get_MTS_state

Function name:	**Fr_get_MTS_state**		
Syntax	Fr_MTS_state_type Fr_get_MTS_state (Fr_channel_type Fr_channel)		
When called:	anytime		
Parameters (in):	Fr_channel	FR_CHANNEL_A	Media test symbol state will be checked for channel A
		FR_CHANNEL_B	Media test symbol state will be checked for channel B
Parameters (out):	none		
Return value:	FR_MTS_RCV	A valid MTS has been received	
	FR_MTS_RCV_SYNERR	A valid MTS has been received and a Syntax Error was detected	
	FR_MTS_RCV_BVIO	A valid MTS has been received and a Boundary Violation has been detected	
	FR_MTS_RCV_SYNERR_ BVIO	A valid MTS has been received and a Syntax Error and a Boundary Violation has been detected	
	FR_MTS_NOT_RCV	No valid MTS has been received or function has been called with an incorrect value of the *Fr_channel* parameter	
	FR_MTS_NOT_RCV_ SYNERR	No valid MTS has been received and a Syntax Error was detected	
	FR_MTS_NOT_RCV_BVIO	No valid MTS has been received and a Boundary Violation has been detected	
	FR_MTS_NOT_RCV_ SYNERR_BVIO	No valid MTS has been received and a Syntax Error and a Boundary Violation has been detected	
	FR_MTS_UNKNOWN	Returned value if an error has occured	
Description:	This function retrieves the MTS state on a given channel. If this function is called with an incorrect *Fr_channel* parameter, the FR_MTS_NOT_RCV value will be returned		

Example of usage:	`Fr_MTS_state_type mts_return_value;`
	
	`mts_return_value = Fr_get_MTS_state(FR_CHANNEL_A);`		

3.3 Data Transmission and Reception Support

3.3.1 Fr_transmit_data

Function name:	**Fr_transmit_data**	
Syntax	Fr_tx_MB_status_type Fr_transmit_data (uint16 Fr_buffer_idx const uint16 *Fr_data_ptr, uint8 Fr_data_length)	
When called:	*POC:normal active*	
Parameters (in):	Fr_buffer_idx	This index is used to select a transmit/receive message buffer of the FlexRay module
	Fr_data_ptr	This reference points to an array where the data to be transmitted is stored
	Fr_data_length	Determines the length of the data to be transmitted. If the Fr_data_length input parameter is zero, the function determines (reads) the length of data directly from the FlexRay module
Parameters (out):	none	
Return value:	FR_TXMB_UPDATED	The transmit message buffer has been successfully updated with new data
	FR_TXMB_NO_ACCESS	The transmit message buffer has not been successfully locked
Description:	This function is used to transmit a data from a given memory array. If the Fr_data_length parameter is zero, the function determines (reads) the length of the data directly from the FlexRay module. If the FlexRay UNIFIED Driver detects that it cannot lock the message buffer, the FR_TXMB_NO_ACCESS parameter is returned. The function clears the message buffer interrupt flag even if the buffer has not been locked	

Example of usage:	`Fr_tx_MB_status_type tx_return_value;`
	`uint16 tx_data_4[16] = {0};`
	
	`tx_return_value = Fr_transmit_data(4, &tx_data_4[0], 16);`
	`if(tx_return_value == FR_TXMB_UPDATED)`
	`{`
	`}`	

3.3.2 Fr_receive_data

Function name:	**Fr_receive_data**	
Syntax	Fr_rx_MB_status_type Fr_receive_data (uint16 Fr_buffer_idx uint16 *Fr_data_ptr, uint8 *Fr_data_length_ptr, uint16 *Fr_slot_status_ptr)	
When called:	*POC:normal active*	
Parameters (in):	Fr_buffer_idx	This index is used to select a transmit/receive message buffer of FlexRay module
	Fr_data_ptr	This reference points to an array where the data to be received shall be stored
Parameters (out):	Fr_data_length_ptr	This reference points to the memory location where the length of received data shall be stored
	Fr_slot_status_ptr	This reference points to the memory location where the slot status of received data shall be stored
Return value:	FR_RXMB_RECEIVED	Data has been received
	FR_RXMB_NOT_RECEIVED	Data has not been received
	FR_RXMB_NULL_FRAME_ RECEIVED	Received frame is a null frame
	FR_RXMB_NO_ACCESS	The receive message buffer has not been successfully locked
Description:	This function is used to receive a data and copy them to given memory array. If a data has been successfully received, the data is copied to the buffer referenced by the Fr_data_ptr pointer, and FR_RXMB_RECEIVED parameters is returned. If data has not been received, no copying takes place to the message array referenced by the Fr_data_ptr pointer, and the FR_RXMB_NOT_RECEIVED parameter is returned. If a null frame has been received, no copying takes place to the message array referenced by the Fr_data_ptr pointer, the FR_RXMB_NULL_FRAME_RECEIVED parameter is returned and the length “0” is stored into memory location referenced by the Fr_data_length_ptr pointer. If the FlexRay UNIFIED Driver detects that it cannot lock the message buffer, the FR_RXMB_NO_ACCESS parameter is returned. The function clears the message buffer interrupt flag	

Example of usage:	`Fr_rx_MB_status_type rx_return_value;`
	`uint16 rx_data_1[16] = {0};`
	`uint8 rx_data_length = 0;`
	`uint16 rx_status_slot = 0;`
	
	`rx_return_value = Fr_receive_data(1, &rx_data_1[0],`
	`&rx_data_length, &rx_status_slot);`
	
	`if(rx_return_value == FR_RXMB_RECEIVED)`
	`{`
	`}`

3.3.3 Fr_receive_fifo_data

Function name:	**Fr_receive_fifo_data**	
Syntax	Fr_FIFO_status_type Fr_receive_fifo_data (uint16 Fr_buffer_read_idx, uint16 *Fr_data_ptr, uint8 *Fr_data_length_ptr, uint16 *Fr_slot_idx_ptr, uint16 *Fr_slot_status_ptr)	
When called:	*POC:normal active*	
Parameters (in):	Fr_buffer_read_idx	This index is used to select the next available receive FIFO buffer that the application can read
	Fr_data_ptr	This reference points to an array where the data to be received shall be stored
Parameters (out):	Fr_data_length	This reference points to the memory location where the length of received data shall be stored
	Fr_slot_idx_ptr	This reference points to the memory location where the received frame ID will be stored
	Fr_slot_status	This reference points to the memory location where the slot status of received data shall be stored
Return value:	FR_FIFO_RECEIVED	Data has been received
	FR_FIFO_NOT_RECEIVED	Data has not been received
Description:	This function is used to receive a data and copy them to a given memory array. If a data has been successfully received, data is copied to the buffer referenced by the Fr_data_ptr pointer, and the FR_FIFO_RECEIVED parameter is returned. If a data has not been received, no copying takes place to the message array referenced by the Fr_data_ptr pointer, and the FR_FIFO_NOT_RECEIVED parameter is returned.	

Example of usage:

```
uint16 fifo_data[8] = {0};
uint8 fifo_data_length = 0;
uint16 fifo_status_slot = 0;
uint16 fifo_slot_idx = 0;
Fr_FIFO_status_type fifo_return_value;

/* The FIFO interrupt has to be enabled and the
Fr_set_fifo_IRQ_callback() function has to be called with the name
of the callback service routine */

Fr_set_fifo_IRQ_callback(&CC_interrupt_FIFO_A, FR_FIFO_A_IRQ);

Fr_enable_interrupts(FR_FIFO_A_IRQ, 0, 0);


void CC_interrupt_FIFO_A(uint16 header_idx)
{
/* The input parameter header_idx is passed by the
Fr_interrupt_handler() function */

    fifo_return_value = Fr_receive_fifo_data(header_idx,
    &fifo_data[0], &fifo_data_length, &fifo_slot_idx,
    &fifo_status_slot);

    if(fifo_return_value == FR_FIFO_RECEIVED)
    {
    }
}
```

3.4 Poll Driven Mode Support

3.4.1 Fr_check_tx_status

Function name:	**Fr_check_tx_status**	
Syntax	Fr_tx_status_type Fr_check_tx_status (uint8 Fr_buffer_idx)	
When called:	*POC:normal active*	
Parameters (in):	Fr_buffer_idx	This index is used to select a transmit message buffer of FlexRay module
Parameters (out):	none	
Return value:	FR_TRANSMITTED	Single buffer: data has been transmitted Double buffer: commit side - transferred transmit side - transmitted
	FR_NOT_TRANSMITTED	Single buffer: data has not been transmitted Double buffer: commit side - not transferred transmit side - not transmitted
	FR_INTERNAL_ MESSAGE_TRANSFER_ DONE	Applied only for double buffered message buffer commit side - transferred transmit side - not transmitted
	FR_TRANSMIT_SIDE_ TRANSMITTED	Applied only for double buffered message buffer commit side - not transferred transmit side - transmitted
Description:	This function checks whether the transmission of the transmit message buffer referenced by the Fr_buffer_idx input parameter has been performed. If a double buffered message buffer is configured, the FlexRay UNIFIED Driver allocates two message buffers (in the ***Fr_buffers_init()*** function), one for the commit side (with an even, lower number) and one for the transmit side (with an odd, higher number). The host must always pass the number of the commit side into the first parameter of the ***Fr_check_tx_status()*** function	

Example of usage:	`Fr_tx_status_type tx_status;`
	
	`/* Check whether data has been transferred or transmitted */`
	`tx_status = Fr_check_tx_status(4);`
	
	`/* Has Internal Message Transfer been performed? */`
	`if((tx_status == FR_TRANSMITTED) || (tx_status == FR_INTERNAL_MESSAGE_TRANSFER_DONE))`
	`{`
	`}`	

3.4.2 Fr_check_rx_status

Function name:	**Fr_check_rx_status**	
Syntax	Fr_rx_status_type Fr_check_rx_status (uint8 Fr_buffer_idx)	
When called:	*POC:normal active*	
Parameters (in):	Fr_buffer_idx	This index is used to select a receive message buffer of FlexRay module
Parameters (out):	none	
Return value:	FR_RECEIVED FR_NOT_RECEIVED FR_NULL_FRAME_RECEIVED	Valid non-null frame has been received No valid frame has been received Null frame has been received
Description:	Check whether or not the reception of the frame has been performed. The FlexRay UNIFIED Driver decides which message buffer is checked according to the Fr_buffer_idx input parameter	

Example of usage:	`Fr_rx_status_type rx_status;`
	
	`rx_status = Fr_check_rx_status(5);`
	
	`if(rx_status == FR_RECEIVED)`
	`{`
	`}`	

3.4.3 Fr_check_CHI_error

Function name:	**Fr_check_CHI_error**
Syntax	Fr_CHI_error_type Fr_check_CHI_error (void)
When called:	anytime
Parameters (in):	none
Parameters (out):	none
Return value:	The current content of the CHIERFR register (16-bit format)
Description:	Check whether or not the CHI related error flags have been occurred. The function returns current content of the CHIERFR register (16-bit format). It is up to the host application to deal with possible error flag(s). For more information about the CHIERFR register, please see [FLEXRAY_MODULE]. The function clears pending flags (in the CHIERFR register)

Example of usage:	`Fr_CHI_error_type chi_error;`
	
	`chi_error = Fr_check_CHI_error();`
	
	`if(chi_error != 0)`
	`{`
	`}`

3.4.4 Fr_check_cycle_start

Function name:	**Fr_check_cycle_start**	
Syntax	boolean Fr_check_cycle_start (uint8 *Fr_cycle_ptr)	
When called:	*POC:normal active*	
Parameters (in):	none	
Parameters (out):	Fr_cycle_ptr	Reference to a memory location where the number of the current FlexRay communication cycle is stored
Return value:	FALSE	Communication cycle has not been started
	TRUE	Communication cycle started
Description:	Check whether or not the FlexRay communication cycle has been started. However, if the host application calls this function at the end of a communication cycle and the flag has not yet been cleared in the current communication cycle, the function also returns a TRUE value. The function clears any pending flag and can be used for synchronisation with the FlexRay global time without using an interrupt (its use is shown in [UNIFIED_APP_EXAMPLES])	

Example of usage:	`uint8 current_cycle;`
	`boolean cycle_starts;`
	
	`cycle_starts = Fr_check_cycle_start(&current_cycle);`
	`if(cycle_starts)`
	`{`
	`}`	

3.4.5 Fr_check_transmission_across_boundary

Function name:	**Fr_check_transmission_across_boundary**		
Syntax	boolean Fr_check_transmission_across_boundary (Fr_channel_type Fr_channel)		
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*		
Parameters (in):	Fr_channel	FR_CHANNEL_A	An error flag is tested for channel A
		FR_CHANNEL_B	An error flag is tested for channel B
		FR_CHANNEL_AB	Error flags are tested for both channels
Parameters (out):	none		
Return value:	FALSE	The frame transmission across boundary error flag has not been occurred	
	TRUE	The frame transmission across boundary error flag has been occurred	
Description:	This function checks whether or not the frame transmission across boundary error flag has been occurred. If the FR_CHANNEL_AB parameter is placed into the Fr_channel input parameter, the function tests both channels and returns a TRUE value, in case where error flag occurs on at least one of the channels. The function clears pending interrupt flag		

Example of usage:	`boolean tx_across;`
	
	`tx_across = Fr_check_transmission_across_boundary(FR_CHANNEL_AB);`
	`if(tx_across)`
	`{`
	`}`		

3.4.6 Fr_check_violation

Function name:	**Fr_check_violation**		
Syntax	boolean Fr_check_violation (Fr_channel_type Fr_channel)		
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*		
Parameters (in):	Fr_channel	FR_CHANNEL_A	An error flag is tested for channel A
		FR_CHANNEL_B	An error flag is tested for channel B
		FR_CHANNEL_AB	Error flags are tested for both channels
Parameters (out):	none		
Return value:	FALSE	The violation flag has not been occurred	
	TRUE	The violation flag has been occurred	
Description:	This function checks whether or not a frame transmission in dynamic segment has exceeded the dynamic segment boundary. If the FR_CHANNEL_AB parameter is placed, the function tests both channels and returns a TRUE value, in case where error flag occurs on at least one of the channels. The function clears pending interrupt flag		

Example of usage:	`boolean tx_violation;`
	
	`tx_violation = Fr_check_violation(FR_CHANNEL_AB);`
	`if(tx_violation)`
	`{`
	`}`		

3.4.7 Fr_check_max_sync_frame

Function name:	**Fr_check_max_sync_frame**	
Syntax	boolean Fr_check_max_sync_frame (void)	
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FALSE	The number of synchronization frames has not exceeded a value of the *node_sync_max*
	TRUE	The number of synchronization frames detected in the current communication cycle has exceeded a value of the *node_sync_max*
Description:	This function checks whether or not the number of synchronization frames detected in the current communication cycle has exceeded a value of the *node_sync_max* field in the *Protocol Configuration Register 30*, for more detail see [FLEXRAY_MODULE]. The function clears pending interrupt flag	

Example of usage:	`boolean max_sync_frames;`
	
	`max_sync_frames = Fr_check_max_sync_frame();`
	`if(max_sync_frames)`
	`{`
	`}`	

3.4.8 Fr_check_clock_correction_limit_reached

Function name:	**Fr_check_clock_correction_limit_reached**
Syntax	boolean Fr_check_clock_correction_limit_reached (void)
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*
Parameters (in):	none
Parameters (out):	none
Return value:	FALSE — Offset or rate correction limit has not been reached TRUE — Offset or rate correction limit has been reached
Description:	This function checks whether or not the internal calculated values have reached or exceeded the configured thresholds, as given by the *offset_correction_out* field in the *Protocol Configuration Register 9* and the *rate_correction_out* field in the *Protocol Configuration Register 14*, for more detail see [FLEXRAY_MODULE]. The function clears pending interrupt flag

Example of usage:	`boolean clk_limit_reached;`
	
	`clk_limit_reached = Fr_check_clock_correction_limit_reached();`
	`if(clk_limit_reached)`
	`{`
	`}`

3.4.9 Fr_check_missing_offset_correction

Function name:	**Fr_check_missing_offset_correction**
Syntax	boolean Fr_check_missing_offset_correction (void)
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*
Parameters (in):	none
Parameters (out):	none
Return value:	FALSE — An insufficient number of measurements for offset correction have not been detected TRUE — An insufficient number of measurements for offset correction have been detected
Description:	This function checks whether or not an insufficient number of measurements are available for offset correction. The function clears pending interrupt flag

Example of usage:	`boolean offset_correction;`
	
	`offset_correction = Fr_check_missing_offset_correction();`
	`if(offset_correction)`
	`{`
	`}`

3.4.10 Fr_check_missing_rate_correction

Function name:	**Fr_check_missing_rate_correction**
Syntax	boolean Fr_check_missing_rate_correction (void)
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*
Parameters (in):	none
Parameters (out):	none
Return value:	FALSE — An insufficient number of measurements for rate correction have not been detected TRUE — An insufficient number of measurements for rate correction have been detected
Description:	This function checks whether or not an insufficient number of measurements are available for rate correction. The function clears pending interrupt flag

Example of usage:	`boolean rate_correction;`
	
	`rate_correction = Fr_check_missing_rate_correction();`
	`if(rate_correction)`
	`{`
	`}`

3.4.11 Fr_check_coldstart_abort

Function name:	**Fr_check_coldstart_abort**	
Syntax	boolean Fr_check_coldstart_abort (void)	
When called:	*POC:startup*, *POC:normal active*, *POC:normal passive*, *POC:halt*	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FALSE	The configured number of allowed cold start attempts has not been reached
	TRUE	The configured number of allowed cold start attempts has been reached and none of these attempts were successful
Description:	This function checks whether or not the configured number of allowed cold start attempts has been reached and none of these attempts were successful. The function clears pending interrupt flag	

Example of usage:	`boolean coldstart_max;`
	
	`coldstart_max = Fr_check_coldstart_abort();`
	`if(coldstart_max)`
	`{`
	`}`	

3.4.12 Fr_check_internal_protocol_error

Function name:	**Fr_check_internal_protocol_error**	
Syntax	boolean Fr_check_internal_protocol_error (void)	
When called:	*POC:halt*	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FALSE	The protocol engine has not detected an internal protocol error
	TRUE	The protocol engine has detected an internal protocol error
Description:	This function checks whether or not the protocol engine has detected an internal protocol error. In this case, the protocol engine goes into the *POC:halt* state immediately. An internal protocol error occurs when the protocol engine has not finished a calculation and a new calculation has been requested. For subsequent FlexRay communication, a re-configuration of the FlexRay module is necessary. The function clears pending interrupt flag	

Example of usage:	`boolean internal_error;`
	
	`internal_error = Fr_check_internal_protocol_error();`
	`if(internal_error)`
	`{`
	`}`	

3.4.13 Fr_check_fatal_protocol_error

Function name:	**Fr_check_fatal_protocol_error**
Syntax	boolean Fr_check_fatal_protocol_error (void)
When called:	*POC:halt*
Parameters (in):	none
Parameters (out):	none
Return value:	FALSE The protocol engine has not detected a fatal protocol error TRUE The protocol engine has detected a fatal protocol error
Description:	This function checks whether or not the protocol engine has detected a fatal protocol error. In this case, the protocol engine goes into the *POC:halt* state immediately. The fatal protocol errors are: 1) *pLatestTx* violation as described in the MAC process of the [FR_PROTOCOL] 2) transmission across slot boundary violation as described in the FSP process of the [FR_PROTOCOL]. For subsequent FlexRay communication, a re-configuration of the FlexRay module is necessary. The function clears pending interrupt flag

Example of usage:	`boolean fatal_error;`
	
	`fatal_error = Fr_check_fatal_protocol_error();`
	`if(fatal_error)`
	`{`
	`}`

3.4.14 Fr_check_protocol_state_changed

Function name:	**Fr_check_protocol_state_changed**
Syntax	boolean Fr_check_protocol_state_changed (void)
When called:	anytime
Parameters (in):	none
Parameters (out):	none
Return value:	FALSE The protocol state has not been changed TRUE The protocol state has been changed
Description:	This function checks whether or not the protocol state has been changed. The function clears pending interrupt flag

Example of usage:	`boolean poc_changed;`
	
	`poc_changed = Fr_check_protocol_state_changed();`
	`if(poc_changed)`
	`{`
	`}`

3.4.15 Fr_check_protocol_engine_com_failure

Function name:	**Fr_check_protocol_engine_com_failure**	
Syntax	boolean Fr_check_protocol_engine_com_failure (void)	
When called:	anytime	
Parameters (in):	none	
Parameters (out):	none	
Return value:	FALSE	The protocol engine communication failure has not been detected
	TRUE	The protocol engine communication failure has been detected
Description:	This function checks whether or not the protocol engine communication failure has been detected. A protocol engine communication failure occurs when there is a communication failure between the protocol engine and the controller host interface. The function clears pending interrupt flag	

Example of usage:	`boolean engine_failure;`
	
	`engine_failure = Fr_check_protocol_engine_com_failure();`
	`if(engine_failure)`
	`{`
	`}`	

3.4.16 Fr_get_global_time

Function name:	**Fr_get_global_time**	
Syntax	void Fr_get_global_time (uint8 * Fr_cycle_ptr, uint16 * Fr_macrotick_ptr)	
When called:	anytime	
Parameters (in):	none	
Parameters (out):	Fr_cycle_ptr	Reference to a memory location where the current FlexRay communication cycle value will be stored
	Fr_macrotick_ptr	Reference to a memory location where the current FlexRay communication macrotick value will be stored
Return value:	none	
Description:	This function queries the global time of the FlexRay cluster - current cycle and macrotick values. These values are stored in a memory locations referenced by the output parameters	

Example of usage:	`uint8 current_cycle;`
	`uint16 current_macrotick;`
	
	`Fr_get_global_time(&current_cycle, &current_macrotick);`	

3.4.17 Fr_get_network_management_vector

Function name:	**Fr_get_network_management_vector**	
Syntax	void Fr_get_network_management_vector (uint16 * Fr_vector_ptr)	
When called:	*POC:normal active*	
Parameters (in):	none	
Parameters (out):	Fr_vector_ptr	This reference points to a array where the network management vectors shall be stored
Return value:	none	
Description:	This function queries the vector management vectors. The vectors are stored in a host application array referenced by the output parameter. It is convenient to use this FlexRay module functionality for detection which node in the FlexRay cluster is connected	

Example of usage:	`uint16 network_management_vector[1] = {0};`
	
	`Fr_get_network_management_vector(&network_management_vector[0]);`
	
	`switch(network_management_vector[0])`
	`{`
	`}`	

3.5 Status Monitoring Support

3.5.1 Fr_get_channel_status_error_counter_value

Function name:	**Fr_get_channel_status_error_counter_value**		
Syntax	void Fr_get_channel_status_error_counter_value (Fr_channel_type Fr_channel, uint16 *Fr_counter_value_ptr)		
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*		
Parameters (in):	Fr_channel	FR_CHANNEL_A	Channel A Status Error Counter value is stored to given memory location
		FR_CHANNEL_B	Channel B Status Error Counter value is stored to given memory location
Parameters (out):	Fr_counter_value_ptr	Reference to a memory location where the required error counter value will be stored	
Return value:	none		
Description:	This function queries the current value of the channel status error counter for a required channel. The protocol engine generates a status vector for each static slot, each dynamic slot, the symbol window, and the NIT. The status vector contains the four protocol related error indicator bits. The FlexRay module increments the error status counter by 1 if, for a slot or segment, at least one error bit is set to 1. The counter wraps around after it has reached the maximum value. See [FLEXRAY_MODULE], the *CASERCR* and *CBSERCR* registers for more information		

Example of usage:	`uint16 error_counter_A = 0;`
	
	`Fr_get_channel_status_error_counter_value(FR_CHANNEL_A, &error_counter_A);`		

3.5.2 Fr_get_slot_status_reg_value

Function name:	**Fr_get_slot_status_reg_value**		
Syntax	Fr_return_type Fr_get_slot_status_reg_value (uint16 Fr_slot_number, Fr_channel_type Fr_channel, Fr_slot_status_required_type Fr_slot_status_required, uint16 *Fr_status_vector_ptr)		
When called:	*POC:normal active*		
Parameters (in):	Fr_slot_number	This index is used to select a slot for which status is read	
	Fr_channel	FR_CHANNEL_A	Channel A is used for slot number comparison
		FR_CHANNEL_B	Channel B is used for slot number comparison
	Fr_slot_status_required	FR_SLOT_STATUS_CURRENT	The newest updated slot status information is required from a given slot
		FR_SLOT_STATUS_PREVIOUS	Not the historic slot status information is required from a given slot
Parameters (out):	Fr_status_vector_ptr	Reference to a memory location where the required status vector is stored	
Return value:	FR_SUCCESS	API call has been successful	
	FR_NOT_SUCCESS	Slot status registers have not been configured or slot status monitoring is not configured for required slot number	
Description:	This function gets the status vector of the required slot for the static/dynamic segment, for the symbol window or for the NIT, on a per channel basis. See section 2.1.5.2 Slot Status Registers for more details on the FlexRay UNIFIED Driver Slot Status Registers implementation		

Example of usage:

```
Fr_return_type return_value;
uint16 slot_status_4;

return_value = Fr_get_slot_status_reg_value(4, FR_CHANNEL_A,
FR_SLOT_STATUS_CURRENT, &slot_status_4);

if(return_value == FR_NOT_SUCCESS)
{
}

// Determine which channels are connected
// Has been a valid frame received on channel A?
if(slot_status_4 & FrSSRn_VFA)
{
}

// Has been a valid frame received on channel B?
if(slot_status_4 & FrSSRn_VFB)
{
}
```

3.5.3 Fr_get_slot_status_counter_value

Function name:	**Fr_get_slot_status_counter_value**		
Syntax	void Fr_get_slot_status_counter_value (Fr_slot_status_counter_ID_type Fr_counter_idx, uint16 *Fr_counter_value_ptr)		
When called:	*POC:normal active*, *POC:normal passive*, *POC:halt*		
Parameters (in):	Fr_counter_idx	FR_SLOT_STATUS_COUNTER_0	Content of the Slot Status Counter 0 is stored to given memory location
		FR_SLOT_STATUS_COUNTER_1	Content of the Slot Status Counter 1 is stored to given memory location
		FR_SLOT_STATUS_COUNTER_2	Content of the Slot Status Counter 2 is stored to given memory location
		FR_SLOT_STATUS_COUNTER_3	Content of the Slot Status Counter 3 is stored to given memory location
Parameters (out):	Fr_counter_value_ptr	Reference to a memory location where the required slot status counter value is stored	
Return value:	none		
Description:	This function reads the current value of relevant slot status counter. See section 2.1.5.3 Slot Status Counter Registers for more details on the FlexRay UNIFIED Driver Slot Status Counter Registers implementation		

Example of usage:	`uint16 slot_status_counter_0;`
	
	`Fr_get_slot_status_counter_value(FR_SLOT_STATUS_COUNTER_0, &slot_status_counter_0);`		

3.5.4 Fr_reset_slot_status_counter

Function name:	**Fr_reset_slot_status_counter**		
Syntax	void Fr_reset_slot_status_counter (Fr_slot_status_counter_ID_type Fr_counter_idx)		
When called:	*POC:normal active*		
Parameters (in):	Fr_counter_idx	FR_SLOT_STATUS_COUNTER_0	The Slot Status Counter 0 is reset
		FR_SLOT_STATUS_COUNTER_1	The Slot Status Counter 1 is reset
		FR_SLOT_STATUS_COUNTER_2	The Slot Status Counter 2 is reset
		FR_SLOT_STATUS_COUNTER_3	The Slot Status Counter 3 is reset
Parameters (out):	none		
Return value:	none		
Description:	This function resets required slot status counter. If the counter is in the multicycle mode (i.e. SSCCRn[MCY] = 1), the application can reset the counter by calling the ***Fr_reset_slot_status_counter()*** function and wait for the next cycle start, when the FlexRay module clears the counter. Subsequently, the counter can be set into the multicycle mode again by means of the ***Fr_slot_status_counter_init()*** function. See section 2.1.5.3 Slot Status Counter Registers for more details on the FlexRay UNIFIED Driver Slot Status Counter Registers implementation		
Example of usage:	`Fr_reset_slot_status_counter(FR_SLOT_STATUS_COUNTER_0);`		

3.6 Interrupt Support

3.6.1 Fr_enable_interrupts

Function name:	**Fr_enable_interrupts**		
Syntax	void Fr_enable_interrupts (uint16 Fr_global_interrupt, uint16 Fr_protocol_0_interrupt, uint16 Fr_protocol_1_interrupt)		
When called:	*POC:config*, *POC:ready*, *POC:normal active*, *POC:normal passive*, *POC:halt*		
Parameters (in):	Fr_global_interrupt	FR_MODULE_IRQ	Enable the Module Interrupt
		FR_PROTOCOL_IRQ	Enable the Protocol Interrupt
		FR_CHI_IRQ	Enable the CHI Interrupt
		FR_WAKEUP_IRQ	Enable the Wakeup Interrupt
		FR_FIFO_B_IRQ	Enable the Receive FIFO Channel B Not Empty Interrupt
		FR_FIFO_A_IRQ	Enable the Receive FIFO Channel A Not Empty Interrupt
		FR_RECEIVE_IRQ	Enable the Receive Buffer Interrupt
		FR_TRANSMIT_IRQ	Enable the Transmit Buffer Interrupt
	Fr_protocol_0_ interrupt	FR_FATAL_PROTOCOL_ ERROR_IRQ	Enable Fatal Protocol Error Interrupt
		FR_INTERNAL_PROTOCOL_ ERROR_IRQ	Enable Internal Protocol Error Interrupt
		FR_ILLEGAL_PROTOCOL_ CONFIGURATION_IRQ	Enable Illegal Protocol Configuration Interrupt
		FR_COLDSTART_ABORT_ IRQ	Enable Cold Start Abort Interrupt
		FR_MISSING_RATE_ CORRECTION_IRQ	Enable Missing Rate Correction Interrupt
		FR_MISSING_OFFSET_ CORRECTION_IRQ	Enable Missing Offset Correction Interrupt
		FR_CLOCK_CORRECTION_ LIMIT_IRQ	Enable Clock Correction Limit Reached Interrupt
		FR_MAX_SYNC_FRAMES_ DETECTED_IRQ	Enable Max Sync Frames Detected Interrupt

		FR_MEDIA_ACCESS_TEST_SYMBOL_RECEIVED_IRQ	Enable Media Test Symbol Received Interrupt
		FR_VIOLATION_CHB_IRQ	Enable Violation on Channel B Interrupt
		FR_VIOLATION_CHA_IRQ	Enable Violation on Channel A Interrupt
		FR_TRANSMISSION_ACROSS_BOUNDARY_CHB_IRQ	Enable Transmission Across Boundary on Channel B Interrupt
		FR_TRANSMISSION_ACROSS_BOUNDARY_CHA_IRQ	Enable Transmission Across Boundary on Channel A Interrupt
		FR_TIMER_2_EXPIRED_IRQ	Enable Timer 2 Expired Interrupt
		FR_TIMER_1_EXPIRED_IRQ	Enable Timer 1 Expired Interrupt
		FR_CYCLE_START_IRQ	Enable Cycle Start Interrupt
	Fr_protocol_1_interrupt	FR_ERROR_MODE_CHANGED_IRQ	Enable Error Mode Changed Interrupt
		FR_ILLEGAL_PROTOCOL_COMMAND_IRQ	Enable Illegal Protocol Command Interrupt
		FR_PROTOCOL_ENGINE_COM_FAILURE	Enable Protocol Engine Communication Failure
		FR_PROTOCOL_STATE_CHANGED_IRQ	Enable Protocol State Changed Interrupt
		FR_SLOT_STATUS_COUNTER_3_INCREMENTED_IRQ	Enable Slot Status Counter 3 Incremented Interrupt
		FR_SLOT_STATUS_COUNTER_2_INCREMENTED_IRQ	Enable Slot Status Counter 2 Incremented Interrupt
		FR_SLOT_STATUS_COUNTER_1_INCREMENTED_IRQ	Enable Slot Status Counter 1 Incremented Interrupt
		FR_SLOT_STATUS_COUNTER_0_INCREMENTED_IRQ	Enable Slot Status Counter 0 Incremented Interrupt
		FR_EVEN_CYCLE_TABLE_WRITTEN_IRQ	Enable Even Cycle Table Written Interrupt
		FR_ODD_CYCLE_TABLE_WRITTEN_IRQ	Enable Odd Cycle Table Written Interrupt
Parameters (out):	none		
Return value:	none		

Description:	This function enables an interrupt in the GIFER, PIER0 and PIER1 registers. The host application passes the combination of the pre-defined macros into three input parameters. If the Fr_global_interrupt input parameter is zero, this function does not update the GIFER register, otherwise the GIFER register is updated. If the Fr_protocol_0_interrupt parameter is zero, this function does not update the PIER0 register, otherwise the PIER0 register is updated. If the Fr_protocol_1_interrupt parameter is zero, this function does not update the PIER1 register, otherwise the PIER1 register is updated. The FR_MODULE_IRQ parameter should always be passed in the first Fr_global_interrupt input parameter in case where the MFR4300 CC is used (no module service callback function has to be defined by the ***Fr_set_global_IRQ_callback()*** function). The FR_PROTOCOL_IRQ parameter should always be passed in the first Fr_global_interrupt input parameter in case where the Fr_protocol_0_interrupt or Fr_protocol_1_interrupt input parameters are not null (an interrupt in the PIER0 or PIER1 registers is enabled)
Example of usage:	`Fr_enable_interrupts((FR_PROTOCOL_IRQ | FR_FIFO_A_IRQ | FR_RECEIVE_IRQ | FR_TRANSMIT_IRQ), (FR_TIMER_1_EXPIRED_IRQ | FR_TIMER_2_EXPIRED_IRQ | FR_CYCLE_START_IRQ), 0);`

3.6.2 Fr_disable_interrupts

Function name:	**Fr_disable_interrupts**		
Syntax	void Fr_disable_interrupts (uint16 Fr_global_interrupt, uint16 Fr_protocol_0_interrupt, uint16 Fr_protocol_1_interrupt)		
When called:	*POC:config*, *POC:ready*, *POC:normal active*, *POC:normal passive*, *POC:halt*		
Parameters (in):	Fr_global_interrupt	FR_MODULE_IRQ	Disable the Module Interrupt
		FR_PROTOCOL_IRQ	Disable the Protocol Interrupt
		FR_CHI_IRQ	Disable the CHI Interrupt
		FR_WAKEUP_IRQ	Disable the Wakeup Interrupt
		FR_FIFO_B_IRQ	Disable the Receive FIFO Channel B Not Empty Interrupt
		FR_FIFO_A_IRQ	Disable the Receive FIFO Channel A Not Empty Interrupt
		FR_RECEIVE_IRQ	Disable the Receive Buffer Interrupt
		FR_TRANSMIT_IRQ	Disable the Transmit Buffer Interrupt
	Fr_protocol_0_ interrupt	FR_FATAL_PROTOCOL_ ERROR_IRQ	Disable Fatal Protocol Error Interrupt
		FR_INTERNAL_PROTOCOL_ ERROR_IRQ	Disable Internal Protocol Error Interrupt
		FR_ILLEGAL_PROTOCOL_ CONFIGURATION_IRQ	Disable Illegal Protocol Configuration Interrupt
		FR_COLDSTART_ABORT_ IRQ	Disable Cold Start Abort Interrupt
		FR_MISSING_RATE_ CORRECTION_IRQ	Disable Missing Rate Correction Interrupt
		FR_MISSING_OFFSET_ CORRECTION_IRQ	Disable Missing Offset Correction Interrupt
		FR_CLOCK_CORRECTION_ LIMIT_IRQ	Disable Clock Correction Limit Reached Interrupt
		FR_MAX_SYNC_FRAMES_ DETECTED_IRQ	Disable Max Sync Frames Detected Interrupt
		FR_MEDIA_ACCESS_TEST_ SYMBOL_RECEIVED_IRQ	Disable Media Test Symbol Received Interrupt

		FR_VIOLATION_CHB_IRQ	Disable Violation on Channel B Interrupt
		FR_VIOLATION_CHA_IRQ	Disable Violation on Channel A Interrupt
		FR_TRANSMISSION_ ACROSS_BOUNDARY_CHB_ IRQ	Disable Transmission Across Boundary on Channel B Interrupt
		FR_TRANSMISSION_ ACROSS_BOUNDARY_CHA_ IRQ	Disable Transmission Across Boundary on Channel A Interrupt
		FR_TIMER_2_EXPIRED_IRQ	Disable Timer 2 Expired Interrupt
		FR_TIMER_1_EXPIRED_IRQ	Disable Timer 1 Expired Interrupt
		FR_CYCLE_START_IRQ	Disable Cycle Start Interrupt
	Fr_protocol_1_ interrupt	FR_ERROR_MODE_ CHANGED_IRQ	Disable Error Mode Changed Interrupt
		FR_ILLEGAL_PROTOCOL_ COMMAND_IRQ	Disable Illegal Protocol Command Interrupt
		FR_PROTOCOL_ENGINE_ COM_FAILURE	Disable Protocol Engine Communication Failure
		FR_PROTOCOL_STATE_ CHANGED_IRQ	Disable Protocol State Changed Interrupt
		FR_SLOT_STATUS_ COUNTER_3_ INCREMENTED_IRQ	Disable Slot Status Counter 3 Incremented Interrupt
		FR_SLOT_STATUS_ COUNTER_2_ INCREMENTED_IRQ	Disable Slot Status Counter 2 Incremented Interrupt
		FR_SLOT_STATUS_ COUNTER_1_ INCREMENTED_IRQ	Disable Slot Status Counter 1 Incremented Interrupt
		FR_SLOT_STATUS_ COUNTER_0_ INCREMENTED_IRQ	Disable Slot Status Counter 0 Incremented Interrupt
		FR_EVEN_CYCLE_TABLE_ WRITTEN_IRQ	Disable Even Cycle Table Written Interrupt
		FR_ODD_CYCLE_TABLE_ WRITTEN_IRQ	Disable Odd Cycle Table Written Interrupt
Parameters (out):	none		
Return value:	none		

Description:	This function disables an interrupt in the GIFER, PIER0 and PIER1 registers. The host application passes the combination of the pre-defined macros into three input parameters. If the Fr_global_interrupt input parameter is zero, this function does not update the GIFER register, otherwise the GIFER register is updated. If the Fr_protocol_0_interrupt parameter is zero, this function does not update the PIER0 register, otherwise the PIER0 register is updated. If the Fr_protocol_1_interrupt parameter is zero, this function does not update the PIER1 register, otherwise the PIER1 register is updated
Example of usage:	`Fr_disable_interrupts(0, (FR_TIMER_1_EXPIRED_IRQ | FR_TIMER_2_EXPIRED_IRQ), 0);`

3.6.3 Fr_interrupt_handler

Function name:	**Fr_interrupt_handler**
Syntax	void Fr_interrupt_handler (void)
When called:	anytime
Parameters (in):	none
Parameters (out):	none
Return value:	none
Description:	This function should be called in an interrupt service routine when an interrupt occurred. If a callback function has been configured for the related interrupt (by the related ***Fr_set_xxx_callback()*** function), this is then called by the ***Fr_interrupt_handler()*** routine. The interrupt flags are cleared except the transmit or receive message buffer interrupts (these are cleared by the ***Fr_transmit_data()*** or the ***Fr_receive_data()***) and the module interrupt. In case where the FR_MODULE_IRQ interrupt is enabled (by the ***Fr_enable_interrupts()*** function) and name of a service callback function for module interrupt is stored in the Fr_module_ptr field by calling the ***Fr_set_global_IRQ_callback()*** function, this callback service routine is called, but no interrupt flags are cleared. It means, that for an interrupt flag clearing, it is also necessary to set a service routine for appropriate individual interrupt (e.g.by the ***Fr_set_wakeup_IRQ_callback()*** function). The module interrupt should always be enabled if the MFR4300 CC is used (however, a service callback routine has not to be defined for the module interrupt). In case where the FR_PROTOCOL_IRQ interrupt is enabled (by the ***Fr_enable_interrupts()*** function) and a name of a service callback function is stored in the Fr_protocol_ptr field by calling the ***Fr_set_global_IRQ_callback()*** function, this callback service routine is called and all protocol interrupt flags in the PFR0 and PFR1 registers are cleared

Example of usage:

```
// Callback service routine
void CC_interrupt_timer_1(void)
{
}
// Callback service routine
void CC_interrupt_timer_2(void)
{
}
// Callback service routine
void CC_interrupt_cycle_start(void)
{
}
// Callback service routine
void CC_interrupt_FIFO_A(uint16 header_idx)
{
}

// The Fr_timers_cfg_00_ptr refers to the timer configuration array
Fr_timers_init(&Fr_timers_cfg_00_ptr);
Fr_set_protocol_0_IRQ_callback(&CC_interrupt_timer_1,
FR_TIMER_1_EXPIRED_IRQ);
Fr_set_protocol_0_IRQ_callback(&CC_interrupt_cycle_start,
FR_CYCLE_START_IRQ);
Fr_set_fifo_IRQ_callback(&CC_interrupt_FIFO_A, FR_FIFO_A_IRQ);

Fr_enable_interrupts((FR_PROTOCOL_IRQ | FR_FIFO_A_IRQ),
(FR_TIMER_1_EXPIRED_IRQ | FR_CYCLE_START_IRQ), 0);

void FLEXRAY_ISR(UWord32 temp) // Interrupt from the FlexRay module
{
    Fr_interrupt_handler();
}
```

3.6.4 Fr_set_MB_callback

Function name:	**Fr_set_MB_callback**	
Syntax	void Fr_set_MB_callback (void (* MB_callback_ptr) (uint8 Fr_int_buffer_idx), uint8 Fr_buffer_idx)	
When called:	anytime	
Parameters (in):	MB_callback_ptr (uint8 Fr_int_buffer_idx)	The reference to the host application callback service routine which is called by the ***Fr_interrupt_handler()*** function when an interrupt occurred
	Fr_buffer_idx	This index is used to select a transmit/receive message buffer for which the callback service routine should be configured
Parameters (out):	none	
Return value:	none	
Description:	This function stores the reference to the host application callback service routine into an internal structure. When a transmit/receive interrupt arises and the ***Fr_interrupt_handler()*** function is called, the function referenced by the MB_callback_ptr pointer is called with the number of message buffer (in the Fr_int_buffer_idx parameter) which generated an interrupt. The function does not set the MBIE bit in the MBCCSRn register or the TBIE/RBIE bit in the GIFER register, i.e. it does not enable a transmit/receive interrupt	

Example of usage:	`// Callback service routine`
	`void CC_interrupt_slot_1(uint8 buffer_idx)`
	`{`
	`}`
	
	`/* The callback service routine (the CC_interrupt_slot_1()) is called if the message buffer 3 generates an interrupt */`
	`Fr_set_MB_callback(&CC_interrupt_slot_1, 3);`	

3.6.5 Fr_set_global_IRQ_callback

Function name:	**Fr_set_global_IRQ_callback**		
Syntax	Fr_return_type Fr_set_global_IRQ_callback (void (* callback_ptr) (void), uint16 Fr_global_interrupt)		
When called:	anytime		
Parameters (in):	callback_ptr (void)	The reference to the function which is called by the ***Fr_interrupt_handler()*** function when an interrupt occurred	
	Fr_global_interrupt	FR_MODULE_IRQ	Store the reference to the callback service routine for the module interrupt
		FR_PROTOCOL_IRQ	Store the reference to the callback service routine for the protocol interrupt
		FR_WAKEUP_IRQ	Store the reference to the callback service routine for the wakeup interrupt
Parameters (out):	none		
Return value:	FR_SUCCESS		API call has been successful
	FR_NOT_SUCCESS		API call has not been successful - a wrong input parameter
Description:	This function stores the reference to the host application callback service routine in an internal structure. When an interrupt arises, the ***Fr_interrupt_handler()*** function calls the callback service routine referenced by the callback_ptr pointer. The function does not enable an interrupt		

Example of usage:	`Fr_return_type return_value;`
	
	`// Callback service routine`
	`void CC_wakeup(Fr_channel_type wakeup_channel)`
	`{`
	`}`
	
	`return_value = Fr_set_global_IRQ_callback(&CC_wakeup, FR_WAKEUP_IRQ);`		

3.6.6 Fr_set_fifo_IRQ_callback

Function name:	**Fr_set_fifo_IRQ_callback**		
Syntax	Fr_return_type Fr_set_fifo_IRQ_callback (void (* callback_ptr) (uint16 header_index), uint16 Fr_global_interrupt)		
When called:	anytime		
Parameters (in):	callback_ptr (void)	The reference to the function which is called by the ***Fr_interrupt_handler()*** function when an interrupt occurred	
	Fr_global_interrupt	FR_FIFO_B_IRQ	Store the reference to the callback service routine for the Receive FIFO channel B Not Empty Interrupt
		FR_FIFO_A_IRQ	Store the reference to the callback service routine for the Receive FIFO channel A Not Empty Interrupt
Parameters (out):	none		
Return value:	FR_SUCCESS		API call has been successful
	FR_NOT_SUCCESS		API call has not been successful - a wrong input parameter
Description:	This function stores the reference to the callback service routine into an internal structure. When an FIFO interrupt arises and the ***Fr_interrupt_handler()*** function is called, the function referenced by the callback_ptr pointer is called with content of the RFARIR or RFBRIR register (in the header_index parameter). This API call does not enable a FIFO interrupt		

Example of usage:	`Fr_return_type return_value;`
	
	`// Callback service routine`
	`void CC_interrupt_FIFO_A(uint16 header_idx)`
	`{`
	`}`
	
	`return_value = Fr_set_fifo_IRQ_callback(&CC_interrupt_FIFO_A, FR_FIFO_A_IRQ);`		

3.6.7 Fr_set_protocol_0_IRQ_callback

Function name:	**Fr_set_protocol_0_IRQ_callback**		
Syntax	Fr_return_type Fr_set_protocol_0_IRQ_callback (void (* callback_ptr) (void), uint16 Fr_protocol_interrupt)		
When called:	anytime		
Parameters (in):	callback_ptr (void)	The reference to the callback service routine which should be called by the ***Fr_interrupt_handler()*** function when an protocol interrupt occurred	
	Fr_protocol_ interrupt	FR_FATAL_PROTOCOL_ ERROR_IRQ	Store the reference to the callback function for the Fatal Protocol Error Interrupt
		FR_INTERNAL_ PROTOCOL_ERROR_IRQ	Store the reference to the callback function for the Internal Protocol Error Interrupt
		FR_ILLEGAL_PROTOCOL_ CONFIGURATION_IRQ	Store the reference to the callback function for the Illegal Protocol Configuration Interrupt
		FR_COLDSTART_ABORT_ IRQ	Store the reference to the callback function for the Cold Start Abort Interrupt
		FR_MISSING_RATE_ CORRECTION_IRQ	Store the reference to the callback function for the Missing Rate Correction Interrupt
		FR_MISSING_OFFSET_ CORRECTION_IRQ	Store the reference to the callback function for the Missing Offset Correction Interrupt
		FR_CLOCK_CORRECTION_ LIMIT_IRQ	Store the reference to the callback function for the Clock Correction Limit Reached Interrupt
		FR_MAX_SYNC_FRAMES_ DETECTED_IRQ	Store the reference to the callback function for the Max Sync Frames Detected Interrupt
		FR_MEDIA_ACCESS_ TEST_SYMBOL_ RECEIVED_IRQ	Store the reference to the callback function for the Media Test Symbol Received Interrupt
		FR_VIOLATION_CHB_IRQ	Store the reference to the callback function for the Violation on Channel B Interrupt
		FR_VIOLATION_CHA_IRQ	Store the reference to the callback function for the Violation on Channel A Interrupt

		FR_TRANSMISSION_ ACROSS_BOUNDARY_ CHB_IRQ	Store the reference to the callback function for the Transmission across boundary on channel B Interrupt
		FR_TRANSMISSION_ ACROSS_BOUNDARY_ CHA_IRQ	Store the reference to the callback function for the Transmission across boundary on channel A Interrupt
		FR_TIMER_2_EXPIRED_ IRQ	Store the reference to the callback function for the Timer 2 Expired Interrupt
		FR_TIMER_1_EXPIRED_ IRQ	Store the reference to the callback function for the Timer 1 Expired Interrupt
		FR_CYCLE_START_IRQ	Store the reference to the callback function for Cycle Start Interrupt
Parameters (out):	none		
Return value:	FR_SUCCESS		API call has been successful
	FR_NOT_SUCCESS		API call has not been successful - a wrong input parameter
Description:	This function stores the reference to the callback service routine into an internal structure. When an protocol interrupt (asserted with the PIFR0 register) arises and the ***Fr_interrupt_handler()*** function is called, the callback service routine referenced by the callback_ptr pointer is called. The function does not enable an interrupt		

Example of usage:	`Fr_return_type return_value;`
	
	`// Callback service routine`
	`void CC_interrupt_timer_1(void)`
	`{`
	`}`
	
	`return_value =`
	`Fr_set_protocol_0_IRQ_callback(&CC_interrupt_timer_1,`
	`FR_TIMER_1_EXPIRED_IRQ);`		

3.6.8 Fr_set_protocol_1_IRQ_callback

Function name:	**Fr_set_protocol_1_IRQ_callback**		
Syntax	Fr_return_type Fr_set_protocol_1_IRQ_callback (void (* callback_ptr) (void), uint16 Fr_protocol_interrupt)		
When called:	anytime		
Parameters (in):	callback_ptr (void)	The reference to the callback service routine which is called by the ***Fr_interrupt_handler()*** function when an protocol interrupt occurred	
	Fr_protocol_ interrupt	FR_ERROR_MODE_ CHANGED_IRQ	Store the reference to the callback function for the Error Mode Changed Interrupt
		FR_ILLEGAL_PROTOCOL_ COMMAND_IRQ	Store the reference to the callback function for the Illegal Protocol Command Interrupt
		FR_PROTOCOL_ENGINE_ COM_FAILURE	Store the reference to the callback function for the Protocol Engine Communication Failure Interrupt
		FR_PROTOCOL_STATE_ CHANGED_IRQ	Store the reference to the callback function for the Protocol State Changed Interrupt
		FR_SLOT_STATUS_ COUNTER_3_ INCREMENTED_IRQ	Store the reference to the callback function for the Slot Status Counter 3 Incremented Interrupt
		FR_SLOT_STATUS_ COUNTER_2_ INCREMENTED_IRQ	Store the reference to the callback function for the Slot Status Counter 2 Incremented Interrupt
		FR_SLOT_STATUS_ COUNTER_1_ INCREMENTED_IRQ	Store the reference to the callback function for the Slot Status Counter 1 Incremented Interrupt
		FR_SLOT_STATUS_ COUNTER_0_ INCREMENTED_IRQ	Store the reference to the callback function for the Slot Status Counter 0 Incremented Interrupt
		FR_EVEN_CYCLE_TABLE_ WRITTEN_IRQ	Store the reference to the callback function for the Even Cycle Counter Incremented Interrupt
		FR_ODD_CYCLE_TABLE_ WRITTEN_IRQ	Store the reference to the callback function for the Odd Cycle Counter Incremented Interrupt
Parameters (out):	none		
Return value:	FR_SUCCESS		API call has been successful

	FR_NOT_SUCCESS	API call has not been successful - a wrong input parameter
Description:	This function stores the reference to the callback service routine into an internal structure. When a protocol interrupt (asserted with the PIFR1 register) arises and the ***Fr_interrupt_handler()*** function is called, the callback service routine referenced by the callback_ptr pointer is called. The function does not enable an interrupt	

Example of usage:	`Fr_return_type return_value;`
	
	`// Callback service routine`
	`void CC_engine_failure(void)`
	`{`
	`}`
	
	`return_value = Fr_set_protocol_1_IRQ_callback(&CC_engine_failure, FR_PROTOCOL_ENGINE_COM_FAILURE);`	

3.6.9 Fr_set_wakeup_IRQ_callback

Function name:	**Fr_set_wakeup_IRQ_callback**	
Syntax	void Fr_set_wakeup_IRQ_callback (void (* callback_ptr) (Fr_channel_type wakeup_channel))	
When called:	anytime	
Parameters (in):	callback_ptr (uint16 chi_error)	The reference to the callback service routine which is called by the ***Fr_interrupt_handler()*** function when the wakeup interrupt occurred
Parameters (out):	none	
Return value:	none	
Description:	This function stores the reference to the callback service routine into an internal structure. When wakeup interrupt arises and the ***Fr_interrupt_handler()*** function is called, the callback service routine referenced by the callback_ptr pointer is called with information on which channel the wakeup symbol has been received (in the wakeup_channel parameter). The function does not enable a wakeup interrupt	

Example of usage:	`// Callback service routine`
	`void CC_wakeup(Fr_channel_type wakeup_channel)`
	`{`
	`}`
	
	`Fr_set_wakeup_IRQ_callback(&CC_wakeup);`	

3.6.10 Fr_set_chi_IRQ_callback

Function name:	**Fr_set_chi_IRQ_callback**	
Syntax	void Fr_set_chi_IRQ_callback (void (* callback_ptr) (uint16 chi_error))	
When called:	anytime	
Parameters (in):	callback_ptr (uint16 chi_error)	The reference to the callback service routine which is called by the ***Fr_interrupt_handler()*** function when a CHI interrupt occurred
Parameters (out):	none	
Return value:	none	
Description:	This function stores the reference to the callback service routine into an internal structure. When a controller host interface (CHI) interrupt arises and the ***Fr_interrupt_handler()*** function is called, the function referenced by the callback_ptr pointer is called with content of the CHIERFR register (in the chi_error parameter). The function does not enable a CHI interrupt	

Example of usage:	`// Callback service routine`
	`void CC_chi(uint16 chi_error)`
	`{`
	`}`
	
	`Fr_set_chi_IRQ_callback(&CC_chi);`	

3.6.11 Fr_clear_MB_interrupt_flag

Function name:	**Fr_clear_MB_interrupt_flag**	
Syntax	void Fr_clear_MB_interrupt_flag (uint8 Fr_buffer_idx)	
When called:	*POC:normal active*	
Parameters (in):	Fr_buffer_idx	This index is used to select a transmit/receive message buffer for which an interrupt flag should be cleared
Parameters (out):	none	
Return value:	none	
Description:	This function clears an interrupt flag of a transmit/receive message buffer. Interrupt flags are cleared also by the ***Fr_transmit_data()*** or ***Fr_receive_data()*** functions. This function can be called, e.g. for clearing an interrupt flag of the transmit side of the double message buffer	

Example of usage:	`Fr_tx_MB_status_type tx_return_value;`
	`uint16 tx_data_1[16] = {0};`
	
	`/* Update double transmit MB with new data`
	`(commit side - MB 2, transmit side - MB 3) */`
	`tx_return_value = Fr_transmit_data(2, &tx_data_1[0], 16);`
	
	`/* It is necessary to clear the flag at transmit side */`
	`Fr_clear_MB_interrupt_flag(3);`	

3.7 Timer Support

3.7.1 Fr_start_timer

Function name:	**Fr_start_timer**
Syntax	void Fr_start_timer (Fr_timer_ID_type timer_ID)
When called:	*POC:normal active*, *POC:normal passive*
Parameters (in):	timer_ID This parameter determines which timer (T1 or T2) to start
Parameters (out):	none
Return value:	none
Description:	This function starts the timer T1 or timer T2

Example of usage:	`void CC_interrupt_timer_1(void)`
	`{`
	`}`
	
	`// The Fr_timers_cfg_00_ptr refers to the timer configuration array`
	`Fr_timers_init(&Fr_timers_cfg_00_ptr);`
	`Fr_set_protocol_0_IRQ_callback(&CC_interrupt_timer_1,`
	`FR_TIMER_1_EXPIRED_IRQ);`
	`Fr_enable_interrupts(FR_PROTOCOL_IRQ, FR_TIMER_1_EXPIRED_IRQ, 0);`
	
	`Fr_start_timer(FR_TIMER_T1);`

3.7.2 Fr_stop_timer

Function name:	**Fr_stop_timer**
Syntax	void Fr_stop_timer (Fr_timer_ID_type timer_ID)
When called:	*POC:normal active*, *POC:normal passive*
Parameters (in):	timer_ID This parameter determines which timer (T1 or T2) to stop
Parameters (out):	none
Return value:	none
Description:	This function stops the timer T1 or timer T2
Example of usage:	`Fr_stop_timer(FR_TIMER_T2);`

3.8 Low Level Access Support

3.8.1 Fr_low_level_access_read_reg

Function name:	**Fr_low_level_access_read_reg**	
Syntax	uint16 Fr_low_level_access_read_reg (uint16 Fr_required_register)	
When called:	anytime	
Parameters (in):	Fr_required_register	This parameter determines which FlexRay register to be read
Parameters (out):	none	
Return value:	uint16	The content of required register
Description:	This function reads the content of the required register	

Example of usage:	`uint16 mvr_register_value;`
	
	`// Read the MVR register`
	`mvr_register_value = Fr_low_level_access_read_reg(FrMVR);`	

3.8.2 Fr_low_level_access_write_reg

Function name:	**Fr_low_level_access_write_reg**	
Syntax	void Fr_low_level_access_write_reg (uint16 Fr_required_register, uint16 Fr_value)	
When called:	anytime, according to the related register write access rules	
Parameters (in):	Fr_required_register	This parameter determines the FlexRay register which should be updated with passed value in the Fr_value parameter
	Fr_value	Value which should be stored into required register
Parameters (out):	none	
Return value:	none	
Description:	This function stores a 16-bit value to the required register	
Example of usage:	`// Store 0x0000 to the PIFR1 register` `Fr_low_level_access_write_reg(FrPIFR1, 0x0000);`	

3.8.3 Fr_low_level_access_read_memory

Function name:	**Fr_low_level_access_read_memory**	
Syntax	uint16 Fr_low_level_access_read_memory (uint16 Fr_memory_address)	
When called:	anytime	
Parameters (in):	Fr_memory_address	This parameter determines which FlexRay memory field to be read. The host has to put the value in bytes and needs to be even number, see FlexRay module documentation [FLEXRAY_MODULE]
Parameters (out):	none	
Return value:	uint16	The content of required FlexRay memory field
Description:	This function reads the content of the required FlexRay memory field. The function uses 16-bit access, however the address of the required memory field has to be put in the number of bytes (as it is described in documentation [FLEXRAY_MODULE], chapter FlexRay Memory Layout). The value of the *Fr_memory_address* parameter is any offset of the corresponding FlexRay memory field with respect to the FlexRay memory base address as provided by the *CC_FlexRay_memory_base_address* parameter in the *Fr_HW_config_type* structure	

Example of usage:	`uint16 data_field_off_value; // Data Field Offset in the message buffer Header Field`
	`uint16 data_mb10[2] = {0};`
	
	`// Read the Data Field Offset of the tenth message buffer Header Field in the FlexRay memory`
	`// Address calculation for Data Field Offset (in bytes):`
	`// (i * 10) + 0x6 ; where i is the message buffer number`
	`// for the tenth message buffer, it is (10 * 10) + 0x6 = 106 bytes`
	`data_field_off_value = Fr_low_level_access_read_memory(106);`
	
	`// Read the data from the message buffer Data Field of the tenth message buffer`
	`data_mb10[0] = Fr_low_level_access_read_memory(data_field_off_value);`	

3.8.4 Fr_low_level_access_write_memory

Function name:	**Fr_low_level_access_write_memory**	
Syntax	void Fr_low_level_access_write_memory (uint16 Fr_memory_address, uint16 Fr_value)	
When called:	anytime, according to the related FlexRay memory write access rules	
Parameters (in):	Fr_memory_address	This parameter determines the FlexRay memory field which should be updated with passed value in the Fr_memory_address parameter The host has to put the value in bytes and needs to be even number, see FlexRay module documentation [FLEXRAY_MODULE]
	Fr_value	Value which should be stored into required register
Parameters (out):	none	
Return value:	none	
Description:	This function stores a 16-bit value to the required FlexRay memory field. The function uses 16-bit access, however the address of the required memory field has to be put in the number of bytes (as it is described in documentation [FLEXRAY_MODULE], chapter FlexRay Memory Layout). The value of the *Fr_memory_address* parameter is any offset of the corresponding FlexRay memory field with respect to the FlexRay memory base address as provided by the *CC_FlexRay_memory_base_address* parameter in the *Fr_HW_config_type* structure	

Example of usage:

```
uint16 data_field_off_value; // Data Field Offset in the message
buffer Header Field
uint16 data_mb4[2];          // The user data array


// Read the Data Field Offset of the fourth message buffer Header
Field in the FlexRay memory
// Address calculation for Data Field Offset (in bytes):
// (i * 10) + 0x6 ; where i is the message buffer number
// for the fourth message buffer, it is (4 * 10) + 0x6 = 46 bytes
data_field_off_value = Fr_low_level_access_read_memory(46);

data_mb4[0] = 0x1234;    // Store data into the user data array
data_mb4[1] = 0x5678;    // Store data into the user data array

// Write the data to the message buffer Data Field of the fourth
message buffer
// For receive message buffers, the application must lock the related
// receive message buffer and retrieve the message buffer header index
// from the Message Buffer Index Registers (FR_MBIDXRn)
Fr_low_level_access_write_memory(data_field_off_value,
data_mb4[0]);
Fr_low_level_access_write_memory((data_field_off_value + 2),
data_mb4[1]);
```

3.9 Type Definitions

3.9.1 boolean

Type:	unsigned integer	
Range:	FALSE	0
	TRUE	1
Description:	The standard type, shall be implemented on basic of an unsigned integer	

3.9.2 Fr_return_type

Type:	enumeration	
Range:	FR_NOT_SUCCESS (=0)	Function call has not been successful
	FR_SUCCESS	Function call has been successful
Description:	This type can be used as standard API return type. The *Fr_return_type* shall be used with the FR_NO_SUCCESS and FR_SUCCESS. If those return values are not sufficient, an own type is defined	

3.9.3 Fr_stop_communication_type

Type:	Enumeration	
Range:	FR_HALT_COMMUNICATION (=0)	Transition to the *POC:halt* state on completion of current communication cycle
	FR_ABORT_COMMUNICATION	Immediately transition to the *POC:halt* state
Description:	These values of this enumeration are used to determine if communication stops immediately or at the end of current FlexRay communication cycle	

3.9.4 Fr_sync_state_type

Type:	Enumeration	
Range:	FR_ASYNC (=0)	The local FlexRay module is asynchronous to the FlexRay global time
	FR_SYNC	The local FlexRay module is synchronous to the FlexRay global time

Description:	These values of this enumeration are used to provide information whether or not the local FlexRay module is synchronous to the FlexRay global time

3.9.5 Fr_POC_state_type

Type:	Enumeration	
Range:	FR_POCSTATE_CONFIG (=0)	*POC:config* state, configuration state
	FR_POCSTATE_DEFAULT_CONFIG	*POC:default config* state, state prior to config, only left with an explicit configuration request
	FR_POCSTATE_HALT	*POC:halt* state, error state, can only be left by a reinitialization
	FR_POCSTATE_NORMAL_ACTIVE	*POC:normal active* state, normal operation
	FR_POCSTATE_NORMAL_PASSIVE	*POC:normal passive* state, errors detected, no transmission of data, but attempting to return to normal operation
	FR_POCSTATE_READY	*POC:ready* state, state reached from the *POC:config* state after concluding the configuration
	FR_POCSTATE_STARTUP	*POC:startup* state, the FlexRay module transmits only startup frames
	FR_POCSTATE_WAKEUP	*POC:wakeup* state, the FlexRay module sends a wakeup pattern if it could not find anything on the bus
	FR_POCSTATE_UNKNOWN	Unknown state due to error
Description:	These values are used to hold the protocol state. This formal definition refers to [FR_PROTOCOL], chapter POC status	

3.9.6 Fr_tx_MB_status_type

Type:	Enumeration	
Range:	FR_TXMB_UPDATED (=0)	A transmit message buffer has been successfully updated with new data
	FR_TXMB_NO_ACCESS	A transmit message buffer has not been successfully locked
Description:	These values are used to determine whether a message buffer has been updated with a new data	

3.9.7 Fr_tx_status_type

Type:	Enumeration	
Range:	FR_TRANSMITTED (=0)	Data has been transmitted. In case of a double buffered message buffer, the data has been transmitted from the transmit side of a double message buffer and also the Internal Message Transfer has been performed, i.e. a data has been transmitted from a double buffered message buffer. Commit and transmit sides of the a double buffered message buffer shall be updated with new data
	FR_NOT_TRANSMITTED	Data has not been transmitted. In case of a double buffered message buffer, the data has not been transmitted from the transmit side of a double message buffer and also the Internal Message Transfer has not been performed, i.e. a data has not been transmitted from a double buffered message buffer. Commit and transmit sides of a double buffered message buffer shall not be updated with new data
	FR_INTERNAL_MESSAGE_TRANSFER_DONE	Applied only for a double buffered message buffer. The data has not been transmitted from the transmit side of a double message buffer and the Internal Message Transfer has been performed, i.e. a data has been transferred from commit to transmit side. Only commit side of a double buffered message buffer shall be updated with new data
	FR_TRANSMIT_SIDE_TRANSMITTED	Applied only for double buffered message buffer. The data has been transmitted from the transmit side of a double message buffer and the Internal Message Transfer has not been performed, i.e. a data has been transmitted from transmit side
Description:	These values are used to determine if a data has been transmitted. For single message buffers only the FR_TRANSMITTED and FR_NOT_TRANSMITTED values are used	

3.9.8 Fr_rx_MB_status_type

Type:	Enumeration	
Range:	FR_RXMB_RECEIVED (=0)	Data has been received
	FR_RXMB_NOT_RECEIVED	Data has not been received
	FR_RXMB_NULL_FRAME_RECEIVED	Received frame is a NULL frame
	FR_RXMB_NO_ACCESS	The receive message buffer has not been successfully locked
Description:	These values are used to determine if a data has been received	

3.9.9 Fr_rx_status_type

Type:	Enumeration	
Range:	FR_RECEIVED (=0)	Valid non-null frame received
	FR_NOT_RECEIVED	No valid frame received
	FR_NULL_FRAME_RECEIVED	Received frame is a NULL frame
Description:	These values are used to determine if a message buffer has received a frame	

3.9.10 Fr_FIFO_status_type

Type:	Enumeration	
Range:	FR_FIFO_RECEIVED (=0)	Data has been received
	FR_FIFO_NOT_RECEIVED	Data has not been received
Description:	These values are used to determine if a data has been received into a FIFO buffer	

3.9.11 Fr_wakeup_state_type

Type:	Enumeration	
Range:	FR_WAKEUPSTATE_UNDEFINED (=0)	See [FR_PROTOCOL]
	FR_WAKEUPSTATE_RECEIVED_HEADER	See [FR_PROTOCOL]
	FR_WAKEUPSTATE_RECEIVED_WUP	See [FR_PROTOCOL]
	FR_WAKEUPSTATE_COLLOSION_ HEADER	See [FR_PROTOCOL]
	FR_WAKEUPSTATE_COLLISION_WUP	See [FR_PROTOCOL]

	FR_WAKEUPSTATE_COLLISION_ UNKNOWN	See [FR_PROTOCOL]
	FR_WAKEUPSTATE_TRANSMITTED	See [FR_PROTOCOL]
Description:	These values determine the state of the wakeup mechanism. This formal definition refers to [FR_PROTOCOL], chapter 2.2.1.3 POC status	

3.9.12 Fr_MTS_state_type

Type:	Enumeration	
Range:	FR_MTS_RCV (=0)	A valid MTS has been received
	FR_MTS_RCV_SYNERR	A valid MTS has been received and a Syntax Error was detected
	FR_MTS_RCV_BVIO	A valid MTS has been received and a Boundary Violation has been detected
	FR_MTS_RCV_SYNERR_BVIO	A valid MTS has been received and a Syntax Error and a Boundary Violation has been detected
	FR_MTS_NOT_RCV	No valid MTS has been received
	FR_MTS_NOT_RCV_SYNERR	No valid MTS has been received and a Syntax Error was detected
	FR_MTS_NOT_RCV_BVIO	No valid MTS has been received and a Boundary Violation has been detected
	FR_MTS_NOT_RCV_SYNERR_BVIO	No valid MTS has been received and a Syntax Error and a Boundary Violation has been detected
	FR_MTS_UNKNOWN	The state of MTS is unknown due to some error
Description:	These values determine the state of the Media Access Test Symbol (MTS) reception in the symbol window of a communication cycle. These values are derived from [FR_PROTOCOL], *Table 9-1*	

3.9.13 Fr_channel_type

Type:	Enumeration	
Range:	FR_CHANNEL_A (=0)	Refers to channel A
	FR_CHANNEL_B	Refers to channel B
	FR_CHANNEL_AB	Refers to both channels
	FR_NO_CHANNELS	No channel is used
Description:	These values are used to reference a FlexRay channel	

3.9.14 Fr_FIFO_range_filter_mode_type

Type:	Enumeration	
Range:	FR_ACCEPTANCE (=0)	Receive FIFO filter operates in the acceptance mode
	FR_REJECTION	Receive FIFO filter operates in the rejection mode
Description:	These values are used for the initialization of the receive FIFO filters	

3.9.15 Fr_transmit_MB_type

Type:	Enumeration	
Range:	FR_SINGLE_TRANSMIT_BUFFER (=0)	A transmit message buffer is configured as single buffered message buffer
	FR_DOUBLE_TRANSMIT_BUFFER	A transmit message buffer is configured as double buffered message buffer
Description:	These values are used to initialize a transmit message buffer and define the message buffering type	

3.9.16 Fr_transmission_type

Type:	Enumeration	
Range:	FR_EVENT_TRANSMISSION_MODE (=0)	A transmit message buffer operates in the event transmission mode
	FR_STATE_TRANSMISSION_MODE	A transmit message buffer operates in the state transmission mode
Description:	These values are used to initialize a transmit message buffer and define the transmission mode	

3.9.17 Fr_transmission_commit_type

Type:	Enumeration	
Range:	FR_STREAMING_COMMIT_MODE (=0)	A double transmit message buffer operates in the streaming commit mode
	FR_IMMEDIATE_COMMIT_MODE	A double transmit message buffer operates in the immediate commit mode

Description:	These values are used to initialize a transmit message buffer and defines the commit mode for a double buffered transmit message buffer

3.9.18 Fr_connected_HW_type

Type:	Enumeration	
Range:	FR_MFR4300 (=0)	Freescale MFR4300 FlexRay Communication controller is used
	FR_MC9S12XFR128	Freescale MC9S12XFR128 Microcontroller with integrated FlexRay module is used
	FR_MPC5567	Freescale MPC5567 Microcontroller (Rev 0) with integrated FlexRay module is used
	FR_MFR4310	Freescale MFR4310 FlexRay Communication controller is used
	FR_MPC5561	Freescale MPC5561 Microcontroller with integrated FlexRay module is used
	FR_MPC5510_0M76F	Freescale MPC5516 Microcontroller (with MaskSet 0M76F) with integrated FlexRay module is used
	FR_MC9S12XF	One of following Freescale Microcontrollers with integrated FlexRay module is used: MC9S12XF512 MC9S12XF384 MC9S12XF256 MC9S12XF128
	FR_MPC5567_REVA	Freescale MPC5567 Microcontroller (Rev A) with integrated FlexRay module is used
	FR_MPC5510_0M22M	Freescale MPC5516 or MPC5517 Microcontroller (with MaskSet 0M22M) with integrated FlexRay module is used
	FR_MPC560xP	Freescale MPC5602P, MPC5603P or MPC5604P Microcontroller with integrated FlexRay module is used
Description:	These values are used to determine which type of the FlexRay communication controller is controlled by the FlexRay UNIFIED Driver	

3.9.19 Fr_clock_source_type

Type:	Enumeration	
Range:	FR_EXTERNAL_OSCILLATOR (=0)	External oscillator is used
	FR_INTERNAL_SYSTEM_BUS_CLOCK	Internal system bus clock is used
Description:	These values are used to determine which type of the clock is used for the FlexRay module	

3.9.20 Fr_buffer_type

Type:	Enumeration	
Range:	FR_TRANSMIT_BUFFER (=0)	Transmit message buffer
	FR_RECEIVE_BUFFER	Receive message buffer
	FR_RECEIVE_FIFO	Receive FIFO buffer
	FR_RECEIVE_SHADOW	Receive shadow message buffer
Description:	These values are used to determine which type of the message buffer is used	

3.9.21 Fr_timer_ID_type

Type:	Enumeration	
Range:	FR_TIMER_T1 (=0)	Timer T1 is configured
	FR_TIMER_T2	Timer T2 is configured
Description:	These values are used to determine which FlexRay timer is configured	

3.9.22 Fr_timer_timebase_type

Type:	Enumeration	
Range:	FR_ABSOLUTE (=0)	Timer is configured as absolute
	FR_RELATIVE	Timer is configured as relative
Description:	These values are used to determine which type of the timer is configured	

3.9.23 Fr_timer_repetition_type

Type:	Enumeration	
Range:	FR_NON_REPETITIVE (=0)	Timer is configured as non repetitive
	FR_REPETITIVE	Timer is configured as repetitive
Description:	These values are used to determine which type of the repetition mode is configured	

3.9.24 Fr_slot_status_required_type

Type:	Enumeration	
Range:	FR_SLOT_STATUS_CURRENT (=0)	The newest updated slot status information is required from a given slot
	FR_SLOT_STATUS_PREVIOUS	Not the historic slot status information is required from a given slot

Description:	These values are used to determine which slot status information is read by the ***Fr_get_slot_status_reg_value()*** function. If the FR_SLOT_STATUS_CURRENT parameter is used in the ***Fr_get_slot_status_reg_value()*** function, the function reads the newest slot status information for a required slot. It means that the FlexRay UNIFIED Driver queries the current communication cycle number and also the current slot number, it determines if the current cycle is odd or even. In case where the current slot number is greater than the required slot number, the slot status information is read for the current cycle otherwise for previous cycle from related odd or even part of the slot status register. If the FR_SLOT_STATUS_PREVIOUS parameter is used in the ***Fr_get_slot_status_reg_value()*** function, this function queries the previous (not the historic) slot status information for the required slot

3.9.25 Fr_slot_status_counter_channel_type

Type:	Enumeration	
Range:	FR_SLOT_STATUS_CHANNEL_A (=0)	Slot Status Counter is incremented by 1 if its increment condition is fulfilled on channel A
	FR_SLOT_STATUS_CHANNEL_B	Slot Status Counter is incremented by 1 if its increment condition is fulfilled on channel B
	FR_SLOT_STATUS_CHANNEL_AB_BY_1	Slot Status Counter is incremented by 1 if its increment condition is fulfilled on at least one channel
	FR_SLOT_STATUS_CHANNEL_AB_BY_2	Slot Status Counter is incremented either - by 2 if its increment condition is fulfilled on both channels or - by 1 if its increment condition is fulfilled on only one channel
Description:	These values control the channel related incrementing of the slot status counter	

3.9.26 Fr_slot_status_counter_ID_type

Type:	Enumeration	
Range:	FR_SLOT_STATUS_COUNTER_0 (=0)	Slot Status Counter 0 is configured
	FR_SLOT_STATUS_COUNTER_1	Slot Status Counter 1 is configured
	FR_SLOT_STATUS_COUNTER_2	Slot Status Counter 2 is configured
	FR_SLOT_STATUS_COUNTER_3	Slot Status Counter 3 is configured
Description:	These values are used to determine which slot status counter is configured	

3.9.27 Fr_CHI_error_type

Type:	uint16
Description:	These values are used to return the CHI related error flags. See [FLEXRAY_MODULE] for more information, chapter CHI Error Flag Register

3.9.28 Fr_index_selector_type

Type:	uint8
Description:	The numeric values are used to determine which message buffers defined in a *Fr_buffer_info_type* structure will be used for the FlexRay module configuration

3.9.29 Fr_single_channel_mode_type

Type:	Enumeration	
Range:	FR_DUAL_CHANNEL_MODE (=0)	Dual channel mode is configured
	FR_SINGLE_CHANNEL_MODE	Single channel mode is configured
Description:	These values control the device channel mode. The single channel device mode supports devices that have only one FlexRay port available	

How to Reach Us:

USA/Europe/Locations not listed:
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale Semiconductor products, please visit
http://www.freescale.com

UG
Rev. 1.3.8, 01/2010

freescale™
semiconductor