STAR COOPERATION®

Your Partners in Excellence

FlexCard

FCBASE API Documentation

Contact Information

STAR ELECTRONICS GmbH & Co. KG
A Company of the STAR COOPERATION Group
Jahnstraße 86
73037 Göppingen
Phone: +49 (0)7031 6288-5656
Phone: +49 (0)7031 6288-5330 (Support)

Sales: sales-ee@star-cooperation.com
Support: support-ee@star-cooperation.com
www.star-cooperation.com/ee-solutions

Company Data

STAR ELECTRONICS GmbH & Co. KG, registered offices: Göppingen, register court Ulm, HRA 721096
Partner liable to unlimited extent: STAR ELECTRONICS Verwaltungs-GmbH, registered offices: Göppingen, register court Ulm, HRB 722565
Represented by the executive board: Rolf Wittig, Henning Lange

Copyright Notice

Trademarks

Disclaimer

The information contained in this document does not affect or change General Terms and Conditions of *STAR ELECTRONICS GmbH & Co. KG*. *STAR ELECTRONICS GmbH & Co. KG* does not guarantee the completeness and accuracy of the content of this document and assumes no responsibility for any errors which may appear in this document or due to this document. The content of this document or the associated products are subject to change without notice at any time.

Based on currently state of arts and science it is impossible to develop software that is bug-free in all applications. Therefore, the product is only allowed to be used in the sense of the product use case described herein.

*STAR ELECTRONICS GmbH & Co. KG m*akes no warranty express or implied, as to this document or the information content, materials or products for any particular purpose, nor does *STAR ELECTRONICS GmbH & Co. KG* assume any liability arising out of the application or use of this product, and disclaims all liabilities, including without limitation resulting damages, as permissible by applicable law.

All operating parameters which are provided in this document can vary in different applications or over time. The herein described product solely is allowed to be used as described in chapter “Intended use”.

Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written consent of *STAR ELECTRONICS GmbH & Co. KG*.

STAR ELECTRONICS GmbH & Co. KG may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly stated in a written license agreement from *STAR ELECTRONICS GmbH & Co. KG* the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Any semiconductor devices have an inherent chance of failure. You have to protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
The safety and handling instructions in this document have to be followed strictly.

Revision History

Version	Date	Description
D1V0-F	06-Mar-2006	Initial release.
D1V1-F	02-Nov-2006	API functions added and updated. Multicard usage.
D1V2-F	02-May-2007	New API functions added and updated. PMC and XENOMAI usage.
D1V3-F	10-May-2007	Corrected description and changed Xenomai usage.
D1V4-F	21-Jun-2007	VxWorks API functions added.
D1V5-F	30-Aug-2007	PMC, VxWorks and Linux functions changed and added.
D1V6-F	02-Dec-2007	FlexCard Cyclone II (SE) support self startup/synchronization.
D1V7-F	28-Jan-2008	Support of CC Timer, API functions added and updated.
D1V8-F	25-Feb-2008	VxWorks chapter updated.
D1V9-F	11-Jul-2008	FlexCard Cyclone II (SE) support CAN. New API functions added.
D1V10-F	29-Oct-2008	FlexCard PMC/PCI support CAN. New FlexRay API functions added.
D1V11-F	27-Feb-2009	FlexCard PMC-II support and new API functions added.
D1V12-F	16-Apr-2009	Corrected description. Linux driver supports FlexCard PMC-II.
D1V13-F	10-Jul-2009	Xenomai driver supports FlexCard PMC-II. Windows driver is compatible to DMA Firmwares. Added extended message buffer configurations. Functions for FlexCard PMC-II firmware added.
D1V14-F	11-Dec-2009	Windows driver supports FlexCard USB-M. Redesigned API documentation. Added API functions for time stamp configuration.
D1V15-F	28-May-2010	Added CAN transmit FIFO feature.

Version	Date	Description
D1V16-F	06-Oct-2010	Updated description.
D1V17-F	19-May-2011	Updated VxWorks description.
D1V18-F	06-Jul-2012	Windows 64 bit release.
D1V19-F	12-Sep-2013	Linux S6V5 release.
D2V0-F	16-Nov-2015	Layout adapted to STAR ELECTRONICS GmbH & Co. KG. Replaced product names.
D2V1b-F	22-Jun-2016	Added CAN-FD support. Updated Reference to Bosch ERay User Manual.
D2V2-F	25-Jan-2017	Update for Xenomai S6V5 release.
D2V3-F	10-Dec-2018	Linux 64 bit S6V6 release.
D2V4-4	30-Jul-2019	Preliminary: Support for FlexCard PXIe3 and FlexCard PCIe3 added.
D2V4-9	04-Nov-2019	Preliminary: Updated description for FlexCard PXIe3 and FlexCard PCIe3.
D2V4-14	07-Feb-2020	Preliminary: Extended description for Ethernet.
D2V4-19	17-Mar-2020	Preliminary: Added HwCom Devices and command “SetGlobalConfig”
D2V4-F	10-Dec-2020	Release for fcBase API S6V7 Added API support for FlexDevice family Removed driver support for Windows 7 x86 and x64. Removed driver support for Windows 10 x86 (the library still supports x86 and x64 applications)
D2V5-F	05-Oct-2021	Release for fcBase API S6V8

Related Hardware / Software Versions

Product	Reference No.	Version (Major and Minor)	Remarks
FlexCard Cyclone II Firmware	3-0009-0C04	S6V4	Current version
FlexCard Cyclone II SE Firmware	3-0009-0C05	S6V4	Current version
FlexCard PMC Firmware	3-0033-0B01	S6V4	Current version
FlexCard PMC-II Firmware Windows	3-0055-0C01	S6V6 (6.5.0.33)	Current version
FlexCard PMC-II Firmware Linux	3-0055-0C01	S6V5	Current version
FlexCard PMC-II Firmware Xenomai	3-0055-0C01	S6V5	Current version
FlexCard USB-M Firmware	3-0058-0B01	S6V4	Current version
FlexCard Cyclone II Hardware	3-0009-0A04	H1V1	Initial version
FlexCard Cyclone II SE Hardware	3-0009-0A05	H1V1	Initial version
FlexCard PMC Hardware	3-0033-0A01	H1V0	Initial version
FlexCard PMC-II Hardware	3-0055-0A01	H1V1	Initial version
FlexCard USB-M Hardware	3-0058-0A01	H1V2	Initial version
FlexCard PXIe3 Hardware	3-0094-0B01	50	Support for this hardware is preliminary
FlexCard PCIe3 Hardware	3-0095-0B01	20	Support for this hardware is preliminary

Product	Reference No.	Version (Major and Minor)	Remarks
FlexDevice-L	3-0087-0A01	03	
FlexDevice-L	3-0087-0A02	01	
FlexDevice-L	3-0087-0A03	01	
FlexDevice-L²	3-0087-0S02	01	
FlexDevice-L²	3-0087-0S03	01	
FlexDevice-S	3-0086-0A01	20	Supports only battery supply
FlexDevice-S	3-0086-0A01	30	

Related Software Versions (Windows)

Component	Reference No.	Version (Major and Minor)	Remarks
fcBase API	3-0009-0K03	S6V8	Current version supports : FlexCard Cyclone II (SE), FlexCard PMC (II), FlexCard USB-M, FlexCard PXIe3, FlexCard PCIe3, FlexDevice-L, FlexDevice-L², FlexDevice-S
Driver fce05	3-0009-0E05	S6V8	Current version supports only FlexCard Cyclone II (SE), FlexCard PMC (II), FlexCard USB-M
Driver fcx0d01	3-0094-0D01	S1V0	Supports FlexCard PXIe3 and FlexCard PCIe3
PC Hardware Interface	3-9999-0C01	2.4	Supports FlexDevices, Flexcard PXIe3, FlexCard PCIe3.

Related Software Versions (Linux)

Component	Reference No.	Version (Major and Minor)	Remarks
libflexcard API (libfcBase.so)	3-0009-0U01	S6V6	FlexCard Linux driver. This version supports only FlexCard PMC-II
Kernel module (flexcard)	3-0009-0U01	S6V6	FlexCard Linux driver. This version supports only FlexCard PMC-II.
libflexcard API (libfcBase_ng.so)	3-0095-0D01	S6V8	New FlexCard Linux driver. Current version supports : FlexCard PCIe3
Kernel module (flexcard_ng)	3-0095-0D01	S6V8	New FlexCard Linux driver. Current version supports FlexCard PCIe3

Related Software Versions (Xenomai)

Component	Reference No.	Version (Major and Minor)	Remarks
libfcBase API	3-0009-0V01	S6V5	Current version supports: FlexCard PMC (II)
Kernel module	3-0009-0V01	S6V5	Current version

Related Software Versions (VxWorks)

Component	Reference No.	Version (Major and Minor)	Remarks
FlexCard PMC Driver	3-0033-0D01	S2V1	Current version supports: FlexCard PMC

Related Documents

Document	Ordering number
FlexCard Cyclone II SE Instructions for Use	3-0009-0T01-D01
FlexCard PMC Instructions for Use	3-0033-0P01-D01
FlexCard PMC-II Instructions for Use	3-0055-0P01-D05
FlexCard USB-M Instructions for Use	3-0058-0P01-D03
FlexCard PXIe3/PCIe3 Instructions for Use	3-0094-0A01-D10
FlexCard Cyclone II SE Getting started	3-0009-0S01-D02
FlexCard PMC-II Getting started	3-0055-0P01-D07
FlexCard USB-M Getting started	3-0058-0P01-D04
Release Notes FlexCard Windows	3-0009-0S01-D13
Release Notes FlexCard Linux	3-0009-0U01-D01
Release Notes FlexCard Xenomai	3-0009-0V01-D01
FlexDevice-L/L² Instructions for Use	3-0087-0A01-D19
FlexDevice-S Instructions for Use	3-0086-0A01-D09

Contents

1 General

1.1 Intended User Group

This product may only be used by expert technicians and/or engineers who are qualified and familiar with electronic components and systems.

Each person involved with setup or operation of the product must

- be a qualified technician or engineer
- strictly adhere to this manual
- receive a briefing by an authorized person

❗	**NOTICE** If you are unsure of how to use the product as intended or have any questions about the use of the product, please discontinue use of the product immediately and contact the STAR Electronics Support.

⚠	**⚠ WARNING** The product may only be used by expert technicians and/or engineers who are qualified and familiar with electronic components and systems! The use of the product by non-professionals is not permitted and strictly forbidden!

The FlexCard fcBase API is a testing equipment. It was developed to test the communication behavior of automotive bus systems and Ethernet together with Electronics Control Units and sensors in a fully controlled testing and/or laboratory environment.
For this intended use, the FlexCard fcBase API offers the following options:

- Transmit and receive data (e.g. Use Case “Remaining Bus Simulation”)
- Exchange of data traffic between two or more bus systems (e.g. Use Case “Gateway”)
- Manipulation of data traffic (e.g. Use Case “Manipulation of signal values based on user configuration”)
- Recording of data traffic (e.g. Use Case “Logging”)

Any deviation from the intended use and/or installation in a testing vehicle is only permitted with specific **prior written approval** of STAR ELECTRONICS GmbH & Co. KG.

⚠	**⚠ WARNING** The FlexCard fcBase API may be used to communicate with networked electronic systems. E.g. FlexRay, CAN or Ethernet. Any use of the product outside a fully controlled testing and/or laboratory environment may result in death or serious injury due to unpredictable behavior of a vehicle and/or potentially missing, deactivated, or malfunctioning safety devices on a vehicle! The user is responsible to ensure the safety of the entire system. This includes amongst other things a safety shutdown.

⚠	⚠ **WARNING**
	Any use of the device to control an actuator outside a fully controlled testing and/or laboratory environment may result in death or serious injury due to unpredictable behavior of a vehicle and/or potentially missing, deactivated or malfunctioning safety devices on a vehicle!

❗	**NOTICE**
	The device is not a calibrated measurement device. STAR ELECTRONICS GmbH & Co. KG accepts no liability whatsoever for the correctness of any measurement results.

⚠	⚠ **WARNING**
	The FlexCard fcBase API is **NOT** designed, intended, or authorized any may **NOT** be used for or in connection with the following purposes and/or devices: - use as part of medical systems - life support applications - aviation, space, nuclear, or military applications - use in areas where combustible or explosive gas mixtures are likely to occur - any other purposes / devices deviating from the intended use of the product specified by STAR ELECTRONICS GmbH & Co. KG.

⚠	⚠ **WARNING**
	The product may only be used by expert technicians and/or engineers who are qualified and familiar with electronic components and systems! The use of the product by non-professionals is not permitted and strictly forbidden!

1.3 Used Pictograms

The meaning of used pictograms is shortly described below.

Follow the specific instructions in the document where these pictograms are placed:

⚡	⚠ **WARNING**
	Used to indicate a potentially hazardous situation which, if not avoided, could result in death or serious injury.

⚠	⚠ **CAUTION**
	Used to indicate a potentially hazardous situation which, if not avoided, could result in minor or moderate injury.

❗	**NOTICE**
	Used to indicate a situation which may result in an operating failure. Damage of the product may occur, but there is no hazard of injury if not avoided.

ℹ	**Information**
	Used to indicate information provided only for purposes of clarification, illustration, and general information.

1.4 Safety and Handling Instructions

Please read the instructions for use carefully. To protect the device or the application against damage, or to avoid personal injury the FlexCard fcBase API has to be handled as described herein.

Changes or modifications of the FlexCard fcBase API are not allowed for safety and warranty reasons!

STAR ELECTRONICS GmbH & Co. KG is not liable for any damages arising from non-observance of the product information.

Follow the
a) specific safety and handling instructions placed at dedicated document positions
b) general safety and handling instructions below:

1.5 Meaning of Text Styles

In this document *filenames*, `source code`, FlexRay Protocol Variable, **`functions`** and **`structs`** are marked with a different text format.

2 Product Description

2.1 FlexCard fcBase API at a glance

This document describes the application programming interface (API) *fcBase API* for the FlexCard and FlexDevices. For simplicity, the devices of the FlexDevice family are also called "FlexCard" if no special handling is needed. The API defines the basic functions and structures which are used to communicate with the FlexCard hardware, the FlexRay and CAN bus. With these functions the developer can integrate the FlexCard in a FlexRay cluster and CAN network.

The following figure illustrates a typical approach of accessing the FlexRay and CAN bus via the FlexCard:

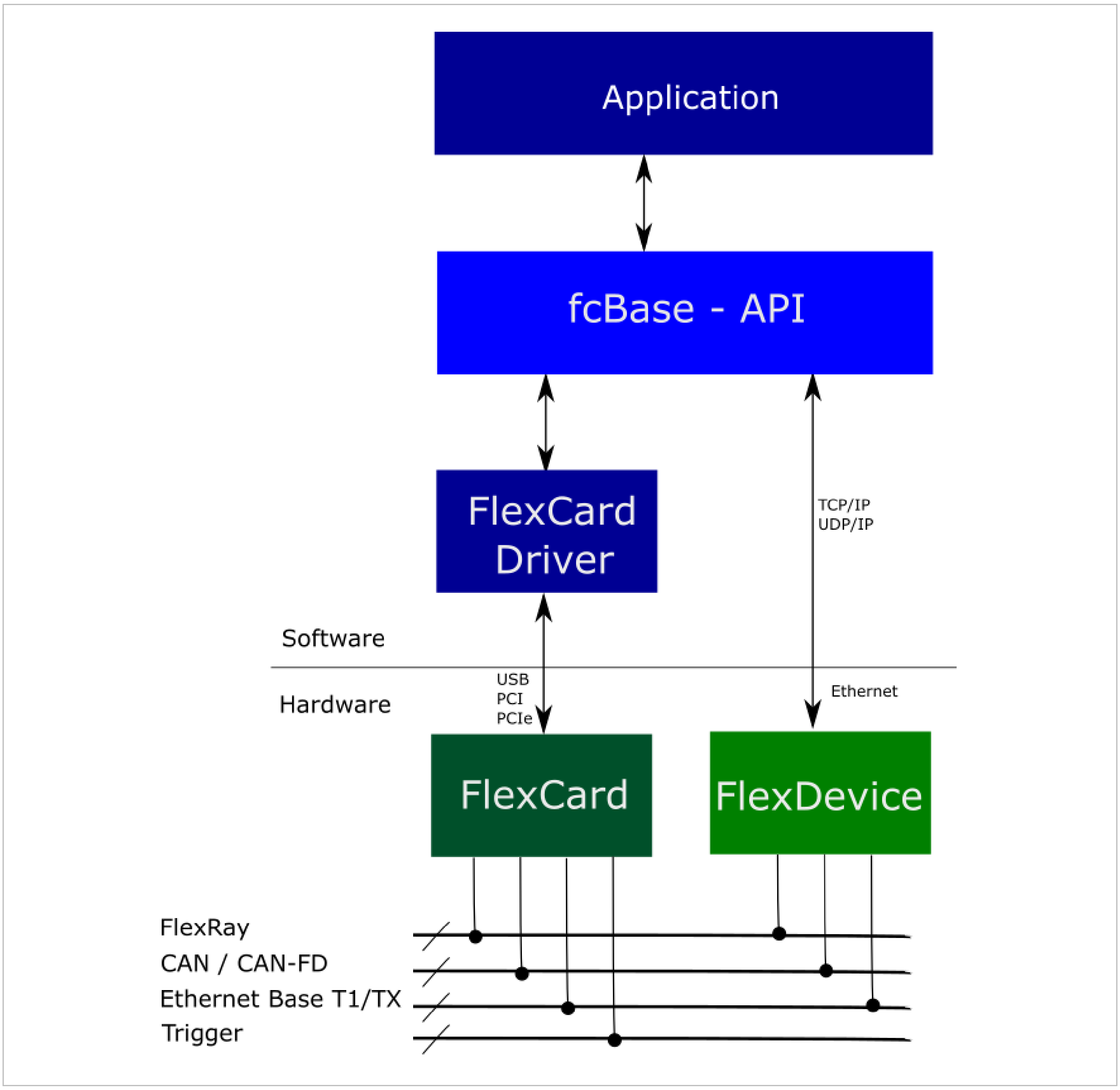


Figure 1: Overview of a typical FlexCard system with hardware and software

The *fcBase API* consists of the following groups of functions:

- **Error handling** → Functions to get detailed error information
- **Configuration** → Functions and structures to configure the available Communication Controller and the FlexCard hardware. For example, bus parameters, message buffers and the triggers may be configured.

- **Initialization** → Functions to enumerate the FlexCards in the system, to establish a connection to a FlexCard and to start and stop the monitoring of the FlexRay and CAN bus.
- **Transmit / Receive** → Functions to receive FlexRay and CAN frames or informational frames (e.g. Trigger information), or to transmit a FlexRay and CAN frame on a specific slot or id.
- **Event handling** → Functions to obtain event handles which are signalled if a specific time elapses, a wake-up pattern is detected or at the start of a new FlexRay cycle.

Additional there is a tracing module, which can only be accessed by the tracing control application. For further information refer to chapter 18 in this document.

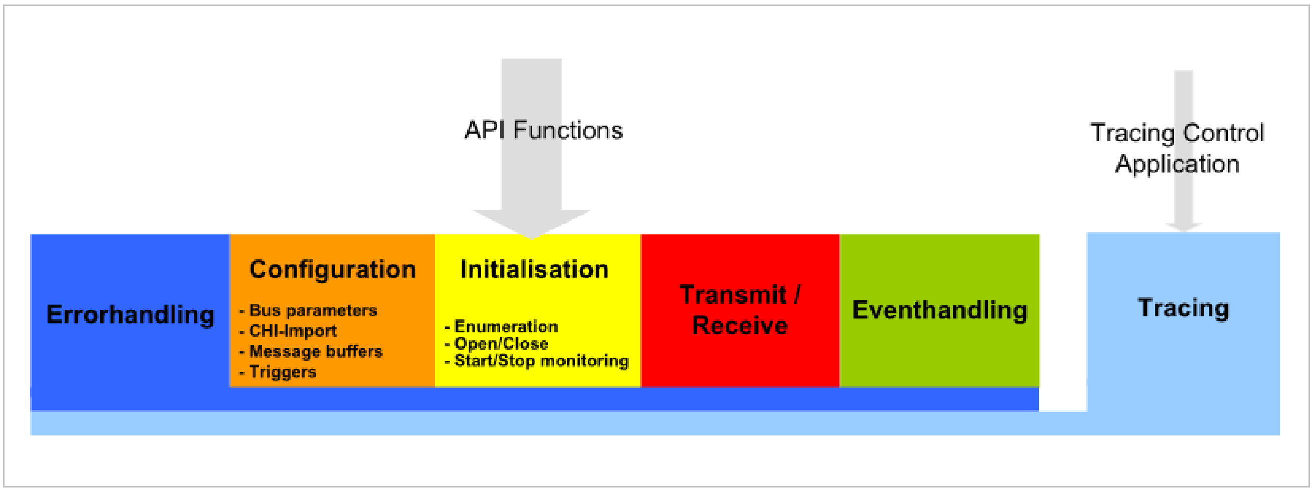


Figure 2: *fcBase API* groups

The FlexCard API uses a well-defined naming convention. Each function, structure or enumeration is prefixed with fc or fcb. The prefix fcb (fcBase) stands for a function, a structure or an enumeration which is only available in the fcBase API. Functions, structures, or enumerations which are prefixed with fc are not limited to the fcBase API and could also be available in other FlexCard APIs.

Each function of this library (except some error handling functions) returns an error code. If the return value is equal to zero, no error occurred. A number greater than zero indicates an error. To get more information about it, use the error handling functions described in chapter 4.1.

Some functions will allocate memory for you. In such a case the **`fcFreeMemory`** function needs to be called to release this memory.

2.2 General Function Availability

There are some functional hardware and software differences between FlexCard products which demand additional functions or enumerations. The differences are listed in the table below:

Functions	FlexCard Cyclone II (SE)	FlexCard USB-M	FlexCard PMC	FlexCard PMC-II	FlexDevice S / L / L^2	FlexCard PXIe3 / PCIe3
General FlexCard API Description		Available				
FlexRay API	Available (depends on firmware configuration)				Limited. See additional notes in section 2.3.2.3 for details	Depends on firmware configuration. See additional notes in section 2.3.2 for details.
CAN API	Available (depends on firmware configuration)					
CAN-FD API	Not available			Available		
Trigger API	2 trigger lines, unidirectional		2 trigger lines, configurable		Not available	
Termination API	Not available	Not available	Available	Available	Not available	
Firmware API	Not available	Available	Not available	Available	Not available	Not available

The functions are in different header files, but all functions are provided by fcBase.dll (or libfcBase API under Linux/Xenomai). If the function you called is not supported on the selected hardware, the FlexCard API will return an error.

Following a list of the supported number of Communication Controllers per FlexCard device:

Functions	FlexCard Cyclone II (SE)	FlexCard USB-M	FlexCard PMC	FlexCard PMC-II	FlexCard PXIe3 / PCIe3 and FlexDevice
CC count	1 CC for FlexRay and 2 CCs for CAN-HS	1 CC for FlexRay and 2 CCs for CAN-HS, 1 CC for CAN-LS	2 CCs for FlexRay or 1 CC for FlexRay and 2 CCs for CAN-HS	Variable interface configurations for FlexRay and CAN max. 4 FlexRay CCs or max. 8 CAN-HS CCs with FlexTiny II possible.	Variable interface configuration possible (FlexRay, CAN, CAN-FD, Ethernet)

The features are only available with the correct firmware, mounted FlexTiny module, driver and license.

2.3 Exceptions for Function Availability

This chapter lists the differences in the functional range compared to the general function availability (see chapter 2.2). Exceptions may also be found in the Operation System chapters.

2.3.1 FlexCard USB-M

The FlexCard USB-M driver doesn’t support the following functions:

```
➢ fcbSetEventHandleV2
➢ fcbSetReceiveBufferLevelNotification
➢ fcbSetTimer
➢ fcbFRSetTransceiverState
➢ fcbFRGetTransceiverState
➢ fcbFRSetCcTimerConfig
```

- fcbFRGetCcTimerConfig
- fcbFRCalculateMacrotickOffset

2.3.2 FlexCard PXIe3 and FlexCard PCIe3

The FlexCard PXIe3 and FlexCard PCIe3 supports two different operating modes which are dependent on the used FPGA firmware.

2.3.2.1 FlexDevice Mode

The FlexCard runs a restbus simulation / gateway application on the embedded ARM Cortex processor. The embedded application has full access to the communication controllers so only a limited subset of API features will work. The monitoring part is done via extra bus decoder modules which are completely independent from the existing communication controllers.

Use FlexConfig RBS to configure the restbus simulation / gateway. Refer to the document FlexConfig_RBS_UserManual (3-0016-0Q01-D04) [8].

ℹ	**Information**
	The settings of the busses (for example the baudrate) in the FlexDevice Mode are done via FlexConfig RBS, **not** via fcBase API.

The frames received via the fcBase API in this mode have the direction set accordingly. If the embedded application transmits the packet, the fcBase API packet has direction tx. Otherwise the packet has the direction set to rx.

The following functions can be used in the FlexDevice Mode:

- fcGetErrorType
- fcGetErrorCode
- fcGetErrorText
- fcFreeMemory
- fcbGetEnumFlexCardsV3
- fcbCheckVersion
- fcbOpen
- fcbClose
- fcbGetInfoFlexCard
- fcbGetNumberCcs
- fcbSetContinueOnPacketOverflow
- fcbGetCurrentTimeStamp
- fcbResetTimestamp
- fcbConfigureFlexCardTimeStamp
- fcbGetCurrentHighResTimeStamp
- fcbReceive
- fcbFRMonitoringStart
- fcbFRMonitoringStop
- fcbCANMonitoringStart
- fcbCANMonitoringStop
- fcbEthMonitoringStart
- fcbEthMonitoringStop

2.3.2.2 FlexCard Mode

In the FlexCard Mode only the host computer controls the communication controller. It is not required to run a FlexConfig RBS project on the embedded processor. Almost all fcBase API functions are available.

The FlexCard mode **does not support** the following functions:

- fcbSetReceiveBufferLevelNotification
- fcbNotificationPacket
- fcbSetUserDefinedCardId
- fcbGetUserDefinedCardId
- fcbFWGetImageInfo
- fcbFWSelectImage
- fcbSetBusTermination
- fcbGetBusTermination

- **fcbNotificationPacket**
- **fcbTrigger**
- **fcbFRSetSoftwareAcceptanceFilter**
- **fcbFRSetHardwareTransmitFilter**
- **fcbFRSetOffsetSynchronization**
- **fcbCANSetCcConfiguration**
- **fcbCANSetMessageBuffer**
- **fcbCANGetMessageBuffer**
- **fcbCANSetTxFifoConfiguration**
- **fcbCANGetTxFifoConfiguration**
- **fcbCANTxFifoReset**
- **fcbCANSetFilterConfiguration**
- **fcbCANTransmit**
- **fcbCANTxFifoTransmit**
- **fcbEthMonitoringStart**
- **fcbEthMonitoringStop**
- **fcbEthSetCcConfiguration**
- **fcbEthSetFilterConfiguration**
- **fcbEthTransmit**
- **fcbUARTOverCANMonitoringStart**
- **fcbUARTOverCANMonitoringStop**
- **fcbUARTOverCANSetCcConfiguration**
- **fcbUARTOverCANGetCcConfiguration**
- **fcbUARTOverCANTransmit**

2.3.2.3 Supported packet types

The following table shows the supported packets depending on the operation mode of the device.

Packet type	FlexDevice Mode	FlexCard Mode
fcFlexRayFrame	**Yes**	**Yes**
fcInfoPacket	No	**Yes**
fcErrorPacket	No	**Yes**
fcStatusPacket	No	**Yes**
fcTriggerInfoPacket	No	No
fcTxAcknowledgePacket	**Yes**	**Yes**
fcNMVectorPacket	No	No
fcNotificationPacket	No	No
fcTriggerExInfoPacket	No	**Yes**
fcCANPacket	No	No
fcCANErrorPacket	No	No
fcCANFDPacket	**Yes**	**Yes**
fcCANFDErrorPacket	**Yes**	**Yes**
fcEthernetPacket	**Yes** (CRC is not checked)	No
fcEthernetErrorPacket	**Yes**	No
fcUARTOverCANPacket	No	No
fcUARTOverCANErrorPacket	No	No
fcUARTOverCANTxAckPacket	No	No

2.3.3 FlexDevices

Internally, the fcBase API uses for FlexDevices the PC Hardware Interface (HwCom). The HwCom documentation is available from STAR ELECTRONICS GmbH & Co. KG. The physical connection to the device depends on the hardware type. E.g., the FlexDevice L^{2} uses the medium Ethernet to connect the host pc to the device. It is possible that other device types use other mediums.

The following functions can be used with FlexDevice:

- **`fcGetErrorType`**
- **`fcGetErrorCode`**
- **`fcGetErrorText`**
- **`fcFreeMemory`**
- **`fcbGetEnumFlexCardsV3`**
- **`fcbCheckVersion`**
- **`fcbOpen`**
- **`fcbClose`**
- **`fcbGetInfoFlexCard`**
- **`fcbGetNumberCcs`**
- **`fcbReceive`**
- **`fcbCANSetCcConfiguration`**
- **`fcbCANMonitoringStart`**
- **`fcbCANMonitoringStop`**
- **`fcbCANTxFifoTransmit`**
- **`fcbCANSetFilterConfiguration`**
- **`fcbCANFDSetCcConfiguration`**
- **`fcbFRSetCcConfigurationChi`**
- **`fcbFRSetCcConfiguration`**
- **`fcbFRGetCcConfiguration`**
- **`fcbFRMonitoringStart`**
- **`fcbFRMonitoringStop`**
- **`fcbFRTransmit`**
- **`fcbEthSetCcConfiguration`**
- **`fcbEthMonitoringStart`**
- **`fcbEthMonitoringStop`**
- **`fcbEthTransmit`**
- **`fcbSetGlobalConfig`**

2.3.3.1 Device Discovery

For the FlexDevice family, the device discovery uses currently ICMP ping with raw sockets which requires that the process executing FlexCard API needs administrator privileges. Otherwise no FlexDevices will be found.

2.3.3.2 Required TCP/UDP Ports

Using the FlexDevices with the FlexConfig SDK on a network protected by a Windows Firewall, the firewall must be configured to permit FlexCard API to access the FlexDevice network resources.

The following ports must be open on the computer running the FlexCard API:

Ports	Incoming/Outgoing	Protocol	Description
1500	Outgoing	TCP	Used for HW-Com normal protocol operation
15300	Incoming	UDP	Used for HW-Com Streaming. In case the port is already used, the next free port after 15300 will be used
15300	Outgoing	UDP	Used for HW-Com frame transmissions

2.3.4 Exclusive access limitiation

For FlexDevice it cannot be guaranteed that no other application will get access to the API functionality at the same time as the lock is based on the IP address of the PC.

If multiple fcBase API applications are running on the PC at the same time, the user must ensure that the device is only accessed by the API from one application. Otherwise, an undefined behavior can occur.

2.4 API Changes From Previous Versions

2.4.1 From S1V0-F to S2V0-F

Change	Reason	Page	Remark
Definition of type fcQuad corrected	Portability	48	Downwardly compatible. Works with applications which are designed for S1V0-F.
Enumeration fcTransceiverState added	New feature	113	
Function fcbSetTransceiverState added	New feature	225	
Function fcbGetTransceiverState added	New feature	226	
Structure fcMsgBufCfgTx modified. New configuration options TxAcknowledgeShowNullFrames and TxAcknowledgeShowPayload added. TxAcknowledge packets work in all transmission modes.	Feature extended	123	Downwardly compatible. Works with applications which are designed for S1V0-F, if the reserved member of this structure was set to zero.
New member fcNoticiationTypeWakeup for enumeration fcNotifyType added for getting notification if one of the transceivers has detected a wakeup event.	Feature extended	76	Downwardly compatible. Works with applications which are designed for S1V0-F.
Function fcbNotificationPacket added	New feature	78	
Structure fcInfoPacket modified. Rate and offset correction information added.	Feature extended	80	Downwardly compatible. Works with applications which are designed for S1V0-F.
Structure fcFlexRayFrame modified. Timestamp information added.	Feature extended.	80	Downwardly compatible. Works with applications which are designed for S1V0-F.
Structure fcTxAcknowledgePacket modified. Additional information about the transmitted frame added.	Feature extended.	82	Downwardly compatible. Works with applications which are designed for S1V0-F.
Structure fcNotificationPacket added	New feature	88	
Structure fcPacket modified. fcNotificationPacket information added	Feature extended.	91	Downwardly compatible. Works with applications which are designed for S1V0-F.
Enumeration fcErrorPacketFlag extended.	Feature extended.	97	Downwardly compatible. Works with applications which are designed for S1V0-F.

2.4.2 From S2V0-F to S2V2-F

Change	Reason	Page	Remark
PMC functions added: fcbSetCCIndex, fcbGetCCIndex, fcbSetTermination, fcbGetTermination PMC Enumerations added: fcBusChannel, fcBusType	New features	189, 240	
Added new trigger functionality for FlexCard Cyclone II and FlexCard Cyclone SE. Triggers can be OR-ed now.	Feature extended	72	
Added Xenomai support function for event handling	New feature	201	

2.4.3 From S2V2-F to S3V0-F

Change	Reason	Page	Remark
Added Self synchronization for FlexCard Cyclone II (SE)	New features	163	Firmware-Version S3V0-F is needed

2.4.4 From S3V0-F to S4V0-F

Change	Reason	Page	Remark
Added CAN API for FlexCard Cyclone II (SE)	New features	143	Firmware-Version S4V0-F is needed
Added function fcbResetTimestamp.	New feature	73	
Added function fcbGetNumberCcs.	New feature	72	
Added function fcbSetContinueOnPacketOverflow.	New feature	73	
Added function fcbCalculateMacrotickOffset.	New feature	235	
Added function fcbGetCcTimerConfig.	New feature	234	
Added function fcbSetCcTimerConfig.	New feature	234	
Added function fcbCheckVersion	New feature	63	
New packets CAN packet and CAN error packet.	New feature	89, 91	
Extended enumeration fcPacketType	Feature extended	96	Downwardly compatible.
Extended enumeration fcCC	Feature extended	49	Downwardly compatible.
Extended enumeration fcTriggerConditionEx	Feature extended	183	Downwardly compatible.
Structure fcTriggerExInfoPacket modified. Reserved1 added	Feature extended	88	Downwardly compatible.
Structure fcCcTimerCfg added.	New feature	125	
Enumeration fcCyclePos added.	New feature	115	
Enumeration fcNotificationType modified. fcNotificationTypeCcTimer added.	Feature extended	76	Downwardly compatible.
Enumeration fcMemoryType modified. fcMemoryTypeInfoV2 added.	Feature extended	47	Downwardly compatible.
Structure fcInfoV2 added.	New feature	220	
Added function fcbGetEnumFlexCardsV2.	New feature	221	
Added function fcbReinitializeCcMessageBuffer	New feature	71	
Added function fcbReinitializeCcMessageBufferSelfSynchronization	New feature	181	
Added function fcbGetCurrentTimeStamp	New feature	73	

2.4.5 From S4V0-F to S4V2-F

Change	Reason	Page	Remark
CAN API is supported by FlexCard Cyclone II (SE) and FlexCard PMC/PCI.	New features	143	Firmware-Version S4V2-F is needed
Added additional Linux API.	New feature	196	
Xenomai: Added thread-safe function for event handling.	New feature	199	
Added new thread-safe FlexRay API for all supported devices.	New features	103	
Extended enumeration fcNotificationType.	Feature extended	76	Downwardly compatible.
Added thread-safe function for event handling.	New feature	76	
Structure fcFlexRayFrame modified. AsyncMode added.	Feature extended	80	Downwardly compatible.

2.4.6 From S4V2-F to S5V1-F

Change	Reason	Page	Remark
Structure fcMemoryType modified. fcMemoryTypeInfoHwSW added.	Feature extended	47	Downwardly compatible.
Enumeration fcFlexCardDeviceId modified. fcFlexCardPMCII added.	Feature extended	50	Downwardly compatible.
Structure fcVersionCC modified. IncorrectPhysicalLayer added.	Feature extended	57	Downwardly compatible.
Structure fcInfoHw added.	New feature	59	
Structure fcInfoSw added.	New feature	60	
Structure fcInfoHwSw added.	New feature	61	
Added function fcbGetEnumFlexCardsV3	New feature	62	
Added function fcbGetInfoFlexCard	New feature	65	
Added function fcbSetUserDefinedCardId	New feature	66	Firmware-Version S5V1-F is needed
Added function fcbGetUserDefinedCardId	New feature	67	Firmware-Version S5V1-F is needed
Added function fcbFRSetHardwareAcceptanceFilter	New feature	135	Firmware-Version S5V1-F is needed
Structure fcFlexRayFrame modified. FrameCRC added.	Feature extended	80	Downwardly compatible.
Structure fcTxAcknowledgePacket modified. ValidFrame, SyntaxError, ContentError added.	Feature extended	82	Downwardly compatible.
Structure fcCANMonitoringMode modified. fcCANMonitoringSilent, fcCANMonitoringActive, fcCANMonitoringPassive added.	Feature extended	145	Downwardly compatible.
Structure fcCANBufCfgTx modified. newData added.	Feature extended	150	Downwardly compatible.
Structure fcCANBufCfgRemoteTx modified. newData added.	Feature extended	152	Downwardly compatible.
Added FlexCard PMC-II description	New features	17	
Enumeration fcBusChannel modified. fcBusChannel5 to fcBusChannel8 added.	Feature extended	189	Downwardly compatible.

2.4.7 From S5V1-F to S6V1-F

Change	Reason	Page	Remark
Added typedef fcBool.	New feature	48	
Structure fcVersionCC modified. Ansi-C conformity.	Feature extended	57	Downwardly compatible.
Structure fcInfo modified. Ansi-C conformity.	Feature extended	220	Downwardly compatible.
Structure fcInfoV2 modified. Ansi-C conformity.	Feature extended	220	Downwardly compatible.
Structure fcInfoHwSw modified. Ansi-C conformity.	Feature extended	61	Downwardly compatible.
Structure fcPacket modified. Ansi-C conformity.	Feature extended	91	Downwardly compatible.
Enumeration fcNotificationType modified. fcNotificationTypeReceiveBufferLevel added.	Feature extended	76	Downwardly compatible.
Added function fcbSetReceiveBufferLevelNotification.	New feature	79	Firmware-Version S6V1-F is needed
Function fcbGetEnumFlexCardsV3 modified. Ansi-C conformity.	Feature extended	62	Downwardly compatible.
Function fcbMonitoringStart modified. Ansi-C conformity.	Feature extended	223	Downwardly compatible.
Function fcbTrigger modified. Ansi-C conformity.	Feature extended	240	Downwardly compatible.
Function fcbSetTimer modified. Ansi-C conformity.	Feature extended	78	Downwardly compatible.
Function fcbSetCcTimerConfig modified. Ansi-C conformity.	Feature extended	234	Downwardly compatible.
Function fcbSetContinueOnPacketOverflow modified. Ansi-C conformity.	Feature extended	73	Downwardly compatible.
Function fcbNotificationPacket modified. Ansi-C conformity.	Feature extended	78	Downwardly compatible.
Added enumerations fcFRMsgBufCfgMode.	New features	113	
Added function fcbFRSetMsgBufCfgMode.	New feature	131	
Added function fcbFRSetHardwareTransmitFilter.	New feature	137	Firmware-Version S6V1-F is needed
Function fcbFRMonitoringStart modified. Ansi-C conformity.	Feature extended	105	Downwardly compatible.
Function fcbFRSetCcTimerConfig modified. Ansi-C conformity.	Feature extended	138	Downwardly compatible.
Function fcbFRSetHardwareAcceptanceFilter modified. Ansi-C conformity.	Feature extended	135	Downwardly compatible.
Function fcbCANMonitoringStart modified. Ansi-C conformity.	Feature extended	146	Downwardly compatible.
Function fcbCANSetMessageBuffer modified. Ansi-C conformity.	Feature extended	156	Downwardly compatible.
Function fcbCANTransmit modified. Ansi-C conformity.	Feature extended	160	Downwardly compatible.
Added function fcbSetBusTerminationCc.	New features	189	
Added function fcbGetBusTerminationCc.	New features	190	
Function fcbSetBusTermination modified. Ansi-C conformity.	Feature extended	191	Downwardly compatible.
Function fcbGetBusTermination modified. Ansi-C conformity.	Feature extended	192	Downwardly compatible.

Change	Reason	Page	Remark
Structure fcInfoSw modified. LicensedForLabviewDriver added.	Feature extended	60	Downwardly compatible.
Structure fcFWInfo added.	New features	194	
Added function fcbFWGetImageInfo.	New features	194	Firmware-Version S6V1-F is needed
Added function fcbSetSelectImage.	New features	195	Firmware-Version S6V1-F is needed

2.4.8 From S6V1-F to S6V2-F

Change	Reason	Page	Remark
Enumeration fcTriggerConditionEx modified. fcTriggerOutOnTimeStampChanged added.	Feature extended	183	Downwardly compatible
Enumeration fcTriggerConditionPMC modified. fcTriggerPMCOutOnTimeStampChanged added.	Feature extended	185	Downwardly compatible
Enumeration fcTimeStampSourceMode added.	New features	70	
Enumeration fcFlexCardDeviceId modified.	Feature extended	50	Downwardly compatible.
Structure fcTimeStampCfg added.	New features	70	
Added function fcbConfigureFlexCardTimeStamp.	New features	74	Firmware-Version S6V2-F is needed
Added function fcbGetCurrentHighResTimeStamp.	New features	75	Firmware-Version S6V2-F is needed
Structure fcNumberCC modified. FlexRaySelfSync added.	Feature extended	56	Downwardly compatible
Structure fcVersionCC modified. FaultTolerantCAN added.	Feature extended	57	Downwardly compatible

2.4.9 From S6V2-F to S6V3-F

Change	Reason	Page	Remark
Structure fcCANTxFifoConfig added.	New features	154	
Added function fcbCANSetTxFifoConfiguration.	New features	157	Firmware-Version S6V3-F is needed
Added function fcbCANGetTxFifoConfiguration.	New features	158	Firmware-Version S6V3-F is needed
Added function fcbCANTxFifoReset.	New features	159	Firmware-Version S6V3-F is needed
Added function fcbCANTxFifoTransmit.	New features	161	Firmware-Version S6V3-F is needed

2.4.10 From S6V3-F to S6V4-F

No API change.

2.4.11 From S6V4-F to S6V5-F

No API change.

2.4.12 From S6V5-F to S6V6-F

Change	Reason	Page	Remark
Added enum fcCANFDFrameFormat, structure fcCANCcBitTime, structure fcCANFDCcConfig, structure fcCANFDTxFrame.	New features	165	
Enum fcTimeStampSourceMode extended.	Feature extended	70	Downwardly compatible
Added function fcbCANFDSetCcConfiguration, fcbCANFDTransmit.	New features	167	Firmware-Version S6V6-F (6.5.0.33) is needed
Added function fcbGetTinyInfo.	New features	68	Firmware-Version S6V6-F (6.5.0.33) is needed

2.4.13 From S6V6-F to S6V7-F

Change	Reason	Page	Remark
Enum fcBusType extended to support CAN-FD and Ethernet	Feature extended	49	Downwardly compatible
Enum fcCCType extended to support CAN-FD and Ethernet	Feature extended	50	Downwardly compatible
Enum fcFlexCardDeviceId extended to support FlexCard PXIe3 and FlexCard PCIe3	Feature extended	50	Downwardly compatible
Added Enum fcConnector to support mapping connectors to cc index.	New feature	53	Only FlexCard PXIe3 and PCIe3 and FlexDevice
Enum fcNumberCC extended to support Ethernet	Feature extended	56	Downwardly compatible
Enum fcVersionCC extended to support mapping connectors to cc index.	Feature extended	57	Downwardly compatible
Added structure fcEthernetPacket.	New feature	93	Downwardly compatible
Structure fcPacket extended to support fcEthernetPacket	Feature extended	94	Downwardly compatible
Enum fcPacketType extended to support fcEthernetPacket	Feature extended	96	Downwardly compatible
Added new functions fcbEthMonitoringStart fcbEthMonitoringStop	New feature	171	Only FlexCard PXIe3 and PCIe3 and FlexDevice
Added support for FlexDevices	New feature	20	Only FlexDevice
Added new function fcbSetGlobalConfig	New feature	68	Only FlexDevice
Added new functions fcbEthSetCcConfiguration and fcbEthTransmit	New featiure	175	Only FlexDevice

2.4.14 From S6V7-F to S6V8-F

Change	Reason	Page	Remark
Enum fcTinyType extended.	Feature extended	52	Downwardly compatible
Enum fcInfoHw extended to support field FullCommunicationControllerAccess.	Feature extended	59	Downwardly compatible
Added function fcbCANSetFilterConfiguration.	New feature	168	Only FlexDevice
Added function fcbEthSetFilterConfiguration.	New feature	176	Only FlexDevice

2.5 Support

There is support available by *STAR ELECTRONICS GmbH & Co. KG* regarding software (device driver and API) and hardware. Before you submit a problem, ensure that you have the latest release of the software. The latest versions of the device driver and API are available from our support team or on our web site: http://www.star-cooperation.com/ee-solutions

If you encounter a problem, please send an email to support-ee@star-cooperation.com, including the following information:

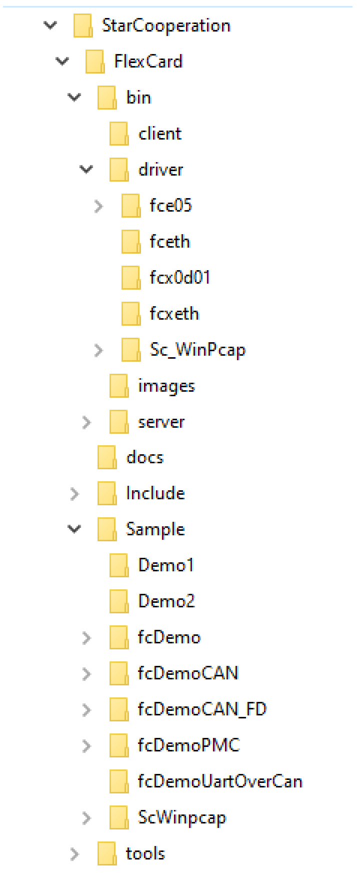
- Description of your problem
- Detailed steps to reproduce the problem
- Version number of the device driver or loadable kernel module
- Version number of the DLL or shared object library
- Version number of the hardware
- Version number of the firmware
- Serial number of your FlexCard
- The application you are using
- Your computer system (manufacturer and type of PC, e.g. Dell Inspiron 7500)
- Your operating system (Windows 7, Linux, Xenomai, VxWorks)
- The cardbus or PCI adapter in your PC (e.g. Texas Instruments, ...)
- If possible the CC configuration file or string or a CC parameter list

3 Getting Started

In this section the necessary steps for developing a FlexCard application with Windows operating systems are specified. First, the setup of the files and the integration in an Integrated Development Environment (IDE) is described. The next section provides a guideline with important steps to create an application. This includes the functions and structures which should normally be used. A more in depth view about the used functions can be found in chapter 4 et seq

3.1 Installation

For details about the installation process please refer to the installation section in [1]. After a successful installation of the developer package the following directory structure should exist:



```
StarCooperation
  FlexCard
    bin
      client
      driver
        fce05
        fceth
        fcx0d01
        fcxeth
        Sc_WinPcap
      images
      server
    docs
    Include
    Sample
      Demo1
      Demo2
      fcDemo
      fcDemoCAN
      fcDemoCAN_FD
      fcDemoPMC
      fcDemoUartOverCan
      ScWinpcap
    tools
```

Figure 3: FlexCard directory structure

The directory *Docs* contains the documentation (Instructions for Use and Getting started guides) in PDF format. The API documentation is only present if during the installation the setup type Developer was selected.

The directory *bin/driver* contains the folders for the manual installation of the device driver:

- *Fce05 (FlexCard PMC2 and USB-M device driver for the Windows™ 10 and later 64 Bit operating system)*
- *fceth (FlexCard PMC2 ethernet device driver for the Windows™ 10 and later 64 Bit operating system)*
- *Fcxd01 (FlexCard PXIe3 and PCIe3 device driver for the Windows™ 10 and later 64 Bit operating system)*
- *fcethx (FlexCard PXIe3 and PCIe3 ethernet device driver for the Windows™ 10 and later 64 Bit operating system)*

- *Sc_WinPcap (FlexCard PMC2, PXIe3 and PCIe3 WinPcap packet filter device driver for the Windows™ 10 and later 64 Bit operating system)*

The previous directory is not required for developing user defined application, whereas the two following directories are a must-have for developers.

The directory *Include* contains the API definition, namely the *fcBase* header files and library files for 64 Bit Windows™ 10 and later operating system plattforms.

- *fcBase.h*: The file includes the definition of the basic API functions.
- *fcBaseTypes.h*: The file contains the data types and enumerations (e.g. possible error codes) used by the basic functions.
- *fcBaseFlexRay.h*: This file contains definitions of functions specific for FlexRay.
- *fcBaseTypesFlexRay.h*: The data types and enumerations for the FlexRay functions are defined here.
- *fcBasePMC.h*: The file includes additional definitions of API functions which are to use with FlexCard PMC and FlexCard PMC-II only.
- *fcBaseTypesPMC.h*: The file (for the FlexCard PMC and FlexCard PMC-II only) contains additional data types and enumerations used in this library.
- *fcBaseCAN.h*: The file includes the definition of the API functions which are to be used with a CAN license only. Contains CAN/ CAN-FD functionality.
- *fcBaseTypesCAN.h*: The file (for FlexCards with CAN license only) contains additional data types and enumerations for CAN/ CAN-FD used in this library.
- *fcBaseUARTOverCAN.h*: The file includes the definition of the API functions which are used for the UARTOverCAN functionality.
- *fcBaseTypesUARTOverCAN.h*: The file contains additional data types and enumerations for UARTOverCAN used in this library.
- *Lib/amd64*: Contains the fcBase.dll and fcBase.lib as 64 Bit version.
- *Lib/x86*: Contains the fcBase.dll and fcBase.lib as 32 Bit version.

The directory *Sample* contains the following directories:

- *Demo1*: Configuration files for a cluster composed of two FlexCards
- *Demo2*: Configuration files for a cluster composed of one FlexCard and two FlexNodes.
- *fcDemo:* Contains the source files for the demo application.
- *fcDemoCAN*: Contains the source files for the CAN demo application for a FlexCard.
- *fcDemoCANFD*: Contains the source files for the CAN-FD demo application for a FlexCard.
- *fcDemoPMC:* Contains the source files for the demo application for a FlexCard PMC-II.
- *fcDemoUARTOverCAN*: Contains the source files for the demo application for the UARTOverCAN functionality.
- *ScWinPcap*: Contains the source files for the demo application for accessing WinPcap functionality.

The directory *Tools* contains the following applications:

- *CANBaudrateCalculator.exe*: Application to calculate CAN CC configuration for fcBase CAN API.
- *fcDemo.exe:* The demo application for one FlexRay CC.
- *fcDemoCAN.exe*: The demo application for two CAN CCs.
- *fcDemoCANFD.exe*: The demo application for two CAN-FD CCs.
- *fcDemoPMC.exe:* The demo application for two FlexRay CCs.
- *fcDemoUARTOverCAN.exe:* The demo application for two UARTOverCAN CCs.

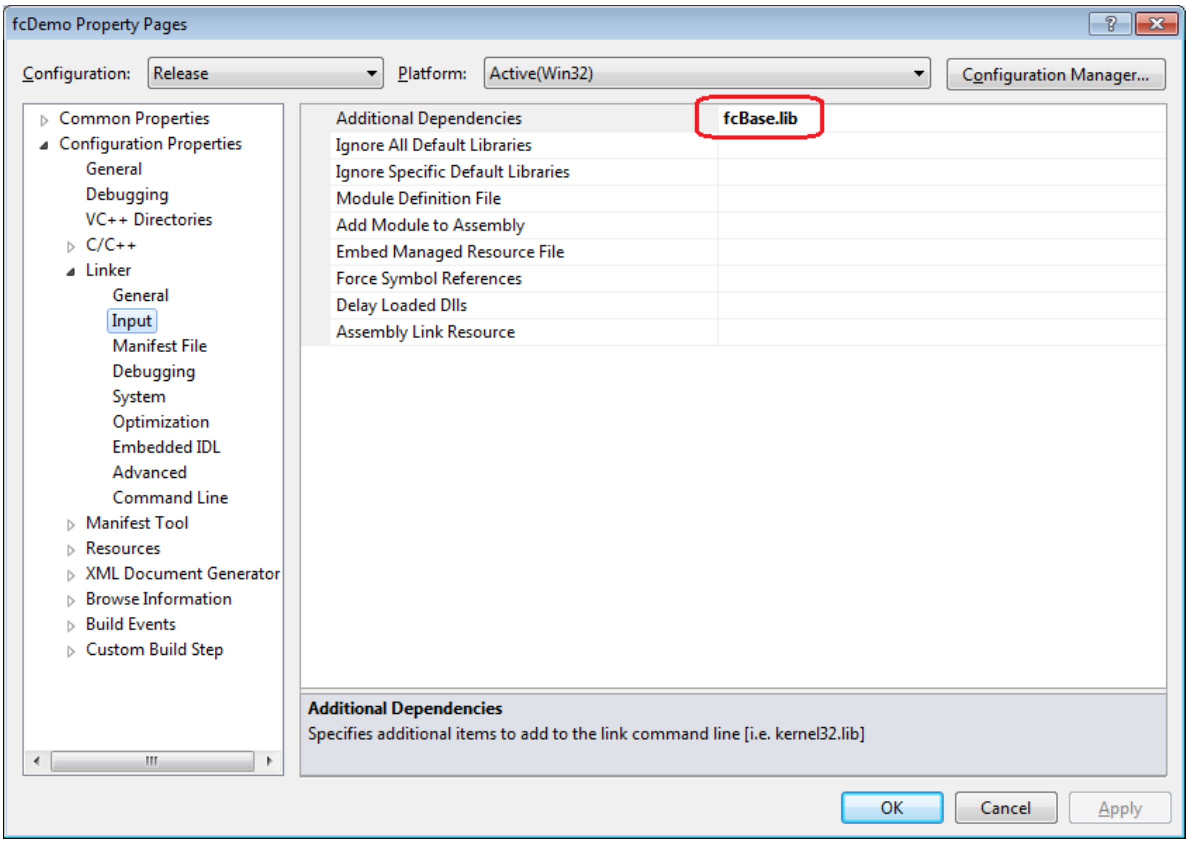
- *FlexAlyzerV2.exe*: FlexRay and CAN monitoring application for FlexCard products.
- *FlexUpdate.exe:* Firmware and license update application for FlexCard products.

Information

The windows installer will copy the *fcBase.dll* into your *<windows>\system32* directory. On Windows 64 bit, *fcBase.dll* 32 Bit will be copied to *<windows>\SysWOW64*. If you do not use the windows installer, please check if the desired version of the DLL is loaded. A description of the DLL search order which is used by the Windows operating system can be found in [2].

3.2 Integration

There are different ways to integrate the fcBase DLL into your application depending on the development platform and language. Under Microsoft Visual Studio the integration is done via the property pages/project settings.



fcDemo Property Pages

Configuration: Release

Platform: Active(Win32)

Additional Dependencies	**fcBase.lib**
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies

Specifies additional items to add to the link command line [i.e. kernel32.lib]

Figure 4: Integration under Microsoft Visual Studio 2010

Another alternative for Microsoft compiler users is to include the fcBase API via the Microsoft specific pre-processor command:

```
#pragma comment( lib , “fcBase.lib”)
```

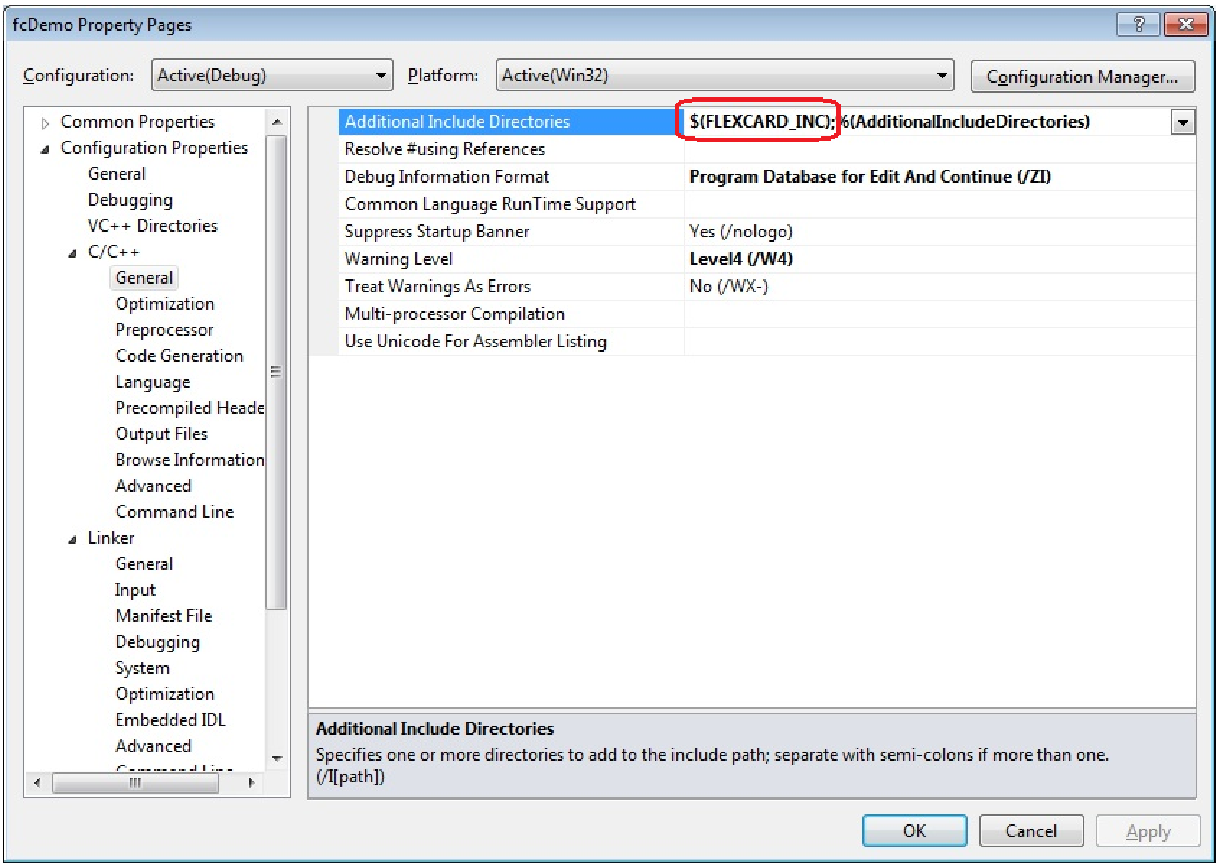
To complete the integration of the fcBase API into your user defined application, you have to add the files *fcBaseTypes.h*, *fcBaseTypesFlexRay.h*, *fcBase.h* and *fcBaseFlexRay.h*. The include order is important because the file *fcBase.h* uses definitions which are declared in *fcBaseTypes.h* and the file *fcBaseFlexRay.h* uses definitions which are declared in *fcBaseTypes.h* and *fcBaseTypeFlexRay.h*. For FlexCard PMC-II usage please also include the files *fcBaseTypesPMC.h* and *fcBasePMC.h* in the right order. In case you want to access the CAN functionality of the FlexCard, the files *fcBaseTypesCAN.h* and *fcBaseCAN.h* should be also included.

```
#include "fcBaseTypes.h"
#include "fcBaseTypesFlexRay.h"
#include "fcBase.h"
#include "fcBaseFlexRay.h"

//Additional for PMC usage
#include "fcBaseTypesPMC.h"
#include "fcBasePMC.h"

//Additional for CAN usage
#include "fcBaseTypesCAN.h"
#include "fcBaseCAN.h"
```

The setup program sets the environment variable FLEXCARD_INC which directly points to the fcBase include directory. This variable can be used as shown in the figures below.

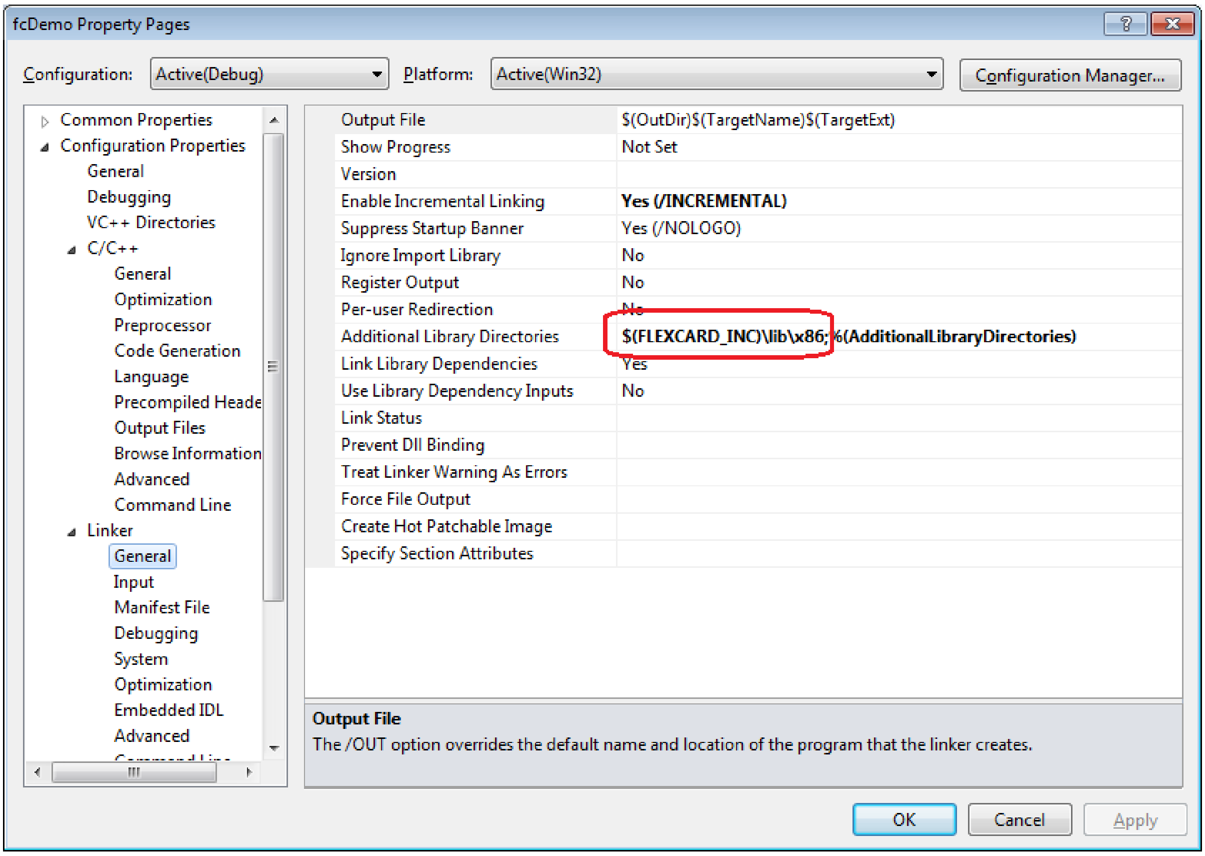

fcDemo Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)

Additional Include Directories	$(FLEXCARD_INC);%(AdditionalIncludeDirectories)
Resolve #using References	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Common Language RunTime Support	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level4 (/W4)
Treat Warnings As Errors	No (/WX-)
Multi-processor Compilation	
Use Unicode For Assembler Listing	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Figure 5: Using the variable FLEXCARD_INC under Microsoft Visual Studio 2010 (Compiler)



Configuration: Active(Debug) Platform: Active(Win32)

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**$(FLEXCARD_INC)\lib\x86;%(AdditionalLibraryDirectories)**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Output File
The /OUT option overrides the default name and location of the program that the linker creates.

Figure 6: Using the variable FLEXCARD_INC under Microsoft Visual Studio 2010 (Linker)

NOTICE

Ensure you use the directory of the fcBase library and header files which corresponds to the loaded DLL. A description of the DLL search order which is used by the windows operating system can be found in [2].

3.2.1 Calling Convention

The dynamic link library for Windows was developed under Microsoft Visual Studio 2010 C++. The Microsoft C/C++ compiler supports several calling conventions (__cdecl, __stdcall, __fastcall, this, naked). To provide access to the *fcBase.dll* 32 Bit for other languages (e.g. Visual Basic), the functions are declared with __stdcall calling convention (function arguments are pushed onto the stack from right to left, the callee cleans the stack). On *fcBase.dll* 64 bit the user does not have to specify a calling convention, because there is only one.

3.2.2 Loading the Dll

When you use the standard “inf” installation, you don’t have to append the *fcBase.dll* path to the Windows PATH environment variable. On a Windows 64 Bit installation, loading *fcBase.dll* in a 64 bit application will load it from *<windows>\system32* and loading it in a 32 bit application will load it from *<windows>\SysWOW64.*

3.2.3 Multithreading

All functions, which are not declared as obsolete, of the fcBase library are thread-safe. If you are using the fcBase functions in the context of a multi-threaded program, the library ensures that only one thread is accessing the internal shared data at any given time.

3.3 Basic Workflow

This section will guide you through the necessary workflow for creating an application for the FlexCard. The following figure shows a typical workflow. For FlexRay refer to 5.1 Basic FlexRay Workflow, for CAN refer to 6.1 Basic CAN Workflow. The main functions and principles for building a user defined application are introduced in this chapter. Demo applications for the FlexRay/CAN usage (source code and binary) are installed with the FlexCard Windows Developer Setup.

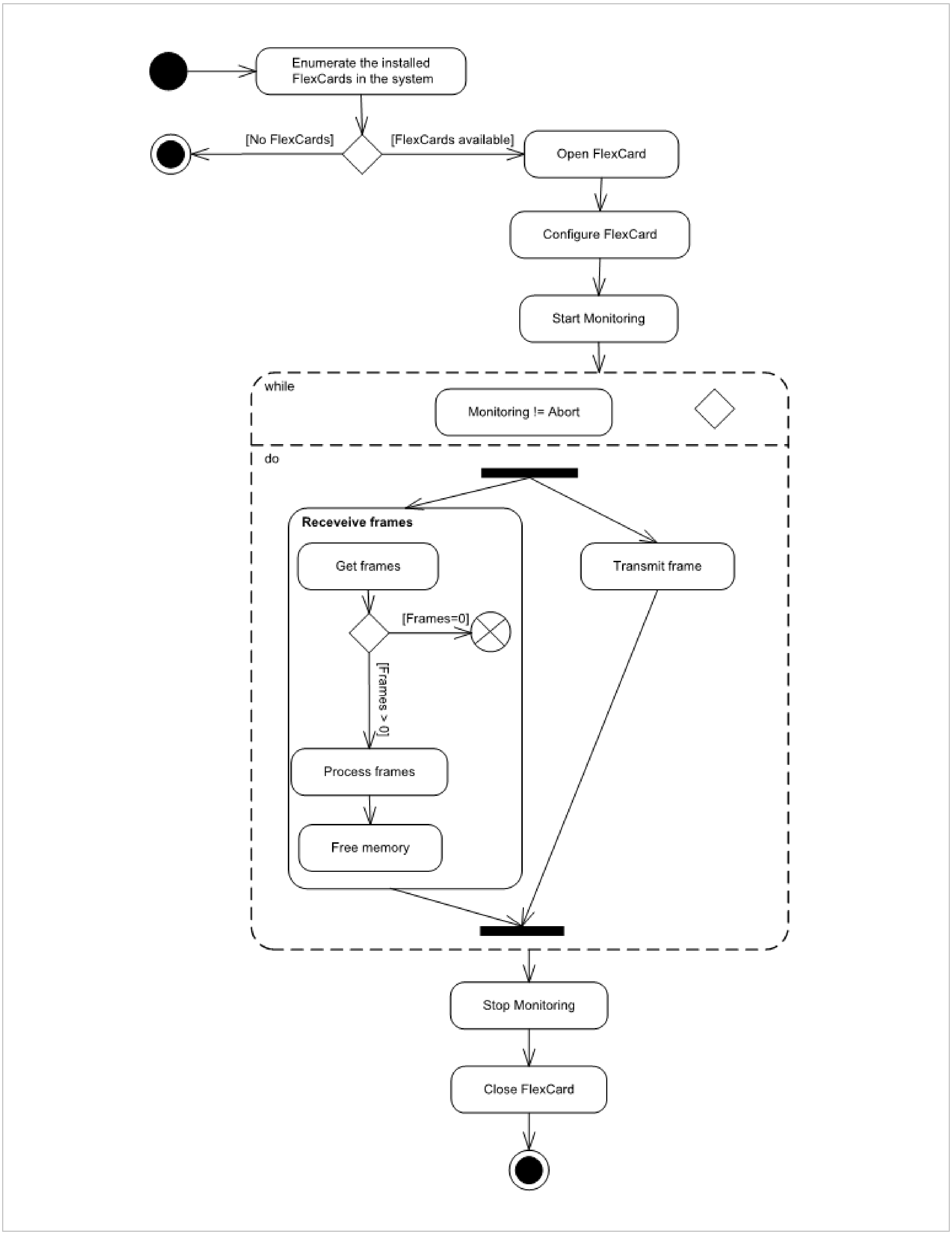


Figure 7: Typical FlexCard workflow

3.3.1 Setting Up the Project

The sample code projects are installed by the setup on Windows to the Program Files path, to the directory StarCooperation\FlexCard\Sample. This directory cannot be used directly because only users with administrator rights may write there. That is why the user should copy the sample directory to different place.

For the development of the example project we will use Microsoft Visual Studio 2003 with the programming language C/C++ and the Microsoft Foundation Class (MFC). The Visual Studio project wizard will generate the framework for our MFC dialog based application (for more details, please refer to the documentation of Microsoft Visual Studio 2003).

As described in the chapter Integration, we must add the library and header files of the fcBase API. This can be done easily at the end of the file *stdafx.h*. Ensure your compiler and linker use the correct path to the fcBase header and library files.

```
// fcBase API
#pragma comment(lib,”fcBase.lib”)
#include “fcBaseTypes.h”
#include “fcBase.h”

// additional for FlexCards with FlexRay
#include “fcBaseTypesFlexRay.h”
#include “fcBaseFlexRay.h”

// additional for FlexCards with CAN
#include “fcBaseTypesCAN.h”
#include “fcBaseCAN.h”

// additional for FlexCard PMC and FlexCard PMC-II
#include “fcBaseTypesPMC.h”
#include “fcBasePMC.h”

// own headers
```

3.3.2 Get the Installed FlexCards

Before we can open a connection to a FlexCard, we require a valid FlexCard identifier. This can be done with the function **`fcbGetEnumFlexCardsV3`** which returns a list of FlexCards found in the system. In the method `CselectFlexCardDlg::OnInitDialog()` in our example we call **`fcbGetEnumFlexCardsV3`** to fill the combo box with available FlexCards found in the system.

ℹ	**Information**
	The **`fcInfoHwSw`** structure contains valid FlexCard information only if the member `FlexCardId` is greater than 0. The `FlexCardId` is later used to open a connection to the FlexCard.

```
fcError e = fcbGetEnumFlexCardsV3(&m_pInfoHwSw, false);
if (0 == e)
{
      // Iterate through the list of flexcards
      fcInfoHwSw* pCurrent = m_pInfoHwSw;
      while (NULL != pCurrent)
      {
            // only if we got a valid flexcard identfier
            if ( 0 != pCurrent->FlexCardId )
            {
                  Cstring szItem;
                  szItem.Format(“FlexCard #%d”,pCurrent->InfoHardware.Serial);

                  // Add the string to the combo box
                  int nIndex = m_FlexCardComboBox.InsertString(0,szItem);
                  m_FlexCardComboBox.SetItemDataPtr(nIndex,pCurrent);
            }
```

```
                // get the next flexcard
                pCurrent = pCurrent->pNext;
        }
}
```

If the user selects one of the items in the combo box, we save the member FlexCardId from the structure **fcInfoHwSw** into the member variable `m_flexcardIdentifier` (see `CselectFlexCardDlg::UpdateVersionInformation`).

```
Int nCurrentSelection = m_FlexCardComboBox.GetCurSel();
if (-1 != nCurrentSelection)
{
        fcInfo* pCurrent =
                (fcInfo*)m_FlexCardComboBox.GetItemDataPtr(nCurrentSelection);

        // Save the flexcard identifier
        m_flexcardIdentifier = pCurrent->FlexCardId;
        …
}
```

Once finished with the selection of a FlexCard, we have to free the memory which was allocated by the function **fcbGetEnumFlexCardsV3**.

```
CselectFlexCardDlg::~CselectFlexCardDlg()
{
        if (NULL != m_pInfo)
        {
                fcFreeMemory(fcMemoryTypeInfoHwSw,m_pInfo);
                m_pInfo = NULL;
        }
}
```

3.3.3 Open a Connection

After getting a valid FlexCard identifier, we use it to open a connection to the FlexCard. The function **fcbOpen** expects this identifier and returns a handle (to the previously selected FlexCard) which is later used in all other functions.

	Information
ℹ	The function **fcbOpen** resets all configuration settings. That means that all Communication Controller registers are set to their default value and the FlexRay message buffers are configured as FIFO buffer

```
...
m_hFlexCard = NULL;
fcError e = fcbOpen(&m_hFlexCard,dlg.FlexCardIdentifier());
...
```

3.3.4 FlexRay Configuration behavior FlexCard

To integrate the FlexCard into a FlexRay cluster it is essential to configure its Communication Controller registers. These registers describe global cluster parameters (e.g. gdStaticSlot), node parameters (e.g. pMicroPerCycle) and Communication Controller specific settings. The global cluster parameters are identical for all nodes of a cluster, whereas the node parameters are set for each node individually.

The FlexRay CC configuration is possible via directly entering the bus parameters, or by passing a CHI-file. If one of the parameters is not correct, the integration of the FlexCard and/or the communication may fail. Therefore, it is recommended to use a tool which helps generate a valid FlexRay configuration for each node. FlexConfig Developer from STAR ELECTRONICS GmbH & Co. KG is such a tool. It exports the bus parameters as CHI-file.

In our example we use a CHI-compatible string to configure the FlexCard. As the function **fcbFRSetCcConfigurationChi** expects a string, we read and parse the configuration file into a string.

```
Std::string s;
std::ifstream file(szPath);
if (! File.is_open())
{
      // Print error message
      return;
}

char ch;
while (file.get(ch)) s += ch;
file.close();
```

This string is passed to the function **fcbFRSetCcConfigurationChi** which will configure the specified Communication Controller registers described in the chi file. Setting a configuration via this function will override the previous configuration of this CC.

```
fcCC eCC = fcCC1;
fcError e = fcbFRSetCcConfigurationChi(m_hFlexCard, eCC, s.c_str());
```

As we want to transmit messages on the FlexRay bus, we have to configure transmit buffers for the FlexCard. To configure such a buffer two options exist: Using the function **fcbFRConfigureMessageBuffer** or via the CHI configuration string. There is a significant difference between these two methods: While **fcbFRConfigureMessageBuffer** returns the index of the configured message buffer, **fcbFRSetCcConfigurationChi** does not. And considering that to transmit the content of a message buffer, the function **fcbFRTransmit** requires its index; we need a way to retrieve it. The following code performs this task for all configured transmit message buffers.

```
// Get all transmit message buffers
unsigned int bufferIdx = 1; // The first valid buffer is 1
while (true)
{
      fcMsgBufCfg cfg;
      fcCC eCC = fcCC1;

      // as long no error occurs we try to get each buffer
      fcError e = fcbFRGetMessageBuffer(m_hFlexCard,eCC,bufferIdx,&cfg);
      if (0 != e) break;

      // is this a tx buffer, then add it to our list
      if (fcMsgBufTx == cfg.Type)    Buffers[bufferIdx] = cfg;

      // next buffer index
      bufferIdx++;
}
```

The function **fcbFRConfigureMessageBuffer** is used to add a message buffer dynamically. This function checks the given message buffer settings and will report an error in the case of a mismatch with a global cluster parameter or a node specific parameter. The returned error informs the user about the mismatch. Before setting the members of a struct, initialize it to zero.

```
fcMsgBufCfg cfg;

memset(&cfg, 0, sizeof(fcMsgBufCfg)); // Initialize to zero
cfg.Type = fcMsgBufTx;
cfg.ChannelFilter = fcChannelA;
cfg.CycleCounterFilter = 0;
cfg.Tx.FrameId = 5;
cfg.Tx.TxAcknowledgeEnable = 1;
cfg.Tx.PayloadLength = 16;
cfg.Tx.PayloadLengthMax = 16;
cfg.Tx.PayloadPreambleIndicator = 0;
cfg.Tx.StartupFrameIndicator = 0;
cfg.Tx.SyncFrameIndicator = 0;
cfg.Tx.TransmissionMode = fcMsgBufTxSingleShot;
cfg.Tx.TxAcknowledgeShowNullFrames = 0;
cfg.Tx.TxAcknowledgeShowPayload = 0;

fcCC eCC=fcCC1;
```

```
unsigned int bufferIdx = 0;
fcError e = fcbFRConfigureMessageBuffer(m_hFlexCard,eCC,&bufferIdx,cfg);

if (0 != e)
{
      ShowErrorI;
}
```

Via the function **`fcbFRReconfigureMessageBuffer`** and with some restrictions the user can modify an existing message buffer.

3.3.5 Start and Stop a FlexRay Measurement

After having successfully configured the FlexCard, the monitoring can be started through the function **`fcbFRMonitoringStart`**. To use the FlexCard as a wake-up node, the flag `enableWakeup` has to be set to true (the FlexCard must have been previously configured with the correct wake-up settings). To use the FlexCard as a start-up node, the flag `enableColdstart` has to be set to true (one transmit buffer with both start-up and sync flags set must have been previously configured). In the case of the integration of a FlexCard into a running cluster, none of these two parameters has to be set. To be notified at the start of each cycle, the flag `enableCycleStartEvents` has to be set to true and the user has to provide an event object (used to signal when a new cycle starts) to the function **`fcbSetEventHandleV2`**.

```
// create the event handle which is signaled when a new cycle starts
const bool cyclestart = true;
fcCC eCC=fcCC1;

HANDLE hEvent = ::CreateEvent(NULL,FALSE,FALSE,NULL);

// inform the api that the event should be used when a new cycle starts
fcError e = fcbSetEventHandleV2(m_hFlexCard, eCC,
      hEvent, fcNotificationTypeCycleStarted);

// no coldstart and wake-up attempt have to be done
const bool coldstart = false;
const bool wakeup = false;

fcError e = fcbFRMonitoringStart(m_hFlexCard,eCC,fcMonitoringNormal,true,
             cyclestart,coldstart,wakeup);
```

After starting the monitoring, it is highly recommended to verify that the integration has succeeded. It can be determined either by receiving (via **`fcbReceive`**) a status packet with the flag `fcStatusStartupCompletedSuccessfully` set or by calling the function **`fcbFRGetCcState`** and checking that the return value is `fcStateNormalActive`.

Calling the function **`fcbFRMonitoringStop`** will stop the monitoring and set back the Communication Controller in its configuration state, `fcStateConfig`.

3.3.6 Receive FlexRay Frames

Once the monitoring started, the FlexCard begins to monitor the FlexRay bus. The received FlexRay frames and the FlexCard generated packets (Info frame, Error frame, etc.) can be fetched by the function **`fcbReceive`**. A call to this function will get all available packets from the FlexCard.

The code below uses the cycle start event to collect the received data of the previous cycles. If the event is signalled or if the timeout elapses, we get the available received packets (fcPacket) by calling the function **`fcbReceive`**. The timeout is used as a fallback if the FlexCard is not successfully integrated and no cycle start events could be generated. The FlexCard USB-M does not support events, so WAIT_TIMEOUT is signalled.

```
DWORD CdemoDlg::Thread()
{
      // … Code removed for listing …
      fcCC eCC=fcCC1;
      // create the event handle which is signaled when a new cycle starts

      hEvents[1] = ::CreateEvent(NULL,FALSE,FALSE,NULL);

      // inform the api that the event should be used when a new cycle starts
      fcError e = fcbSetEventHandleV2(m_hFlexCard,eCC,hEvents[1],
            fcNotificationTypeCycleStarted); // not for FlexCard USB-M

      // … Code removed for listing

      while (endlessLoop)
      {
            // Wait until an event is signaled or until timeout has elapsed
            DWORD dwResult = ::WaitForMultipleObjects(2,hEvents,false,
                  dwTimeOutMilliseconds);

            switch (dwResult)
            {
            case WAIT_OBJECT_0+1:  // Cycle start event
            case WAIT_TIMEOUT:     // or time is elasped
                  {
                        //Update our transmit buffers
                        AutomaticTransmit();

                        fcPacket* pPacket = NULL;
                        e = fcbReceive(m_hFlexCard, &pPacket);
                        if (0 == e)
                        {
                              ProcessPackets(pPacket);
                              e =   fcFreeMemory(fcMemoryTypePacket, pPacket);
                        }
                        else
                        {
                         // … Code removed for listing
                        }
                  }
                  break;

            // … Code removed for listing
      }
      // … Code removed for listing
}
```

The **fcbReceive** function returns the received data as a linked list of packets. The code below goes through the whole list and processes each packet.

```
Void CdemoDlg::ProcessPackets(fcPacket* pPackets)
{
      fcPacket* p = pPackets;
      while (NULL != p)
      {
            switch (p->Type)
            {
            case fcPacketTypeInfo:
                  // … Code removed for listing
                  break;
            case fcPacketTypeFlexRayFrame:
                  // … Code removed for listing
                  break;
            // … Code removed for listing
            default:

            };
            // get the next packet
            p = p->pNextPacket;
      }
}
```

Information

If an error packet with the flag *fcErrFlexcardOverflow* is received, the monitoring can not continue in case the FlexCard is set to stop if a packet overflow occurred (**fcbSetContinueOnPacketOverflow**). This error occurs if the application is too slow to receive and process the packets. In such a case it is necessary to stop the monitoring and start it again.

After processing the packets, the memory allocated by the packet list must be released.

```
ProcessPackets(pPacket);
e =   fcFreeMemory(fcMemoryTypePacket, pPacket);
```

3.3.7 Transmit FlexRay Frames

To transmit a frame on the FlexRay bus you need to have previously configured a transmit buffer and to know its index. The transmission is done by calling the **fcbFRTransmit** function.

```
fcCC eCC=fcCC1;
fcWord payload[fcPayloadMaximum];
payload[0] = 0x0001; // Update your payload data

fcError e = fcbFRTransmit(m_hFlexCard,eCC,bufferIdx,
            payload,payloadLength);
```

The transmit function expects the index of the Communication Controller, the index of the transmit buffer, the payload of the frame (the data) and the length of the payload section (the data length). The configured payload length (set during configuration of the transmit buffer) and the payload length to transmit (set during call to **fcbFRTransmit**) must match. It is recommended to check the error that is returned by the function.

3.3.8 Close a Connection

Once the measurement finished, closing the connection to the FlexCard is done by calling the function **fcbClose**.

3.3.9 Connector/CC Mapping (FlexCard PXIe3 and FlexCard PCIe3)

The FlexCard PXIe3 and FlexCard PCIe3 have five connectors at the front panel. The connectors are numbered from 1 to 5. Each connector can have 8 channels, they are numbered A to H. You can use the PcHwIf and the command COM_GetRBSChannelsInfoReq to get a list of the channels from the RBS perspective. The rbs channel number and the rbs channel bus type is contained. A different way to get the rbs channel number for a bus is in FlexConfig RBS. After you added a bus on a connector, you can hover the mouse over it. The HW-Channel displayed is the rbs channel number.

The fcBase API uses a bus type and a cc index to identify a channel. A table of the mapping of a rbs channel bus type to a fcBase bus type follows.

Rbs channel bus type	fcBase bus type
COM_BUSTYPE_FlexRay	fcBusTypeFlexRay
COM_BUSTYPE_CAN	fcBusTypeCAN
COM_BUSTYPE_CAN_LS	
COM_BUSTYPE_CAN_HS	
COM_BUSTYPE_CAN_FD	

Rbs channel bus type	fcBase bus type
COM_BUSTYPE_BROADR_REACH	fcBusTypeEth
COM_BUSTYPE_BROADR_REACH_SWITCHED	
COM_BUSTYPE_ETH	
COM_BUSTYPE_ETH_SWITCHED	

The fcBase API goes through all rbs channel numbers from 1 to maximum. It gives the first occurrence of a bus type the fcBase index fcCC1. The next occurrence will get the fcBase index fcCC2, and so on.

The FlexRay bus type is a special case. FlexRay channel A and FlexRay channel B get different rbs channel numbers. In the fcBase API they are addressed together via fcBase bus type and a fcCC.

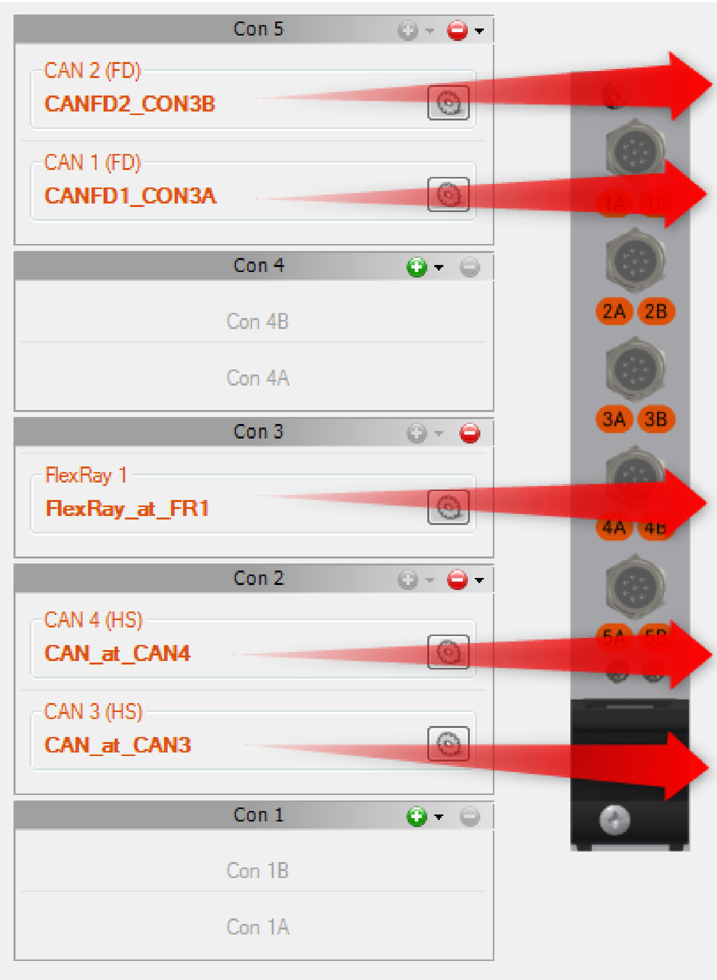
When you use the fcBase API and call fcbGetEnumFlexCardsV3, you get a list of the devices in the pc. For each device, the available ccs are listed. For each cc, the connector that it corresponds to is listed. The variable ConnectorIndex can be found in the struct fcVersionCC.

In order that analyzing is working on a channel, several conditions have to be met:

- A compatible FlexTiny module has to be mounted on the device
- A FlexConfig RBS application needs to be running
- A decoder must be present in the firmware

When not all of these conditions are met, the channel will not be visible in the struct fcVersionCC that is returned by fcbGetEnumFlexCardsV3.

The following example shows the device configuration in FlexConfig RBS and the mapping to the fcBase bus type and cc index.



Rbs channel 10 - fcBusTypeCAN, fcCC4

Rbs channel 9 - fcBusTypeCAN, fcCC3

Rbs channel 5,6 - fcBusTypeFlexRay, fcCC1

Rbs channel 4 - fcBusTypeCAN, fcCC2

Rbs channel 3 - fcBusTypeCAN, fcCC1

Figure 8: The device configuration in FlexConfig RBS and the mapping to the fcBase bus type and cc index.

3.4 Library compatibility

The Library offers a C interface to the application. For compatibility reasons, new features are added as new Library functions.

To make extensions to existing Library functions possible, it uses reserved fields in structs. When a new Library version introduces a new field, a reserved field is used for it. That way the size of the function parameter stays the same.

3.4.1 Library getter function

App old. Library old.	No problem.
App old. Library new (backwards compatibility).	App must ignore the reserved fields. E.g. the App must not binary compare two structs.
App new. Library old (upwards compatibility).	App should consider checking the library version.
App new. Library new.	No problem.

3.4.2 Library setter function

App old. Library old.	No problem.
App old. Library new (backwards compatibility).	App must zero the reserved fields.
App new. Library old (upwards compatibility).	App should consider checking the library version.
App new. Library new.	No problem.

4 General FlexCard API Description

This chapter describes the application programming interface in detail. Each section represents a group of operations dedicated to a common purpose (configuring, initializing, receiving…). For each group, the data definition (structures and enumerations) is first described, followed by the API functions illustrated with code samples.

For additional API description, which depends on the used operating system and/or used FlexCard device, please refer to the following major chapters.

Please note the following limitations of the FlexCard driver:

- The latency in transmit direction is influenced by the underlying operating system. Jitter is possible when the driver is interrupted by processes with a higher priority.
- If the PC lacks performance, it may lead to a buffer overflow in the receive path. In this case the measurement must be restarted.
- The FlexCard API does not support the loading of network databases directly.

4.1 Error Handling

Almost every function in this library returns with an error status number. The meaning of this status code can be retrieved with the following functions and enumerations. Additional to this status code it is possible to get hints about the error if you use the tool *fcTracerControl.exe*. For more information about the tracing tool please refer to Tracing.

ℹ	**Information**
	In a few situations you will not get a meaningful error text. This happens for example if the device driver reports an error to the API. In such a case only the error code ACTION_FAILED is returned. To get a more detailed error description it may be helpful to use the tracing module.

4.1.1 Type Definitions

4.1.1.1 fcError

This type provides information about an error. A zero value means no error occurred. To extract the detailed information about an error, use the functions **fcGetErrorType**, **fcGetErrorText** and **fcGetErrorCode**.

```
Typedef unsigned int fcError;
```

ℹ	**Information**
	fcError is not equivalent to fcErrorCode (see **fcErrorCode**)

4.1.2 Enumerations

4.1.2.1 fcErrorCode

This enumeration contains all error codes which are reported by the fcBase API. To extract the error code from a **fcError** use the **fcGetErrorCode** function. To get information for the error code, use the **fcGetErrorText** function. For detailed error description please refer to the Headerfile *fcBaseTypes.h*.

See Also

fcGetErrorCode, **fcGetErrorText**, **fcGetErrorType**

4.1.2.2 fcErrorType

This enumeration defines the type of error information. To get the fcErrorType from a **fcError**, use the **fcGetErrorType** function.

```
Typedef enum fcErrorType
{
      fcErrorTypeSuccess         = 0,
      fcErrorTypeInformation     = 1,
      fcErrorTypeWarning         = 2,
      fcErrorTypeError           = 3,
} fcErrorType;
```

Members

fcErrorTypeSuccess

No error.

fcErrorTypeInformation

The error should be treated as an information message. The function has succeeded but wants to give additional information.

fcErrorTypeWarning

The error should be treated as a warning message. The function has succeeded but the input parameters are modified or not completely accepted.

fcErrorTypeError

The error should be treated as an error message. The function has failed.

See Also

fcGetErrorType, **fcGetErrorText**, **fcGetErrorCode**

4.1.3 fcGetErrorCode

This function returns the error code for a given error. A zero value indicates no error occurred. The list of all error codes can be found in the **fcErrorCode** enumeration (see *fcBaseTypes.h*).

```
fcErrorCode fcGetErrorCode(
      fcError error
);
```

Parameters

error

[IN] An error value of type fcError

Return values

Error code

See Also

fcErrorCode, **fcGetErrorType**, **fcGetErrorText**

4.1.4 fcGetErrorType

This function returns the error type for a given error. Please, refer to **fcErrorType** for more details.

```
fcErrorType fcGetErrorType(
      fcError error
);
```

Parameters

error

[IN] An error value of type fcError

Return values

Error type

See Also

`fcErrorType`, **`fcGetErrorCode`**, **`fcGetErrorText`**

Example

```
fcError e = fcbFRSetCcConfigurationChi(hFlexCard,eCC,pszChi);
if (0 != e)
{
       // oops, something went wrong
       fcErrorType etype = fcGetErrorTypeI;
       if (fcErrorTypeInformation == etype || fcErrorTypeWarning == etype)
       {
              // ok, the function succeeds, but the function wants to give us some
              // information
              PrintInfo(e);
       }
       else
       {
              PrintError(e);
       }
}
```

4.1.5 fcGetErrorText

This function returns the corresponding error text for a given error. To free the memory which was allocated by this function, please use the function **`fcFreeMemory`** with the type *`fcMemoryTypeString`* (see **`fcMemoryType)`**. Some text will be generated at runtime to provide a more detailed error description.

```
fcError fcGetErrorText(
       fcError error,
       char** szText
);
```

Parameters

`error`

[IN] An error value of type `fcError` for which an error text should be returned.

`szText`

[OUT] Address of a char pointer which holds the address for the generated error text.

Return values

If the function succeeds (return value is zero), the parameter `szText` contains the requested error text. If the function fails `szText` isn't valid. The fcErrorCode `NULL_PARAMETER` is returned if the *`szText`* parameter is a null pointer, `TEXT_NOT_DEFINIED` if no error text for the given error could be found, or `MEMORY_ALLOCATION_FAILED` to indicate that the memory allocation for the error text failed.

Example

```
fcError e = fcbOpen(&hFlexCard,flexcardId);
if (fcErrorTypeSuccess != fcGetErrorTypeI)
{
       char* szErrorText = NULL;
       if (0 == fcGetErrorText(e, &szErrorText))
       {
              // Print the error text and then free up the memory
              PrintErrorText(szErrorText);
              fcFreeMemory(fcMemoryTypeString, reinterpret_cast<void*>(szErrorText));
       }
}
```

See Also

`fcFreeMemory`, **`fcGetErrorType`**, **`fcGetErrorCode`**

4.2 Memory Handling

As the API allocates memory for you, it has to free up this memory. For this task the API provides the function **`fcFreeMemory`** which frees only the memory allocated by a function from the API. The reason why the API provides this mechanism is that your application may be linked to a different C/C++ run-time library than this library. Allocating memory in one module and freeing it in another one (with different run-time libraries) may fail or cause a run-time error. Another reason for this implementation is that the API can use its own memory management in order to reuse the memory blocks.

4.2.1 Enumerations

4.2.1.1 fcMemoryType

This enumeration defines the memory types needed to release the memory allocated by the functions **`fcGetErrorText`**, **`fcbGetEnumFlexCards (Obsolete)`**, **`fcbGetEnumFlexCardsV2 (Obsolete)`**, **`fcbGetEnumFlexCardsV3`**, **`fcbGetInfoFlexCard`** and **`fcbReceive`**.

```
Typedef enum fcMemoryType
{
      fcMemoryTypeString,
      fcMemoryTypeInfo,
      fcMemoryTypePacket,
      fcMemoryTypeInfoV2,
      fcMemoryTypeInfoHwSw,
} fcMemoryType;
```

Members

`fcMemoryTypeString`

Memory is of the type char[]

`fcMemoryTypeInfo`

Memory is of the type `fcInfo`

`fcMemoryTypePacket`

Memory is of the type fcPacket

`fcMemoryTypeInfoV2`

Memory is of the type fcInfoV2

`fcMemoryTypeInfoHwSw`

Memory is of the type fcInfoHwSw

See Also

`fcFreeMemory`, **`fcGetErrorText`**, **`fcbGetEnumFlexCards (Obsolete)`**, **`fcbGetEnumFlexCardsV2 (Obsolete)`**, **`fcbGetEnumFlexCardsV3`**, **`fcbGetInfoFlexCard`**, **`fcbReceive`**

4.2.2 fcFreeMemory

This function releases the memory allocated by one of the API functions **`fcGetErrorText`**, **`fcbGetEnumFlexCards (Obsolete)`**, **`fcbGetEnumFlexCardsV2 (Obsolete)`**, **`fcbGetEnumFlexCardsV3`**, **`fcbGetInfoFlexCard`** or **`fcbReceive`**. The allocated memory can be used as long as necessary. If the connection to the FlexCard is closed, all allocated memory blocks must be released with this function.

```
fcError fcFreeMemory(
      const fcMemoryType memoryType,
      void* p
);
```

Parameters

`memoryType`

Type of memory to be released.

`P`

Pointer to the memory to be released.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information. The **fcErrorCode** INVALID_PARAMETER is returned if the memoryType parameter wasn't correct, NULL_PARAMETER if the p parameter is a null pointer.

Example

```
fcError e = fcbOpen(&hFlexCard, flexcardId);
if (0 != e)
{
        char* szErrorText = NULL;
        if (0 == fcGetErrorText(e, &szErrorText))
        {
                // Print the error text and then free up the memory
                PrintErrorText(szErrorText);
                fcFreeMemory(fcMemoryTypeString, reinterpret_cast<void*>(szErrorText));
        }
}
```

See Also

fcMemoryType, **fcGetErrorText**, **fcbGetEnumFlexCards (Obsolete)**, **fcbGetEnumFlexCardsV2 (Obsolete)**, **fcbGetEnumFlexCardsV3**, **fcbGetInfoFlexCard**, **fcbReceive**

4.3 Initialization

The following section describes the data structures and features used for initialization.

4.3.1 Type Definitions

4.3.1.1 fcHandle

It defines a handle to a FlexCard object. A handle is returned by the function **fcbOpen** (assuming that a valid FlexCard identifier has been used).

```
Typedef void* fcHandle;
```

4.3.1.2 fcByte

Unsigned 8-bit value.

```
Typedef unsigned char fcByte;
```

4.3.1.3 fcWord

Unsigned 16-bit value.

```
Typedef unsigned short fcWord;
```

4.3.1.4 fcDword

Unsigned 32-bit value.

```
Typedef unsigned int fcDword;
```

4.3.1.5 fcQuad

Unsigned 64-bit value.

```
Typedef unsigned long long fcQuad;
```

4.3.1.6 fcBool

Boolean value.

```
Typedef unsigned char fcBool;
```

4.3.2 Enumerations

4.3.2.1 fcBusType

This enumeration defines the available FlexCard bus types.

```
Typedef enum fcBusType
{
      fcBusTypeFlexRay = 0,
      fcBusTypeCAN,
      fcBusTypeCANFD,
      fcBusTypeEth,

} fcBusType;
```

Members

fcBusTypeFlexRay

The FlexRay bus is selected.

fcBusTypeCAN

The CAN bus is selected.

fcBusTypeCANFD

The CAN-FD bus is selected.

fcBusTypeEth

The ethernet bus is selected.

See Also

fcVersionCC

4.3.2.2 fcCC

This enumeration defines the available FlexCard Communication Controller index depending on the communication bus type.

```
Typedef enum fcCC
{
      fcCC1 = 0x00,
      fcCC2 = 0x01,
      fcCC3 = 0x02,
      fcCC4 = 0x03,
      fcCC5 = 0x04,
      fcCC6 = 0x05,
      fcCC7 = 0x06,
      fcCC8 = 0x07,
} fcCC;
```

Members

fcCC1

The Communication Controller 1 is selected.

fcCC2

The Communication Controller 2 is selected.

fcCC3

The Communication Controller 3 is selected.

fcCC4

The Communication Controller 4 is selected.

fcCC5

The Communication Controller 5 is selected.

fcCC6

The Communication Controller 6 is selected.

fcCC7

The Communication Controller 7 is selected.

fcCC8

The Communication Controller 8 is selected.

4.3.2.3 fcCCType

This enumeration defines the Communication Controller types supported by the API. The parameter *CCType* of the structure **fcVersionCC**, which is returned by the functions **fcbGetEnumFlexCardsV3**, indicates the Communication Controller used by the FlexCard.

```
Typedef enum fcCCType
{
      Undefined = 0,
      FreeScale_FPGA,
      Bosch_E_Ray,
      Bosch_DCAN,
      Bosch_M_CAN,
      Ethernet,
}fcCCType;
```

Members

Undefined
Undefined Communication Controller.

FreeScale_FPGA
Communication controller is a FreeScale FPGA

Bosch_E_Ray
Communication controller is a Bosch E-Ray

Bosch_DCAN
Communication controller is a Bosch DCAN

Bosch_M_CAN
Communication controller is a Bosch M_CAN

Ethernet

Controller is Ethernet controllerSee Also

fcVersionCC, **fcbGetEnumFlexCardsV3**

Remarks

Current FlexCard hardware (FlexCard PMC (II), FlexCard Cyclone II (SE), FlexCard USB-M) supported by the latest driver versions integrate Bosch E-Ray and D-CAN Communication Controllers.

4.3.2.4 fcFlexCardDeviceId

This enumeration defines the different FlexCard types. The driver supports the FlexCard products except FlexCard PXI (aka FlexCard Cyclone II PXI).

```
Typedef enum fcFlexCardDeviceId
{
      fcNoDevice                         = 0,
      fcFlexCardCycloneII                = 5,
      fcFlexCardCycloneIIPXI             = 6,
      fcFlexCardPMC                      = 7,
      fcFlexCardCycloneIISE              = 8,
      fcFlexCardPMCII                    = 9,
      fcFlexDevice_M_0P01                = 10,
      fcFlexXConMidgetBinderV1           = 10,
      fcFlexDevice_M_0P02                = 11,
      fcFlexXConMidgetBinderV2           = 11,
      fcFlexDevice_M_0R01                = 12,
      fcFlexXConMidgetLemo               = 12,
      fcFlexXConExpert                   = 13,
      fcFlexCardUSB_M                    = 14,
      fcFlexCardUSB                      = 14,
      fcFlexXConCompact                  = 15,
      frPCIeNGP                          = 16,
      fcFlexDevice_L_0A01                = 17,
      fcFlexDevice_L_2SOC_0S01           = 18,
      fcFlexDevice_L_2SOC_HDSUB_AS_0S01 = 19,
      fcFlexDevice_L_0A02                = 20,
      fcFlexDevice_L_2SOC_0S02           = 21,
      fcFlexDevice_L_2SOC_HDSUB_AS_0S02 = 22,
      fcFlexDevice_S_0A01                = 23,
      fcFlexCardPXIe3                    = 24,
      fcFlexCardPCIe3                    = 25,
      fcFlexDevice_L_0A03                = 26,
      fcFlexDevice_L_2SOC_0S03           = 27

} fcFlexCardDeviceId;
```

Members

fcNoDevice

No FlexCard device was detected.

fcFlexCardCycloneII

The device identifier of a FlexCard Cyclone II.

fcFlexCardCycloneIIPXI

The device identifier of a FlexCard PXI.

fcFlexCardPMC

The device identifier of a FlexCard PMC / PCI.

fcFlexCardCycloneIISE

The device identifier of a FlexCard Cyclone II SE.

fcFlexCardPMCII

The device identifier of a FlexCard PMC-II.

fcFlexDevice_M_0P01

The device identifier of a FlexDevice-M with black case; Layout Version 1.1

fcFlexXConMidgetBinderV1

The device identifier of a FlexXCon Midget Binder V1.

fcFlexDevice_M_0P02

FlexDevice-M with grey case; Layout Version 2.x; Lemo connectors.

fcFlexXConMidgetBinderV2

The device identifier of a FlexXCon Midget Binder V2.

fcFlexDevice_M_0R01

The device identifier of a FlexDevice-M with grey case; Layout Version 2.x; Lemo connectors.

fcFlexXConMidgetLemo

The device identifier of a FlexXCon Midget Lemo.

fcFlexXConExpert

The device identifier of a FlexXCon Expert.

fcFlexCardUSB_M

The device identifier of a FlexCard USB-M.

fcFlexCardUSB
The device identifier of a FlexCard USB.

fcFlexXConCompact
A ‘FlexXCon Compact’ was detected.

frPCIeNGP
A ‘PCIeNGP’ was detected.

fcFlexDevice_L_0A01
A ‘FlexDevice-L 0A01’ Revision 1 was detected.

fcFlexDevice_L_2SOC_0S01
A ‘FlexDevice-L²’ Revision 1 was detected.

fcFlexDevice_L_2SOC_HDSUB_AS_0S01
A ‘FlexDevice-L²’ Revision 1 with HDSUB Connectors was detected.

fcFlexDevice_L_0A02
A ‘FlexDevice-L’ Revision 2 was detected.

fcFlexDevice_L_2SOC_0S02
A ‘FlexDevice-L²’ Revision 2 was detected.

fcFlexDevice_L_2SOC_HDSUB_AS_0S02
A ‘FlexDevice-L²’ Revision 2 with HDSUB Connectors was detected.

fcFlexDevice_S_0A01
A ‘FlexDevice-S 0A01’ was detected.

fcFlexCardPXIe3
A ‘FlexCard PXIe3’ was detected.

fcFlexCardPCIe3
A ‘FlexCard PCIe3’ was detected.

fcFlexDevice_L_0A03
A ‘FlexDevice-L 0A03-01’ Revision 3 was detected.

fcFlexDevice_L_2SOC_0S03
A ‘FlexDevice-L² 0S03-01’ Revision 3 was detected.

See Also

fcInfoHw, **fcbGetEnumFlexCardsV3**, **fcbGetInfoFlexCard**

4.3.2.5 fcTinyType

This enumeration defines the FlexCard FlexTiny types. The FlexTiny modules are small PCBs that contain physical layer components.

```
Typedef enum fcTinyType
{
       fcTinyTypeUnknown = 0,
       fcTinyTypeFlexRay,
       fcTinyTypeCAN,
       fcTinyTypeEthernet,
       fcTinyTypeRS232,
       fcTinyTypeCAN_LS,
       fcTinyTypeCAN_HS,
       fcTinyTypeLIN,
       fcTinyTypeK_Line,
       fcTinyTypeFlexRayIso,
       fcTinyTypeCAN_HS_Iso,
       fcTinyTypeCAN_LS_Iso,
       fcTinyTypeLIN_Iso,
       fcTinyTypeRS232_Iso,
       fcTinyTypeCAN_FD,
       fcTinyTypeCAN_FD_Iso,
       fcTinyTypeUSB,
       fcTinyTypeBroadR_Reach,
       fcTinyType100BASE_T1 = fcTinyTypeBroadR_Reach
}fcTinyType;
```

Members

fcTinyTypeUnknown

fcTinyTypeFlexRay

fcTinyTypeCAN

fcTinyTypeEthernet

fcTinyTypeRS232

fcTinyTypeCAN_LS

fcTinyTypeCAN_HS

fcTinyTypeLIN

fcTinyTypeK_Line

fcTinyTypeFlexRayIso

fcTinyTypeCAN_HS_Iso

fcTinyTypeCAN_LS_Iso

fcTinyTypeLIN_Iso

fcTinyTypeRS232_Iso

fcTinyTypeCAN_FD

fcTinyTypeCAN_FD_Iso

fcTinyTypeUSB

fcTinyTypeBroadR_Reach

The bus type is BroadR-Reach (Automotive Ethernet)

fcTinyType100BASE_T1

The bus type is 100BASE-T1 (Automotive Ethernet)

See Also

4.3.2.6 fcConnector

This enumeration defines the different connector mappings for the communication controllers.

```
Typedef enum fcConnector
{
      fcConnectorNotSupported,

      fcConnector1A,
      fcConnector1B,
      fcConnector1AB,
      fcConnector1C,
      fcConnector1D,
      fcConnector1E,
      fcConnector1F,
      fcConnector1G,
      fcConnector1H,

      fcConnector2A,
      fcConnector2B,
      fcConnector2AB,
      fcConnector2C,
      fcConnector2D,
      fcConnector2E,
      fcConnector2F,
      fcConnector2G,
      fcConnector2H,

      fcConnector3A,
      fcConnector3B,
      fcConnector3AB,
      fcConnector3C,
      fcConnector3D,
      fcConnector3E,
      fcConnector3F,
      fcConnector3G,
      fcConnector3H,

      fcConnector4A,
      fcConnector4B,
      fcConnector4AB,
      fcConnector4C,
      fcConnector4D,
      fcConnector4E,
      fcConnector4F,
      fcConnector4G,
      fcConnector4H,

      fcConnector5A,
      fcConnector5B,
      fcConnector5AB,
      fcConnector5C,
      fcConnector5D,
      fcConnector5E,
      fcConnector5F,
      fcConnector5G,
      fcConnector5H,

}fcConnector;
```

Members

fcConnectorNotSupported

Connector mapping is not supported

fcConnector1A

Controller uses connector 1A

fcConnector1B

Controller uses connector 1B

fcConnector1AB

Controller uses connector 1A and 1B

fcConnector1C

Controller uses connector 1C

fcConnector1D

Controller uses connector 1D

fcConnector1E

Controller uses connector 1E

fcConnector1F

Controller uses connector 1F

fcConnector1G

Controller uses connector 1G

fcConnector1H

Controller uses connector 1H

fcConnector2A

Controller uses connector 2A

fcConnector2B

Controller uses connector 2B

fcConnector2AB

Controller uses connector 2A and 2B

fcConnector2C

Controller uses connector 2C

fcConnector2D

Controller uses connector 2D

fcConnector2E

Controller uses connector 2E

fcConnector2F

Controller uses connector 2F

fcConnector2G

Controller uses connector 2G

fcConnector2H

Controller uses connector 2H

fcConnector3A

Controller uses connector 3A

fcConnector3B

Controller uses connector 3B

fcConnector3AB

Controller uses connector 3A and 3B

fcConnector3C

Controller uses connector 3C

fcConnector3D

Controller uses connector 3D

fcConnector3E

Controller uses connector 3E

fcConnector3F

Controller uses connector 3F

fcConnector3G

Controller uses connector 3G

fcConnector3H

Controller uses connector 3H

fcConnector4A

Controller uses connector 4A

fcConnector4B

Controller uses connector 4B

fcConnector4AB

Controller uses connector 4A and 4B

fcConnector4C

Controller uses connector 4C

fcConnector4D

Controller uses connector 4D

fcConnector4E

Controller uses connector 4E

fcConnector4F

Controller uses connector 4F

fcConnector4G

Controller uses connector 4G

fcConnector4H

Controller uses connector 4H

fcConnector5A

Controller uses connector 5A

fcConnector5B

Controller uses connector 5B

fcConnector5AB

Controller uses connector 5A and 5B

fcConnector5C

Controller uses connector 5C

fcConnector5D

Controller uses connector 5D

fcConnector5E

Controller uses connector 5E

fcConnector5F

Controller uses connector 5F

fcConnector5G

Controller uses connector 5G

fcConnector5H

Controller uses connector 5H

See Also

fcInfoHw, fcVersionCC

Remarks

Only the FlexCard PXIe3, FlexCard PCIe3 and FlexDevice uses the connector mapping.

4.3.3 Structures

4.3.3.1 fcNumberCC

This structure provides information about the available number of Communication Controllers of the FlexCard hardware.

```
Typedef struct fcNumberCC
{
      fcByte FlexRay;
      fcByte CAN;
      fcByte LIN;
      fcByte MOST;
      fcByte FlexRaySelfSync;
      fcByte Ethernet;
      fcByte UartOverCAN;
      fcByte Reserved[1];
} fcNumberCC;
```

Members

`FlexRay`

Number of FlexRay Communication Controllers.

`CAN`

Number of CAN Communication Controllers.

`LIN`

Number of LIN Communication Controllers. This parameter is currently not supported.

`MOST`

Number of MOST Communication Controllers. This parameter is currently not supported.

`FlexRaySelfSync`

Number of FlexRay self sync Communication Controllers.

`Ethernet`

Number of Ethernet controllers.

`UARTOverCAN`

Number of UARTOverCAN controllers.

`Reserved`

Reserved for future use.

See Also

`fcInfoHw`, **`fcbGetNumberCcs`**, **`fcbGetInfoFlexCard`**

	Information
ℹ	This structure is initially supported by FlexCard API version S4V0-F. The parameter *`FlexRaySelfSync`* is initially supported by version S6V2-F. The parameter Ethernet is initially supported by version S6V7-F The parameter UARTOverCAN is initially supported by version S6V8-F

4.3.3.2 fcVersionCC

This structure provides version information about the available FlexCard Communication Controllers.

```
Typedef struct fcVersionCC
{
       fcBusType BusType;
       fcCC CCIndex;
       fcCCType CCType;
       fcVersionNumber CCVersion;
       fcVersionNumber Protocol;
       struct fcVersionCC* pNext;
       fcDword IncorrectPhysicalLayer : 1;
       fcDword FaultTolerantCAN : 1;
       fcConnector ConnectorIndex;
       fcDword Reserved[1];

} fcVersionCC;
```

Members

BusType

Communication controller bus type

CCIndex

Communication controller identifier

CCType

Communication controller type

CCVersion

Communication controller version

Protocol

Protocol version of the bus type

pNext

Pointer to the next CC version. If the pointer is NULL, there are no more CC versions available.

IncorrectPhysicalLayer

Physical layer module detection. A value <> 0 indicates a mismatch between Communication Controller type and physical layer module.

FaultTolerantCAN

Low speed CAN bus detection. A value <> 0 indicates a fault tolerant CAN compatible physical layer module.

ConnectorIndex

The connector which is used for the controller. This value is only valid for FlexCard PXIe3, FlexCard PCIe3 and FlexDevice hardware.

Reserved

Reserved for future use

See Also

fcInfoHw

	Information
ℹ	This structure is initially supported by FlexCard API version S4V0-F. The parameter *IncorrectPhysicalLayer* is initially supported by version S5V1-F. The parameter *FaultTolerantCAN* is initially supported by version S6V2-F. The parameter *ConnectorIndex* is initially supported by version S6V7-F.

4.3.3.3 fcVersionNumber

This structure describes the version of a FlexCard component (hardware or software). The function **fcbGetEnumFlexCardsV3** returns the version numbers for the FlexCard components.

```
Typedef struct fcVersionNumber
{
      fcDword Major;
      fcDword Minor;
      fcDword Update;
      fcDword Release;
} fcVersionNumber;
```

Members

Major

An increment indicates a modification which is not downwardly compatible

Minor

An increment indicates a light-weight modification

Update

Indicates an update (bug fix) for a minor version

Release

0 indicates a release version, greater than 0 indicates a test version

Remarks

Software version numbers are displayed as SmVn-r, with m = major number, n = minor number, r = release number. Released software is displayed with an “F” as release number. In binary values like this struct, the “F” is replaced with a zero. Example: S1V2-F may also be displayed as 1.2.0.0.

See Also

fcInfoHw, **fcInfoSw**, **fcbGetEnumFlexCardsV3**, **fcbGetInfoFlexCard**

4.3.3.4 fcInfoHw

This structure provides information about the hardware components of a FlexCard.

```
Typedef struct fcInfoHw
{
      fcQuad Serial;
      fcFlexCardDeviceId DeviceId;
      fcVersionNumber VersionFirmware;
      fcVersionNumber VersionHardware;
      fcNumberCC SupportedCCs;
      fcNumberCC LicensedCCs;
      fcNumberCC UseableCCs;
      fcVersionCC* pVersionCC;
      fcDword FullCommunicationControllerAccess;
      fcDword Reserved[7];
} fcInfoHw;
```

Members

Serial

FlexCard serial number. A zero value indicates a non-valid FlexCard serial number.

DeviceId

FlexCard Device ID

VersionFirmware

Firmware (gateway software) version

VersionHardware

FlexCard hardware version

SupportedCCs

Possible FlexCard Communication Controller counts with the hardware.

LicensedCCs

Licensed FlexCard Communication Controller counts with the hardware.

UseableCCs

Useable FlexCard Communication Controller counts for the software.

pVersionCC

Pointer to version information about the useable Communication Controllers.

FullCommunicationControllerAccess

Depending on device base software (e.g FPGA image type) the API has limited access to the communication controllers e.g. the FlexCard PXIe3 or FlexCard PCIe3 supports the "FlexDevice Mode" and the " FlexCard Mode". If the FlexDevice Mode is active, the value is set to 0 (the communication controllers are controlled by the restbus simulation / gateway application from the embedded ARM Cortex processor). If the FlexCard Mode is active, the value is set to 1 (the API controls the complete communication controller).

Reserved

Reserved for future use

See Also

fcInfoSw, **fcInfoHwSw**, **fcVersionCC**, **fcbGetEnumFlexCardsV3**, **fcbGetInfoFlexCard**

	Information
ℹ	This structure is initially supported by FlexCard API version S5V1-F.

4.3.3.5 fcInfoSw

This structure provides information about the software components of a FlexCard. For correct operation, the base driver, the device driver and the firmware should have the same major version number. An exception is the FlexCard USB-M, where the base driver and the device driver have completely different versions. Refer to the document FlexCard USB-M Instructions for Use to find out what versions should be installed. You may use the function **fcbCheckVersion** to ensure correct component versions.

```
Typedef struct fcInfoSw
{
      fcVersionNumber VersionBaseDll;
      fcVersionNumber VersionDeviceDriver;
      fcDword LicensedForLinuxDriver : 1;
      fcDword LicensedForWindowsDriver : 1;
      fcDword LicensedForXenomaiDriver : 1;
      fcDword LicensedForLabviewDriver : 1;
      fcDword Reserved[4];
} fcInfoSw;
```

Members

VersionBaseDll

DLL Base Version.

VersionDeviceDriver

Device driver version.

LicensedForLinuxDriver

Valid license for FlexCard Linux driver.

LicensedForWindowsDriver

Valid license for FlexCard Windows driver.

LicensedForXenomaiDriver

Valid license for FlexCard Xenomai driver.

LicensedForLabviewDriver

Valid license for FlexCard Labview driver.

Reserved

Reserved for future use.

See Also

fcInfoHw, **fcInfoHwSw**, **fcbGetEnumFlexCardsV3**, **fcbGetInfoFlexCard**

ℹ	**Information**
	This structure is initially supported by FlexCard API version S5V1-F. The parameter *LicensedForLabviewDriver* is initially supported by version S6V1-F.

4.3.3.6 fcInfoHwSw

This structure provides information about the components, the identifiers and the current device state of a FlexCard. If more than one FlexCard is detected on the system, the **fcbGetEnumFlexCardsV3** function returns a linked list of this structure; the function **fcbGetInfoFlexCard** function returns an item of this structure.

```
Typedef struct fcInfoHwSw
{
      fcDword FlexCardId;
      fcDword UserDefinedCardId;
      fcInfoSw InfoSoftware;
      fcInfoHw InfoHardware;
      fcDword Busy : 1;
      struct fcInfoHwSw* pNext;
      fcDword Reserved[2];
} fcInfoHwSw;
```

Members

FlexCardId

Unique number used to identify a FlexCard. This id is required to open a connection to the FlexCard.

UserDefinedCardId

User defined number used to identify a FlexCard. This id is not unique! A zero value indicates a non-valid or non-existing identifier.

InfoSoftware

Information about software components of the FlexCard.

InfoHardware

Information about hardware components of the FlexCard.

Busy

The current device state. A value <> 0 indicates a connection to this FlexCard is already opened.

pNext

Pointer to the next available FlexCard. If no other FlexCard exists, *pNext* is a null pointer.

Reserved

Reserved for future use.

See Also

fcInfoHw, **fcInfoSw**, **fcbGetEnumFlexCardsV3**, **fcbGetInfoFlexCard**

ℹ	**Information**
	This structure is initially supported by FlexCard API version S5V1-F.

4.3.3.7 fcTinyInfo

This structure contains information about a FlexTiny.

```
Typedef struct fcTinyInfo
{
      fcTinyType TinyType;
      fcDword Reserved;
} fcTinyInfo;
```

Members

TinyType

The type of the FlexTiny.

Reserved

Reserved for future use.

See Also

ℹ	Information
	This structure is initially supported by FlexCard API version S6V6-F.

4.3.3.8 fcTinyInfoCollection

This structure contains information about all the FlexTiny modules that are mounted on a FlexCard.

```
Typedef struct fcTinyInfoCollection
{
      fcTinyInfo info[255];
} fcTinyInfoCollection;
```

Members

info

Information about the mounted FlexTiny modules.

See Also

ℹ	Information
	This structure is initially supported by FlexCard API version S6V6-F.

4.3.4 fcbGetEnumFlexCardsV3

This function returns a linked list of the installed FlexCards found on the system. To free the memory, which was allocated by this function, please use the function **fcFreeMemory** with type **fcMemoryTypeInfoHwSw.**

```
fcError fcbGetEnumFlexCardsV3(
      fcInfoHwSw** pInfoHwSw,
      fcBool getBusyDevices
)
```

Parameters

pInfoHwSw

[OUT] linked list of fcInfoHwSw objects

getBusyDevices

[IN] Show busy devices in linked list. Set this parameter to 0 to get a linked list of the unused FlexCards found on the system.

Return values

If the function succeeds, the return value is 0. If the function fails the content of *pInfoHwSw* is not valid. The error code NULL_PARAMETER is returned if *pInfoHwSw* parameter is a null pointer. If the memory allocation fails, the error code MEMORY_ALLOCATION_FAILED is returned.

Remarks

If the function succeeds, there will always be one valid fcInfoHwSw structure regardless if there is a FlexCard in the system or not. This functionality is given to provide version information about the DLL/library. The version information concerning the hardware is only valid if the identifier (***pInfoHwSw->FlexCardId***) is not 0.

ℹ	**Information**
	This function allocates memory for use. To prevent memory leaks, you have to free it up by calling the function **`fcFreeMemory`** with the type *`fcMemoryTypeInfoHwSw`*.

ℹ	**Information**
	Restriction: FlexCard USB-M devices are only enumerated by this function, if it’s opened with the calling application. Devices opened in other applications are not returned in the linked list.

See Also

`fcInfoHwSw`

Example

```
//
// Get the installed FlexCards in the system and print the version numbers
//
fcInfoHwSw* pInfoHwSw = NULL;
fcError e = fcbGetEnumFlexCardsV3(&pInfoHwSw, true);
if (0 != e) return; // if it fails, return directly

fcInfoHwSw* pLoop = pInfoHwSw;
while (NULL != pLoop)
{
      // if FlexCard ID is equal to zero, we got NO FlexCard in the system
      bool bFlexCardAvailable = (0 != pLoop->FlexCardId);

      printf(“\r\nFlexCard ID\t:  “);
      if (bFlexCardAvailable) printf(“%d\r\n”,pLoop->FlexCardId);
      else printf(“not available\r\n”);

      // if FlexCard isn’t in use, we print out the version numbers
      if (bFlexCardAvailable && (0 == pLoop->Busy))
      {/*... print out the version numbers ...*/}
      else printf(“FlexCard is in use\r\n”);

      pLoop = pLoop->pNext; // get the next flexcard
}

// Don’t forget to free the memory
fcFreeMemory(fcMemoryTypeInfoHwSw, pInfoHwSw);
```

ℹ	**Information**
	This function is initially supported by FlexCard API version S5V1-F.

4.3.5 fcbCheckVersion

This function checks the version combination of the installed FlexCard driver and firmware. On Windows the files *fcBase.DLL*, *fce05xp.SYS* / *fce052k.SYS and* and *FCFTBUS.sys* are checked. On Linux, *flexcard.ko* and *libfcBase.so* are checked. This function can only be called after **`fcbOpen`**. The major version number of the files must be identical when you want to use the FlexCard. An exception is the FlexCard USB-M: With this device, the major driver version may differ. Refer to the document FlexCard USB-M Instructions for Use to find out what versions should be installed.

```
fcError fcbCheckVersion(
      fcHandle hFlexCard
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard.

Return values

If the function succeeds, the return value is 0 and the opened FlexCard can be used with the SYS and DLL. If the return value is <> 0, use the functions described in the section Error Handling to get extended error information and the opened FlexCard cannot be used because of incompatible SYS and DLL versions.

Example

```
fcError e = fcbCheckVersion(hFlexCard);
if (0 != e)
{
     fcbClose(hFlexCard);
     // Error handling
}
```

ℹ	Information
	This function is initially supported by FlexCard API version S4V0-F.

4.3.6 fcbOpen

This function opens a connection to a specified FlexCard and returns a handle to this FlexCard. The function modifies some Communication Controller registers (e.g. set the Communication Controller in its configuration state, *fcStateConfig*) and all message buffers are configured as receive FIFO buffers with maximum payload length..

```
fcError fcbOpen(
      fcHandle* phFlexCard,
      fcDword flexCardId
)
```

Parameters

`phFlexCard`

[OUT] Handle to a specific FlexCard.

`flexCardId`

[IN] Number which indicates the FlexCard you want to use. This identifier is stored in **`fcInfoHwSw`** objects returned by the function **`fcbGetEnumFlexCardsV3`**. Only `FlexCardId` greater than zero are valid FlexCard identifier.

Return values

If the function succeeds, `phFlexCard` holds a valid FlexCard handle and the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

Use the functions **`fcbGetEnumFlexCardsV3`** to get a valid `FlexCardId`. The function **`fcbClose`** is used to close a connection previously opened with **`fcbOpen`**.

Information

If the FlexCard is closed and reopened, all previous (before closing) configuration settings are lost. After opening a connection, it is necessary to configure the FlexCard.

See Also

`fcbGetEnumFlexCardsV3`, **`fcbClose`**

Example

```
...
fcInfoHwSw* pInfoHwSw = NULL;
fcHandle hFlexCard = NULL;

if (0 == fcbGetEnumFlexCardsV3(&pInfoHwSw, true))
{
      // Open the flexcard using the first flexcard identifier
      fcError e = fcbOpen(&hFlexCard, pInfoHwSw ->FlexCardId);

      // always free the memory which was allocated by fcbGetEnumFlexCardsV3
      fcFreeMemory(fcMemoryTypeInfoHwSw, pInfoHwSw);
      if (0 != e) // handle isn’t valid
            printErrorI;
}
...
```

4.3.7 fcbClose

This function closes the connection to a FlexCard..

```
fcError fcbClose(
      fcHandle hFlexCard
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

If a monitoring is active, this function will first stop the monitoring and then close the connection.

See Also

`fcbGetEnumFlexCardsV3`, **`fcbOpen`**

Example

```
fcError e = fcbClose(hFlexCard);
if (0 == e)
{
      // This handle isn’t valid anymore
      hFlexCard = NULL;
}
```

4.3.8 fcbGetInfoFlexCard

This function returns an item of the structure **`fcInfoHwSw`**, which provides information about the components, the identifiers and the current device state of the opened FlexCard device. The pointer pNext in the struct fcInfoHwSw is empty. To free the memory which was allocated by this function, please use the function **fcFreeMemory** with type *fcMemoryTypeInfoHwSw*.

```
fcError fcbGetInfoFlexCard(
      fcHandle hFlexCard,
      fcInfoHwSw** pInfoHwSw
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

pInfoHwSw

[OUT] Hardware and software information of a FlexCard.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

ℹ	**Information**
	This function allocates memory for use. To prevent memory leaks, you have to free it up by calling the function **fcFreeMemory** with the type *fcMemoryTypeInfoHwSw*.

See Also

fcbGetEnumFlexCardsV3, **fcbOpen**, **fcInfoHwSw**

Example

```
...
fcInfoHwSw* pInfoHwSw = NULL;
fcError e = fcbGetInfoFlexCard(hFlexCard, &pInfoHwSw);
if (0 == e)
{
      // Check open device
      …

      // always free the memory which was allocated by fcbGetInfoFlexCard
      fcFreeMemory(fcMemoryTypeInfoHwSw, pInfoHwSw);
      if (0 != e) // handle isn't valid
            printErrorI;
}
...
```

ℹ	**Information**
	This function is initially supported by FlexCard API version S5V1-F.

4.3.9 fcbSetUserDefinedCardId

This function writes a persistent user ID to the FlexCard. The ID stays the same even if the device is powered off and on.

```
fcError fcbSetUserDefinedCardId (
      fcHandle hFlexCard,
      fcDword UserDefinedCardId
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

UserDefinedCardId

[IN] The ID that will be given to the FlexCard.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbGetUserDefinedCardId

Example

```
fcDword UserDefinedCardId = 0xef000001;
fcError e = fcbSetUserDefinedCardId (hFlexCard, UserDefinedCardId);
if (0 != e)
{
// error handling
}
```

	Information
ℹ	This function is initially supported by FlexCard API version S5V1-F.

4.3.10 fcbGetUserDefinedCardId

This function reads the persistent ID from the FlexCard. The ID stays the same even if the device is powered off and on.

```
fcError fcbGetUserDefinedCardId (
      fcHandle hFlexCard,
      fcDword* pUserDefinedCardId
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

UserDefinedCardId

[OUT] The user defined FlexCard ID.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbSetUserDefinedCardId

Example

```
fcDword UserDefinedCardId = 0x0;
fcError e = fcbGetUserDefinedCardId (hFlexCard, &UserDefinedCardId);
if (0 != e)
{
      // error handling
}
else
{
      printf(“FlexCard UserID: 0x%X“, UserDefinedCardId);
}
```

ℹ	Information
	This function is initially supported by FlexCard API version S5V1-F.

4.3.11 fcbGetTinyInfo

Gets information about the FlexTiny modules mounted on the FlexCard. Depending on the hardware type this is the FlexTiny II or FlexTiny III type.

```
fcError fcbGetTinyInfo (
       fcHandle hFlexCard,
       fcTinyInfoCollection* pTinyInfo,
       fcByte* pNumberOfTinySlots
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

pTinyInfo

[OUT] This struct contains the FlexTiny information.

pNumberOfTinySlots

[OUT] The number of FlexTiny slots that are available on the hardware. The caller must provide memory for this parameter.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcTinyInfoCollection

Example

```
fcTinyInfoCollection tinyCollection;
memset(&tinyCollection, 0, sizeof(tinyCollection));
fcByte numberOfTinySlots;
fcError error = fcbGetTinyInfo(handle, &tinyCollection, &numberOfTinySlots);
if (0 == error)
{
    for(int i = 0; i < numberOfTinySlots; i++)
    {
        printf(“Tiny index: %u, type: %u\n”, i, tinyCollection.info[i].TinyType);
    }
}
```

ℹ	Information
	This function is **not** available for FlexCard Cyclone II (SE) and FlexCard USB-M. This function is initially supported by FlexCard API version S6V6-F.

4.3.12 fcbSetGlobalConfig

Sets global config parameters which can be used by the API during its livetime (only optional!). Global parameters should always be set first. Before even starting initialization with fcbGetEnumFlexCardsV3. All global parameters consist of a key and a value, pass both to the function to set a parameter. For each parameter the function must be called. If this function is not called, default parameters are used.

```
fcError fcbSetGlobalConfig(
        const char* key,
        char* value
)
```

Parameters

key

[IN] Describes key of config parameter. (terminated c-string!)

value

[IN] Describes value of config parameter. (terminated c-string!)

List of Global Parameters

Possible key-value pairs that together form a global parameter are listed below:

Key	Value
„HwComAddress“	Unicast or Broadcast address for finding FlexDevices. (e.g. “192.168.1.15” or “192.168.1.255”)
„TargetIPv4“	Unicast or Multicast address where FlexDevices sends analyzing data to. Most of the time the current PC. (e.g. “192.168.1.10”)
“AnalyzerConfigVersion“	Version oft the Analyzer Config Command (1 byte)(HwCom Analyzing)
“DataInterval“	Maximum Interval of Container Messages. (2 byte)(HwCom Analyzing)
“StatisticInterval“	Interval of Statistic Messages (2 byte)(HwCom Analyzing)
”ThresholdSize“	Maximum size of Container Payload (2 byte)(HwCom Analyzing)
“TransportProtocol“	Used Transport Protocol (1 byte)(HwCom Analyzing)
“ProtocolFormat“	Used Protocol Format (1 byte)(HwCom Analyzing)
“TargetPort“	Target Port (2 byte)(HwCom Analyzing)
“TargetType“	Target Type Unicast/Multicast (2 byte)(HwCom Analyzing)

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
fcError error = fcbSetGlobalConfig(“hwComAddress”, “192.168.2.255”);
if(error != 0) return;
error = fcbSetGlobalConfig(“TargetIPv4”, “192.168.2.10”);
if(error != 0) return;
error = fcbSetGlobalConfig(“TransportProtocol”, “2”); // UDP
if(error != 0) return;
```

ℹ	Information
	This function is initially supported by FlexCard API version S6V7-F.

4.4 Configuration

4.4.1 Enumerations

4.4.1.1 fcTimeStampSourceMode

This enumeration defines the modes available for the time stamp clock source configuration of the FlexCard hardware.

```
Typedef enum fcTimeStampSourceMode
{
	fcTimeStampModeDefault = 0,
	fcTimeStampModeTriggerIn,
	fcTimeStampModeUserIo,
	fcTimeStampModePxiClock10,
} fcTimeStampSourceMode;
```

Members

fcTimeStampModeDefault

The internal 1 MHz timestamp clock is used which results in a resolution of 1 µs.

fcTimeStampModeTriggerIn

The trigger in line is used to generate the time stamp. Time stamp resolution depends on the frequency at the configured trigger line. For example, if the external clock is1 MHz, 10 MHz or 100 MHz, the timestamp is in units of 1 µs, 100 ns or 10 ns.

fcTimeStampModeUserIo

For FlexCard PMC-II. The PMC UserIo lines are used to control the FlexCard timestamp. UserIo pin 63 is used to increment the timestamp and UserIo pin 64 is used to reset it. For example, if the external clock is1 MHz, 10 MHz or 100 MHz, the timestamp is in units of 1 µs, 100 ns or 10 ns.

fcTimeStampModePxiClock10

For FlexCard PXIe3. PXI CLOCK 10 is used to increment the timestamp and PXI TRIGGER 0 is used to reset the timestamp value. The device only reacts to PXI TRIGGER 0 if the timestamp source is set to this value. The timestamp value is synchronized to PXI CLOCK 10 and increments with microsecond resolution. That means, after ten pxi clock ticks the FlexCard timestamp makes one tick. PXI TRIGGER 0 is a 0..5V signal with a pullup resistor on the pxi rack backplane. When the signal is 0, the FlexCard timestamp is zero. After a positive edge on the signal, the FlexCard timestamp starts to increment.

See Also

fcTimeStampCfg

	Information
ℹ	With fcTimeStampModeTriggerIn and fcTimeStampModeUserIo the FlexCards time stamp doesn't represent a real time. It's a clock counter value. All received packets (member TimeStamp) and the FlexCards time stamp itself (**fcbGetCurrentTimeStamp**, **fcbGetCurrentHighResTimeStamp**) must be translated to the expected time by the user application.

	Information
ℹ	This enumeration is initially supported by FlexCard API version S6V2-F.

4.4.2 Structures

4.4.2.1 fcTimeStampCfg

This structure defines the time stamp configuration of the FlexCard. Default configuration uses the internal FlexCard time stamp (1 µs resolution).

```
Typedef struct fcTimeStampCfg
{
       fcTimeStampSourceMode mode;
       fcDword Reserved1;
       union
       {
       fcDword TriggerLine;
       fcDword Reserved[2];
       } AdditionalCfg;
       fcDword Reserved2[4];
} fcTimeStampCfg;
```

Members

mode

Time stamp clock source mode

Reserved1

Reserved for future use

AdditionalCfg

- *TriggerLine*

  For FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3. The trigger line number used for external time stamp generation. Valid values range from 1 to 2.

- *Reserved*

  Reserved for architecture compatibility

Reserved2

Reserved for future use

See Also

fcTimeStampSourceMode

	Information
	This structure is initially supported by FlexCard API version S6V2-F.

4.4.3 fcbReinitializeCcMessageBuffer

This function re-initializes the message buffer configuration of the specified bus type and Communication Controller index. After calling this function the Communication Controller does not send old payload data. Re-initialization of message buffers is only allowed if the Communication Controller is in configuration state. Currently this function only supports the bus type *fcBusTypeFlexRay*.

```
fcError fcbReinitializeCcMessageBuffer(
       fcHandle hFlexCard,
       fcBusType BusType,
       fcCC CC
)
```

Paramaters

hFlexCard

[IN] Handle to a FlexCard

BusType

[IN] The bus type.

CC

[IN] Index of the Communication Controller.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
// FlexRay network running, FlexCard is sending data

// monitoring is stopped, e.g. because of user interaction
fcbFRMonitoringStop(hFlexCard,eCC);

fcError e = fcbReinitializeCcMessageBuffer(hFlexCard, fcBusTypeFlexRay, eCC);
if (0 == e)
{
      // error handling
}

// fcbReinitializeCcMessageBuffer ist not needed when calling
//    fcbFRSetCcConfiguration, fcbFRSetCcConfigurationChi or
//    fcbFRConfigureMessageBuffer before MonitoringStart

fcError e = fcbFRMonitoringStart(hFlexCard,eCC,fcMonitoringNormal,true,
                                  false,false,false);
if (0 == e)
{
      // error handling
}

// now the CC does not send data from the previous monitoring
```

ℹ	Information
	This function is initially supported by FlexCard API version S4V0-F.

4.4.4 fcbGetNumberCcs

This function reads the number of the various Communication Controllers which are available on the FlexCard.

```
fcError fcbGetNumberCcs(
      fcHandle hFlexCard,
      fcNumberCC* pNumberCC
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

pNumberCC

[OUT] Pointer to the structure of the available Communication Controller numbers.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
fcNumberCC numberCC;
fcError e = fcbGetNumberCcs(hFlexCard, &numberCC);
if (0 == e)
{
      printf("Communication controllers: FlexRay: %d, FlexRay SelfSync: %d, CAN: %d,
LIN:  %d,  MOST:  %d”,  numberCC.FlexRay,  numberCC.FlexRaySelfSync,  numberCC.CAN,
numberCC.LIN, numberCC.MOST);
}
```

See Also

fcNumberCC

ℹ	Information
	This function is initially supported by FlexCard API version S4V0-F.

4.4.5 fcbSetContinueOnPacketOverflow

This function configures the packet overflow handling of the FlexCard. The FlexCards default behavior is to stop the monitoring if a data overflow was detected. This is the case, when the application receives the data too slowly. This function can configure the FlexCard to continue with the monitoring when an amount of free RAM space is available again. An error packet fcErrFlexCardOverflow is generated in both cases.

```
fcError fcbSetContinueOnPacketOverflow(
      fcHandle hFlexCard,
      fcBool bContinue
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

bContinue

[IN] Set this flag to <> 0 to continue the monitoring in case of a packet buffer overflow being detected when RAM space is available again. Set to 0 to stop the monitoring.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
// Configure the FlexCard to continue on a data overflow
fcError e = fcbSetContinueOnPacketOverflow (hFlexCard, true);
if (0 == e)
{
      printf(“FlexCard will continue receiving after a data overflow.”);
}
```

ℹ	Information
	This function is initially supported by FlexCard API version S4V0-F.

4.4.6 fcbGetCurrentTimeStamp

This function returns the current time stamp of the FlexCard device and the correlated performance counter value of the operating system. The unit of the timestamp depends on the timestamp configuration that is done with fcbConfigureFlexCardTimeStamp. Details about 32/64 bit timestamps and timestamp units can be found there. fcbGetCurrentTimeStamp returns the 32 bit hardware timestamp.

```
fcError fcbGetCurrentTimeStamp(
      fcHandle hFlexCard,
      fcDword* pTimeStamp,
      fcQuad* pPerformanceCounter
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

pTimeStamp

[OUT] Current 32 bit time stamp

pPerformanceCounter

[OUT] Correlated performance counter

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcQuad

ℹ	Information
	This function is initially supported by FlexCard API version S4V0-F.

4.4.7 fcbResetTimestamp

This function sets the FlexCard timestamp to 0.

```
fcError fcbResetTimestamp (
      fcHandle hFlexCard
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
fcError e = fcbResetTimestamp(hFlexCard);
if (0 == e)
{
      printf(“Timestamp was reset.”);
}
```

ℹ	Information
	This function is initially supported by FlexCard API version S4V0-F.

4.4.8 fcbConfigureFlexCardTimeStamp

This function configures the FlexCards time stamp. By default the FlexCard uses an internal clock (1 MHz) to generate a time stamp with 1 µs resolution. This function cannot be used with FlexCard Cyclone II (SE) devices.

The FlexCard Firmware runs with a 64 bit timestamp. Certain API functions return the full 64 bit timestamp, others return only the lower 32 bits of the 64 bit timestamp. fcbGetCurrentTimeStamp and most of the packets returned by fcbReceive (member TimeStamp) contain the 32 bit timestamp. fcbGetCurrentHighResTimeStamp and some packets returned by fcbReceive return the 64 bit timestamp.

See the description of fcTimeStampCfg and fcTimeStampSourceMode for details.

```
fcError fcbConfigureFlexCardTimeStamp(
      fcHandle hFlexCard,
      fcTimeStampCfg cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

cfg

[IN] The time stamp configuration

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcTimeStampSourceMode, **fcTimeStampCfg**

ℹ	**Information**
	This function is initially supported by FlexCard API version S6V2-F.

4.4.9 fcbGetCurrentHighResTimeStamp

This function returns the current high resolution time stamp of the FlexCard device and the correlated performance counter value of the operating system. The unit of the timestamp depends on the timestamp configuration that is done with fcbConfigureFlexCardTimeStamp. Details about 32/64 bit timestamps and timestamp units can be found there. fcbGetCurrentHighResTimeStamp returns the 64 bit hardware timestamp.

```
fcError fcbGetCurrentHighResTimeStamp(
      fcHandle hFlexCard,
      fcQuad* pTimeStamp,
      fcQuad* pPerformanceCounter
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

pTimeStamp

[OUT] Current 64 bit time stamp

pPerformanceCounter

[OUT] Correlated performance counter

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcQuad

	Information
ℹ	This function is initially supported by FlexCard API version S6V2-F.

4.5 Event

4.5.1 Enumerations

4.5.1.1 fcNotificationType

This enumeration defines different notification types. These types are used in the functions **fcbSetEventHandleV2**, **fcbSetEventHandleSemaphore** or **fcbWaitForEventV2** to specify on which kind of event the application has to be notified.

```
Typedef enum fcNotificationType
{
      fcNotificationTypeCycleStarted       = 1,
      fcNotificationTypeFRCycleStarted     = fcNotificationTypeCycleStarted,
      fcNotificationTypeTimer              = 2,
      fcNotificationTypeWakeup             = 3,
      fcNotificationTypeFRWakeup           = fcNotificationTypeWakeup,
      fcNotificationTypeCcTimer            = 12,
      fcNotificationTypeFRCcTimer          = fcNotificationTypeCcTimer,
      fcNotificationTypeSurpriseRemoval    = 13,
      fcNotificationTypeStandby            = 14,
      fcNotificationTypeReceiveBufferLevel = 15,
} fcNotifyType, fcNotificationType;
```

Members

fcNotificationTypeCycleStarted
fcNotificationTypeFRCycleStarted

Used to notify that a new cycle has started and that probably new data has been received.

fcNotificationTypeTimer

Used to notify that the timer interval has elapsed. This notification requires the internal timer of the FlexCard to be enabled (See **fcbSetTimer**).

fcNotificationTypeWakeup
fcNotificationTypeFRWakeup

Used to notify that one of the transceivers has received a wake-up event (only if one of the transceivers was in sleep mode).

fcNotificationTypeCcTimer
fcNotificationTypeFRCcTimer

Used to notify that the configured CC timer macrotick offset has elapsed. This notification requires the E-Ray CC Timer0 to be enabled (See **fcbFRSetCcTimerConfig**).

fcNotificationTypeSurpriseRemoval
fcNotificationTypeStandby

For internal use only.

fcNotificationTypeReceiveBufferLevel

Used to notify that the configured FlexCard receive buffer filling level has reached (See **fcbSetReceiveBufferLevelNotification**).

See Also

fcbFRMonitoringStart, **fcbSetEventHandleV2**, **fcbSetEventHandleSemaphore**, **fcbSetTimer**, **fcbFRSetCcTimerConfig**, **fcbWaitForEventV2**, **fcbSetReceiveBufferLevelNotification**

4.5.2 fcbSetEventHandleV2

This function registers an event handle for a specific notification type.

```
fcError fcbSetEventHandleV2(
       fcHandle hFlexCard,
       fcCC CC,
       fcHandle hEvent,
       fcNotificationType type
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

hEvent

[IN] Event handle to be registered to signal when a new cycle starts, a timer interval has elapsed or the FlexCard receive buffer reaches a specific filling level depending on the given *type*.

Type

[IN] The notification type for which the event must be registered.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcNotificationType

Example

```
// Create the event objects
HANDLE hCycleStartEvent = ::CreateEvent(NULL,FALSE,FALSE,NULL);
fcCC eCC = fcCC1;

// Register our event handles
fcbSetEventHandleV2(hFlexCard, eCC, hCycleStartEvent,
   fcNotificationTypeFRCycleStarted);

// ...
// Use the event objects
// ...
```

	Information
ℹ	This function is initially supported by FlexCard API version S4V2-F.

	Information
ℹ	This function is not supported by the *FlexCard USB-M*.

ℹ	**Information**
	Please don't use this function with the FlexCard Linux driver, because it's not async-signal safe. To avoid deadlocks with the API use the function **fcbSetEventHandleSemaphore** instead. On Xenomai, use the function fcbWaitForEventV2.

4.5.3 fcbSetTimer

This function enables or disables the internal FlexCard timer. To become notified when the timer interval has elapsed, an event of type *fcNotificationTypeTimer* has to be registered by the function **fcbSetEventHandleV2**, **fcbSetEventHandleSemaphore** or **fcbWaitForEventV2**.

```
fcError fcbSetTimer(
      fcHandle hFlexCard,
      fcBool enable,
      fcDword timerInterval
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

enable

[IN] Set to <> 0 to enable the timer and to 0 to disable it.

timerInterval

[IN] Specifies the timer period in µs

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcNotificationType, **fcbSetEventHandleV2**, **fcbSetEventHandleSemaphore**, **fcbWaitForEventV2**

Example

```
// Create the event objects
HANDLE hCycleStartEvent = ::CreateEvent(NULL,FALSE,FALSE,NULL);
HANDLE hTimerEvent = ::CreateEvent(NULL,FALSE,FALSE,NULL);
fcCC eCC = fcCC1;

// Register our event handles
fcbSetEventHandleV2(hFlexCard, eCC, hCycleStartEvent,
   fcNotificationTypeCycleStarted);
fcbSetEventHandleV2(hFlexCard, eCC, hTimerEvent, fcNotificationTypeTimer);

// Enable the timer (1ms Interval)
fcbSetTimer(hFlexCard,true,1000);

// ...
// Use the event objects
// ...
```

4.5.4 fcbNotificationPacket

This function generates a notification packet each time the configured timer timeout has elapsed. This timer can be enabled / disabled by this function and the timeout can be set. The notification packets are inserted in the stream and received through the function **fcbReceive**.

```
fcError fcbNotificationPacket(
      fcHandle hFlexCard,
      fcBool enable,
      fcDword timerInterval
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

enable

[IN] Set to <> 0 to enable the timer and to 0 to disable it.

timerInterval

[IN] Specifies the time-out interval, in microseconds. A packet is generated as soon as the time-out has elapsed. The timer interval must be greater than 50µs and smaller than 655350µs. The value must be rounded to 10µs units.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

ⓘ	**Information**
	This function is initially supported by FlexCard API version S2V0-F.

4.5.5 fcbSetReceiveBufferLevelNotification

This function enables or disables the FlexCard receive buffer level notification. To become notified, when the receive buffer reaches the filling level, an event of type *fcNotificationTypeReceiveBufferLevel* has to be registered by one of the functions **fcbSetEventHandleV2**, **fcbSetEventHandleSemaphore** or **fcbWaitForEventV2**.

IMPORTANT: This function currently does not work!

```
fcError fcbSetReceiveBufferLevelNotification(
      fcHandle hFlexCard,
      fcBool enable,
      fcDword percentage
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

enable

[IN] Set to 0 to disable the receive buffer level notification, a value <> 0 enables the notification.

Percentage

[IN] Specifies the percentage filling level of the FlexCard receive buffer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

ℹ	Information
	This function is initially supported by FlexCard API version S6V1-F.

4.6 Receive

4.6.1 Typedefinitions

4.6.1.1 fcInfoPacket

This structure describes an information packet. This packet type informs you about the start of a new cycle. All packets received between two consecutive info packets are part of the current cycle.

```
Typedef struct fcInfoPacket
{
    fcDword CurrentCycle;
    fcDword TimeStamp;
    fcDword RateCorrection : 12;
    fcDword OffsetCorrection : 19;
    fcDword ClockCorrectionFailedCounter : 4;
    fcDword PassiveToActiveCount : 5;
    fcCC    CC;
}fcInfoPacket;
```

Members

CurrentCycle

The current cycle (FlexRay Protocol Specification: vRF!Header!CycleCount)

Timestamp

The FlexCard timestamp (per default 1 µs resolution). The timestamp marks the point in time where the FlexCard detects the internal FlexRay cycle start interrupt. The event fcNotificationTypeFRCycleStarted does not have to be configured to receive the timestamp.

RateCorrection

Rate correction value (two’s complement). Indicates by how many microticks the node’s cycle length should be changed.

OffsetCorrection

Offset correction value (two’ complement). Indicates the number of microticks that are added to the offset correction segment of the network idle time.

ClockCorrectionFailedCounter

FlexRay Protocol Specification: vClockCorrectionFailed.

PassiveToActiveCount

FlexRay Protocol Specification: vAllowPassiveToActive

CC

The FlexCard CC which created this packet. This parameter will always be set to fcCC1 if only one FlexRay CC is available.

Remarks

A timestamp overflow occurs after approximately 4295 seconds.

See Also

fcPacket

4.6.1.2 fcFlexRayFrame

This structure is equivalent to the FlexRay frame described in the FlexRay specification [3].

```
Typedef struct fcFlexRayFrame
{
    fcDword ID : 11;
    fcDword STARTUP : 1;
    fcDword SYNC : 1;
    fcDword NF : 1;
    fcDword PP : 1;
    fcDword R : 1;
    fcDword PayloadLength : 7;
    fcDword CycleCount : 6;
    fcDword HeaderCRC : 11;
    fcWord* pData;

    fcChannel Channel;
    fcDword ValidFrame : 1;
    fcDword SyntaxError : 1;
    fcDword ContentError : 1;
    fcDword SlotBoundaryViolation : 1;
    fcDword AsyncMode : 1;
    fcDword FrameCRC : 24;

    fcDword TimeStamp;
    fcCC    CC;
} fcFlexRayFrame;
```

Members

ID

The frame id defines the slot in which the frame was transmitted.
(FlexRay Protocol Specification: vRF!Header!FrameID)

STARTUP

Indicates if the frame is a start-up frame (=1) or not (=0)
(FlexRay Protocol Specification: vRF!Header!SuFIndicator)

SYNC

Indicates if the frame is a sync frame (=1) or not (=0)
(FlexRay Protocol Specification: vRF!Header!SyFIndicator)

NF

Set to 0, the null frame indicator indicates that *pData* contains no valid data. Set to 1, it indicates that *pData* contains valid data.
(FlexRay Protocol Specification: vRF!Header!NFIndicator)

PP

The payload preamble indicator indicates whether or not an optional vector is contained within the payload segment of the frame transmitted. In the static segment, it indicates the presence of a network management vector at the beginning of the payload. In the dynamic segment it indicates the presence of a message id at the beginning of the payload, (FlexRay Protocol Specification: vRF!Header!PPIndicator).

R

Reserved Bit (FlexRay Protocol Specification: vRF!Header!Reserved)

PayloadLength

Defines the number of 16-bit words contained in *pData*
(FlexRay Protocol Specification: vRF!Header!Length)

CycleCount

The cycle in which the frame was received. (FlexRay Protocol Specification: vRF!Header!CycleCount)

HeaderCRC

The header CRC containing the cyclic redundancy check code is computed over the sync frame indicator, the start-up frame indicator, the frame id and the payload length.(FlexRay Protocol Specification: vRF!Header!HeaderCRC)

pData

The pointer to the payload data. The payload is given in 16-bit words.
(FlexRay Protocol Specification: vRF!Payload)

If Data0 is the first byte that was received and Data1 the second byte received, then the high byte (Bit 8 – 15) of payload[0] contains Data1, the low byte (Bit 0-7) of payload[0] contains Data0, etc.

Parameter data	data[0] (Word 0)		data[1] (Word 1)		...
	High byte	Low byte	High byte	Low byte	...
FlexRay payload segment	Data 1	Data 0	Data 3	Data 2	...

Channel

The channel (A or B) on which the frame was received.
(FlexRay Protocol Specification: vRF!Channel)

ValidFrame

If a valid frame was received, this parameter is set to 1 (FlexRay Protocol Specification: vSS!ValidFrameA or vSS!ValidFrameB depends on Channel - Table 9-2: Slot status interpretation)

SyntaxError

If a syntax error was observed, this parameter is set to 1 (frame is syntactically incorrect). (FlexRay Protocol Specification: vSS!SyntaxErrorA or vSS!SyntaxErrorB depends on Channel)

ContentError

If a content error was observed, this parameter is set to 1 (frame is semantically incorrect). (FlexRay Protocol Specification: vSS!ContentErrorA or vSS!ContentErrorB depends on Channel)

SlotBoundaryViolation

If a slot boundary violation was observed, this parameter is set to 1 (FlexRay Protocol Specification: vSS!BviolationA or vSS!BviolationB depends on Channel)

AsyncMode

If the packet was generated by the asynchronous debug mode, this parameter is set to 1.

FrameCRC

If the packet was generated by the asynchronous debug mode, the FrameCRC contains the cyclic redundancy check code is computed over complete frame. In synchronous monitoring mode, this parameter is not set.

TimeStamp

The FlexCard timestamp (per default 1 µs resolution). FlexCard PMC-II and FlexCard USB-M: The timestamp marks the point in time where the FlexCard detects the transition from the ChannelIdle state to the FlexRay frame header of the received frame.

CC

The FlexCard CC which created this packet. This parameter will always be set to fcCC1 if only one FlexRay CC is available.

See Also

fcPacket, **fcChannel**

ℹ	Information
	The payload length is a multiple of 16-bit words. The payload data is also given in 16-bit words.

4.6.1.3 fcTxAcknowledgePacket

This structure provides information about a transmit acknowledge packet. Transmit acknowledge packets are used to inform the user when a frame is transmitted.

```
Typedef struct fcTxAcknowledgePacket
{
    fcDword BufferId;
    fcDword TimeStamp;
    fcDword CycleCount;

    fcDword ID : 11;
    fcDword STARTUP : 1;
    fcDword SYNC : 1;
    fcDword NF : 1;
    fcDword PP : 1;
    fcDword R : 1;
    fcDword PayloadLength : 7;
    fcDword ValidFrame : 1;
    fcDword SyntaxError : 1;
    fcDword ContentError : 1;
    fcDword HeaderCRC : 11;
    fcWord* pData;
    fcChannel Channel;
    fcCC CC;
} fcTxAcknowledgePacket;
```

Members

BufferId

The buffer id used to transmit the frame (equivalent to the buffer id returned by the function **fcbFRConfigureMessageBuffer**).

TimeStamp

The FlexCard timestamp (per default 1 µs resolution). FlexCard PMC-II and FlexCard USB-M: The timestamp marks the point in time where the FlexCard detects the transition from the ChannelIdle state to the FlexRay frame header of the transmitted frame.

CycleCount

Indicates the cycle in which the frame was transmitted. (FlexRay Protocol Specification: vTF!Header!CycleCount)

ID

The frame id defines the slot in which the frame was transmitted.

STARTUP

Indicates if the frame was a start-up frame (=1) or not (=0)

SYNC

Indicates if the frame was a sync frame (=1) or not (=0)

NF

Set to 0, the null frame indicator indicates that *pData* contains no valid data. Set to 1, it indicates that *pData* contains valid data.

PP

The payload preamble indicator indicates whether or not an optional vector is contained within the payload segment of the frame transmitted. In the static segment, it indicates the presence of a network management vector at the beginning of the payload. In the dynamic segment it indicates the presence of a message id at the beginning of the payload.

R

Reserved Bit

PayloadLength

Defines the number of 16-bit words contain in *pData*

ValidFrame

If a valid frame was received, this parameter is set to 1 (FlexRay Protocol Specification: vSS!ValidFrameA or vSS!ValidFrameB depends on Channel - Table 9-2: Slot status interpretation)

SyntaxError

If a syntax error was observed, this parameter is set to 1 (frame is syntactically incorrect). (FlexRay Protocol Specification: vSS!SyntaxErrorA or vSS!SyntaxErrorB depends on Channel)

ContentError

If a content error was observed, this parameter is set to 1 (frame is semantically incorrect). (FlexRay Protocol Specification: vSS!ContentErrorA or vSS!ContentErrorB depends on Channel)

HeaderCRC

The header CRC contains the cyclic redundancy check code is computed over the sync frame indicator, the start-up frame indicator, the frame id and the payload length.

pData

The pointer to the payload data. The payload is given in 16-bit words.

If Data0 is the first byte that was transmitted and Data1 the second byte transmitted, then the high byte (Bit 8 – 15) of payload[0] contains Data1, the low byte (Bit 0-7) of payload[0] contains Data0, etc.

Parameter data	data[0] (Word 0)		data[1] (Word 1)		...
	High byte	Low byte	High byte	Low byte	...
FlexRay payload segment	Data 1	Data 0	Data 3	Data 2	...

Channel

The channel (A or B) on which the frame was transmitted.
(FlexRay Protocol Specification: vRF!Channel)

CC

The FlexCard CC which created this packet. This parameter will always be set to fcCC1 if only one FlexRay CC is available. This parameter will always be set to fcCC2 for used SelfSync feature packets.

See Also

fcPacket, **fcChannel**

4.6.1.4 fcErrPOCErrorModeChangedInfo

This structure provides additional information about the *fcErrPOCErrorModeChanged* error.

```
Typedef struct fcErrPOCErrorModeChangedInfo
{
    fcState State;
}fcErrPOCErrorModeChangedInfo;
```

Members

State

Contains the new POC error mode (HALT, NORMAL_ACTIVE or NORMAL_PASSIVE)

See Also

fcErrorPacket, **fcState**

4.6.1.5 fcErrSyncFramesInfo

This structure provides additional information about the *fcErrSyncFramesBelowMinimum* and *fcErrSyncFrameOverflow* errors.

```
Typedef struct fcErrSyncFramesInfo
{
    fcDword SyncFramesEvenA : 4;
    fcDword SyncFramesEvenB : 4;
    fcDword SyncFramesOddA : 4;
    fcDword SyncFramesOddB : 4;
}fcErrPOCErrorModeChangedInfo;
```

Members

SyncFramesEvenA

Valid sync frame received and transmitted on channel A in even communication cycles

SyncFramesEvenB

Valid sync frame received and transmitted on channel B in even communication cycles

SyncFramesOddA

Valid sync frame received and transmitted on channel A in odd communication cycles

SyncFramesOddB

Valid sync frame received and transmitted on channel B in odd communication cycles

See Also

fcErrorPacket

4.6.1.6 fcErrClockCorrectionFailureInfo

This structure provides additional information about the *fcErrClockCorrectionFailure* error.

```
Typedef struct fcErrClockCorrectionFailureInfo
{
    fcDword MissingRateCorrection : 1;
    fcDword RateCorrectionLimitReached : 1;

    fcDword OffsetCorrectionLimitReached : 1;
    fcDword MissingOffsetCorrection : 1;

    fcDword SyncFramesEvenA : 4;
    fcDword SyncFramesEvenB : 4;
    fcDword SyncFramesOddA : 4;
    fcDword SyncFramesOddB : 4;
}fcErrClockCorrectionFailureInfo;
```

Members

MissingRateCorrection

Is set to 1 if no rate correction can be performed because no pairs of even/odd sync frames were received.

RateCorrectionLimitReached

Is set to 1 if the maximum rate correction limit is reached.

OffsetCorrectionLimitReached

Is set to 1 if the maximum offset correction limit is reached.

MissingOffsetCorrection

Is set to 1 if no offset correction can be performed because no sync frames were received.

SyncFramesEvenA

Valid sync frame received and transmitted on channel A in even communication cycles

SyncFramesEvenB

Valid sync frame received and transmitted on channel B in even communication cycles

SyncFramesOddA

Valid sync frame received and transmitted on channel A in odd communication cycles

SyncFramesOddB

Valid sync frame received and transmitted on channel B in odd communication cycles

See Also

fcErrorPacket

4.6.1.7 fcErrSlotInfo

This structure provides additional information about the *fcErrSyntax*, *fcErrContent*, *fcErrSlotBoundaryViolation*, *fcErrTransmissionAcrossBoundary*, *fcErrLatestTransmitViolation* fcErrSyntaxSW, *fcErrSlotBoundaryViolationSW*, *fcErrTransmissionConflictSW*, *fcErrSyntaxNIT* and *fcErrSlotBoundaryViolationNIT* errors.

```
Typedef struct fcErrSlotInfo
{
    fcChannel Channel;
    fcDword SlotCount;
}fcErrSlotInfo;
```

Members

Channel

The channel on which the error was observed.

SlotCount

The approximate slot count when the error occurred.

See Also

fcErrorPacket

4.6.1.8 fcErrorPacket

This structure provides information about an error packet.

```
Typedef struct fcErrorPacket
{
    fcErrorPacketFlag Flag;
    fcDword TimeStamp;
    fcDword CycleCount;

    union
    {
         fcErrPOCErrorModeChangedInfo         ErrPOCErrorModeChangedInfo;
         fcErrSyncFramesInfo                  ErrSyncFramesInfo;
         fcErrSlotInfo                        ErrSlotInfo;
         fcErrClockCorrectionFailureInfo      ErrClockCorrectionFailureInfo;
    }AdditionalInfo;
    fcCC CC;

    fcDword Reserved;
}fcErrorPacket;
```

Members

Flag

Error type

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated.

CycleCount

The cycle in which the error occurred.

AdditionalInfo

- *ErrPOCErrorModeChangedInfo*
  Additional information about the fcErrPOCErrorModeChanged error.
- *ErrSyncFramesInfo*
  Additional information about the fcErrSyncFramesBelowMinimum, fcErrSyncFrameOverflow errors
- *ErrSlotInfo*
  Additional information about the fcErrSyntax, fcErrContent, fcErrSlotBoundaryViolation, fcErrTransmissionAcrossBoundary and fcErrLatestTransmitViolation errors
- *ErrClockCorrectionFailureInfo*
  Additional information about the fcErrClockCorrectionFailure error.

CC

The FlexCard CC which created this packet. This parameter will always be set to fcCC1 if only one FlexRay CC is available.

Reserved

Reserved for future use.

See Also

fcPacket, **fcErrorPacketFlag**, **fcErrPOCErrorModeChangedInfo**, **fcErrSyncFramesInfo**, **fcErrSlotInfo**, **fcErrClockCorrectionFailureInfo**

4.6.1.9 fcStatusWakeupInfo

This structure provides additional information about the cStatusWakeupStatus status.

```
Typedef struct fcStatusWakeupInfo
{
    fcWakeupStatus WakeupStatus;
} fcStatusWakeupInfo;
```

Members

WakeupStatus

Current wake-up state.

See Also

fcStatusPacket, **fcWakeupStatus**

4.6.1.10 fcStatusPacket

This structure provides information about a status packet.

```
Typedef struct fcStatusPacket
{
    fcStatusPacketFlag Flag;
    fcDword TimeStamp;
    fcDword CycleCount;

    union
    {
         fcStatusWakeupInfo StatusWakeupInfo;
    }AdditionalInfo;
    fcCC    CC;
    fcDword Reserved[2];
}fcStatusPacket;
```

Members

Flag

Status type

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated.

CycleCount

The cycle in which the status has changed.

AdditionalInfo

StatusWakeupInfo

Additional information about `fcStatusWakeupStatus` status

CC

The FlexCard CC which created this packet. This parameter will always be set to fcCC1 if only one FlexRay CC is available. This parameter will always be set to fcCC2 for used SelfSync feature packets.

Reserved

Reserved for future use.

See Also

fcPacket, **fcStatusPacketFlag**, **fcStatusWakeupInfo**

4.6.1.11 fcNMVectorPacket

This structure provides information about a network management vector. (FlexRay Protocol Specification V2.0: Section 4.3.1 NMVector)

```
typedef struct fcNMVectorPacket
{
    fcDword TimeStamp;
    fcDword CycleCount;
    fcDword NMVectorLength;
    fcByte NMVector[12];
    fcCC CC;
    fcDword Reserved;
} fcNMVectorPacket;
```

Members

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated.

CycleCount

The cycle in which the network management vector was updated.

NMVectorLength

Length of network management vector in number of bytes. (FlexRay Protocol Specification: gNetworkManagementVectorLength)

NMVector

The data bytes of the network management vector.

CC

The FlexCard CC which created this packet. This parameter will always be set to fcCC1 if only one FlexRay CC is available.

Reserved

Reserved for future use.

See Also

fcPacket, **fcCC**

4.6.1.12 fcNotificationPacket

This structure provides information about a notification packet. A notification packet is generated each time the configured time out elapses. The generation of this packet can be controlled with the function **fcbNotificationPacket**.

```
Typedef struct fcNotificationPacket
{
    fcDword TimeStamp;
    fcDword SequenceCounter;
    fcDword Reserved;
} fcNotificationPacket;
```

Members

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated.

SequenceCounter

This parameter is incremented each time a notification packet is generated.

Reserved

See Also

fcPacket, **fcbNotificationPacket**

	Information
ℹ	This packet type is initially supported by FlexCard API version S2V0-F.

4.6.1.13 fcTriggerExInfoPacket

This structure provides information about a trigger packet.

```
Typedef struct fcTriggerExInfoPacket
{
       fcDword Condition;
       fcDword TimeStamp;
       fcDword SequenceCount;
       fcDword Reserved1;
       fcQuad PerformanceCounter;
       fcDword Edge;
       fcDword TriggerLine;
       fcDword reserved[4];
} fcTriggerInfoPacket;
```

Members

`Condition`

The fulfilled condition which has caused the trigger packet generation. For hardware FlexCard Cyclone II SE or FlexCard USB-M, this variable is fcTriggerConditionEx. For hardware FlexCard PMC-II, this variable is fcTriggerConditionPMC.

`TimeStamp`

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated.

`Reserved1`

Reserved for future use.

`SequenceCount`

Sequence count for each signal.

`PerformanceCounter`

Variable that receives the current performance-counter value. This value is only valid when all of the following conditions are met:
-Trigger condition is fcTriggerInOnSWPulse or fcTriggerInOnSWTimer
-The hardware is FlexCard USB-M
-The operating system is Windows

`Edge`

The edge on which the trigger was signalled.

`TriggerLine`

The trigger line which detected a trigger signal. This value is only valid for hardware triggers of FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3.

`Reserved[4]`

Reserved for future use.

See Also

`fcPacket`

	Information
ℹ	This packet type is initially supported by FlexCard API version S2V2-F.

4.6.1.14 fcCANPacket

This structure provides information about a CAN packet.

```
Typedef struct fcCANPacket
{
      fcDword ID           : 29;
      fcDword ExtendedId   : 1;
      fcDword TimeStamp;
      fcDword BufferNumber : 8;
      fcDword DLC          : 4;
      fcDword Direction    : 1;
      fcDword RemoteFrame  : 1;
      fcDword MessageLost  : 1;
      fcDword Reserved;
      fcCC CC;
      fcByte Data[8];
}fcCANPacket;
```

Members

ID

The CAN message identifier which was received or transmitted.

ExtendedId

If this flag is 1 the CAN message is an extended frame. If set to 0 it is a standard frame.

TimeStamp

The CAN timestamp (per default 1 µs resolution). FlexCard PMC-II and FlexCard USB-M: The timestamp marks the point in time where the FlexCard detects the Ack Slot of the frame.

BufferNumber

Indicates the corresponding buffer number for the CAN packet.

DLC

Indicates the data length (in bytes).

Direction

This flag depends on the parameter *RemoteFrame*. If *Direction* is 0 and *RemoteFrame* is 0, the CAN packet is a received data frame. If *Direction* is 1 and *RemoteFrame* is 0 the CAN packet is a transmit acknowledge frame generated by the FlexCard. If *RemoteFrame* is 1, see *RemoteFrame* for further description.

When a FlexCard PXIe3/FlexCard PCIe3 received a frame via fcBase API in the FlexDevice mode, it has the direction set to 0 (the frame was received by the FlexCard).

For FlexDevice the flag tells if the Frame was received (0) or sent (1) by the Device.

RemoteFrame

This flag depends on the parameter *Direction*. If *RemoteFrame* is 1 and *Direction* is 0, the CAN packet is a remote rx frame. If *RemoteFrame* is 1 and *Direction* is 1, the CAN packet is a remote tx frame. If *Direction* is 0, see *Direction* for further description.

MessageLost

If this flag is 1 the CAN Communication Controller has lost a message. If 0 no message has been lost. This flag is only valid with *Direction* = 0.

Reserved

Reserved for future use.

CC

The CAN Communication Controller on which the frame was received or transmitted.

Data

The received or transmitted data. All of the 8 data bytes can be read. The corresponding parameter *DLC* indicates the length of the valid values.

See Also

fcPacket

ℹ	Information
	This packet type is initially supported by FlexCard API version S4V0-F.

4.6.1.15 fcCANErrorPacket

This structure provides information about a CAN error packet.

```
Typedef struct fcCANErrorPacket
{
      fcCANErrorType Type;
      fcCANCcState State;
      fcDword TimeStamp;
      fcDword ReceiveErrorCounter;
      fcDword TransmitErrorCounter;
      fcCC CC;
      fcDword Reserved[2];
}fcCANErrorPacket;
```

Members

Type

Error type

State

Communication controller state

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated.

ReceiveErrorCounter

Actual state of the Receive Error Counter. Valid values range from 0 to 127.

TransmitErrorCounter

Actual state of the Transmit Error Counter. Values values range from 0 to 255.

CC

The CC on which the packet was created.

Reserved[4]

Reserved for future use.

See Also

fcPacket, **fcCANErrorType**, **fcCANCcState**

ℹ	Information
	This packet type is initially supported by FlexCard API version S4V0-F.

4.6.1.16 fcCANFDPacket

This structure provides information about a CAN-FD packet.

```
Typedef struct fcCANFDPacket
{
      fcDword ID           :29;
      fcDword ExtendedId   :1;
      fcDword TimeStampLow;
      fcDword TimeStampHigh;
      fcDword BufferNumber :8;
      fcDword DLC          :4;
      fcDword Direction    :1;
      fcDword RemoteFrame  :1;
      fcDword MessageLost  :1;
      fcDword FdFormat  :1;
      fcDword BitRateSwitch  :1;
      fcDword ESI  :1;
      fcDword Reserved;
      fcCC CC;
      fcByte* pData;
} fcCANFDPacket;
```

Members

ID

The CAN message identifier which was received or transmitted.

ExtendedId

If this flag is 1 the CAN message is an extended frame. If set to 0 it is a standard frame.

TimeStampLow

The CAN timestamp low (per default 1 µs resolution). FlexCard PMC-II and FlexCard USB-M: The timestamp marks the point in time where the FlexCard detects the Ack Slot of the frame.

TimeStampHigh

The CAN timestamp high (per default 1 µs resolution). FlexCard PMC-II and FlexCard USB-M: The timestamp marks the point in time where the FlexCard detects the Ack Slot of the frame.

BufferNumber

Unused field.

DLC

Indicates the data length. It is coded with four bits according to the CAN/ CAN-FD standard.

Direction

If Direction is 0 the CAN packet is a received data frame. If Direction is 1 the packet is a transmit acknowledge frame generated by the FlexCard.

When a FlexCard PXIe3/FlexCard PCIe3 received a frame via fcBase API in the FlexDevice mode, it has the direction set to 0 (the frame was received by the FlexCard).

For FlexDevices the flag tells if the Frame was received (0) or sent (1) by the Device.

RemoteFrame

Unused field.

MessageLost

Unused field.

FdFormat

If this flag is 1 the message has the CAN-FD frame format.

BitRateSwitch

If this flag is 1 the message uses CAN-FD bit rate switching. This bit is only valid when FdFormat is 1.

ESI

If ESI (Error state indicator) is 0, the transmitting node is in the error active state. That means the node is allowed to send error frames. If ESI is 1, the transmitting node is in the error passive state. This bit is only valid when FdFormat is 1.

Reserved

Reserved for future use.

CC

The CAN Communication Controller on which the frame was received or transmitted.

pData

The pointer to the payload data. The payload is given in byte. The corresponding parameter DLC indicates the length of the valid values.

See Also

fcPacket

	Information
ℹ	This packet type is initially supported by FlexCard API version S6V6-F.

4.6.1.17 fcCANFDErrorPacket

This structure provides information about a CAN-FD error packet.

```
Typedef struct fcCANFDErrorPacket
{
      fcCANErrorType Type;
      fcCANCcState State;
      fcDword TimeStampLow;
      fcDword TimeStampHigh;
      fcDword ReceiveErrorCounter;
      fcDword TransmitErrorCounter;
      fcCC CC;
      fcDword Reserved[3];
}fcCANFDErrorPacket;
```

Members

Type

Error type

State

Current CAN CC state

TimeStampLow

The FlexCard time stamp low (per default 1 µs resolution). Indicates the time at which the packet was generated.

TimeStampHigh

The FlexCard time stamp high (per default 1 µs resolution). Indicates the time at which the packet was generated.

ReceiveErrorCounter

Current state of the Receive Error Counter. Valid values range from 0 to 127.

TransmitErrorCounter

Current state of the Transmit Error Counter. Values values range from 0 to 255.

CC

The CC on which the packet was created.

Reserved[4]

Reserved for future use.

See Also

fcPacket, **fcCANErrorType**, **fcCANCcState**

	Information
ℹ	This packet type is initially supported by FlexCard API version S6V6-F.

4.6.1.18 fcEthernetPacket

This structure provides information about an Ethernet packet.

```
Typedef struct fcEthernetPacket
{
      fcDword TimeStamp;
      fcCC CC;
      fcWord PayloadLength;
      fcWord Reserved;
      fcByte* pData;
}fcEthernetPacket;
```

Members

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated

CC

The CC on which the frame was transmitted/received.

PayloadLength

Defines the number of octets contained in pData, including following information:
MAC destination, MAC source, 802.1Q tag (if it exists), ethertype, payload, crc.

Reserved

Reserved for future use.

pData

The pointer to the payload data. The payload is given in octets.
pData[0..n] contains following information:
MAC destination, MAC source, 802.1Q tag (if it exists), ethertype, payload, crc.
In case you are not interested in the crc, cut off the last 4 octets.

See Also

fcPacket

4.6.1.19 fcEthernetErrorPacket

This structure provides information about an Ethernet error packet.

```
Typedef struct fcEthernetErrorPacket
{
      fcDword TimeStamp;
      fcCC CC;
      fcEthernetErrorType Type;
      fcDword Reserved[5];
}fcEthernetPacket;
```

Members

TimeStamp

The FlexCard time stamp (per default 1 µs resolution). Indicates the time at which the packet was generated

CC

The CC on which the packet was created.

fcEthernetErrorType

Contains the ethernet error type.

Reserved

Reserved for future use.

See Also

fcPacket

	Information
ℹ	This packet type is initially supported by FlexCard API version S6V7-F.

4.6.1.20 fcPacket

This structure provides information about a packet.

```
Typedef struct fcPacket
{
    fcPacketType Type;
    union
    {
        fcFlexRayFrame*         FlexRayFrame;
        fcInfoPacket*           InfoPacket;
        fcErrorPacket*          ErrorPacket;
        fcStatusPacket*         StatusPacket;
        fcTriggerInfoPacket*    TriggerPacket;
        fcTxAcknowledgePacket*  TxAcknowledgePacket;
        fcNMVectorPacket*       NMVectorPacket;
        fcNotificationPacket*   NotificationPacket;
        fcTriggerExInfoPacket*  TriggerExPacket;
        fcCANPacket*            CANPacket;
        fcCANErrorPacket*       CANErrorPacket;
        fcCANFDPacket*          CANFDPacket;
        fcCANFDErrorPacket*     CANFDErrorPacket;
        fcEthernetPacket*       EthernetPacket;
        fcEthernetErrorPacket*  EthernetErrorPacket;
    };
    struct fcPacket* pNextPacket;
}fcPacket;
```

Members

Type

Type of packet.

FlexRayFrame

Pointer to the packet data. The content depends on the type of packet.

InfoPacket

Pointer to the packet data. The content depends on the type of packet.

ErrorPacket

Pointer to the packet data. The content depends on the type of packet.

StatusPacket

Pointer to the packet data. The content depends on the type of packet.

TriggerPacket

Pointer to the packet data. The content depends on the type of packet.

TxAcknowledgePacket

Pointer to the packet data. The content depends on the type of packet.

NMVectorPacket

Pointer to the packet data. The content depends on the type of packet.

NotificationPacket

Pointer to the packet data. The content depends on the type of packet.

TriggerExPacket

Pointer to the packet data. The content depends on the type of packet.

CANPacket

Pointer to the packet data. The content depends on the type of packet.

CANErrorPacket

Pointer to the packet data. The content depends on the type of packet.

CANFDPacket

Pointer to the packet data. The content depends on the type of packet.

CANFDErrorPacket

Pointer to the packet data. The content depends on the type of packet.

EthernetPacket

Pointer to the packet data. The content depends on the type of packet.

EthernetErrorPacket

Pointer to the packet data. The content depends on the type of packet.

pNextPacket

Pointer to the next packet. If the pointer is NULL, there are no more packets available.

See Also

fcInfoPacket, **fcErrorPacket**, **fcStatusPacket**, **fcTriggerInfoPacket**, **fcTriggerExInfoPacket**, **fcNotificationPacket**, **fcFlexRayFrame**, **fcTxAcknowledgePacket**, **fcNMVectorPacket, fcCANPacket, fcCANErrorPacket, fcCANFDPacket, fcCANFDErrorPacket, fcEthernetPacket, fcEthernetErrorPacket**

4.6.2 Enumerations

4.6.2.1 fcPacketType

This enumeration contains the different packet types.

```
Typedef enum fcPacketType
{
      fcPacketTypeInfo              = 1,
      fcPacketTypeFlexRayFrame      = 2,
      fcPacketTypeError             = 3,
      fcPacketTypeStatus            = 4,
      fcPacketTypeTrigger           = 5,
      fcPacketTypeTxAcknowledge     = 6,
      fcPacketTypeNMVector          = 7,
      fcPacketTypeNotification      = 8,
      fcPacketTypeTriggerEx         = 9,
      fcPacketTypeCAN               = 10,
      fcPacketTypeCANError          = 11,
      fcPacketTypeCANFD             = 12,
      fcPacketTypeCANFDError        = 13,
      fcPacketTypeEthernet          = 14,
      fcPacketTypeEthernetError     = 15,
}fcPacketType;
```

Members

fcPacketTypeInfo

Frame is an info packet

fcPacketTypeFlexRayFrame

Frame is a FlexRay frame

fcPacketTypeError

Frame is an error packet

fcPacketTypeStatus

Frame is a status packet

fcPacketTypeTrigger

Frame is a trigger packet (obsolete)

fcPacketTypeTxAcknowledge

Frame is a transmit acknowledge packet

fcPacketTypeNMVector

Frame is a network management vector packet

fcPacketTypeNotification

Frame is a notification packet

fcPacketTypeTriggerEx

Frame is a trigger packet

fcPacketTypeCAN

Frame is a CAN packet

fcPacketTypeCANError

Frame is a CAN error packet

fcPacketTypeCANFD

Frame is a CAN-FD packet

fcPacketTypeCANFDError

Frame is a CAN-FD error packet

fcPacketTypeEthernet

Frame is an ethernet packet

fcPacketTypeEthernetError

Frame is an ethernet error packet

See Also

fcPacket, **fcInfoPacket**, **fcFlexRayFrame**, **fcTxAcknowledgePacket**, **fcErrorPacket**, **fcStatusPacket**, **fcTriggerInfoPacket**, **fcNMVectorPacket**, **fcNotificationPacket**, **fcTriggerExInfoPacket**, **fcCANPacket**, **fcCANErrorPacket**, **fcCANFDPacket**, **fcCANFDErrorPacket**, **fcEthernetPacket**, **fcEthernetErrorPacket**

4.6.2.2 fcErrorPacketFlag

This enumeration contains the different error types reported by an error packet.

```
Typedef enum fcErrorPacketFlag
{
    fcErrNone = 0,
    fcErrFlexcardOverflow,
    fcErrPOCErrorModeChanged,
    fcErrSyncFramesBelowMinimum,
    fcErrSyncFrameOverflow,
    fcErrClockCorrectionFailure,
    fcErrParityError,
    fcErrReceiveFIFOOverrun,
    fcErrEmptyFIFOAccess,
    fcErrIllegalInputBufferAccess,
    fcErrIllegalOutputbufferAccess,
    fcErrSyntax,
    fcErrContent,
    fcErrSlotBoundaryViolation,
    fcErrTransmissionAcrossBoundary,
    fcErrLatestTransmitViolation,
    fcErrSyntaxSW,
    fcErrSlotBoundaryViolationSW,
    fcErrTransmissionConflictSW,
    fcErrSyntaxNIT,
    fcErrSlotBoundaryViolationNIT,
} fcErrorPacketFlag;
```

Members

fcErrNone

No error occurred

fcErrFlexcardOverflow

FlexCard buffer overflow. This error occurs if the application was too slow to receive and process the packets. If the FlexCard is configured to stop the monitoring it is necessary to stop and start the monitoring again. Else the FlexCard continue the monitoring when an amount of free RAM space is available again. In such a case the FlexCard loses packets.

fcErrPOCErrorModeChanged

Protocol Operation Control error. Additional information is described in the structure `fcErrPOCErrorModeChangedInfo`

fcErrSyncFramesBelowMinimum

Additional information is described in the structure `fcErrSyncFramesInfo`

fcErrSyncFrameOverflow

Additional information described in the structure `fcErrSyncFramesInfo`

fcErrClockCorrectionFailure

Additional information is described in the structure `fcErrClockCorrectionFailureInfo`

fcErrParityError

Internal E-Ray error. No additional information is available

fcErrReceiveFIFOOverrun

No additional information exists for the internal FlexCard error
(`fcErrorPacket.AdditionalInfo` is not valid)

fcErrEmptyFIFOAccess

No additional information exists for the internal FlexCard error *fcErrIllegalInputBufferAccess*

No additional information exists for the internal FlexCard error

fcErrIllegalOutputbufferAccess

No additional information exists for the internal FlexCard error

fcErrSyntax

A syntax error was observed (frame is syntactically incorrect). (FlexRay Protocol Specification: vSS!SyntaxErrorA or vSS!SyntaxErrorB depends on Channel) Additional information are described in the structure `fcErrSlotInfo`

fcErrContent

A content error was observed (frame is semantically incorrect). (FlexRay Protocol Specification: vSS!ContentErrorA or vSS!ContentErrorB depends on Channel) Additional information is described in the structure `fcErrSlotInfo`

fcErrSlotBoundaryViolation

A slot boundary violation was observed. (FlexRay Protocol Specification: vSS!BviolationA or vSS!BviolationB depends on Channel) Additional information is described in the structure `fcErrSlotInfo`

fcErrTransmissionAcrossBoundary

Additional information is described in the structure `fcErrSlotInfo`

fcErrLatestTransmitViolation

Additional information is described in the structure `fcErrSlotInfo`

fcErrSyntaxSW

Syntax error in symbol window was observed. Additional information is described in the structure `fcErrSlotInfo`.

fcErrSlotBoundaryViolationSW

Slot boundary violation in symbol window was observed. Additional information is described in the structure `fcErrSlotInfo`.

fcErrTransmissionConflictSW

Transmission conflict in symbol window was observed. Additional information is described in the structure `fcErrSlotInfo`.

fcErrSyntaxNIT

Syntax error in network idle time was observed. Additional information is described in the structure `fcErrSlotInfo`.

fcErrSlotBoundaryViolationNIT

Slot boundary violation in network idle time was observed. Additional information is described in the structure `fcErrSlotInfo`.

See Also

fcErrorPacket, **fcErrPOCErrorModeChangedInfo**, **fcErrSyncFramesInfo**, **fcErrClockCorrectionFailureInfo**, **fcErrSlotInfo**

4.6.2.3 fcStatusPacketFlag

Possible hardware status flags are reported by a status packet.

```
Typedef enum fcStatusPacketFlag
{
    fcStatusNone = 0,
    fcStatusWakeupStatus,
    fcStatusCollisionAvoidanceSymbol,
    fcStatusStartupCompletedSuccessfully,
    fcStatusWakeupPatternChannelA,
    fcStatusWakeupPatternChannelB,
    fcStatusMTSReceivedonChannelA,
    fcStatusMTSReceivedonChannelB,
} fcStatusPacketFlags;
```

Members

fcStatusNone

No status changes.

fcStatusWakeupStatus

Wakeup status has changed

fcStatusCollisionAvoidanceSymbol

Collision avoidance symbol was received

fcStatusStartupCompletedSuccessfully

Start-up has been successfully completed

fcStatusWakeupPatternChannelA

Wakeup pattern received on Channel A

fcStatusWakeupPatternChannelB

Wakeup pattern received on Channel B

fcStatusMTSReceivedonChannelA

Media Access Test Symbol received on Channel A

fcStatusMTSReceivedonChannelB

Media Access Test Symbol received on Channel B

See Also

fcPacket, **fcStatusPacket**, **fcStatusWakeupInfo**

4.6.2.4 fcCANErrorType

This enumeration contains the different error types reported by a CAN error packet.

```
Typedef enum fcCANErrorType
{
      fcCANErrorNone = 0,
      fcCANErrorStuff,
      fcCANErrorForm,
      fcCANErrorAcknowledge,
      fcCANErrorBit1,
      fcCANErrorBit0,
      fcCANErrorCRC,
      fcCANErrorParity,
}fcCANErrorType;
```

Members

fcCANErrorNone

No error occurred.

fcCANErrorStuff

More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed.

fcCANErrorForm

A fixed format part of a received frame has the wrong format.

fcCANErrorAcknowledge

The message the CAN Communication Controller transmitted was not acknowledged by another node.

fcCANErrorBit1

During the transmission of a message (with the exception of the arbitration field), the device wanted to send a recessive level (Bit of logical value 1), but the monitored bus value was dominant (Bit of logical value 0).

fcCANErrorBit0

During the transmission of a message, the device wanted to send a dominant level (data or identifier Bit logical value 0), but the monitored bus value was recessive (data or identifier Bit logical value 1).

fcCANErrorCRC

The CRC check sum was incorrect in the message received; the CRC received for an incoming message does not match with the calculated CRC for the received data.

fcCANErrorParity

The parity check mechanism has detected a parity error in the message RAM of the Communication Controller.

See Also

fcCANErrorPacket

	Information
ℹ	This enumeration is initially supported by FlexCard API version S4V0-F.

4.6.2.5 fcEthernetErrorType

This enumeration contains the different error types reported by an ethernet error packet.

```
Typedef enum fcEthernetErrorType
{
      fcEthernetErrorNone = 0,
      fcEthernetErrorFrameCheckSequence,
}fcEthernetErrorType;
```

Members

fcEthernetErrorNone

No error occurred.

fcEthernetErrorFrameCheckSequence

A frame check sequence error was detected.

See Also

fcEthernetErrorPacket

	Information
ℹ	This enumeration is initially supported by FlexCard API version S6V7-F.

4.6.3 fcbReceive

This function reads all available packets from the FlexCard memory into a memory block allocated by the fcBase API. The frames are stored into a linked list. To free the memory allocated by this function, use the function **fcFreeMemory** with the type *fcMemoryTypePacket*.

```
fcError fcbReceive(
      fcHandle hFlexCard,
      fcPacket** pPacket
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

pPacket

[OUT] Address of the fcPacket object pointer. The memory for this structure and its content is allocated by the fcBase API. Packets are available if the return code is 0 and *pPacket* is not a null pointer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

	Information
ℹ	This function allocates memory. To prevent memory leaks the memory has to be released after having processed the packets.

Example

```
fcPacket* pPackets = NULL;
fcError e = fcbReceive(m_hFlexCard, &pPackets);
if (0 == e)
{
      fcPacket* pCurrentPacket = pPackets;
      while (NULL != pCurrentPacket)
      {
            switch (pCurrentPacket->Type)
            {
            case fcPacketTypeInfo:
                  printf(“[fcPacketTypeInfo] TimeStamp: %f Cycle: %d\n”,
                        (float)pCurrentPacket->InfoPacket->TimeStamp* 0.000001,
                        pCurrentPacket->InfoPacket->CurrentCycle);

            break;

            case fcPacketTypeFlexRayFrame:
            {
                  fcFlexRayFrame* pFrame = pCurrentPacket->FlexRayFrame;
                  printf(“[fcPacketTypeFlexRayFrame] Cycle: %d ID: %d Channel:”
                        “%d PayloadLength: %d”,  pFrame->CycleCount,
                        pFrame->ID,
                        pFrame->Channel,
                        pFrame->PayloadLength);

                  for (int i = 0; i < pFrame->PayloadLength; i++)
                  {
                        printf(“%04X “, pFrame->pData[i]);
                  }

                  if (pFrame->PP) printf(“ PP”);
                  if (pFrame->NF) printf(“ NF”);
                  if (pFrame->SYNC) printf(“ SYNC”);
                  if (pFrame->STARTUP) printf(“ STARTUP”);
                  if (pFrame->SyntaxError) printf(“ SyntaxError”);
                  if (pFrame->ContentError) printf(“ ContentError”);
                  if (pFrame->ValidFrame) printf(“ ValidFrame”);
                  if (pFrame->SlotBoundaryViolation)
                        printf(“ SlotBoundaryViolation”);
                  if (pFrame->AsyncMode)
                        printf(“ AsyncMode FrameCRC: 0x%06X”, pFrame->FrameCRC);
                  printf(“\n”);
                  break;
            }
            case fcPacketTypeError:
                  printf(“[fcPacketTypeError]\n”);
                  break;

            case fcPacketTypeStatus:
                  printf(“[fcPacketTypeStatus]\n”);
                  break;

            case fcPacketTypeTrigger:
                  printf(“[fcPacketTypeTrigger]\n”);
                  break;

            case fcPacketTypeTxAcknowledge:
                  printf(“[fcPacketTypeTxAcknowledge]\n”);
                  break;

            case fcPacketTypeNMVector:
                  printf(“[fcPacketTypeNMVector]\n”);
                  break;
            }

            pCurrentPacket = pCurrentPacket->pNextPacket;
      }
      fcFreeMemory(fcMemoryTypePacket, pPackets);
}
```

5 FlexRay API

The following section describes the data structures and features used for FlexRay functionality. To use these functions the FlexCard must have a firmware with a FlexRay CC and the FlexCard must be licensed for FlexRay.

	Information
ℹ	All enumerations, structures and function in this chapter are initially supported by FlexCard API version S4V2-F.

5.1 Basic FlexRay Workflow

The following figure shows a typical FlexRay workflow. Refer to the chapter General Function Availability to find out what FlexCard supports what functions. The FlexCard Windows Developer Setup installs the example applications *fcDemo.exe*, *fcDemoPMC.exe* and their source code to the installation directory.

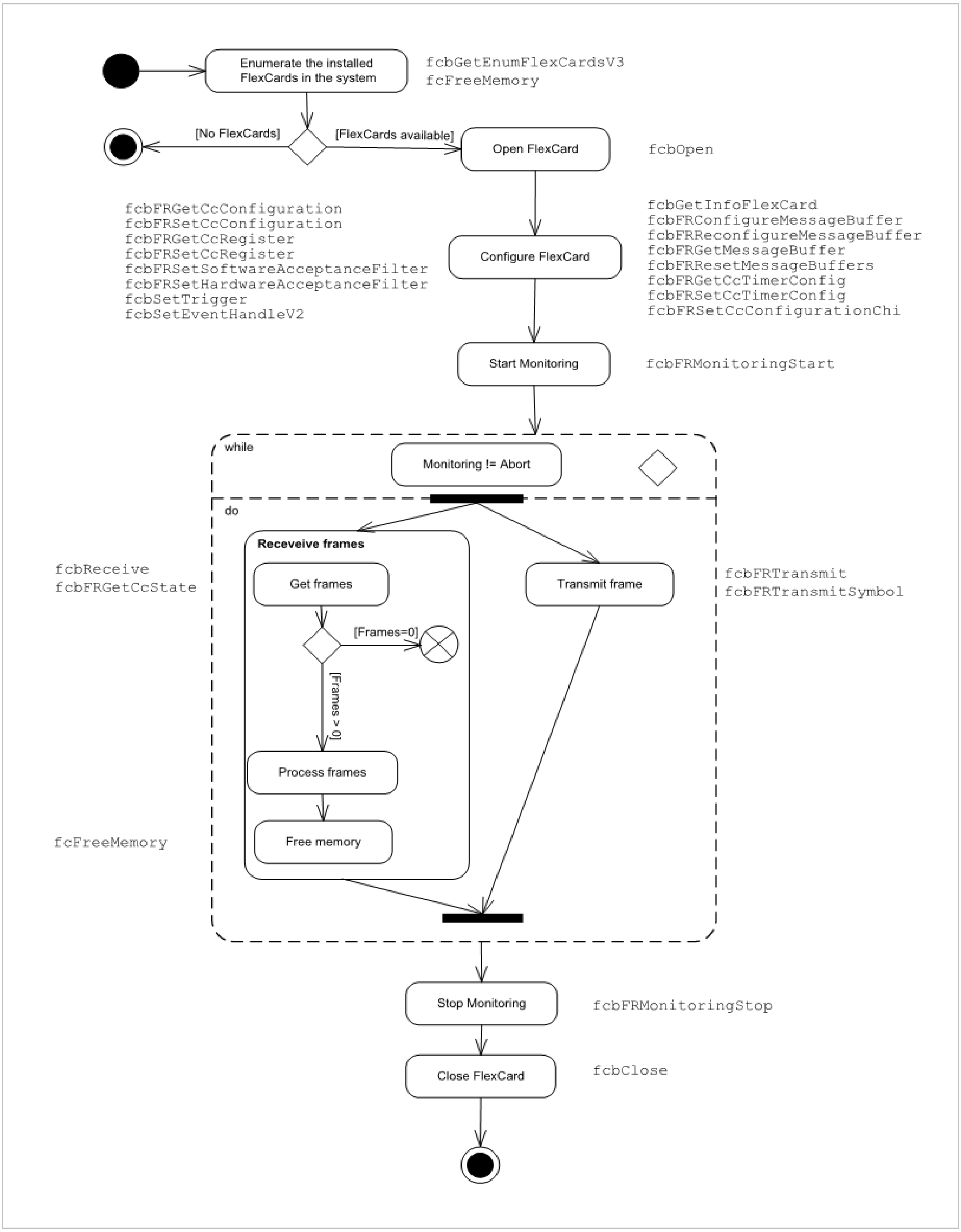


Figure 9: Typical FlexRay function workflow

The following table gives information about what message buffer functions may be called **while Monitoring is active**.

Functionality	Action during monitoring
New configuration of a message buffer.	Not possible.
Read message buffer configuration.	Not possible. Application has to store the buffer information.
Reconfiguration of a static FlexRay ID (Reconfigure channel, id, payload length).	Only in extended mode (fcFRMsgBufCfgModeReconfigurationDuringMonitoring)
Reconfiguration of a static FlexRay Sync-ID.	Not possible due to limitations of the Communication Controller Bosch E-Ray.
Reconfiguration of a dynamic FlexRay ID (Reconfigure channel, id, payload length).	Possible.
Turning off/on a static FlexRay ID (Reconfigure to fcChannelNone).	Only in extended mode (fcFRMsgBufCfgModeReconfigurationDuringMonitoring)
Turning off/on a static FlexRay Sync ID (Reconfigure to fcChannelNone).	Not possible due to limitations of the Communication Controller Bosch E-Ray.
Turning off/on a dynamic FlexRay ID (Reconfigure to fcChannelNone).	Only in extended mode (fcFRMsgBufCfgModeReconfigurationDuringMonitoring)

5.2 Initialization

5.2.1 Enumerations

5.2.1.1 fcMonitoringModes

This enumeration defines the different modes available, used to monitor a FlexRay cluster.

```
Typedef enum fcMonitoringModes
{
       fcMonitoringNormal,
       fcMonitoringDebug,
       fcMonitoringDebugAsynchron,
       fcMonitoringDebugAsynchronBeforeStartup,
} fcMonitoringModes;
```

Members

fcMonitoringNormal

First, the FlexCard tries to synchronize itself with the cluster. Once the synchronization succeeds, the FlexCard enters in the NORMAL_ACTIVE state and is able to transmit and receive FlexRay frames, symbols and errors, as previously configured. The timestamp accuracy in this mode is +/-1 µs.

fcMonitoringDebug

This mode is provided by the E-Ray FlexRay core. The FlexCard does not try to synchronize itself with the cluster and is only able to receive FlexRay frames, symbols and errors from the FlexRay bus. This mode does not allow transmission; it is therefore not possible to perform a start-up or a wake-up. This mode is adapted for debugging purpose (e.g. start-up of a FlexRay network fails).

Note: To receive frames within this mode using an E-Ray version older than 1.3, you must configure RX receive buffers. The FIFO receive buffers aren't working in this mode. With CC version 1.3 it's possible to receive frames wih FIFO receive buffers. You may call **fcbGetInfoFlexCard** and read the variable *CCVersion* to check the E-Ray version that is present on the FlexCard.

fcMonitoringDebugAsynchron

This debug operation mode of the FlexCard allows the reception of all frames without any message buffer (API configures FIFO buffers automatically) and controller configuration. The only parameter to be set is the baudrate (Register 0x0090: 10 Mbit/s: 0x00000000, 5 Mbit/s: 0x00004000, 2.5 Mbit/s: 0x00008000). This mode does not allow transmission. It is therefore not possible to perform a start-up or a wake-up. This mode is adapted for debugging purpose (e.g. start-up of a FlexRay network fails or to monitor an unknown network). The timestamp accuracy in this mode is +/-2 µs. Incorrect data will be interpreted as received FlexRay frames (the Valid Frame Bit is not set for such frames).

fcMonitoringDebugAsynchronBeforeStartup

This mode combines the mode *fcMonitoringDebugAsynchron* and *fcMonitoringNormal*. The mode *fcMonitoringDebugAsynchronBeforeStartup* is used to receive all frames during start-up. Unlike *fcMonitoringDebug* this mode allows to send sync frames. After the start-up completed successfully, the FlexCard switchs directly to the mode *fcMonitoringNormal*.

See Also

fcbFRMonitoringStart

5.2.1.2 fcState

This enumeration defines the possible Communication Controller POC states (FlexRay Protocol Specification: vPOC!State). For more details about Communication Controller POC states, please refer to [3].

```
Typedef enum fcState
{
      fcStateUnknown,
      fcStateConfig,
      fcStateNormalActive,
      fcStateNormalPassive,
      fcStateHalt,
      fcStateReady,
      fcStateStartup,
      fcStateWakeup,
      fcStateMonitorMode,
} fcState;
```

Members

fcStateUnknown

Communication controller state is not known.

fcStateConfig

Communication controller is in CONFIG state.

fcStateNormalActive

Communication controller is in NORMAL_ACTIVE state.

fcStateNormalPassive

Communication controller is in NORMAL_PASSIVE state.

fcStateHalt

Communication controller is in HALT state.

fcStateReady

Communication controller is in READY state.

fcStateStartup

Communication controller is in STARTUP state.

fcStateWakeup

Communication controller is in WAKEUP state.

fcStateMonitorMode

Communication controller is in MONITORMODE state.

See Also

fcbFRGetCcState, **fcbFRMonitoringStart**

5.2.2 fcbFRMonitoringStart

This function is used to start the monitoring of a FlexRay bus. Once the user calls the function with *fcMonitoringNormal*, it changes the Communication Controller state from configuration state to normal active state if the cluster integration succeeds. The function returns immediately and does not wait for the Communication Controller to get synchronous. The current Communication Controller state can be read using the function **fcbFRGetCCState**. If the FlexCard is synchronized with the cluster the function **fcbFRGetCCState** will return the value *fcStateNormalActive*.

```
fcError fcbFRMonitoringStart(
      fcHandle hFlexCard,
      fcCC CC,
      fcMonitoringModes mode,
      fcBool restartTimestamps,
      fcBool enableCycleStartEvents
      fcBool enableColdstart,
      fcBool enableWakeup
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index

mode

[IN] The monitoring mode. Not every monitoring mode is supported by all Communication Controllers. See **fcMonitoringModes** for details.

restartTimestamps

[IN] Set this parameter to 0 to restart the measurement without resetting the FlexCard timestamp. Set it to <> 0 to start the measurement from the beginning. The timestamps have microsecond resolution.

enableCycleStartEvents

[IN] Set this parameter to <> 0 to enable the cycle start events in order that at the beginning of every cycle the event *fcNotificationTypeFRCycleStarted* is signalled. On the *FlexCard USB-M*, this feature is not supported. **fcbFRMonitoringStart** does not return an error when this parameter is set, but no events are signalled.

enableColdstart

[IN] Set this parameter to <> 0 to allow the FlexCard to initialize the cluster communication, otherwise the coldstart inhibit mode is active. This feature can not be used in the monitoring modes *fcMonitoringDebug* and *fcMonitoringDebugAsynchron*.

enableWakeup

[IN] Set this parameter to <> 0 to transmit a wake-up pattern on the configured wake-up channel (FlexRay Protocol Specification: pWakeupChannel). A cluster wake-up must precede the communication start-up to ensure that all nodes in a cluster are awake. The minimum requirement for a cluster wake-up is that all bus drivers are supplied with power. This feature can not be used in the monitoring modes *fcMonitoringDebug* and *fcMonitoringDebugAsynchron*.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

After the monitoring with *fcMonitoringNormal* has started, the user should check if the integration in the cluster was successful: **fcbFRGetCCState** should return the state *fcStateNormalActive*.

ℹ	**Information**
	After the monitoring has successfully started, the receive process must be started as soon as possible to avoid an overflow (error packet *fcErrFlexcardOverflow* is received). Once an overflow occurred, no more packets can be received. The monitoring must be stopped and started again.

See Also

fcCC, **fcbFRMonitoringStop**, **fcbFRGetCcState**, **fcMonitoringModes**, **fcbSetEventHandleV2**, **fcbSetEventHandleSemaphore**

Example

```
// Precondition: valid flexcard handle exists and the flexcard is
// already configured.
fcCC eCC = fcCC1;
fcError e = fcbFRMonitoringStart(hFlexCard,eCC,fcMonitoringNormal,true,
                                   false,false,false);
if (0 == e)
{
      bool synchronized = false;
      bool timeout = false;
      DWORD maxTime = ::GetTickCount() + 2000;
      fcState currentState = fcStateUnknown;

      // Check if the FlexCard is synchronized
      do
      {
            fcbFRGetCcState(hFlexCard, eCC, &currentState);
            synchronized= (currentState == fcStateNormalActive);
            timeout            = ::GetTickCount() >= maxTime;

      } while ( ! synchronized && ! timeout);

      if (synchronized)
      {
            // Start your receive thread/routine
            //   ...
      }
      else
      {
            // if we timed out, we stop the monitoring
            fcbFRMonitoringStop(hFlexCard,eCC);
      }
}
else
{
      // error handling …
}
```

5.2.3 fcbFRMonitoringStop

This function stops the FlexRay bus measurement. The Communication Controller is set back in its configuration state.

```
fcError fcbFRMonitoringStop(
      fcHandle hFlexCard,
      fcCC CC
)
```

Parameters

hFlexCard

[IN] Handle to FlexCard

CC

[IN] Communication controller index

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcbFRMonitoringStart**

5.2.4 fcbFRGetCcState

This function returns the current Communication Controller POC state. For a description of possible states, refer to the enumeration **fcState**. This function should be used to check if the integration into a FlexRay cluster has succeeded.

On the FlexCard PXIe3/PCIe3 in the FlexDevice mode, this function is only allowed for debugging purposes. It should not be used in this case because the decoder and the communication controller should be independent.

```
fcError fcbFRGetCcState(
       fcHandle hFlexCard,
       fcCC CC,
       fcState* pState
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

pState

[OUT] Current Communication Controller state

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See

fcCC, fcState, fcbFRMonitoringStart, fcbFRMonitoringStop

Example

See example **fcbFRMonitoringStart**

5.2.5 fcbFRSetTransceiverState

This function sets the transceiver mode individually for each channel.

```
fcError fcbFRSetTransceiverState (
       fcHandle hFlexCard,
       fcCC CC,
       fcTransceiverState stateChA,
       fcTransceiverState stateChB
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

stateChA

[IN] The new transceiver state for channel A

stateChB

[IN] The new transceiver state for channel B

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

If one of the transceivers is in the sleep mode and the transceiver detects a wake-up event, the notification event *fcNotificationTypeFRWakeup* is fired once only. Note that the transceiver state stays the same after closing the FlexCard and opening it again. When the FlexCard is powered off and on again, the transceiver state is reset. This e.g. happens during the stand-by of the computer. When you want to make sure that the FlexCard is in its default state, set the transceiver state to normal before starting the monitoring.

See

fcCC, **fcTransceiverState**, **fcbFRMonitoringStart**, **fcbFRSetTransceiverState**

5.2.6 fcbFRGetTransceiverState

This function gets the transceiver state of a selected Communication Controller individually for each channel.

```
fcError fcbFRGetTransceiverState (
       fcHandle hFlexCard,
       fcCC CC,
       fcTransceiverState* pStateChA,
       fcTransceiverState* pStateChB
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

pStateChA

[OUT] The current transceiver state for channel A

pStateChB

[OUT] The current transceiver state for channel B

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

If one of the transceivers is in the sleep mode and the transceiver detects a wake-up event, the notification event *fcNotificationTypeFRWakeup* is fired once only. Note that the transceiver state stays the same after closing the FlexCard and opening it again. When the FlexCard is powered off and on again, the transceiver state is reset. This e.g. happens during the stand-by of the computer. When you want to make sure that the FlexCard is in its default state, set the transceiver state to normal before starting the monitoring.

See

fcCC, **fcTransceiverState**, **fcbFRMonitoringStart**, **fcbFRSetTransceiverState**

5.3 Configuration

This chapter describes the functions and data types used to configure both Communication Controller and hardware of a FlexCard. The configuration phase of a FlexCard is an essential part of its integration into a cluster and can not be skipped. Entering the bus parameters of an existent network is possible directly or by CHI-Import. If one of the FlexCard configuration settings does not match the cluster ones, the FlexCard may not be able to monitor the bus. Therefore, it is highly recommended to use a configuration tool for designing a new FlexRay network. FlexConfig Developer from *STAR ELECTRONICS GmbH & Co. KG* is such a tool that outputs a CHI file. It automatically validates and generates for each FlexRay parameter the corresponding register values of each node in a cluster. Manual configuration of the FlexCard is also a possibility but will be a complex, time-consuming, and error-prone method due to the large number of E-Ray registers used for configuration.

As the FlexCard uses the receive FIFO functionality from the Communication Controller to monitor the FlexRay frames, the fcBase API has to ensure that enough FIFO message buffers are configured, that means that not all message buffers are available for the user. Modifying the FIFO message buffers settings may affect the ability to correctly monitor the FlexRay bus.

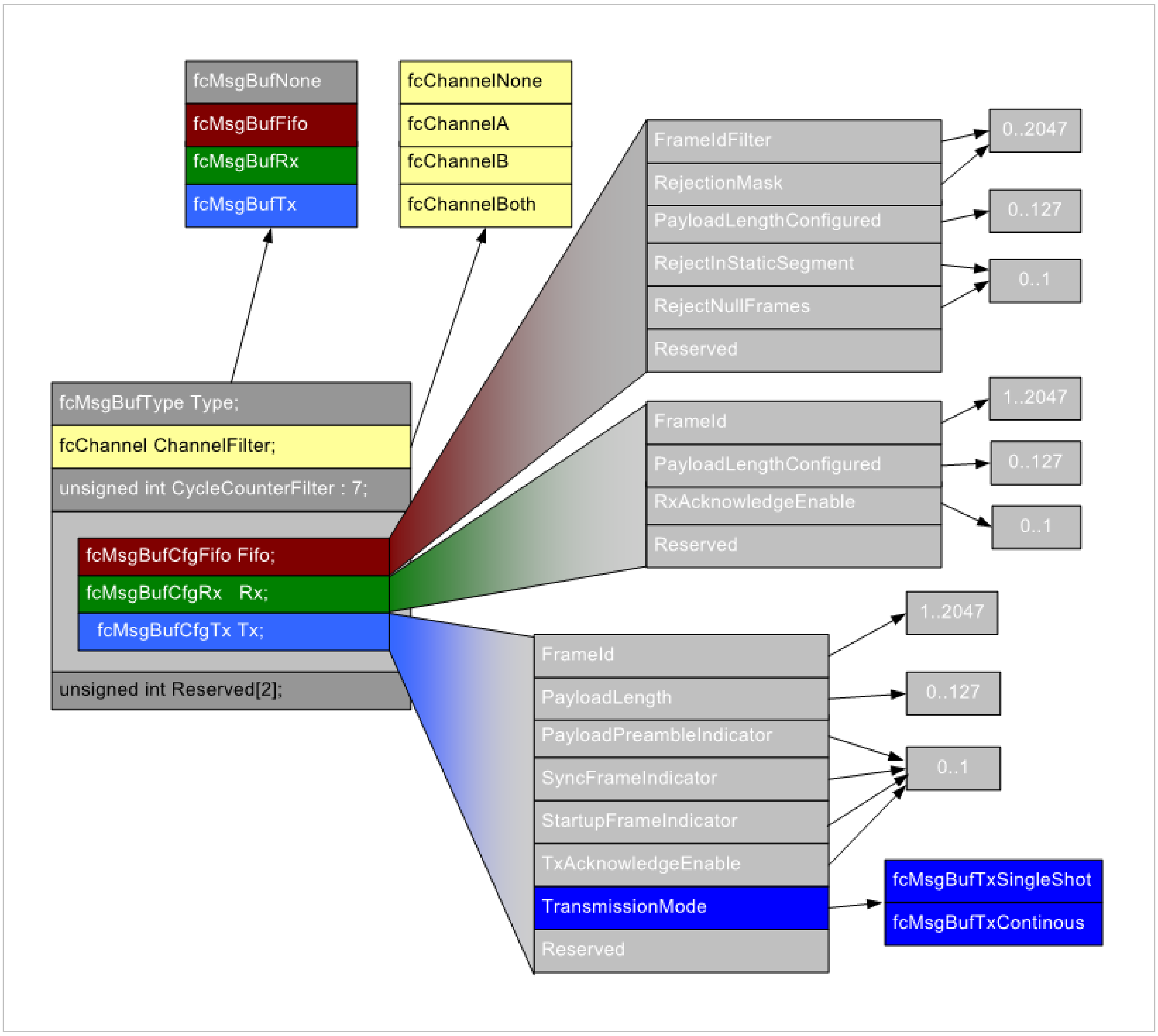


Figure 10: Overview fcbMsgBufCfg structure

The message buffer RAM is 2048 32-bit words in total. It contains a 16 byte administration data structure for each message buffer, the rest may be used for the payload. This leads to a possible maximum configuration for example of 30 message buffers with 254 byte payload, 56 message buffers with 128 byte payload, or 128 message buffers with 48 byte payload. For further information on configuration of the message RAM see [5] chapter 5.12.

5.3.1 Constants

5.3.1.1 fcPayloadMaximum

Maximum number of 2-byte payload data words

```
const fcByte fcPayloadMaximum = 127
```

5.3.2 Enumerations

5.3.2.1 fcChannel

This enumeration defines the available channel combination of the FlexCard.

```
Typedef enum fcChannel
{
       fcChannelNone = 0x00,
       fcChannelA    = 0x01,
       fcChannelB    = 0x02,
       fcChannelBoth = fcChannelA | fcChannelB,
} fcChannel;
```

Members

fcChannelNone

No FlexRay channel selected

fcChannelA

Only FlexRay channel A is selected

fcChannelB

Only FlexRay channel B is selected

fcChannelBoth

FlexRay channel A and B are selected

See Also

fcMsgBufCfg

5.3.2.2 fcWakeupStatus

This enumeration defines the possible Communication Controller wake-up states (FlexRay Protocol Specification: vPOC!WakeupStatus). For more details about Communication Controller wake-up states, please refer to [3].

```
Typedef enum fcWakeupStatus
{
       fcWakeupStatusUndefined = 0,
       fcWakeupStatusReceiveHeader,
       fcWakeupStatusReceiveWUP,
       fcWakeupStatusCollisionHeader,
       fcWakeupStatusCollisionWUP,
       fcWakeupStatusCollisionUnknown
       fcWakeupStatusTransmitted,
} fcWakeupStatus;
```

Members

fcWakeupStatusUndefined

FlexRay Protocol Specification: UNDEFINED

fcWakeupStatusReceiveHeader

FlexRay Protocol Specification: RECEIVE_HEADER

fcWakeupStatusReceiveWUP

FlexRay Protocol Specification: RECEIVE_WUP

fcWakeupStatusCollisionHeader

FlexRay Protocol Specification: COLLISION_HEADER

fcWakeupStatusCollisionWUP

FlexRay Protocol Specification: COLLISION_WUP

fcWakeupStatusCollisionUnknown

FlexRay Protocol Specification: COLLISION_UNKNOWN

fcWakeupStatusTransmitted

FlexRay Protocol Specification: TRANSMITTED

See Also

fcStatusWakeupInfo, **fcStatusPacket**

5.3.2.3 fcTransceiverState

This enumeration defines the different states of the FlexRay transceivers.

```
Typedef enum fcTransceiverState
{
      fcTransceiverNormal,
      fcTransceiverSleep,
} fcTransceiverState;
```

Members

fcTransceiverNormal

Transceiver is in normal mode and can transmit and receive data via the FlexRay bus.

fcTransceiverSleep

Transceiver is in low power mode and is not able to transmit and receive data but is able to detect wake-up events on the bus. If a wake-up is detected the event fcNotificationTypeFRWakeup is fired.

See Also

fcbFRSetTransceiverState, **fcbFRGetTransceiverState**

	Information
ℹ	This enumeration is initially supported by FlexCard API version S2V0-F.

5.3.2.4 fcFRBaudRate

This enumeration defines the various baud rates on the FlexRay bus.

```
Typedef enum fcFRBaudRate
{
      fcFRBaudRateNone = 0,
      fcFRBaudRate2M5,
      fcFRBaudRate5M,
      fcFRBaudRate10M,
} fcFRBaudRate;
```

Members

fcFRBaudRateNone

No baud rate defined

fcFRBaudRate2M5

Defines the baud rate 2.5 Mbit/s

fcFRBaudRate5M

Defines the baud rate 5 Mbit/s

fcFRBaudRate10M

Defines the baud rate 10 Mbit/s

See Also

fcFRCcConfig

5.3.2.5 fcFRMsgBufCfgMode

This enumeration defines the available message buffer configuration modes with the FlexCard fcBase API. In normal mode the message buffer configuration is very strict and safe; in expert mode some special configurations are allowed. These message buffer configuration modes can be binary Ored.

```
Typedef enum fcFRMsgBufCfgMode
{
      fcFRMsgBufCfgModeNone = 0,
      fcFRMsgBufCfgModeNormal = fcFRMsgBufCfgModeNone,
      fcFRMsgBufCfgModeUnequalStaticPayloadLength = 1,
      fcFRMsgBufCfgModeReconfigurationDuringMonitoring = 2,
      fcFRMsgBufCfgModeCycleMultiplexInStartupSyncFrame = 4,
      fcFRMsgBufCfgModeAll = fcFRMsgBufCfgModeUnequalStaticPayloadLength |
                             fcFRMsgBufCfgModeReconfigurationDuringMonitoring |
                             fcFRMsgBufCfgModeCycleMultiplexInStartupSyncFrame,
      fcFRMsgBufCfgModeExpert = fcFRMsgBufCfgModeAll,
} fcFRMsgBufCfgMode;
```

Members

fcFRMsgBufCfgModeNone

fcFRMsgBufCfgModeNormal

Normal (safe) message buffer configuration mode.

fcFRMsgBufCfgModeUnequalStaticPayloadLength

Allows transmit message buffer configurations in static segment with PayloadLength between 0 and PayloadLengthMax. For further description see **fcMsgBufCfgTx** and **fcMsgBufCfgRx**.

fcFRMsgBufCfgModeReconfigurationDuringMonitoring

Allows extended reconfigurations of message buffers during monitoring. Is this mode set, message buffer transmission and reception can be configured with parameter *ChannelFilter* in normal active mode. For further description see **fcMsgBufCfg**.

fcFRMsgBufCfgModeCycleMultiplexInStartupSyncFrame

Allows extended configurations of start-up/sync message buffers. Is this mode set, cycle counter filtering for more than two start-up/sync message buffers is possible. A reconfiguration of start-up/sync isn’t allowed while monitoring is active. For further description see **fcMsgBufCfg**.

fcFRMsgBufCfgModeAll

fcFRMsgBufCfgModeExpert

Expert (unsafe) message buffer configuration mode

See Also

fcbFRSetMsgBufCfgMode, **fcMsgBufCfg**, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**

	Information
ℹ	This enumeration is initially supported by FlexCard API version S6V1-F.

5.3.2.6 fcMsgBufType

For the transmission and reception of FlexRay frames the Communication Controller provides different types of message buffers. Each message buffer can be assigned with one of the following specific types.

```
Typedef enum fcMsgBufType
{
      fcMsgBufNone,
      fcMsgBufRx,
      fcMsgBufTx,
      fcMsgBufFifo,
} fcMsgBufType;
```

Members

fcMsgBufNone

The message buffer is not used.

fcMsgBufRx

The message buffer is used as a receive buffer (e.g. to analyse a specific frame).

fcMsgBufTx

The message buffer is used as a transmit buffer (e.g. to transmit a message on a specific communication slot).

fcMsgBufFifo

The message buffer is used as a receive FIFO buffer. In that case, it will receive frames from different communication slots.

	Information
ℹ	In certain cases, it is not possible to receive all frames with only receive message buffers. To ensure that all frames will be received, we recommend to configure some FIFO message buffers.

See Also

fcMsgBufCfg

5.3.2.7 fcMsgBufTxMode

This enumeration defines the different modes of transmission.

```
Typedef enum fcMsgBufTxMode
{
       fcMsgBufTxSingleShot,
       fcMsgBufTxContinous,
}fcMsgBufTxMode;
```

Members

fcMsgBufTxSingleShot

Frame is transmitted once only if its corresponding message buffer content has been set and both frame id and cycle filter are matching. The function **fcbFRTransmit** sets the content of a given message buffer.

fcMsgBufTxContinous

Frame is transmitted each time when both the frame id and cycle filter are matching, regardless if its corresponding message buffer content has been set or not.

See Also

fcMsgBufCfgTx

5.3.2.8 fcCyclePos

This enumeration defines various positions in a FlexRay cycle.

```
Typedef enum fcCyclePos
{
       fcCyclePosNotDefined = 0,

       fcCyclePosStaticSlot,
       fcCyclePosDynamicMiniSlot,

       fcCyclePosEndStaticSegment,
       fcCyclePosStartDynamicSegment,
       fcCyclePosEndDynamicSegment,
       fcCyclePosStartSymbolWindow,
       fcCyclePosEndSymbolWindow,
       fcCyclePosStartNetworkIdleTime,
} fcCyclePos;
```

Members

fcCyclePosNotDefined

No cycle position defined

fcCyclePosStaticSlot

Defines the start of a static slot

fcCyclePosDynamicMiniSlot

Defines the start of a dynamic mini slot

fcCyclePosEndStaticSegment

Defines the end of the static segment

fcCyclePosStartDynamicSegment

Defines the start of the dynamic segment

fcCyclePosEndDynamicSegment

Defines the end of the dynamic segment

fcCyclePosStartSymbolWindow

Defines the start of the symbol window

fcCyclePosEndSymbolWindow

Defines the end of the symbol window

fcCyclePosStartNetworkIdleTime

Defines the start of the network idle time

See Also

fcbFRCalculateMacrotickOffset

ℹ	Information
	This enumeration is initially supported by FlexCard API version S4V0-F.

5.3.3 Structures

5.3.3.1 fcFRCcConfig

This structure describes the configuration of the FlexRay Communication Controller. The struct contains the variables from the FlexRay specification. The FlexCard driver makes the conversion to/from the registers the FlexRay core E-Ray uses. For example, gListenNoise is not the same in the FlexRay specification compared to the E-Ray register.

The variable descriptions were extracted from [5] (Bosch E-Ray FlexRay IP-Module User's Manual).

```
Typedef struct fcFRCcConfig
{
       fcFRBaudRate BaudRate;
       fcDword gdActionPointOffset;
       fcDword gdCASRxLowMax;
       fcDword gdDynamicSlotIdlePhase;
       fcDword gdMinislot;
       fcDword gdMinislotActionPointOffset;
       fcDword gdNIT;
       fcDword gdStaticSlot;
       fcDword gdTSSTransmitter;
       fcDword gdWakeupSymbolRxIdle;
       fcDword gdWakeupSymbolRxLow;
       fcDword gdWakeupSymbolRxWindow;
       fcDword gdWakeupSymbolTxIdle;
       fcDword gdWakeupSymbolTxLow;
       fcDword gColdStartAttempts;
       fcDword gListenNoise;
       fcDword gMacroPerCycle;
       fcDword gMaxWithoutClockCorrectionFatal;
       fcDword gMaxWithoutClockCorrectionPassive;
       fcDword gNetworkManagementVectorLength;
       fcDword gNumberOfMinislots;
       fcDword gNumberOfStaticSlots;
       fcDword gOffsetCorrectionStart;
       fcDword gPayloadLengthStatic;
       fcDword gSyncNodeMax;
       fcDword pdAcceptedStartupRange;
       fcDword pdListenTimeout;
       fcDword pdMaxDrift;
       fcDword pAllowHaltDueToClock;
       fcDword pAllowPassiveToActive;
       fcChannel pChannelsMTS;
       fcChannel pChannels;
       fcDword pClusterDriftDamping;
       fcDword pDecodingCorrection;
       fcDword pDelayCompensationA;
       fcDword pDelayCompensationB;
       fcDword pExternOffsetCorrection;
       fcDword pExternRateCorrection;
       fcDword pKeySlotUsedForStartup; //NOT USED.
       fcDword pKeySlotUsedForSync; //NOT USED.
       fcDword pLatestTx;
       fcDword pMacroInitialOffsetA;
       fcDword pMacroInitialOffsetB;
       fcDword pMicroInitialOffsetA;
       fcDword pMicroInitialOffsetB;
       fcDword pMicroPerCycle;
       fcDword pOffsetCorrectionOut;
       fcDword pRateCorrectionOut;
       fcDword pSingleSlotEnabled;
       fcChannel pWakeupChannel;
       fcDword pWakeupPattern;
       fcDword vExternOffsetControl;
       fcDword vExternRateControl;

       fcDword Reserved[16];
} fcFRCcConfig;
```

Members

BaudRate

Configures the baud rate on the FlexRay bus.

gdActionPointOffset

Configures the action point offset in macroticks within static slots and symbol window. Must be identical in all nodes of a cluster. Valid values are 1 to 63 MT.

gdCASRxLowMax

Configures the upper limit of the acceptance window for a collision avoidance symbol (CAS). Valid values are 67 to 99 bit times.

gdDynamicSlotIdlePhase

The duration of the dynamic slot idle phase has to be greater or equal than the idle detection time. Must be identical in all nodes of a cluster. Valid values are 0 to 2 Minislot.

gdMinislot

Configures the duration of a minislot in macroticks. The minislot length must be identical in all nodes of a cluster. Valid values are 2 to 63 MT.

gdMinislotActionPointOffset

Configures the action point offset in macroticks within the minislots of the dynamic segment. Must be identical in all nodes of a cluster. Valid values are 1 to 31 MT.

gdNIT

Configures the starting point of the Network Idle Time NIT at the end of the communication cycle expressed in terms of macroticks from the beginning of the cycle. The start of NIT is recognized if Macrotick = gMacroPerCycle – gdNIT -1 and the increment pulse of Macrotick is set. Must be identical in all nodes of a cluster. Valid values of "gMacroPerCycle – gdNIT -1" are 7 to 15997 MT. Therefore, valid values for the parameter gdNIT are 2 to 805 MT.

gdStaticSlot

Configures the duration of a static slot in macroticks. The static slot length must be identical in all nodes of a cluster. Valid values are 4 to 659 MT.

gdTSSTransmitter

Configures the duration of the Transmission Start Sequence (TSS) in terms of bit times (1 bit time = 4 µT = 100ns@10Mbps). Must be identical in all nodes of a cluster. Valid values are 3 to 15 bit times.

gdWakeupSymbolRxIdle

Configures the number of bit times used by the node to test the duration of the idle phase of the received wake-up symbol. Must be identical in all nodes of a cluster. Valid values are 14 to 59 bit times.

gdWakeupSymbolRxLow

Configures the number of bit times used by the node to test the duration of the low phase of the received wake-up symbol. Must be identical in all nodes of a cluster. Valid values are 10 to 55 bit times.

gdWakeupSymbolRxWindow

Configures the number of bit times used by the node to test the duration of the received wake-up pattern. Must be identical in all nodes of a cluster. Valid values are 76 to 301 bit times.

gdWakeupSymbolTxIdle

Configures the number of bit times used by the node to transmit the idle phase of the wake-up symbol. Must be identical in all nodes of a cluster. Valid values are 45 to 180 bit times.

gdWakeupSymbolTxLow

Configures the number of bit times used by the node to transmit the low phase of the wake-up symbol. Must be identical in all nodes of a cluster. Valid values are 15 to 60 bit times.

gColdStartAttempts

Configures the maximum number of attempts that a cold starting node is permitted to try to start-up the network without receiving any valid response from another node. It can be modified in DEFAULT_CONFIG or CONFIG state only. Must be identical in all nodes of a cluster. Valid values are 2 to 31.

gListenNoise

Configures the upper limit for start-up and wake-up listen timeout in the presence of noise expressed as a multiple of `pdListenTimeout`. The range for gListenNoise is 2 to 16.

gMacroPerCycle

Configures the duration of one communication cycle in macroticks. The cycle length must be identical in all nodes of a cluster. Valid values are 10 to 16000 MT.

gMaxWithoutClockCorrectionFatal

Defines the number of consecutive even/odd cycle pairs with missing clock correction terms that will cause a transition from NORMAL_ACTIVE or NORMAL_PASSIVE to HALT state. Must be identical in all nodes of a cluster. Valid values are 1 to 15 cycle pairs.

gMaxWithoutClockCorrectionPassive

Defines the number of consecutive even/odd cycle pairs with missing clock correction terms that will cause a transition from NORMAL_ACTIVE to NORMAL_PASSIVE state. Must be identical in all nodes of a cluster. Valid values are 1 to 15 cycle pairs.

gNetworkManagementVectorLength

Configures the length of the NM vector. The configured length must be identical in all nodes of a cluster. Valid values are 0 to 12 bytes.

gNumberOfMinislots

Configures the number of minislots within the dynamic segment of a cycle. The number of minislots must be identical in all nodes of a cluster. Valid values are 0 to 7986.

gNumberOfStaticSlots

Configures the number of static slots in a cycle. At least 2 coldstart nodes must be configured to start-up a FlexRay network. The number of static slots must be identical in all nodes of a cluster. Valid values are 2 to 1023.

gOffsetCorrectionStart

Determines the start of the offset correction within the NIT phase, calculated from start of cycle. Must be identical in all nodes of a cluster. Valid values are 9 to 15999 MT.

gPayloadLengthStatic

Configures the cluster-wide payload length for all frames sent in the static segment in double bytes. The payload length must be identical in all nodes of a cluster. Valid values are 0 to 127.

gSyncNodeMax

Maximum number of frames within a cluster with sync frame indicator Bit SYN set to ’1’. Must be identical in all nodes of a cluster. Valid values are 2 to 15.

pdAcceptedStartupRange

Number of microticks constituting the expanded range of measured deviation for start-up frames during integration. Valid values are 0 to 1875 µT.

pdListenTimeout

Configures wake-up/start-up listen timeout in µT. The range for pdListenTimeout is 1284 to 1283846 µT.

pdMaxDrift

Maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle in µT. Valid values are 2 to 1923 µT.

pAllowHaltDueToClock

Controls the transition to HALT state due to a clock synchronization error. Valid values are 0 to 1. If a clock sync error occurred the CC will enter HALT state or enter/remain in NORMAL_PASSIVE state.

pAllowPassiveToActive

Defines the number of consecutive even/odd cycle pairs that must have valid clock correction terms before the CC is allowed to transit from NORMAL_PASSIVE to NORMAL_ACTIVE state. If set to zero the CC is not allowed to transit from NORMAL_PASSIVE to NORMAL_ACTIVE state. It can be modified in DEFAULT_CONFIG or CONFIG state only. Valid values are 0 to 31 even/odd cycle pairs.

pChannelsMTS

Selects channels for MTS symbol transmission. The flag is reset by default and may be modified only in DEFAULT_CONFIG or CONFIG state.

pChannels

Configures which channel the node is connected to.

pClusterDriftDamping

Configures the cluster drift damping value used in clock synchronization to minimize accumulation of rounding errors. Valid values are 0 to 20 µT.

pDecodingCorrection

Configures the decoding correction value used to determine the primary time reference point. Valid values are 14 to 143 µT.

pDelayCompensationA

Used to compensate for reception delays on the indicated channel. This covers assumed propagation delay up to `cPropagationDelayMax` for microticks in the range of 0.0125 to 0.05µs. In practice, the minimum of the propagation delays of all sync nodes should be applied. Valid values are 0 to 200 µT.

pDelayCompensationB

Used to compensate for reception delays on the indicated channel. This covers assumed propagation delay up to `cPropagationDelayMax` for microticks in the range of 0.0125 to 0.05µs. In practice, the minimum of the propagation delays of all sync nodes should be applied. Valid values are 0 to 200 µT.

pExternOffsetCorrection

Holds the external offset correction value in microticks to be applied by the internal clock synchronization algorithm. The value is subtracted / added from / to the calculated offset correction value. The value is applied during NIT. May be modified in DEFAULT_CONFIG or CONFIG state only. Valid values are 0 to 7 µT.

pExternRateCorrection

Holds the external rate correction value in microticks to be applied by the internal clock synchronization algorithm. The value is subtracted / added from / to the calculated rate correction value. The value is applied during NIT. May be modified in DEFAULT_CONFIG or CONFIG state only. Valid values are 0 to 7 µT.

pKeySlotUsedForStartup

Defines whether the key slot is used to transmit start-up frames. The Bit can be modified in DEFAULT_CONFIG or CONFIG state only.

1 = Key slot used to transmit start-up frame, node is leading or following coldstarter

0 = No start-up frame transmission in key slot, node is non-coldstarter

Not used during configuration. Is set when configuring a message buffer.

pKeySlotUsedForSync

Defines whether the key slot is used to transmit sync frames. The Bit can be modified in DEFAULT_CONFIG or CONFIG state only.

1 = Key slot used to transmit sync frame, node is sync node

0 = No sync frame transmission in key slot, node is neither sync nor coldstart node

Not used during configuration. Is set when configuring a message buffer.

pLatestTx

Configures the maximum minislot value allowed before inhibiting frame transmission in the dynamic segment of the cycle. There is no transmission in dynamic segment if it is set to zero. Valid values are 0 to 7981 minislots.

pMacroInitialOffsetA

Configures the number of macroticks between the static slot boundary and the subsequent macrotick boundary of the secondary time reference point based on the nominal macrotick duration. Must be identical in all nodes of a cluster. Valid values are 2 to 72 MT.

pMacroInitialOffsetB

Configures the number of macroticks between the static slot boundary and the subsequent macrotick boundary of the secondary time reference point based on the nominal macrotick duration. Must be identical in all nodes of a cluster. Valid values are 2 to 72 MT.

pMicroInitialOffsetA

Configures the number of microticks between the actual time reference point on channel A and the subsequent macrotick boundary of the secondary time reference point. The parameter depends on pDelayCompensationA and therefore has to be set for each channel independently. Valid values are 0 to 240 µT.

pMicroInitialOffsetB

Configures the number of microticks between the actual time reference point on channel B and the subsequent macrotick boundary of the secondary time reference point. The parameter depends on pDelayCompensationB and therefore has to be set for each channel independently. Valid values are 0 to 240 µT.

pMicroPerCycle

Configures the duration of the communication cycle in microticks. Valid values are 640 to 640000 µT.

pOffsetCorrectionOut

Holds the maximum permitted offset correction value to be applied by the internal clock synchronization algorithm (absolute value). The CC checks only the internal offset correction value against the maximum offset correction value. Valid values are 5 to 15266 µT.

pRateCorrectionOut

Holds the maximum permitted rate correction value to be applied by the internal clock synchronization algorithm. The CC checks only the internal rate correction value against the maximum rate correction value (absolute value). Valid values are 2 to 1923 µT.

pSingleSlotEnabled

Selects the initial transmission slot mode. In SINGLE slot mode the CC may only transmit in the preconfigured key slot.
1 = SINGLE Slot Mode (default after hard reset)
0 = ALL Slot Mode.

pWakeupChannel

With this Bit the Host selects the channel on which the CC sends the Wakeup pattern. The CC ignores any attempt to change the status of this Bit when not in DEFAULT_CONFIG or CONFIG state.
1 = Send wake-up pattern on channel B
0 = Send wake-up pattern on channel A

pWakeupPattern

Configures the number of repetitions (sequences) of the Tx wake-up symbol. Valid values are 2 to 63.

vExternOffsetControl

By setting this parameter the external offset correction is enabled as specified below. Should be modified only outside NIT.
00, 01 = No external offset correction
10 = External offset correction value subtracted from calculated offset correction value
11 = External offset correction value added to calculated offset correction value.

vExternRateControl

By setting this parameter the external rate correction is enabled as specified below. Should be modified only outside NIT.
00, 01 = No external rate correction
10 = External rate correction value subtracted from calculated rate correction value
11 = External rate correction value added to calculated rate correction value.

Reserved[16]

Reserved Dwords for possible later use.

See Also

fcbFRSetCcConfiguration, **fcbFRGetCcConfiguration**

5.3.3.2 fcMsgBufCfgFifo

This structure specifies the configuration of a FIFO buffer. The FIFO message buffers are used to receive FlexRay frames from different communication slots and allow therefore to receive more frames than message buffers exist.

```
Typedef struct fcMsgBufCfgFifo
{
      fcDword FrameIdFilter : 11;
      fcDword RejectionMask : 11;
      fcDword PayloadLengthConfigured : 7;
      fcDword RejectInStaticSegment : 1;
      fcDword RejectNullFrames : 1;
      fcDword Reserved;
}fcMsgBufCfgFifo;
```

Members

FrameIdFilter

Defines the acceptance filter used for frame id rejection. A zero value means that no frame is rejected. It is recommended to use the extra acceptance and rejection filter functions and leave this parameter to zero.

RejectionMask

Specifies the relevant bits used for rejection filtering. It is recommended to use the extra acceptance and rejection filter functions and leave this parameter to zero.

PayloadLengthConfigured

Defines the maximum number of 2-byte payload words received.

RejectInStaticSegment

Set this flag to 1 to reject all received static frames of the FIFO. A zero value deactivates the FIFO static segment rejection.

RejectNullFrames

Set this flag to 1 to reject all received null frames of the FIFO. A zero value deactivates the FIFO null frame rejection.

Reserved

Reserved for future use.

	Information
ℹ	Modifying the FIFO configuration may affect the ability to receive all frames (e.g. setting the *RejectInStaticSegment* flag to 1 will disable the FlexCard to monitor frames in the static segment). Configuring (**fcbFRConfigureMessageBuffer**) the FIFO is only possible when the Communication Controller is in its configuration state, *fcStateConfig*. A reconfiguration (**fcbFRReconfigureMessageBuffer**) is allowed for this buffer type. The FIFO can be accessed with buffer ID 1 (if buffer ID 1 was not reconfigured to a different buffer type by user).

See Also

fcMsgBufCfg

Example

```
// Configure fifo receive buffers
// -> Channels A+B, all frames (including null frames) on every cycles

fcMsgBufCfg cfg;
cfg.Type = fcMsgBufFifo;
cfg.ChannelFilter = fcChannelBoth;
cfg.CycleCounterFilter = 0;

cfg.Fifo.FrameIdFilter = 0;
cfg.Fifo.RejectionMask = 0;
cfg.Fifo.PayloadLengthConfigured = 127;
cfg.Fifo.RejectInStaticSegment = 0;
cfg.Fifo.RejectNullFrames = 0;

unsigned int bufferIdx = 0;
fcError e = fcbFRConfigureMessageBuffer(hFlexCard,fcCC1,&bufferIdx,cfg);
```

5.3.3.3 fcMsgBufCfgRx

This structure specifies the configuration of a receive message buffer. This buffer type should be used to analyse a specific communication slot (=frame id).

```
Typedef struct fcMsgBufCfgRx
{
        fcDword FrameId : 11;
        fcDword PayloadLengthConfigured : 7;
        fcDword PayloadLengthMax : 7;
        fcDword RxAcknowledgeEnable: 1;
        fcDword Reserved;
} fcMsgBufCfgRx;
```

Members

FrameId

Defines the slot (=frame id) to be received in this message buffer. With the function **fcbFRReconfigureMessageBuffer**, this parameter can be changed while monitoring is active.

PayloadLengthConfigured

Defines the number of 2-byte payload words to be received. This parameter can be changed while monitoring is active. To do so, call the function **fcbFRReconfigureMessageBuffer** and set this parameter with a value between 0 and *PayloadLengthMax.* The reconfiguration of this parameter for message buffers assigned to the static segment is only allowed with *fcFRMsgBufCfgModeUnequalStaticPayloadLength*.

PayloadLengthMax

Defines the maximum payload reserved for this buffer in the message ram. This E-Ray specific parameter sets the range for the payload reconfiguration. This parameter can not be changed while monitoring is active.

RxAcknowledgeEnable

This flag is obsolete and can be ignored.
Enables message buffer interrupt. This flag must be set to 1 to allow the function **fcbReceive** to get the received frame. This parameter can be changed while monitoring is active. To do so, call the function **fcbFRReconfigureMessageBuffer**.

Reserved

Reserved for future use.

See Also

fcMsgBufCfg

	Information
	FlexCards cannot receive null frames with receive message buffers. For receiving null frames a matched FIFO message buffer configuration is necessary.

5.3.3.4 fcMsgBufCfgTx

This structure specifies the configuration of a transmit message buffer. This buffer type is used to transmit a frame on a specific communication slot.

```
Typedef struct fcMsgBufCfgTx
{
       fcDword FrameId : 11;
       fcDword PayloadLength : 7;
       fcDword PayloadLengthMax : 7;
       fcDword PayloadPreambleIndicator : 1;
       fcDword SyncFrameIndicator : 1;
       fcDword StartupFrameIndicator : 1;
       fcDword TxAcknowledgeEnable: 1;
       fcMsgBufTxMode TransmissionMode;
       fcDword TxAcknowledgeShowNullFrames : 1;
       fcDword TxAcknowledgeShowPayload : 1;
       fcDword Reserved : 29;
} fcMsgBufCfgTx;
```

Members

FrameId

Defines the slot (=frame id) assigned to the transmit message buffer. With the function **fcbFRReconfigureMessageBuffer**, this parameter can be changed while monitoring is active.

PayloadLength

Defines the number of 2-byte payload words to be transmitted. This parameter can be changed while monitoring is active. To do so, call the function **fcbFRReconfigureMessageBuffer** and set this parameter with a value between 0 and *PayloadLengthMax.* The reconfiguration of this parameter for message buffers assigned to the static segment is only allowed with *fcFRMsgBufCfgModeUnequalStaticPayloadLength*.

PayloadLengthMax

Defines the maximum payload reserved for this buffer in the message ram. This E-Ray specific parameter sets the range for the payload reconfiguration. This parameter can not be changed while monitoring is active.

PayloadPreambleIndicator

This parameter is protocol specific. For more information, refer to FlexRay Protocol Specification. With the function **fcbFRReconfigureMessageBuffer**, this parameter can be changed while monitoring is active.

SyncFrameIndicator

Set this flag to 1 to indicate that the frame is a sync frame. This parameter can not be changed while monitoring is active.

StartupFrameIndicator

Set this flag to 1 to indicate that the frame is a start-up frame. This parameter can not be changed while monitoring is active.

TxAcknowledgeEnable

Set this flag to 1 to get an acknowledge packet (**fcTxAcknowledgePacket**) once a frame is transmitted (includes null frames). With the function **fcbFRReconfigureMessageBuffer**, this parameter can be changed while monitoring is active. This feature is only available on FlexCard based on E-Ray Communication Controller.

TransmissionMode

Type of transmission (refer to **fcMsgBufTxMode**). With the function **fcbFRReconfigureMessageBuffer**, this parameter can be changed while monitoring is active.

TxAcknowledgeShowNullFrames

Set this flag to 1 to get TxAcknowledge packet for transmitted null frames. This flag is only evaluated if the `TxAcknowledgeEnable` flag is set.

TxAcknowledgeShowPayload

Set this flag to 1 to get the payload of the transmitted frame. The payload length of generated TxAcknowledge packet will otherwise be set to zero. This flag is only evaluated if the `TxFrameEnable` flag is set.

Reserved

Reserved for future use

See Also

fcMsgBufCfg

5.3.3.5 fcMsgBufCfg

This structure describes the configuration of a message buffer.

```
Typedef struct fcMsgBufCfg
{
	fcMsgBufType Type;
	fcChannel ChannelFilter;
	fcDword CycleCounterFilter : 7;

	union
	{
		fcMsgBufCfgFifo Fifo;
		fcMsgBufCfgRx   Rx;
		fcMsgBufCfgTx   Tx;
	};

	fcDword Reserved[2];
} fcMsgBufCfg;
```

Members

Type

Defines the buffer type (FIFO, receive or transmit buffer)

ChannelFilter

Defines the channel(s) assigned to this buffer. With the function **fcbFRReconfigureMessageBuffer**, this parameter can only be changed while monitoring is active for receive and transmit buffer. For the configuration of a transmit buffer or a receive message buffer assigned to a dynamic frame *fcChannelBoth* isn't allowed. With *fcChannelNone* the buffer can be en- or disabled during an active monitoring. The reconfiguration of this parameter with *fcChannelNone* is only allowed with *fcFRMsgBufCfgModeReconfigurationDuringMonitoring*.

CycleCounterFilter

Defines the filter used by the message buffer for cycle counter filtering. A zero value means that no cycle counter filtering is used. The cycle counter filter is composed of two parameters. The first one determines the cycle repetition and the second one the offset (the first cycle). The cycle repetition must be given in the form of 2^x where x is a number between 0 and 6. The offset must be less than the cycle repetition value. The two values are added. With the function **fcbFRReconfigureMessageBuffer**, this parameter can only be changed while monitoring is active for receive and transmit buffer.

Fifo

FIFO buffer configuration

Rx

Receive buffer configuration

Tx

Transmit buffer configuration

Reserved

Reserved for future use

See Also

fcbFRConfigureMessageBuffer, **fcbFRReconfigureMessageBuffer**, **fcbFRGetMessageBuffer**, **fcMsgBufType**, **fcMsgBufCfgFifo**, **fcMsgBufCfgRx**, **fcMsgBufCfgTx**, **fcFRMsgBufCfgMode**

Example

```
// The following code configures a transmit buffer, which only transmits on cycles
6,14,22,30, …

fcMsgBufCfg cfg;
cfg.Type = fcMsgBufTx;
cfg.ChannelFilter = fcChannelA;


// Repetition: each 8 cycles
// Offset: 6 (First cycle will be cycle number 6)

cfg.CycleCounterFilter = 0x8 + 0x6;

cfg.Tx.FrameId = 61;
cfg.Tx.PayloadLength = 10;
cfg.Tx.PayloadLengthMax = 127;
cfg.Tx.PayloadPreambleIndicator = 0;
cfg.Tx.SyncFrameIndicator = 0;
cfg.Tx.StartupFrameIndicator = 0;
cfg.Tx.TxAcknowledgeEnable= 0;
cfg.Tx.TransmissionMode = fcMsgBufTxSingleShot;


unsigned int bufferIdx = 0;
fcError e = fcbFRConfigureMessageBuffer(hFlexCard,fcCC1,&bufferIdx,cfg);
```

```
// The following code configures 2 sync frame buffers with the same frame id // but
with different channelfilters. This way a different payload can be sent // on the sync
frame id on channel a and on channel b.

fcMsgBufCfg cfg;
cfg.Type = fcMsgBufTx;
//first message buffer on channel a
cfg.ChannelFilter = fcChannelA;
cfg.CycleCounterFilter = 0;
cfg.Tx.FrameId = 1;//this id has to be in the static range
cfg.Tx.PayloadLength = 16;//payload length in the static segment
cfg.Tx.PayloadLengthMax = 16;
cfg.Tx.PayloadPreambleIndicator = 0;
cfg.Tx.SyncFrameIndicator = 1;
cfg.Tx.StartupFrameIndicator = 1;
cfg.Tx.TxAcknowledgeEnable= 1;
cfg.Tx.TransmissionMode = fcMsgBufTxSingleShot;

unsigned int bufferIdx = 0;
fcError e = fcbFRConfigureMessageBuffer(hFlexCard,fcCC1,&bufferIdx,cfg);
```

```
//second message buffer on channel b
cfg.ChannelFilter = fcChannelB;
e = fcbFRConfigureMessageBuffer(hFlexCard,fcCC1,&bufferIdx,cfg);
```

5.3.3.6 fcCcTimerCfg

This structure describes the configuration of a Communication Controller timer.

```
Typedef struct fcCcTimerCfg
{
        fcDword ContinuousMode : 1;
        fcDword CycleCounterFilter : 7;
        fcDword MacrotickOffset : 14;
} fcCcTimerCfg;
```

Members

ContinuousMode

Defines the Communication Controller timer mode. Set to 1 for continuous mode or 0 for single-shot mode.

CycleCounterFilter

Defines the filter used by the CC timer for cycle counter filtering. A zero value means that no cycle counter filtering is used. The cycle counter filter is composed of two parameters. The first one determines the cycle repetition and the second one the offset (the first cycle). The cycle repetition must be given in the form of 2^x where x is a number between 0 and 6. The offset must be less than the cycle repetition value.

MacrotickOffset

Defines the macrotick offset from the beginning of the cycle when the CC timer interrupt must occur. The CC timer interrupt occurs at this offset for each cycle of the cycle counter filter.

See Also

fcbFRSetCcTimerConfig, **fcbFRGetCcTimerConfig**, **fcbFRCalculateMacrotickOffset**

	Information
ℹ	This structure is initially supported by FlexCard API version S4V0-F.

5.3.4 fcbFRSetCcRegister

This function writes a value in each register of the selected Communication Controller. Not every register can be written (e.g. the registers belonging to the message buffer configuration or some interrupt settings).

```
fcError fcbFRSetCcRegister(
        fcHandle hFlexCard,
        fcCC CC,
        fcDword address,
        fcDword value
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

address

[IN] Address of the CC register to be written. Must be a multiple of 4 bytes, otherwise an error will be returned

value

[IN] The value to be written

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information. If the register can not be written the error code `REGISTER_NOT_WRITEABLE` is returned.

Remarks

For a register description, refer to the specification of the corresponding Communication Controller. Modifying one of the following registers will reset message buffers with their default settings (FIFO receive buffers). The user's message buffers configuration will not be valid anymore.
Bosch E-Ray: MHDC (0x0098) and GTUC7 (0x00B8).

On FlexCard driver version Windows S6V4-F or later, on E-Ray, the register GTUC11 (0x00C8) may be read/written when monitoring is not activated. After monitoring, it may be read/written, but for the write operation, only External Offset Correction Control and External Rate Correction Control may be changed. External Offset Correction and External Rate Correction may not be modified.

ℹ	**Information**
	Not all registers of a Communication Controller can be set. The base API will modify some parameters so that the operating of the FlexCard is guaranteed (e.g. interrupt settings). Access is denied to all registers which are used for message buffer configuration.

See Also

`fcCC`, **`fcbFRGetCcRegister`**

5.3.5 fcbFRGetCcRegister

This function reads and returns the content of a given register of the selected Communication Controller.

On the FlexCard PXIe3/PCIe3 in the FlexDevice mode, this function is only allowed for debugging purposes. It should not be used in this case because the decoder and the communication controller should be independent.

```
fcError fcbFRGetCcRegister(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword address,
      fcDword* pValue
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

`CC`

[IN] Communication controller index

`address`

[IN] Address of the CC register to be read. Must be a multiple of 4 bytes, otherwise an error will be returned

`pValue`

[OUT] The content of the desired CC register.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information. If the register cannot be read the error code `REGISTER_NOT_READABLE` is returned.

Remarks

Not every register can be read. For a register description, refer to the specification of the corresponding Communication Controller.

See Also

fcCC, **fcbFRSetCcRegister**

Example

```
fcDword value = 0xFFFFFFFF;
fcDword address = 0x0B8;
fcCC eCC = fcCC1;

if (0 != address % 4) return; //address not a multiple of 4 bytes!

fcError e = fcbFRGetCcRegister(hFlexCard,eCC,address,&value);
if (0 == e)
{
      printf("Register 0x%X=0x%X", address, value);
}
```

5.3.6 fcbFRSetCcConfigurationChi

This function configures the selected Communication Controller of the FlexCard with a FlexConfig compatible configuration string (CHI File). The configuration string contains the global FlexRay parameter and/or the message buffer configuration. The payload data for transmit message buffers is not set by this function. Before the configuration of the Communication Controller starts, all message buffers are reset to their default settings (FIFO buffer).

```
fcError fcbFRSetCcConfigurationChi(
      fcHandle hFlexCard,
      fcCC CC,
      const char* szChi
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index

szChi

[IN] Pointer to null-terminated CHI content string (refer to the CHI string example section).

Please note: Do not use the CHI file name here, but the content of the CHI file as parameter value.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

	Information
ℹ	Internally, the function uses the **fcbFRSetCcRegister** function; therefore the same restrictions as for writing registers exist.

See Also

fcCC, **fcbFRSetCcRegister**

Example

See **fcbFRSetCcConfigurationChi**

5.3.7 fcbFRSetCcConfiguration

This function configures the FlexRay Communication Controller.

```
fcError fcbFRSetCcConfiguration(
       fcHandle hFlexCard,
       fcCC CC,
       fcFRCcConfig cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index.

Cfg

[IN] The FlexRay Communication Controller configuration.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, fcFRCcConfig, fcbFRGetCcConfiguration

Example

```
fcCC eCC = fcCC1;
fcFRCcConfig frCcConfigSet;
memset(&frCcConfigSet, 0, sizeof(fcFRCcConfig));

// SUCC1
frCcConfigSet.pKeySlotUsedForStartup = 0;
frCcConfigSet.pKeySlotUsedForSync = 0;
frCcConfigSet.gColdStartAttempts = 31;
frCcConfigSet.pAllowPassiveToActive = 0;
frCcConfigSet.pWakeupChannel = fcChannelA;
frCcConfigSet.pSingleSlotEnabled = 0;
frCcConfigSet.pAllowHaltDueToClock = 1;
frCcConfigSet.pChannelsMTS = fcChannelNone;
frCcConfigSet.pChannels = fcChannelBoth;

// SUCC2
frCcConfigSet.pdListenTimeout = 80242 ;
frCcConfigSet.gListenNoise = 2 ;

// SUCC3
frCcConfigSet.gMaxWithoutClockCorrectionPassive = 2;
frCcConfigSet.gMaxWithoutClockCorrectionFatal = 2;

// NEMC
frCcConfigSet.gNetworkManagementVectorLength = 0;

// PRTC1
frCcConfigSet.gdTSSTransmitter = 7;
frCcConfigSet.gdCASRxLowMax = 99;
frCcConfigSet.BaudRate = fcFRBaudRate10M;
frCcConfigSet.gdWakeupSymbolRxWindow = 301;
frCcConfigSet.pWakeupPattern = 2;

// PRTC2
frCcConfigSet.gdWakeupSymbolRxIdle = 59;
frCcConfigSet.gdWakeupSymbolRxLow = 54;
frCcConfigSet.gdWakeupSymbolTxIdle = 180;
frCcConfigSet.gdWakeupSymbolTxLow = 60;

// MHDC
frCcConfigSet.gPayloadLengthStatic = 4;
frCcConfigSet.pLatestTx = 0;

// GTUC1
```

```
frCcConfigSet.pMicroPerCycle = 40000;

// GTUC2
frCcConfigSet.gMacroPerCycle = 1000;
frCcConfigSet.gSyncNodeMax = 2;

// GTUC3
frCcConfigSet.pMicroInitialOffsetA = 0;
frCcConfigSet.pMicroInitialOffsetB = 0;
frCcConfigSet.pMacroInitialOffsetA = 2;
frCcConfigSet.pMacroInitialOffsetB = 2;

// GTUC4
frCcConfigSet.gdNIT = 40;
frCcConfigSet.gOffsetCorrectionStart = 991;

// GTUC5
frCcConfigSet.pDelayCompensationA = 0;
frCcConfigSet.pDelayCompensationB = 0;
frCcConfigSet.pClusterDriftDamping = 1;
frCcConfigSet.pDecodingCorrection = 40;

// GTUC6
frCcConfigSet.pdAcceptedStartupRange = 258;
frCcConfigSet.pdMaxDrift = 121;

// GTUC7
frCcConfigSet.gdStaticSlot = 22;
frCcConfigSet.gNumberOfStaticSlots = 43;

// GTUC8
frCcConfigSet.gdMinislot = 11;
frCcConfigSet.gNumberOfMinislots = 0;

// GTUC9
frCcConfigSet.gdActionPointOffset = 1;
frCcConfigSet.gdMinislotActionPointOffset = 5;
frCcConfigSet.gdDynamicSlotIdlePhase = 2;

// GTUC10
frCcConfigSet.pOffsetCorrectionOut = 81;
frCcConfigSet.pRateCorrectionOut = 121;

// GTUC11
frCcConfigSet.vExternOffsetControl = 0;
frCcConfigSet.vExternRateControl = 0;
frCcConfigSet.pExternOffsetCorrection = 0;
frCcConfigSet.pExternRateCorrection = 0;

// Configure the FlexRay CC
e = fcbFRSetCcConfiguration(hFlexCard, eCC, frCcConfigSet);
if (0 != e) {/* Error handling */};
```

5.3.8 fcbFRGetCcConfiguration

This function reads the FlexRay Communication Controller configuration.

```
fcError fcbFRGetCcConfiguration (
      fcHandle hFlexCard,
      fcCC CC,
      fcFRCcConfig* pCfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index.

pCfg

[OUT] Pointer to the configuration parameters of the FlexRay Communication Controller.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcFRCcConfig**, **fcbFRSetCcConfiguration**

Example

```
fcCC eCC = fcCC1;
fcFRCcConfig frCcConfigGet;

e = fcbFRGetCcConfiguration(hFlexCard, eCC, &frCcConfigGet);
if (0 != e) {/* Error handling */};
```

5.3.9 fcbFRSetMsgBufCfgMode

This function configures the fcBase APIs message buffer configuration handling for the FlexRay Communication Controllers. The message buffer configuration mode can be changed while monitoring is active.

```
fcError fcbFRSetMsgBufCfgMode(
      fcHandle hFlexCard,
      fcFRMsgBufCfgMode mode
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

Mode

[IN] The message buffer configuration mode.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcFRMsgBufCfgMode, **fcMsgBufCfg**, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**

ℹ	Information
	This function is initially supported by FlexCard API version S6V1-F.

5.3.10 fcbFRConfigureMessageBuffer

This function configures transmit, receive and FIFO message buffers of the selected Communication Controller. Configuring message buffers is only allowed if the Communication Controller is in its configuration state, `fcStateConfig`.

```
fcError fcbFRConfigureMessageBuffer(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword* pBufferId,
      fcMsgBufCfg cfg
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

pBufferId

[OUT] Message buffer identifier. If the configuration was successful, the message buffer identifier is greater than 0. This identifier will be required to transmit the content of the buffer (in the case of a transmit buffer).

Cfg

[IN] Message buffer configuration parameters

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

Before configuring the message buffers, it is necessary to set up the global communication parameters (cluster parameters). Internally the FlexCard uses the FIFO buffers as receive buffers, therefore we recommend using FIFO message buffers as much as possible.

See Also

fcCC, **fcMsgBufCfg**, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**, **fcMsgBufCfgFifo**, **fcbFRSetMsgBufCfgMode**

Example

```
// The following code configures a transmit buffer,
// which only transmits on cycles 6,14,22,30, …

fcMsgBufCfg cfg;
memset(&cfg, 0, sizeof(fcMsgBufCfg));
cfg.Type = fcMsgBufTx;
cfg.ChannelFilter = fcChannelA;


// Repetition: each 8 cycles
// Offset: 6 (First cycle will be cycle number 6)

cfg.CycleCounterFilter = 0x8 + 0x6;

cfg.Tx.FrameId = 61;
cfg.Tx.PayloadLength = 10;
cfg.Tx.PayloadLengthMax = 127;
cfg.Tx.PayloadPreambleIndicator = 0;
cfg.Tx.SyncFrameIndicator = 0;
cfg.Tx.StartupFrameIndicator = 0;
cfg.Tx.TxAcknowledgeEnable= 0;
cfg.Tx.TransmissionMode = fcMsgBufTxSingleShot;

fcCC eCC = fcCC1;
unsigned int bufferIdx = 0;
fcError e = fcbFRConfigureMessageBuffer(hFlexCard,eCC,&bufferIdx,cfg);
```

5.3.11 fcbFRReconfigureMessageBuffer

This function reconfigures transmit, receive and FIFO message buffers of the selected Communication Controller. A reconfiguration is only allowed for message buffers which are already configured. This function is available in all states of the CC. Not all configuration settings can be modified in monitoring state. Refer to the documentation of the message buffer structures for further details.

```
fcError fcbFRReconfigureMessageBuffer(
       fcHandle hFlexCard,
       fcCC CC,
       fcDword bufferId,
       fcMsgBufCfg cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

bufferId

[IN] The identifier of the message buffer which should be reconfigured.

Cfg

[IN] Message buffer configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcMsgBufCfg**, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**, **fcMsgBufCfgFifo**, **fcbFRConfigureMessageBuffer**, **fcbFRGetMessageBuffer**, **fcbFRSetMsgBufCfgMode**

5.3.12 fcbFRGetMessageBuffer

This function reads a specific message buffer configuration.

```
fcError fcbFRGetMessageBuffer(
       fcHandle hFlexCard,
       fcCC CC,
       fcDword bufferId,
       fcMsgBufCfg* pCfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

bufferId

[IN] The identifier of the message buffer to be read

pCfg

[OUT] The configuration parameters of the specified message buffer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcMsgBufCfg**, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**, **fcMsgBufCfgFifo**, **fcbFRConfigureMessageBuffer**

	Information
ℹ	The buffer with id 1 is always a FIFO message buffer.

Example

```
// Get all configured transmit message buffers of Communication Controller 1
fcCC eCC = fcCC1;
std::map<unsigned int, fcMsgBufCfg> Buffers;
unsigned int bufferIdx = 1; // The first valid buffer is 1
while (true)
{
      fcMsgBufCfg cfg;

      // as long no error occurs we try to get each buffer
      fcError e = fcbFRGetMessageBuffer(m_hFlexCard,eCC,bufferIdx,&cfg);
      if (0 != e) break;

      // is this a tx buffer, then add it to our list
      if (fcMsgBufTx == cfg.Type)    Buffers[bufferIdx] = cfg;

      // next buffer index
      bufferIdx++;
}
```

5.3.13 fcbFRResetMessageBuffers

This function resets the Communication Controller message buffers. After calling this function, all message buffers are configured as receive FIFO – with maximal payload (depends on the Communication Controller).

```
fcError fcbFRResetMessageBuffers(
      fcHandle hFlexCard,
      fcCC CC
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

5.3.14 fcbFRSetSoftwareAcceptanceFilter

This function configures the frame ids accepted by the device driver. Only the FlexRay ids which are in the filter list are forwarded to the user application, all other FlrexRay frames are rejected. One filter can be defined for both channels or two filters can be defined, one for Channel A and one for Channel B. See the configuration notice for further details. The filter behavior differs from the function **fcbFRSetHardwareAcceptanceFilter**.

```
fcError fcbFRSetSoftwareAcceptanceFilter(
      fcHandle hFlexCard,
      fcCC CC,
      fcChannel channel,
      fcDword* pData,
      fcDword size
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

channel

[IN] FlexCard channel(s) concerned by the filter

pData

[IN] Pointer to an `fcDword` array containing the ids accepted by the device driver. Each element (`fcDword`) contains one frame identifier.

fcDword	fcDword	fcDword	fcDword
ID x	ID y	ID z	…

size

[IN] Number of ids in the array.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Configuration notice

channel	pData	size	Behaviour
	NULL	0	Accept all IDs.
fcChannelBoth	ID 0		Accept all IDs.
fcChannelNone	ID x		Does nothing.
fcChannelA	ID 5		Accepts only ID 5 on Channel A, but allows all frames on Channel B (including ID 5) if there wasn't another filter for Channel B defined.
fcChannelA, fcChannelB	ID 5 ID 3		Accepts only ID 5 Channel A and ID 3 Channel B, rejects all other frames.

Example

```
// Configure the filter to get only
// - the frames from frame id 15 and 60 on CC 1, channel A
// - and the frame ids 1,2,3,6 on CC 1, channel B

fcDword idsChA[2] = {15,60};
fcDword idsChB[4] = {1,2,3,6};
fcCC eCC = fcCC1;

fcError e = fcbFRSetSoftwareAcceptanceFilter(hFlexCard,eCC,fcChannelA,idsChA,2);
//…
e = fcbFRSetSoftwareAcceptanceFilter(hFlexCard,eCC,fcChannelB,idsChB,4);
```

5.3.15 fcbFRSetHardwareAcceptanceFilter

This function configures the FlexRay frame ids accepted by the FlexCard firmware. Only the FlexRay ids which are in the filter list are forwarded to the device driver, all other FlexRay frames are rejected. See the configuration notice for further details. The filter behavior differs from the function **`fcbFRSetSoftwareAcceptanceFilter`**. When using this function, receiving frames is faster than using **`fcbFRSetSoftwareAcceptanceFilter`**.

```
fcError fcbFRSetHardwareAcceptanceFilter(
      fcHandle hFlexCard,
      fcCC CC,
      fcChannel channel,
      fcDword* pData,
      fcDword size,
      fcBool reset
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index.

Channel

[IN] FlexCard channel(s) concerned by the filter.

pData

[IN] Pointer to an `fcDword` array containing the ids accepted by the device driver. Each element (`fcDword`) contains one frame identifier.

fcDword	fcDword	fcDword	fcDword
ID x	ID y	ID z	…

size

[IN] Number of ids in the array.

Reset

[IN] Set to <> 0 to reset the filter, before configuring a new filter. The hardware transmit filter and hardware acceptance filter of both channels are resetted, while the software acceptance filter is not touched. Set *reset* to 0 to add the frame identifier to the existing filter.

Configuration notice

channel	pData	size	Behaviour
	NULL	0	Accept all IDs.
fcChannelBoth	ID 0		Accept all IDs.
fcChannelNone	ID 0		Reject all IDs.
fcChannelNone	ID x		Reject ID x.
fcChannelA	ID 5		Accepts only ID 5 Channel A, rejects all other frames.
fcChannelA, fcChannelB	ID 5 ID 3		Accepts only ID 5 Channel A and ID 3 Channel B, rejects all other frames.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, fcChannel, fcbReceive

5.3.15.1 fcEthernetErrorType

This enumeration contains the different error types reported by an ethernet error packet.

```
Typedef enum fcEthernetErrorType
{
      fcEthernetErrorNone = 0,
      fcEthernetErrorFrameCheckSequence,
}fcEthernetErrorType;
```

Members

fcEthernetErrorNone

No error occurred.

fcEthernetErrorFrameCheckSequence

A frame check sequence error was detected.

See Also

fcEthernetErrorPacket

	Information
ℹ	This enumeration is initially supported by FlexCard API version S6V7-F.

fcbReceive

5.3.16 fcbFRSetHardwareTransmitFilter

This function configures the FlexRay frame ids transmitted by the FlexCard firmware. Only the FlexRay ids which are in the transmission filter configuration list are enabled or disabled for transmission. See the configuration notice for further details.

```
fcError fcbFRSetHardwareTransmitFilter(
      fcHandle hFlexCard,
      fcCC CC,
      fcChannel channel,
      fcDword* pData,
      fcDword size,
      fcBool reject,
      fcBool reset
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index.

Channel

[IN] FlexCard channel(s) concerned by the transmission filter.

pData

[IN] Pointer to an `fcDword` array containing the ids which will be configured (en-/disabled). Each element (`fcDword`) contains one frame identifier.

fcDword	fcDword	fcDword	fcDword
ID x	ID y	ID z	…

size

[IN] Number of ids in the array.

Reject

[IN] Set this value to <> 0 to disable the transmission of the ids in the array. Set the value to 0 to enable the transmission.

Reset

[IN] Set this value to <> 0 to reset the transmission filter, before configuring a new filter. The hardware transmit filter and hardware acceptance filter of both channels are resetted, while the software acceptance filter is not touched. If *reset* is set to 0 the frame identifier is added to the existing transmission filter configuration.

Configuration notice

channel	pData	size	behaviour
	NULL	0	Accept all IDs.
fcChannelBoth	ID 0		Transmit or reject transmission of all IDs (depends on reject member).

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcCC`, **`fcChannel`**, **`fcbFRTransmit`**

ℹ	Information
	This function is initially supported by FlexCard API version S6V1-F.

5.3.17 fcbFRSetCcTimerConfig

This function configures the Communication Controller timer interrupt. To get a notification when the Communication Controller timer interval elapsed, an event of type *`fcNotificationTypeFRCcTimer`* has to be registered by the function **`fcbSetEventHandleV2`**. Additionally, the Communication Controller timer can be enabled / disabled by this function.

```
fcError fcbFRSetCcTimerConfig(
        fcHandle hFlexCard,
        fcCC CC,
        fcCcTimerCfg cfg,
        fcBool bEnable
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard.

`CC`

[IN] Communication controller index

`Cfg`

[IN] The Communication Controller timer configuration.

`bEnable`

[IN] Set to <> 0 to enable the CC timer, and to 0 to disable it.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcCC`, **`fcbSetEventHandleV2`**, **`fcbSetEventHandleSemaphore`**, **`fcCcTimerCfg`**, **`fcbFRGetCcTimerConfig`**

Example

See Example **`fcbFRCalculateMacrotickOffset`**

5.3.18 fcbFRGetCcTimerConfig

This function reads the Communication Controller timer configuration.

```
fcError fcbFRGetCcTimerConfig(
      fcHandle hFlexCard,
      fcCC CC,
      fcCcTimerCfg* pCfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index

pCfg

[OUT] The configuration parameters of the CC timer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcCcTimerCfg**, **fcbFRSetCcTimerConfig**

Example

See Example **fcbFRCalculateMacrotickOffset**

5.3.19 fcbFRCalculateMacrotickOffset

This function calculates the macrotick offset for a specific cycle position in a FlexRay cycle. The function requires that the FlexRay CC was configured with the function fcbFRSetCcConfiguration or fcbFRSetCcConfigurationChi.

```
fcError fcbFRCalculateMacrotickOffset(
      fcHandle hFlexCard,
      fcCC CC,
      fcCyclePos CyclePosition,
      fcDword SlotOrMiniSlotId,
      fcDword* pValue
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index

CyclePosition

[IN] The cycle position of type **fcCyclePos**.

SlotOrMiniSlotId

[IN] This parameter is used for a cycle position of *fcCyclePosStaticSlot* and *fcCyclePosDynamicMiniSlot* to calculate the macrotick offset for a static slot or a dynamic mini slot id.

pValue

[OUT] The macrotick offset value.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcCyclePos**, **fcCcTimerCfg**, **fcbFRSetCcTimerConfig**, **fcbFRGetCcTimerConfig**

Example

```
//
// Configure the CC 1 timer to get notified of the static slot id 9 start,
// Check the configuration and start the CC 1 timer
//
fcCC eCC = fcCC1 ;
fcCcTimerCfg ccTimerConfigSet, ccTimerConfigGet ;
memset(&ccTimerConfigSet, 0, sizeof(fcCcTimerCfg)) ;
memset(&ccTimerConfigGet, 0, sizeof(fcCcTimerCfg)) ;

ccTimerConfigSet.CycleCounterFilter = 1 ;
ccTimerConfigSet.ContinuousMode = 1 ;
ccTimerConfigSet.MacrotickOffset = 0 ;

// Calculate the macrotick offset for static slot id 9
fcDword dwMTOffset = 0;
fcDword dwSlotId = 9;
fcError e = fcbFRCalculateMacrotickOffset(hFlexCard, eCC,
      fcCyclePosStaticSlot, dwSlotId, &dwMTOffset);
if (0 != e) {/* Error handling */};
else ccTimerConfigSet.MacrotickOffset = dwMTOffset;

// Configure the CC 1 timer, but don't start
e = fcbFRSetCcTimerConfig(hFlexCard, eCC, ccTimerConfigSet, false) ;
if (0 != e) {/* Error handling */};

// Read the configuration
e = fcbFRGetCcTimerConfig(hFlexCard, eCC, &ccTimerConfigGet) ;
if (0 != e) {/* Error handling */};

// Check the configured timer
if (ccTimerConfigSet.CycleCounterFilter != ccTimerConfigGet.CycleCounterFilter
      || ccTimerConfigSet.ContinuousMode != ccTimerConfigGet.ContinuousMode
      || ccTimerConfigSet.MacrotickOffset != ccTimerConfigGet.MacrotickOffset)
      {return;}

// We passed the check, now start the CC timer with this config
e = fcbFRSetCcTimerConfig(hFlexCard, eCC, ccTimerConfigSet, true);
if (0 != e) {/* Error handling */};

// Wait for the CC 1 timer event ...
```

5.4 Transmit

5.4.1 Enumerations

5.4.1.1 fcSymbolType

This enumeration defines the supported communication symbols when the Communication Controller is in POC state NORMAL_ACTIVE. For more details about these symbols, please refer to the FlexRay Protocol Specification. To send a wake-up symbol (WUS) or collision avoidance symbol (CAS), refer to the function **fcbFRMonitoringStart**.

```
Typedef enum fcSymbolType
{
      fcMediaAccessTestSymbolType = 1,
} fcSymbolType;
```

Members

fcMediaAccessTestSymbolType

Media Access Test Symbol (MTS)

See Also

`fcbFRTransmitSymbol`

5.4.2 fcbFRTransmit

This function writes a data frame into a Communication Controller transmit buffer of the FlexCard. The function returns immediately and does not wait for the data frame to arrive on the bus. The frame should normally be transmitted in the next cycle. In the static segment of the FlexRay cycle, frames are transmitted in the order of the time slot. Example: If the CC is in the current slot 15 and the user transmits the static frame ID 1 and shortly after that ID 30, the CC will transmit ID 30 before ID 1. If the transmit acknowledgment is activated, an acknowledge packet is generated as soon as the frame has been transmitted. This function should only be called when the FlexCard is in normal active state or when all message buffer configurations have been done. When the user transmits several times new data with the same buffer ID in a very short time periode, it may happen that data is overwritten that was not transmitted yet. If you experience that behavior, wait for the TxAck for the data you wanted to send.

```
fcError fcbFRTransmit(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword bufferId,
      fcWord * pPayload,
      fcByte payloadLength
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

bufferId

[IN] The id of the message buffer used for the transmission

pPayload

The payload data to be transmitted

payloadLength

The size of the payload data (number of 2-byte words)

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

The payload data has to be organized as follows: if Data0 is the first byte to transmit and Data1 the second byte to transmit, then the high byte (Bit 8 – 15) of payload[0] contains Data1, the low byte (Bit 0-7) of payload[0] contains Data0, etc.

Parameter `payload`	**`payload[0]` (Word 0)**		**`payload[1]` (Word 1)**		**…**
	High byte	**Low byte**	**High byte**	**Low byte**	**…**
FlexRay payload segment	Data 1	Data 0	Data 3	Data 2	…

Example

```
fcCC eCC = fcCC1;
fcWord payload[fcPayloadMaximum];
payload[0] = 0x0001; // Update your payload data

fcError e = fcbFRTransmit(m_hFlexCard,eCC,bufferIdx,payload,payloadLength);
```

5.4.3 fcbFRTransmitSymbol

This function transmits a symbol in the symbol window segment. It can only be called if the selected Communication Controller is in the POC state NORMAL_ACTIVE. For a list of available symbols to be transmitted, see the enumeration *fcSymbolType*.

```
fcError fcbFRTransmitSymbol(
      fcHandle hFlexCard,
      fcCC CC,
      fcSymbolType type
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

type

[IN] Type of symbol to transmit

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

6 CAN API

The following section describes the data structures and features used for CAN functionality. To use these functions the FlexCard must have a firmware with a CAN CC and the FlexCard must be licensed for CAN.

	Information
ⓘ	All enumerations, structures and function in this chapter are initially supported by FlexCard Windows API version S4V0-F and FlexCard Linux/Xenomai API version S4V2-F.

6.1 Basic CAN Workflow

The following figure shows a typical CAN workflow.

The CAN functions are supported for the CAN Communication Controller type Bosch D_CAN. The Enum function returns the struct fcInfoHwSw. Check the struct fcInfoHw.pVersionCC to see whether the firmware provides this CC type.

The CCs are counted in following order: First FlexRay, then CAN. CAN-FD CCs count like CAN CCs.

For example, if the firmware image has 2 FlexRay CC and 4 CAN CCs, the CCs are referred to like this:

- FlexRay fcCC1, fcCC2
- CAN fcCC1, fcCC2, fcCC3, fcCC4

Please note that the message buffers may be reconfiguered during monitoring, but the CAN configuration may only be changed when monitoring is not active.

The API for CAN-HS and CAN-LS buses is the same. CAN-HS baud rate is 40 Kbit/s to 1 Mbit/s. CAN-LS baud rate is 5 Kbit/s to 125 Kbit/s. The FlexCard installer includes the application “CANBaudRateCalculator” which calculates the bus parameters for a selected baud rate. Refer to the chapter General Function Availability to find out if the connected FlexCard device supports onboard CAN terminations. For CAN-LS the bus termination is integrated on the PCB.

There are several ways to transmit CAN data. Using message buffers either with the function **fcbCANTransmit** or with **fcbCANSetMessageBuffer** with *enableTxRequest* = true. The parameter *ignoreTxRqstLock* should be set to false, so that you notice when data was lost during transmitting or reconfiguring.

Check for the error code that is returned from **fcbCANTransmit** and **fcbCANSetMessageBuffer**. If it is *MSG_BUF_LOCKED_FOR_TRANSMISSION*, try again. This error appears permanently when the configuration is not right, so set a limit to the number of retries.

When you want to transmit data from the start, call **fcbCANSetMessageBuffer** with *enableTxRequest* = false and *newData* = true. After having called **fcbCANMonitoringStart**, use the function **fcbCANTransmit** with *transmitNewData* = false. Using the function **fcbCANSetMessageBuffer** with *enableTxRequest* = true is not recommended before the start-up, because the data is transmitted immediately after starting the CC. When you use two CAN CCs on the FlexCard, the second CC may not be started up and misses the ID you tried to send.

When you want to send new data during monitoring, use **fcbCANSetMessageBuffer** with *enableTxRequest* = true and *newData* = true.

With FlexCard API version S6V3-F and FlexCard firmware version S6V3-F on, it is possible to transmit CAN data messages very easily. FlexCard version S6V3-F introduces a CAN transmit FIFO for every CAN Communication Controller. There are two new configurations (**fcbCANSetTxFifoConfiguration**, **fcbCANTxFifoReset**) and one new transmit function available (**fcbCANTxFifoTransmit**). With these functions CAN data messages can be configured and transmitted without reconfiguration of the corresponding message buffer objects. A configurable hardware timer helps to transmit CAN data messages to a deliberate time slot, i.d. real-time behavior with non real-time capable operating systems are possible.

The FlexCard Windows Developer Setup installs the example application *fcDemoCAN.exe* and its source code to the installation directory.

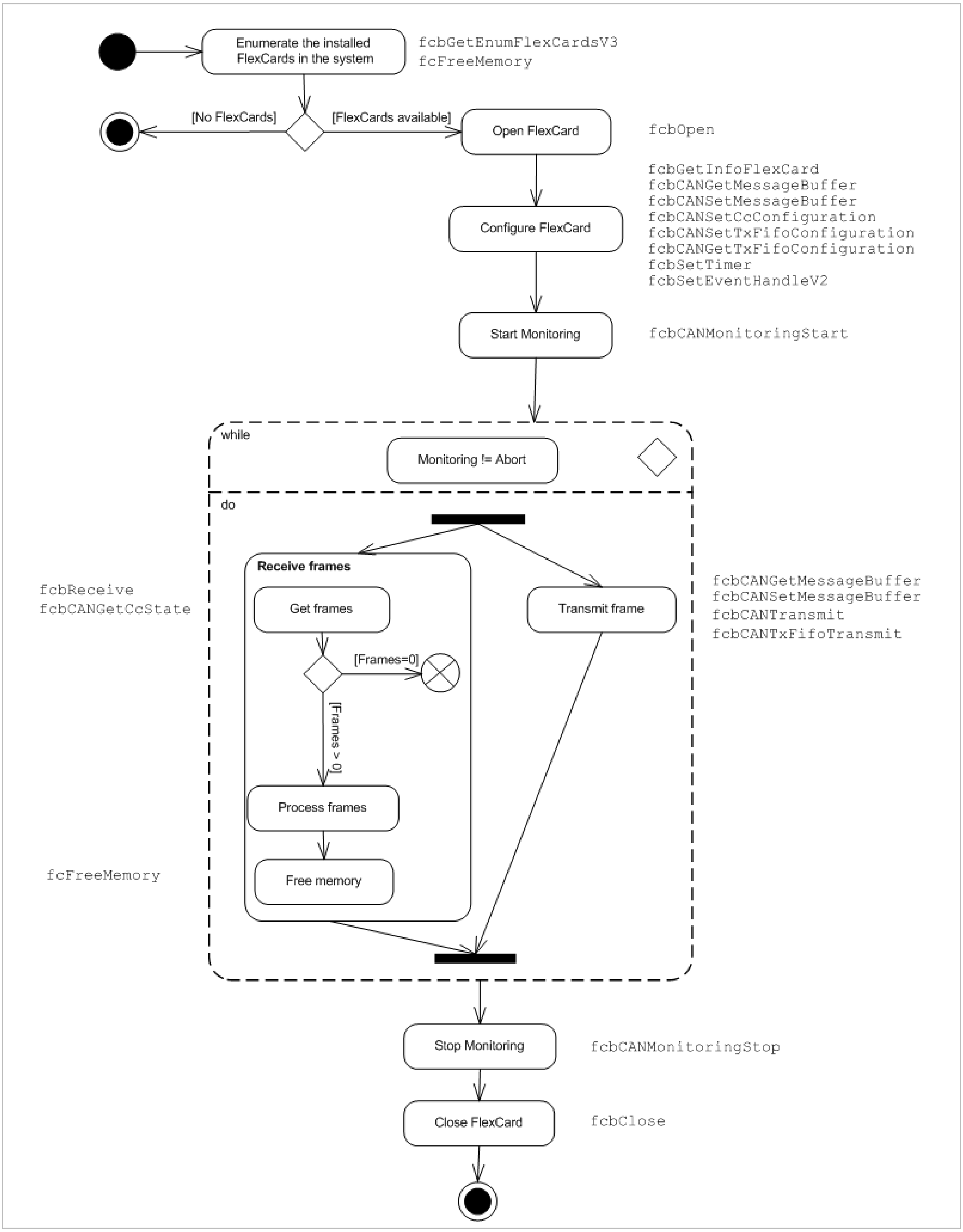


Figure 11: Typical CAN function workflow

6.2 Initialization

6.2.1 Enumerations

6.2.1.1 fcCANCcState

This enumeration defines the CAN Communication Controller states. For more details about CAN Communication Controller states, please refer to [6]/[7] (CAN Protocol Specification).

```
Typedef enum fcCANCcState
{
      fcCANCcStateUnknown = 0,
      fcCANCcStateConfig,
      fcCANCcStateNormalActive,
      fcCANCcStateWarning,
      fcCANCcStateErrorPassive,
      fcCANCcStateBusOff,
}fcCANCcState;
```

Members

fcCANCcStateUnknown

Communication controller state is unknown.

fcCANCcStateConfig

Communication controller is in configuration state.

fcCANCcStateNormalActive

Communication controller is in normal active state.

fcCANCcStateWarning

Communication controller is in error warning state. At least one of the error counters has reached the error warning limit of 96.

fcCANCcStateErrorPassive

Communication controller is in error passive state. No CAN messages can be sent anymore except CAN passive errors.

fcCANCcStateBusOff

Communication controller is in bus off state. No CAN messages can be sent anymore.

See Also

fcbCANGetCcState, **fcbCANMonitoringStart**

6.2.1.2 fcCANMonitoringMode

This enumeration defines the different modes available, used to monitor a CAN cluster.

```
Typedef enum fcCANMonitoringMode
{
      fcCANMonitoringNormal = 0,
      fcCANMonitoringActive = fcCANMonitoringNormal,
      fcCANMonitoringSilent = 1,
      fcCANMonitoringPassive = fcCANMonitoringSilent,
}fcCANMonitoringMode;
```

Members

fcCANMonitoringNormal

fcCANMonitoringActive

The FlexCard will switch from configuration to normal active state as soon as possible. In normal active state CAN frames can be received and transmitted.

fcCANMonitoringSilent

fcCANMonitoringPassive

The FlexCard will switch from configuration to normal passive state as soon as possible. In normal passive state CAN frames can be received only.

See Also

`fcbCANMonitoringStart`

6.2.2 fcbCANMonitoringStart

This function is used to start the monitoring of a CAN bus. Once called, the function changes the Communication Controller state from configuration state to *`fcCANCcStateNormalActive`* if the start-up is successful. This state is entered if either mode *`fcCANMonitoringNormal`* or *`fcCANMonitoringSilent`* is used. The current Communication Controller state can be read using the function **`fcbCANGetCcState`**. The user should call **`fcbCANGetCcState`** after **`fcbCANMonitoringStart`** and check that the CC is in the state *`fcCANCcStateNormalActive`*.

```
fcError fcbCANMonitoringStart(
      fcHandle hFlexCard,
      fcCC CC,
      fcBool resetTimestamps,
      fcCANMonitoringMode mode
);
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard.

`CC`

[IN] Index of the CAN Communication Controller.

`restartTimestamps`

[IN] Set this parameter to 0 to restart the measurement without resetting the FlexCard timestamp. Set it to <> 0 to start the measurement from the beginning. The timestamps have microsecond resolution.

`Mode`

[IN] The monitoring mode. See **`fcCANMonitoringMode`** for details which monitoring mode is supported.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

After the monitoring has started, the user should check if the integration in the cluster was successful, **`fcbCANGetCcState`** should return the state *`fcCANCcStateNormalActive`*.

	Information
ℹ	After the monitoring has successfully started, the receive process must be started as soon as possible to avoid an overflow (error packet *`fcErrFlexcardOverflow`* is received). Once an overflow occurred, no more packets can be received. The monitoring must be stopped and started again.

	Information
ℹ	On a FlexCard PXIe3/PCIe3 using the FlexDevice Mode, the driver checks how the FlexConfig RBS running on the processor is configured. The CAN decoder supports the frame format *`fcCANFDFormatIso11898_1`*, but not the frame format *`fcCANFDFormatBoschSpecV1_0`*. If the FlexConfig RBS is configured to a not supported CAN-FD frame format, this function returns FUNCTION_NOT_IMPLEMENTED.

See Also

`fcbCANMonitoringStop`, **`fcbCANGetCcState`**, **`fcCANMonitoringMode`**

Example

```
// Precondition: valid flexcard handle exists and the flexcard is
// already configured.

fcCC eCC = fcCC1;
fcError e = fcbCANMonitoringStart(hFlexCard,eCC,true,fcCANMonitoringNormal);
if (0 == e)
{
      bool active = false;
      bool timeout = false;
      DWORD maxTime = ::GetTickCount() + 2000;
      fcCANCcState state = fcCANCcStateUnknown;

      // Check if the FlexCard is in CAN normal active state
      do
      {
            fcbCANGetCcState(hFlexCard, eCC, &state);
            active= (state == fcCANCcStateNormalActive);
            timeout      = ::GetTickCount() >= maxTime;

      } while ( ! active && ! timeout);

      if (active)
      {
            // Start your receive thread/routine
            //   ...
      }
      else
      {
            // if we timed out, we stop the monitoring
            fcbCANMonitoringStop(hFlexCard);
      }
}
else
{
      // error handling …
}
```

6.2.3 fcbCANMonitoringStop

This function stops the CAN bus monitoring of the selected Communication Controller. The Communication Controller is set back in its configuration state. After calling this function and emptying the receive buffer with **fcbReceive**, no more messages from this Communication Controller are received. Additionally, if the user transmits messages after calling this function, they do not appear on the CAN bus.

```
fcError fcbCANMonitoringStop(
      fcHandle hFlexCard,
      fcCC CC
)
```

Parameters

hFlexCard

[IN] Handle to FlexCard.

CC

[IN] Index of the CAN Communication Controller.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbCANMonitoringStart

6.2.4 fcbCANGetCcState

This function returns the current CAN Communication Controller state. For a description of possible states, refer to the enumeration **fcCANCcState**.

```
fcError fcbCANGetCcState(
       fcHandle hFlexCard,
       fcCC CC,
       fcCANCcState* pState
)
```

Parameters

hFlexCard
[IN] Handle to a FlexCard.

CC
[IN] Communication Controller index.

pState
[OUT] Current CAN Communication Controller state.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See

fcbCANMonitoringStart, **fcbCANMonitoringStop**, **fcCANCcState**

Example

For an example, see **fcbCANMonitoringStart**.

6.3 Configuration

6.3.1 Enumerations

6.3.1.1 fcCANBufCfgType

For the transmission and reception of CAN frames the Communication Controller provides different types of message buffers. There are 128 message buffers available. These buffers can be freely configured as rx or tx buffers. For sending and receiving error frames or for receiving trigger packets, no message buffer is needed. Each message buffer can be assigned with one of the following specific types.

```
Typedef enum fcCANBufCfgType
{
       fcCANBufCfgTypeNone = 0,
       fcCANBufCfgTypeCommon,
       fcCANBufCfgTypeRxAll,
       fcCANBufCfgTypeRx,
       fcCANBufCfgTypeTx,
       fcCANBufCfgTypeRemoteRx,
       fcCANBufCfgTypeRemoteTx,
}fcCANBufCfgType;
```

Members

fcCANBufCfgTypeNone
The message buffer is not used. It can be used to reset a message buffer.

fcCANBufCfgTypeCommon
The message buffer is reserved for internal use only. (No support.)

fcCANBufCfgTypeRxAll
The message buffer is used for receiving all incoming CAN data and remote frames.

fcCANBufCfgTypeRx
The message buffer is used as a receive buffer for either a specific message or a set of messages.

fcCANBufCfgTypeTx

The message buffer is used as a transmit buffer for a specific CAN message ID.

fcCANBufCfgTypeRemoteRx

The message buffer is used as a remote receive buffer. It is used for sending a remote request and receiving the according replying message.

fcCANBufCfgTypeRemoteTx

The message buffer is used as a remote transmission buffer. It can be transmitted automatically when a remote request is received.

See Also

fcCANBufCfg

6.3.1.2 fcCANBufCfgRxAllCondition

This enumeration defines the acceptance conditions of an **fcCANBufCfgRxAll** buffer. The conditions may be binary Ored.

```
Typedef enum fcCANBufCfgRxAllCondition
{
      fcCANRxAllNone            = 0x0,
      fcCANRxAllIDStandard      = 0x1,
      fcCANRxAllIDExtended      = 0x2,
      fcCANRxAllFrameData       = 0x4,
      fcCANRxAllFrameRemote     = 0x8,
}fcCANBufCfgRxAllCondition;
```

Members

fcCANRxAllNone

Accept no frames.

fcCANRxAllIDStandard

Accept CAN frames with standard identifiers.

fcCANRxAllIDExtended

Accept CAN frames with extended identifiers.

fcCANRxAllFrameData

Accept all CAN data frames.

fcCANRxAllFrameRemote

Accept all CAN remote frames.

See Also

fcCANBufCfgRxAll

6.3.2 Structures

6.3.2.1 fcCANBufCfgRxAll

This structure specifies a special CAN receive message buffer. This buffer type is used to receive all frames of the specified conditions.

```
Typedef struct fcCANBufCfgRxAll
{
      fcDword Condition;
      fcDword Reserved[2];
}fcCANBufCfgRxAll;
```

Members

Condition

The acceptance condition for this buffer, which can be OR-ed.
At least one id condition and one frame condition must be used to receive frames.

Reserved

Reserved for future use.

See Also

`fcCANBufCfg`, **`fcCANBufCfgRxAllCondition`**

6.3.2.2 fcCANBufCfgRx

This structure specifies the configuration of a CAN receive message buffer. This buffer type is used to receive a CAN message with a specific CAN ID only or a range of CAN IDs.

```
Typedef struct fcCANBufCfgRx
{
       fcDword ID;
       fcDword MaskID;
       fcDword enableIDExtended  :1;
       fcDword enableMask        :1;
       fcDword Reserved[2];
}fcCANBufCfgRx;
```

Members

`ID`

Defines the CAN identifier to be received in this message buffer.
Valid values for a standard CAN ID range from 0x0 – 0x7FF.
Valid values for an extended CAN ID range from 0x0 – 0x1FFFFFFF.

`MaskID`

The bit mask. The corresponding bits from the struct member ID are used for acceptance filtering. MaskID configures which bits will be checked during filtering. 1 means that the bit position of the ID will be checked, while 0 means that the bit position of the ID is not used for acceptance filtering. ID configures which value the bit position must contain so that the frame is received.
If MaskID is equal 0, all IDs will be accepted.

`enableIDExtended`

If set to 1 the message buffer is defined for extended CAN identifiers only. If set to 0 the message buffer is defined for standard CAN identifiers. It’s not possible to receive both versions in one message buffer.

`enableMask`

Set this flag to 1 to enable the acceptance mask.

`Reserved`

Reserved for future use.

Configuration notice

Example: IDs 11 and 13 should be received.

ID 11 is in binary code: `1011`

ID 13 is in binary code: `1101`

Set the MaskID to 1001 (0x9) and the ID to one of the IDs that should be received, e.g. 13 (0xD). The filter is not perfect, that means that IDs 9 and 15 are received, too.

Further examples:

ID	MaskID	enableMask	Behaviour
0x1	0x1	1	Only odd frames are received (1, 3, 5, …)
0x3	0x3	1	Received: 3, 7, … Rejected: 1, 2, 4, 5, 9…
0x1	0xF	1	Received: 1, 17. Rejected: 9, 16, …
0x0	0x7F0	1	Received: ID < 16

See Also

`fcCANBufCfg`

6.3.2.3 fcCANBufCfgTx

This structure specifies the configuration of a CAN transmit message buffer. This buffer type is used to transmit a CAN message with a specific CAN ID only.

```
Typedef struct fcCANBufCfgTx
{
      fcDword ID;
      fcByte  Data[8];
      fcDword DLC : 4;
      fcDword enableIDExtended : 1;
      fcDword enableTxAcknowledge : 1;
      fcDword enableTxRequest : 1;
      fcDword newData : 1;
      fcDword Reserved[2];
}fcCANBufCfgTx;
```

Members

ID

Defines the CAN identifier to be transmitted in this message buffer.
Valid values for a standard CAN ID range from 0x0 – 0x7FF.
Valid values for an extended CAN ID range from 0x0 – 0x1FFFFFFF.

Data

Defines the data for transmission. All of the 8 data bytes can be set. The corresponding DLC parameter is used to define the number of bytes to transmit.

DLC

Defines the data length (in bytes) to be transmitted.

enableIDExtended

If set to 1 the message buffer is defined for extended CAN identifiers only. If set to 0 the message buffer is defined for standard CAN identifiers. It is not possible to transmit both versions in one message buffer.

enableTxAcknowledge

Set this flag to 1 to enable the transmit acknowledge. The FlexCard generates a CAN packet with a direction flag = 1 (Tx), if the data is transmitted successfully.

enableTxRequest

Set this flag to 1 to indicate that the message is requested to be sent as soon as the Communication Controller is in state ‘normal active’.

newData

Set this flag to 1 to update the data of the message buffer. Set to 0 if no new data shall be updated.

Reserved

Reserved for future use

See Also

fcCANBufCfg, **fcCANPacket**

6.3.2.4 fcCANBufCfgRemoteRx

This structure specifies a CAN remote receive message buffer. This buffer type is used to send a CAN remote message to request a CAN message with the same CAN identifier. This will be received into the message buffer.

```
Typedef struct fcCANBufCfgRemoteRx
{
      fcDword ID;
      fcDword DLC                     :4;
      fcDword enableIDExtended        :1;
      fcDword enableTxAcknowledge     :1;
      fcDword enableTxRequest         :1;
      fcDword Reserved[2];
}fcCANBufCfgRemoteRx;
```

Members

ID

Defines the CAN identifier to be received in this message buffer.
Valid values for a standard CAN ID range from 0x0 – 0x7FF.
Valid values for an extended CAN ID range from 0x0 – 0x1FFFFFFF.

DLC

Defines the data length (in bytes) to be transmitted.

enableIDExtended

If set to 1 the message buffer is defined for extended CAN identifiers only. If set to 0 the message buffer is defined for standard CAN identifiers. It is not possible to receive both versions in one message buffer.

enableTxAcknowledge

Set this flag to 1 to enable the transmit acknowledge. The FlexCard generates a CAN packet (RemoteTx) if the data are transmitted successfully.

enableTxRequest

Set this flag to 1 to indicate that the message is requested to be sent as soon as the Communication Controller is in state ‘normal active’.

Reserved

Reserved for future use.

See Also

fcCANBufCfg, **fcCANPacket**

6.3.2.5 fcCANBufCfgRemoteTx

This structure specifies a CAN remote transmit message buffer. This buffer type is used to transmit a CAN message when this ID is requested by a corresponding CAN remote frame.

```
Typedef struct fcCANBufCfgRemoteTx
{
      fcDword ID;
      fcByte  Data [8];
      fcDword DLC                 :4;
      fcDword enableIDExtended    :1;
      fcDword enableTxAcknowledge :1;
      fcDword enableTxRequest     :1;
      fcDword enableAutoResponse  :1;
      fcDword newData             :1;
      fcDword Reserved[2];
}fcCANBufCfgRemoteTx;
```

Members

ID

Defines the CAN identifier to be responded with the same id.
Valid values for a standard CAN ID range from 0x0 – 0x7FF.
Valid values for an extended CAN ID range from 0x0 – 0x1FFFFFFF.

Data

Defines the data for transmission. All of the 8 data bytes can be set. The corresponding DLC parameter is used to define the number of bytes to transmit.

DLC

Defines the data length (in bytes) to be transmitted.

enableIDExtended

If set to 1 the message buffer is defined for extended CAN identifiers only. If set to 0 the message buffer is defined for standard CAN identifiers. It’s not possible to receive both versions in one message buffer.

enableTxAcknowledge

Set this flag to 1 to enable the transmit acknowledge. The FlexCard generates a CAN packet (RemoteTx) if the data are transmitted successfully and the parameter *enableAutoResponse* is set to 1 too.

enableTxRequest

Set this flag to 1 to indicate that the message is requested to be sent as soon as the Communication Controller is in state ‘normal active’.

enableAutoResponse

Set this flag to 1 to enable the buffer to transmit a frame as soon as a corresponding CAN remote frame is received.

newData

Set this flag to 1 to update the data of the message buffer. Set to 0 if no new data shall be updated.

Reserved

Reserved for future use.

See Also

fcCANBufCfg, **fcCANPacket**

6.3.2.6 fcCANBufCfg

This structure describes the configuration of a CAN message buffer.

```
Typedef struct fcCANBufCfg
{
      fcCANBufCfgType Type;
      union
      {
            fcCANBufCfgCommon Common;
            fcCANBufCfgRxAll RxAll;
            fcCANBufCfgRx Rx;
            fcCANBufCfgTx Tx;
            fcCANBufCfgRemoteRx RemoteRx;
            fcCANBufCfgRemoteTx RemoteTx;
      };

}fcCANBufCfg;
```

Members

Type

Defines the CAN message buffer type. Using type *fcCANBufCfgTypeNone* disables/resets the message buffer.

Common

Used for internal purposes. (No support).

RxAll

Receive all buffer configuration.

Rx

Receive buffer configuration.

Tx

Transmit buffer configuration.

RemoteRx

Remote receive buffer configuration.

RemoteTx

Remote transmit buffer configuration.

See Also

fcCANBufCfgType, **fcCANBufCfgRxAll**, **fcCANBufCfgRx**, **fcCANBufCfgTx**, **fcCANBufCfgRemoteRx**, **fcCANBufCfgRemoteTx**, **fcbCANSetMessageBuffer**, **fcbCANGetMessageBuffer**

6.3.2.7 fcCANCcConfig

This structure describes the configuration of a CAN Communication Controller. Within this function all message buffers will be reset.

```
Typedef struct fcCANCcConfig
{
      fcWord BaudRatePrescaler;
      fcWord SynchronizationJumpWidth;
      fcWord TimeSegment1;
      fcWord TimeSegment2;
      fcDword enableAutomaticRetransmission :1;
      fcDword Reserved[6];
} fcCANCcConfig;
```

Members

BaudRatePrescaler

Defines the baud rate prescaler (BRP). Valid values are from 0 to 1023.

SynchronizationJumpWidth

Defines the synchronization jump width (SJW). Valid values are from 0 to 3 and must not be larger than TSEG1 and TSEG2.

TimeSegment1

Defines the time segment 1 (TSEG1). Valid values are from 0 to 15.

TimeSegment2

Defines the time segment 2 (TSEG2). Valid values are from 0 to 7.

enableAutomaticRetransmission

Set this flag to 1 to enable automatic retransmission. If the CAN Communication Controller has lost the arbitration or if an error occurred during the transmission, the message will be retransmitted as soon as the CAN bus is free again.

Reserved

Reserved for future use.

See Also

fcbCANSetCcConfiguration

Remarks

The baud rate and the sample point calculation of the CAN Communication Controller depends on *BaudRatePrescaler*, *SynchronizationJumpWidth*, *TimeSegment1* and *TimeSegment2*.
Baud rate [baud] = 16 * 10^6 [Hz] / ((3 + TSEG1 + TSEG2) * (1 + BRP))
Sample point [%] = 100 * (2 + TSEG1) / (3 + TSEG1 + TSEG2)

The unit of the baud rate is either Hz or Bit/sec. 1 Hz equals 1 Bit/sec. 1 Kbit/sec equals 1000 Bit/sec.
The CAN specification defines the BitLength as follows:
BitLength = Sync_Seg + Prop_Seg+ Phase_Seg1 + Phase_Seg2 = 8-25 Tq (Timequantum)
D_CAN uses following definitions:
Tseg1 + 1 = Prop_Seg+ Phase_Seg1
Tseg2 + 1 = Phase_Seg2
1 = Sync_Seg
BitLength = 1 + Tseg1 + 1 + Tseg2 + 1 = 4-25 Tq
The fcBase API and CANBaudRateCalculator use the D_CAN definitions.

	Information
ℹ	*STAR ELECTRONICS GmbH & Co. KG* delivers a calculation tool “CANBaudRateCalculator”, which can be found in the FlexCard tools directory in the Windows program menu.

6.3.2.8 fcCANTxFifoConfig

This structure describes the configuration of the transmit FIFO feature for a CAN Communication Controller. With the transmit FIFO it’s possible to transmit CAN data messages without the configuration of several message buffer objects.

```
Typedef struct fcCANTxFifoConfig
{
      fcDword BufferNumber;
      fcDword TimerInterval;
      fcDword enableRetransmission : 1;
      fcDword enableTxAcknowledge : 1;
      fcDword Reserved[3];
} fcCANTxFifoConfig;
```

Members

BufferNumber

The number of the message buffer used for the transmission. This buffer number is reserved for CAN transmit FIFO only and should not be used with other available message buffer functions to avoid transmit disturbances. Valid values are from 0 to 128. Set to 0 to deactivate the FIFO.

TimerInterval

The transmit FIFO timer interval with microsecond resolution. Valid values are from 200 to 1048575. Timer interval depends on the configured CAN baud rate and will be adjusted to a minimum calculated value by the driver automatically.

enableRetransmission

Set this flag to 1 to enable automatic retransmission. If the transmit FIFO has lost the arbitration or if an error occurred during the transmission, the message will be retransmitted with the configured timer interval. With disabled flag, the transmit FIFO tries to transmit the CAN message once. In case of arbitration lost or error occurrence the next message will be transmitted after the configured timer interval.

enableTxAcknowledge

Set this flag to 1 to enable transmit acknowledge for all transmitted CAN messages with this FIFO.

Reserved

Reserved for future use.

See Also

fcbCANSetTxFifoConfiguration, **fcbCANGetTxFifoConfiguration**, **fcbCANTxFifoTransmit**

Remarks

In case the transmission of any number of message buffers may be requested at the same time, they are transmitted subsequently according to their priority (The message buffer numbers are configurable from 1 up to 128, the lower the message number, the higher is the priority).

	Information
ℹ	This structure is initially supported by FlexCard API version S6V3-F.

6.3.3 fcbCANSetCcConfiguration

This function configures the CAN Communication Controller. This function cannot be called during monitoring. Before the configuration of the CAN CC starts, all CAN message buffers are reset.

```
fcError fcbCANSetCcConfiguration(
        fcHandle hFlexCard,
        fcCC CC,
        fcCANCcConfig cfg
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

Cfg

[IN] CAN Communication Controller configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbCANGetMessageBuffer, **fcCANCcConfig**

6.3.4 fcbCANSetMessageBuffer

This function configures the message buffers of the CAN Communication Controller. Configuring message buffers is allowed in all Communication Controller states.

```
fcError fcbCANSetMessageBuffer(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword bufferNumber,
      fcCANBufCfg cfg,
      fcBool ignoreTxRqstLock
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] CAN Communication Controller identifier.

bufferNumber

[IN] Identifier of the message buffer to be configured. Valid values are 1 to 128.

Cfg

[IN] Message buffer configuration parameters.

ignoreTxRqstLock

Set this flag to <> 0 if you want to force a reconfiguration of this buffer although the previous message in this buffer was (probably) not transmitted yet.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCANBufCfg

6.3.5 fcbCANGetMessageBuffer

This function reads a specific CAN message buffer configuration.

```
fcError fcbCANGetMessageBuffer(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword bufferNumber,
      fcCANBufCfg* pCfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

bufferNumber

[IN] Identifier of the message buffer to be read. Valid values are 1 to 128.

Cfg

[OUT] The configuration parameters of the specified message buffer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbCANSetMessageBuffer, fcCANBufCfg

Example

```
// Get all configured transmit message buffers
std::map<unsigned int, fcCANBufCfg > Buffers;
unsigned int bufferNr = 1; // The first valid buffer is 1
while (true)
{
       fcCANBufCfg cfg;

       // as long as no error occurs we try to get each buffer
       fcError e = fcbCANGetMessageBuffer (m_hFlexCard,fcCC1,bufferNr,&cfg);
       if (0 != e) break;

       // is this a tx buffer, then add it to our list
       if (fcCANBufCfgTypeTx == cfg.Type)
              Buffers[bufferNr] = cfg;

       // next buffer number
       bufferNr++;
}
```

6.3.6 fcbCANSetTxFifoConfiguration

This function configures the transmit FIFO feature for a CAN Communication Controller. With the transmit FIFO it is possible to transmit CAN data messages without the configuration of several message buffer objects. This function cannot be called during monitoring. The FlexCard driver corrects the transmit interval if the configured CAN baudrate is too low. After calling **fcbCANSetTxFifoConfiguration** call **fcbCANGetTxFifoConfiguration** to get the corrected transmit interval.

```
fcError fcbCANSetTxFifoConfiguration(
       fcHandle hFlexCard,
       fcCC CC,
       fcCANTxFifoConfig cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

Cfg

[IN] CAN transmit FIFO configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCANTxFifoConfig, **fcbCANSetTxFifoConfiguration**, **fcbCANTxFifoTransmit**

Example

```
// Configure a transmit CAN FIFO with timer interval of 200 us
// and use with message buffer index 2 for a high priority transmission
fcCANTxFifoConfig FifoConfig;
memset(&FifoConfig, 0, sizeof(fcCANTxFifoConfig));
FifoConfig.BufferNumber = 2;
FifoConfig.TimerInterval = 200;
FifoConfig.enableRetransmission = 1;
FifoConfig.enableTxAcknowledge = 1;

fcError e = fcbCANSetTxFifoConfiguration(m_hFlexCard, fcCC1, FifoConfig);
if (0 != e) {/* Error handling */}
```

ℹ	Information
	This function is initially supported by FlexCard API version S6V3-F.

6.3.7 fcbCANGetTxFifoConfiguration

This function reads the transmit FIFO configuration parameters of a CAN Communication Controller. With the transmit FIFO it’s possible to transmit CAN data messages without the configuration of several message buffer objects.

```
fcError fcbCANGetTxFifoConfiguration(
      fcHandle hFlexCard,
      fcCC CC,
      fcCANTxFifoConfig* pCfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

pCfg

[OUT] CAN transmit FIFO configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCANTxFifoConfig, **fcbCANSetTxFifoConfiguration**, **fcbCANTxFifoTransmit**

Example

```
// Read the configuration parameters of the transmit CAN FIFO
fcCANTxFifoConfig FifoConfig;
memset(&FifoConfig, 0, sizeof(fcCANTxFifoConfig));

fcError e = fcbCANGetTxFifoConfiguration(m_hFlexCard, fcCC1, &FifoConfig);
if (0 != e) {/* Error handling */}
```

	Information
ℹ	This function is initially supported by FlexCard API version S6V3-F.

6.3.8 fcbCANTxFifoReset

This function resets the CAN Communication Controller transmit FIFO buffer. CAN data messages in the FIFO are lost.

```
fcError fcbCANTxFifoReset(
      fcHandle hFlexCard,
      fcCC CC,
      fcBool bOnlyCurrent
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

bOnlyCurrent

[IN] Reset only the current CAN message that is scheduled for transmission by the CAN CC. The next CAN message is going to be scheduled for transmission. Current CAN data message is lost. Other queued CAN data messages in the FIFO aren't lost.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCANTxFifoConfig, **fcbCANTxFifoTransmit**

Example

```
// Reset the current CAN message, which blocks the schedule for next
// CAN messages in the transmit CAN FIFO
bool bResetOnlyCurrent = true;
fcError e = fcbCANTxFifoReset(m_hFlexCard, fcCC1, bResetOnlyCurrent);
if (0 != e) {/* Error handling */}
```

	Information
ℹ	This function is initially supported by FlexCard API version S6V3-F.

6.3.9 fcbCANSetFilterConfiguration

This function configures the accepted frame ids (CAN, CAN-FD). Only the configured ids are forwarded to the user application, all other CAN or CAN-FD frames are rejected. This function can be used on a CAN bus or a CAN-FD bus and depends on the type of communication controller. See the configuration notice for further details.

```
fcError fcbCANSetFilterConfiguration(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword id,
      fcBool bExt,
      fcBool bCanFd,
      fcBool reset
)
```

Parameters

hFlexCard
[IN] Handle to a FlexCard.

CC
[IN] Communication controller index.

Id
[IN] ID of CAN frame, which will be accept.

bExt
[IN] Flag for extended CAN ID. If *bExt* is true, this frame has an extended ID.

bCanFd
[IN] Flag for CANFD frame. If *bCanFd* is false, this frame is a CAN frame.

Reset
[IN] Flag for reset all filters. If Reset is true, all configured filters will be reset.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC

Example

```
// Set CAN filter with ID 55 and the already configured filters should be reset.
fcError e = fcbCANSetFilterConfiguration(handle, ccIndex, 5, false, false, true);
if (0 != e) {/* Error handling */}
```

6.4 Transmit

6.4.1 fcbCANTransmit

This function writes the data bytes in a CAN Communication Controller transmit buffer of the FlexCard. The transmitted data bytes depend on the message buffer configuration. The function returns immediately and does not wait for the data frame to arrive on the bus. The CAN message should normally be transmitted as soon as possible. Example: If the CAN bus is full of high priority messages, and the user transmits a low priority message, this function immediately returns successfully, but the message appears on the bus, only when it wins the arbitration.

In case the transmission of any number of message buffers may be requested at the same time, they are transmitted subsequently according to their priority (The message object numbers are configurable from 1 up to 128, the lower is the message number, the higher is the priority). If the transmit acknowledgment is activated, a CAN packet with a direction flag = 1 (Tx) is generated as soon as the message has been transmitted. This function should only be called when the FlexCard is in normal active state or when all message buffer configurations have been done.

The transmission may fail, if the buffer is already locked for transmission (**fcGetErrorCode** returns MSG_BUF_LOCKED_FOR_TRANSMISSION). In that case retry later or set the parameter *ignoreTxRqstLock* to true.

```
fcError fcbCANTransmit(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword bufferNumber,
      fcByte data[8],
      fcBool transmitNewData,
      fcBool ignoreTxRqstLock
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] CAN Communication Controller identifier.

bufferNumber

[IN] The number of the message buffer used for the transmission. Valid values are 1 to 128.

Data

[IN] The data to be transmitted. The configured DLC in the message buffer determinates the size of bytes which will be copied in the transmit buffer.

transmitNewData

[IN] Set to <> 0 to update the data of the message buffer. Set to 0 if the previous data shall be sent again.

ignoreTxRqstLock

[IN] Set this value to <> 0 if you want to force a reconfiguration of this buffer although the previous message was not transmitted yet.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCANBufCfg, **fcbCANGetMessageBuffer**

Example

```
fcByte data[8];
for (int i=0; i<8; i++) data[i]=0xA;

// Transmit new data
fcError e = fcbCANTransmit(m_hFlexCard, fcCC1, bufferNumber, data, true, true);
if (0 != e) {/* Error handling */}
```

6.4.2 fcbCANTxFifoTransmit

This function writes a CAN data message into the CAN transmit FIFO buffer of the FlexCard. It holds maximum 512 messages. The function returns immediately and does not wait for the data frame to arrive on the bus. The message should normally be transmitted as soon as possible. This function should only be called when the CAN transmit FIFO is configured. In case the FIFO is full, the error code TX_FIFO_FULL is returned.

```
fcError fcbCANTxFifoTransmit(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword id,
      fcBool bExt,
      fcByte* pData,
      fcDword dlc
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] CAN Communication Controller identifier.

Id

[IN] The CAN identifier. Valid values for a standard CAN ID range from 0x0 to 0x7FF. Valid values for an extended CAN ID range from 0x0 to 0x1FFFFFFF.

bExt

[IN] Set to <> 0 to transmit a CAN message with an extended identifier. Set to 0 to transmit a standard CAN message.

pData

[IN] The data to be transmitted.

Dlc

[IN] The size of bytes which will be copied in the FIFO.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbCANSetTxFifoConfiguration

Example

```
fcDword id = 0x789;
bool bExt = false;
const fcDword dlc = 8;
fcByte data[dlc];
for (int i=0; i<dlc; i++) data[i]=i;

// Put CAN message into the transmit FIFO for scheduling as soon as possible
fcError e = fcbCANTxFifoTransmit(m_hFlexCard, fcCC1, id, bExt, data, dlc);
if (0 != e)
{
      // Error handling
}
```

ℹ	**Information**
	This function is initially supported by FlexCard API version S6V3-F.

7 CAN-FD API

The following section describes the data structures and features used for CAN-FD functionality. To use these functions the FlexCard must have a firmware with a CAN-FD CC and the FlexCard must be licensed for CAN.

	Information
ℹ	All enumerations, structures and functions in this chapter are initially supported by FlexCard Windows API version S6V6-F.

7.1 Basic CAN-FD Workflow

The following figure shows a typical CAN-FD workflow.

The CAN-FD functions are supported for the CAN Communication Controller type Bosch M_CAN. The Enum function returns the struct fcInfoHwSw. Check the struct `fcInfoHw.pVersionCC` to see whether the firmware provides this CC type.

When you compare the CAN and CAN-FD API functions, the functions fcbCANFDSetCcConfiguration and fcbCANFDTransmit were added for CAN-FD. The functions fcbCANGetCcState and fcbCANMonitoringStart/Stop are used on the Bosch M_CAN.

The Bosch M_CAN can be used with the new API functions to connect to either a CAN-HS or a CAN-FD network.

The CCs are counted in following order: First FlexRay, then CAN. CAN-FD CCs count like CAN CCs.

For example, if the firmware image has 1 FlexRay CC and 4 CAN-FD CCs, the CCs are referred to like this:

- FlexRay fcCC1
- CAN fcCC1, fcCC2, fcCC3, fcCC4

CAN-HS baud rate is 40 Kbit/s to 1 Mbit/s. The maximum CAN-FD baud rate depends on the transceiver.

Refer to the chapter General Function Availability to find out if the connected FlexCard device supports onboard CAN terminations.

For transmitting CAN data use the function fcbCANFDTransmit. Check the error code to see whether the internal transmit buffer had enough space for the frame.

The FlexCard Windows Developer Setup installs the example application *fcDemoCANFD.exe* and its source code to the installation directory.

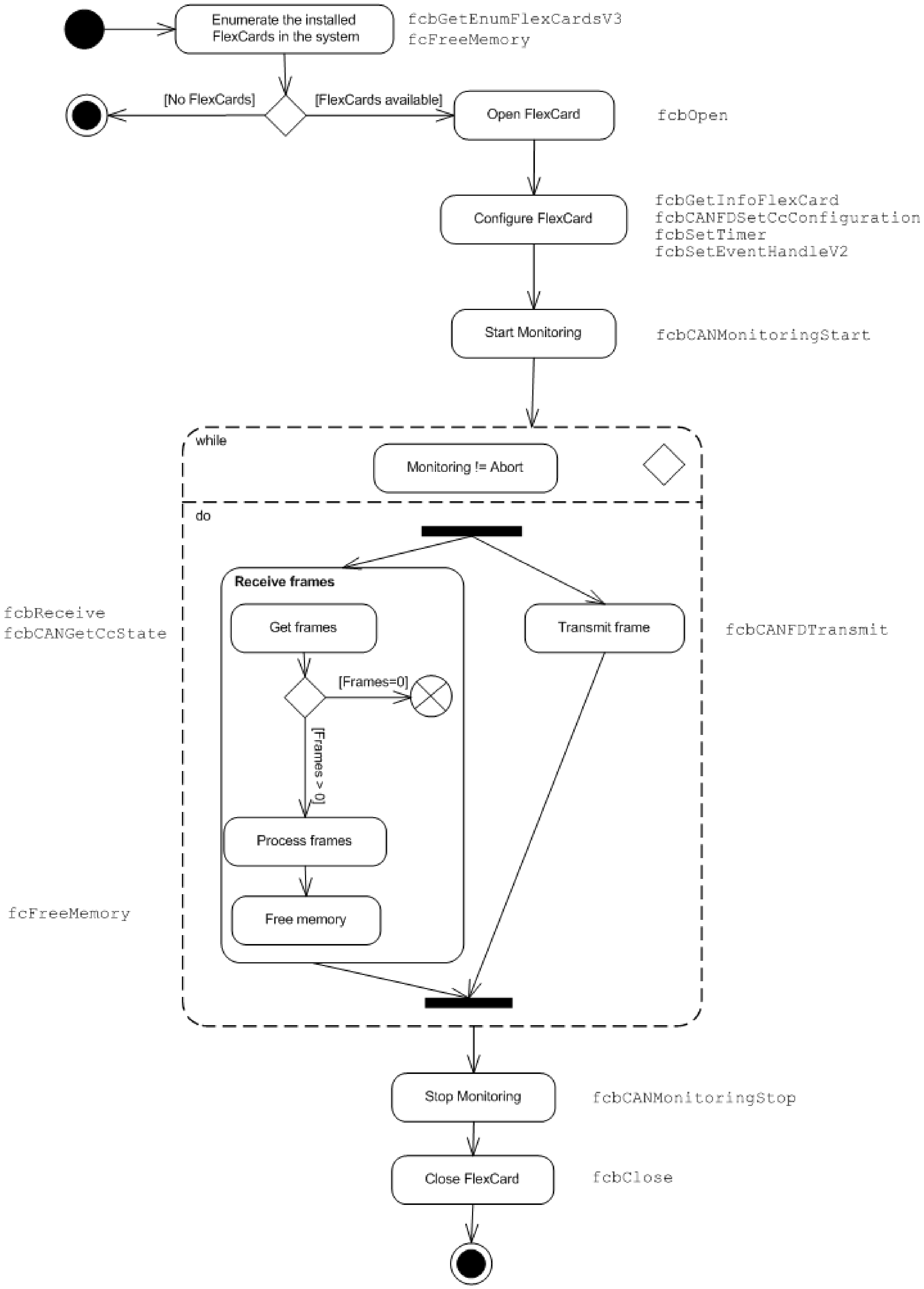


Figure 12: Typical CAN-FD function workflow

7.2 CAN-FD DLC

The CAN-FD specification defines following four bit long DLCs (data length codes).

DLC (decimal)	Number of bytes
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	12
10	16
11	20
12	24
13	32
14	48
15	64

DLCs smaller and equal 8 are possible in a CAN network. DLCs smaller and equal 15 are possible in CAN-FD networks.

7.3 Configuration

7.3.1 Enumerations

7.3.1.1 fcCANFDFrameFormat

This enumeration describes the CAN-FD frame format.

```
Typedef enum fcCANFDFrameFormat
{
      fcCANFDFormatUnspecified = 0,
      fcCANFDFormatIso11898_1,
      fcCANFDFormatBoschSpecV1_0,
} fcCANFDFrameFormat;
```

Members

fcCANFDFormatUnspecified

Unspecified CAN-FD frame format.

fcCANFDFormatIso11898_1

CAN-FD frame format according to ISO 11898-1 (CAN/ CAN-FD).

fcCANFDFormatBoschSpecV1_0

CAN-FD frame format according to CAN-FD Bosch Specification 1.0.

See Also

fcbCANFDSetCcConfiguration

7.3.2 Structures

7.3.2.1 fcCANCcBitTime

This structure describes the configuration of an M_CAN communication controller. An M_CAN controller is capable of CAN-HS and CAN-FD communication.

```
Typedef struct fcCANCcBitTime
{
      fcWord BaudRatePrescaler;
      fcWord SynchronizationJumpWidth;
      fcWord TimeSegment1;
      fcWord TimeSegment2;
      fcDword Reserved[4];
} fcCANCcBitTime;
```

Members

BaudRatePrescaler

Defines the baud rate prescaler (BRP).
Valid range for nominal bit timing: 1 ... 512
Valid range for data bit timing: 1 ... 32

SynchronizationJumpWidth

Defines the synchronization jump width (SJW).
Valid range for nominal bit timing: 1 ... 128
Valid range for data bit timing: 1 ... 16

TimeSegment1

Defines the time segment 1 (TSEG1).
This value is equal Prop_Seg + Phase_Seg1
from the specification.
Valid range for nominal bit timing: 2 ... 256
Valid range for data bit timing: 1 ... 32

TimeSegment2

Defines the time segment 2 (TSEG2).
This value is equal Phase_Seg2 from the specification.
Valid range for nominal bit timing: 1 ... 128
Valid range for data bit timing: 1 ... 16

Reserved

Reserved for future use.

See Also

fcbCANFDSetCcConfiguration

Remarks

The Sync_Seg from the specification is always 1. The information processing time from the specification is 0. The clock frequency for FlexCard USB and FlexCard PMC-2 is 20 MHz, but for FlexDevice and FlexCard PXIe3 and FlexCard PCIe3 the clock frequency is 40 MHz.

The baud rate and the sample point calculation by the CAN communication controller depends on BaudRatePrescaler, SynchronizationJumpWidth, TimeSegment1 and TimeSegment2.

Calculation for FlexCards with 20 MHz clock frequency.

Baud rate [baud] = 20 * 10^6 [Hz] / ((1 + TSEG1 + TSEG2) * BRP)

Sample point [%] = 100 * (1 + TSEG1) / (1 + TSEG1 + TSEG2)

7.3.2.2 fcCANFDCcConfig

This structure describes the configuration of a CAN-FD Communication Controller.

```
Typedef struct fcCANFDCcConfig
{
      fcCANCcBitTime nominalBitTime;
      fcCANCcBitTime dataBitTime;
      fcCANFDFrameFormat frameFormat;
      fcDword enableAutomaticRetransmission :1;
      fcDword enableCANFDOperation   :1;
      fcDword enableCANFDBitRateSwitch      :1;
      fcDword Reserved[6];
} fcCANFDCcConfig;
```

Members

nominalBitTime

Please note: Nominal and data bit time have different ranges. Relevant for CAN and CAN-FD.

dataBitTime

The data bit rate must be greater or equal the nominal bit rate. Relevant for CAN-FD.

frameFormat

Please note: All communication partners in a CAN-FD network must use the same frame format.
Relevant for CAN-FD.

enableAutomaticRetransmission

Set this flag to 1 to enable automatic retransmission. If the CAN communication controller has lost the arbitration or if an error occurred during the transmission, the message will be retransmitted as soon as the CAN bus is free again.
Relevant for CAN and CAN-FD.

enableCANFDOperation

Enables CAN-FD operation. When CAN-FD is enabled, it is also possible to transmit normal CAN frames. This is decided while transmitting the frame.
Relevant for CAN and CAN-FD.

enableCANFDBitRateSwitch

Specifies that it is possible to switch the bit rate for CAN-FD frames.
Relevant for CAN-FD.

Reserved

Reserved for future use.

See Also

fcbCANFDSetCcConfiguration

7.3.3 fcbCANFDSetCcConfiguration

This function configures the CAN-FD Communication Controller. This function cannot be called during monitoring.

```
fcError fcbCANFDSetCcConfiguration(
      fcHandle hFlexCard,
      fcCC CC,
      fcCANFDCcConfig cfg
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

Cfg

[IN] CAN-FD Communication Controller configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcCANFDCcConfig`

Example

```
// Configuration settings for FlexCard PMC-II

fcCANFDCcConfig CcConfig;
memset(&CcConfig, 0, sizeof(fcCANFDCcConfig));

// 500 Kbit/s nominal
CcConfig.nominalBitTime.BaudRatePrescaler = 1;
CcConfig.nominalBitTime.SynchronizationJumpWidth = 1;
CcConfig.nominalBitTime.TimeSegment1 = 29;
CcConfig.nominalBitTime.TimeSegment2 = 10;

// 4 Mbit/s data
CcConfig.dataBitTime.BaudRatePrescaler = 1;
CcConfig.dataBitTime.SynchronizationJumpWidth = 1;
CcConfig.dataBitTime.TimeSegment1 = 2;
CcConfig.dataBitTime.TimeSegment2 = 2;

CcConfig.frameFormat = fcCANFDFormatIso11898_1;
CcConfig.enableAutomaticRetransmission = true;
CcConfig.enableCANFDOperation = true;
CcConfig.enableCANFDBitRateSwitch = true;

fcError e = fcbCANSetCcConfiguration(handle, fcCC1, CcConfig);
if (0 != e) {/* Error handling*/}
```

7.3.4 fcbCANSetFilterConfiguration

This function configures the CAN-FD frame ids accepted by the Analyzer. Only the CAN-FD ids which are configured by setting filter are forwarded to the Analyzer, all other CAN-FD frames are rejected. To configure a CAN-FD filter, *bCanFd* must be set to true.

```
fcError fcbCANSetFilterConfiguration(
      fcHandle hFlexCard,
      fcCC CC,
      fcDword id,
      fcBool bExt,
      fcBool bCanFd,
      fcBool reset
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index.

Id

[IN] ID of CAN-FD frame, which will be accept.

bExt

[IN] Flag for extended CANFD-ID. If *bExt* is true, this frame has an extended ID.

bCanFd

[IN] Flag for CAN-FD frame. If *bCanFd* is true, this frame is a CAN-FD frame.

Reset

[IN] Flag for reset all filters. If Reset is true, all configured filters will be reset.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcCC`

Example

```
// Set CANFD filter with extended ID 268435968 and the already configured filters
should be reset.
fcError e = fcbCANSetFilterConfiguration(handle, ccIndex, 268435968, true, true,
true);
if (0 != e) {/* Error handling */}
```

7.4 Transmit

7.4.1 Structures

7.4.1.1 fcCANFDTxFrame

This structure contains a CAN/ CAN-FD frame that can be transmitted by a CAN-FD communication controller.

```
Typedef struct fcCANFDTxFrame
{
      fcDword ID;
      fcDword DLC :4;
      fcByte data[64];
      fcBool enableIDExtended;
      fcBool enableTxAcknowledge;
      fcBool enableCanFdFormat;
      fcBool enableCanFdBitrateSwitch;
      fcDword reserved;
} fcCANFDTxFrame;
```

Members

`ID`

Defines the CAN identifier to be transmitted in this message buffer. Valid values for a standard CAN Id range from 0x0 to 0x7FF. Valid values for an extended CAN Id range from 0x0 to 0x1FFFFFFF.

`DLC`

Defines the data length to be transmitted. Note that the length is coded with four bits according to the CAN/ CAN-FD standard.

`Data`

The payload data.

`enableIDExtended`

If set to 1 the CAN identifier of the message is defined as extended. If set to 0 the CAN identifier is defined as standard.

`enableTxAcknowledge`

When this bit is selected, the FlexCard generates a Tx acknowledge packet if the frame was transmitted correctly.

`enableCanFdFormat`

Specifies whether the frame has a CAN format or a CAN-FD format.

`enableCanFdBitrateSwitch`

Specifies whether the frame uses bit rate switching. This is only relevant if *`enableCanFdFormat`* is configured.

`Reserved`

Reserved for future use.

See Also

`fcbCANFDTransmit`

7.4.2 fcbCANFDTransmit

This function writes the data bytes in a CAN-FD communication controller of the FlexCard. The function returns immediately and does not wait for the data frame to arrive on the bus. The message should normally be transmitted as soon as possible. The FlexCard internally uses a Tx FIFO which holds maximum

32 messages. The messages appear on the bus in the order in which they were transmitted by the user. If the transmit acknowledgment is activated, a CAN-FD packet with a direction flag = 1 (Tx) is generated as soon as the message has been transmitted. This function should only be called when the FlexCard is in normal active state. If the internal Tx message fifo is full, the error TX_FIFO_FULL is returned and the message is not transmitted.

Sending CAN Remote frames via the function fcbCANFDTransmit is not supported.

```
fcError fcbCANFDTransmit(
       fcHandle hFlexCard,
       fcCC CC,
       const fcCANFDTxFrame* pFrame
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] CAN Communication Controller identifier.

pFrame

[IN] The frame that should be transmitted. This struct contains the payload data and configuration options.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCANFDTxFrame

Example

```
fcCANFDTxFrame frame1;
memset(&frame1, 0, sizeof(frame1));
frame1.DLC = 0xF; // this Dlc means 64 byte payload length
frame1.ID = 1;
frame1.enableIDExtended = false;
frame1.enableCanFdBitrateSwitch = true;
frame1.enableCanFdFormat = true;
frame1.enableTxAcknowledge = true;

fcError e = fcbCANFDTransmit(handle, fcCC1, &frame1);
fcErrorCode ec = fcGetErrorCode(e);
if (TX_FIFO_FULL == ec)
{
    // handle error. E.g. wait and retransmit.
}
else if(NONE != ec)
{
    // handle error
}
```

8 Ethernet API

The following section describes the data structures and features used for Ethernet functionality. To use these functions the FlexCard must have a firmware with an Ethernet Controller. Currently the Ethernet API is only available for the FlexCard PXIe3, FlexCard PCIe3 and FlexDevice.

	Information
ℹ	All enumerations, structures and functions in this chapter are initially supported by FlexCard Windows API version S6V7-F.

8.1 Initialization

8.1.1 Enumerations

8.1.1.1 fcEthMonitoringMode

This enumeration defines the different modes available, used to monitor an ethernet bus.

```
typedef enum fcEthMonitoringMode
{
      fcEthMonitoringPassive= 0,
} fcEthMonitoringMode;
```

Members

fcEthMonitoringPassive

Monitors the ethernet in passive mode without impacting the real bus traffic.

See Also

fcbEthMonitoringStart

8.1.2 fcbEthMonitoringStart

This function is used to start the monitoring of an ethernet channel. Set the configuration with *fcbEthSetCcConfiguration* before calling this function.

```
fcError fcbEthMonitoringStart (
      fcHandle hFlexCard,
      fcCC CC,
      fcEthMonitoringMode mode
      fcBool resetTimestamps,
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Index of the Ethernet controller.

Mode

[IN] The monitoring mode. See **fcEthMonitoringMode** for details which monitoring mode is supported.

restartTimestamps

[IN] Set this parameter to 0 to restart the measurement without resetting the FlexCard timestamp. Set it to <> 0 to start the measurement from the beginning. The timestamps have microsecond resolution.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

> **Information**
>
> After the monitoring has successfully started, the receive process must be started as soon as possible to avoid an overflow (error packet *fcErrFlexcardOverflow* is received). Once an overflow occurred, no more packets can be received. The monitoring must be stopped and started again.

See Also

fcbEthMonitoringStop, **fcEthMonitoringMode**

Example

```
// Precondition: valid flexcard handle exists and the flexcard is
// already configured.

fcCC eCC = fcCC1;
fcError e = fcbEthMonitoringStart(hFlexCard,eCC,true,fcEthMonitoringPassive);
if (0 == e)
{
        // Run your receive thread/routine
        //   ...

        // … when done. Stop it …
        fcbEthMonitoringStop(hFlexCard, eCC);

}
else
{
        // error handling …
}
```

8.1.3 fcbEthMonitoringStop

This function stops the Ethernet channel measurement of the selected controller.

```
fcError fcbEthMonitoringStop(
        fcHandle hFlexCard,
        fcCC CC
)
```

Parameters

hFlexCard

[IN] Handle to FlexCard.

CC

[IN] Index of the Ethernet Controller.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbEthMonitoringStart

8.2 Configuration

8.2.1 Enumerations

8.2.1.1 fcEthMode

This enumeration describes Ethernet Mode, that includes the following information:

Data Transfer Rate, Connection Cable Type, Transmission Direction (relevant only to Automotive Ethernet) Some Modes are only usable for Ethernet FlexTinys, while others for Automotive Ethernet FlexTinys. Abbrevitaions used are

Base	stands for baseband transmission
T1/TX	stands for type of connection cable used
10/100/1000	stands for 10/100/1000 Mbps (Mega bits per second) data transfer rate
Master	stands for Master-Slave direction (relevant to Automotive Ethernet)
Slave	stands for Slave-Master direction
Auto	stands for Auto Negotitaion (relevant only to Ethernet). In this Mode the channel decides the suitable data transfer rate based on negotiation.
Fixed	stands for fixed data transfer used (No Auto Negotiation)
Automotive Ethernet	100BASE-T1, 1000BASE-T1

```
Typedef enum fcEthMode
{
      fcEthMode100BaseT1Master = 1,
      fcEthMode100BaseT1Slave,
      fcEthMode1000BaseT1Master,
      fcEthMode1000BaseT1Slave,
      fcEthModeAuto,
      fcEthMode1000BaseTXFixed,
      fcEthMode100BaseTXFixed,
      fcEthMode10BaseTXFixed,
} fcEthMode;
```

Members

fcEthMode100BaseT1Master
100BASE-T1 Master-Slave Transmission Direction

fcEthMode100BaseT1Slave
100BASE-T1 Slave-Master Transmission Direction

fcEthMode1000BaseT1Master
1000BASE-T1 Master-Slave Transmission Direction

fcEthMode1000BaseT1Slave
1000BASE-T1 Slave-Master Transmission Direction

fcEthModeAuto
Ethernet Mode: Auto Negotiation, possible rates used10/100/1000 Mbps

fcEthMode1000BaseTXFixed
Ethernet Mode: 1000 Mbps

fcEthMode100BaseTXFixed
Ehernet Mode: 100 Mbps

fcEthMode10BaseTXFixed
Ehernet Mode: 10 Mbps

See Also

fcCANBufCfg

8.2.2 Structures

8.2.2.1 fcEthCcConfig

This structure describes the configuration of an Ethernet Communication Controller.

```
Typedef struct fcEthCcConfig
{
       fcEthMode Mode;
       fcBool PromiscuousMode ;
       fcByte MAC[6];
       fcByte VLANCount ;
       fcWord* VLANs;
} fcEthCcConfig;
```

Members

Mode

Ethernet Mode indicates the data transfer rate, the connection cable type and in case of Automotive Ethernet the transmission direction. Some Modes relates to Ethernet while others to Automotive Ethernet.

PromiscuousMode

Promiscous Mode (currently **not** supported)
True: Channel receives all frames Without MAC Address Check.
False: Channel receives frames from specific MAC Addresses.

MAC

Mac Address (Media Access Control) of Hardware Channel.

VLANCount

Total count of VLAN IDs (Virtual Local Area Networks).

VLANs

Pointer to and array of VLAN IDs (Virtual Local Area Networks). VLAN IDs must be provided to be able to receive and send frames from these VLAN IDs on a Channel.

See Also

fcbCANSetCcConfiguration

8.2.2.2 fcEthFilterConfig

This structure describes the configuration of a ethernet filter.

```
Typedef struct fcEthFilterConfig
{
    struct
    {
        fcByte FlagDstMac : 1;
        fcByte FlagSrcMac : 1;
        fcByte FlagVlanId : 1;
        fcByte FlagEtherType : 1;
        fcByte FlagDstIpv4 : 1;
        fcByte FlagSrcIpv4 : 1;
        fcByte FlagDstPort : 1;
        fcByte FlagSrcPort : 1;
    }Flags;

    fcByte Reserved;
    fcByte DstMac[6];
    fcByte SrcMac[6];
    fcWord VlanId;
    fcWord EtherType;
    fcByte DstIpv4[4];
    fcByte SrcIpv4[4];
    fcWord DstPort;
    fcWord SrcPort;
} fcEthFilterConfig;
```

Members

Flags.FlagDstMac

Flag for destination MAC filter.
1: Destination MAC will be used in filter.
0: Destination MAC filter will be not used in filter.

Flags.FlagSrcMac

Flag for source MAC filter.

1: Source MAC will be used in filter.
0: Source MAC will be not used in filter.

Flags.FlagVlanId

Flag for VLAN filter.

1: VLAN will be used in filter.
0: VLAN will be not used in filter.

Flags.FlagEtherType

Flag for EtherType filter.

1: EtherType will be used in filter.
0: EtherType will be not used in filter.

Flags.FlagDstIpv4

Flag for destination IPv4 filter.

1: Destination IPv4 will be used in filter.
0: Destination IPv4 will be not used in filter.

Flags.FlagSrcIpv4

Flag for source IPv4 filter.

1: Source IPv4 will be used in filter.
0: Source IPv4 will be not used in filter.

Flags.FlagDstPort

Flag for desination port filter.

1: Desination port will be used in filter.
0: Desination port will be not used in filter.

Flags.FlagSrcPort

Flag for source port filter.

1: Source port will be used in filter.
0: Source port will be not used in filter.

Reserved

Reserved for future use.

DstMac[6]

Destination MAC address of ethernet frame, which will be accept

SrcMac[6]

Source MAC address of ethernet frame, which will be accept

VlanId

VLAN Id of ethernet frame, which will be accept

EtherType

Type of ethernet frame, which will be accept

DstIpv4[4]

Destination IPv4 address of ethernet frame, which will be accept

SrcIpv4[4]

Source IPv4 address of ethernet frame, which will be accept

DstPort

Destination port of ethernet frame, which will be accept

SrcPort

Source port of ethernet frame, which will be accept

See Also

fcbEthSetFilterConfiguration

8.2.3 fcbEthSetCcConfiguration

This function configures the Ethernet Communication Controller. Call this function before calling *fcbEthMonitoringStart*.

```
fcError fcbEthSetCcConfiguration(
       fcHandle hFlexCard,
       fcCC CC,
       fcEthCcConfig cfg
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Ethernet Communication Controller identifier.

Cfg

[IN] Ethernet Communication Controller configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

8.2.4 fcbEthSetFilterConfiguration

This function configures the ethernet filter for the selected ethernet controller. Only the ethernet frames included in the filter configuration are forwarded, all other ethernet frames are rejected. See the configuration notice for further details. This function can also be called while the ethernet monitoring is running.

```
fcError fcbEthSetFilterConfiguration(
       fcHandle hFlexCard,
       fcCC CC,
       fcBool combinedFilter,
       fcEthFilterConfig filterCfg,
       fcBool reset
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Communication controller index.

combinedFilter

[IN] Flag for combined filter. If value is 1, the filter is a combined filter.
If value is 0, the filter is not a combined filter.

filterCfg

[IN] Filter configuration parameters

Reset

[IN] If value is 1, all configured filters will be reset.
If value is 0, no filter will be reset. New filter will be added to the existing list of filters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcEthFilterConfig

Example

```
// Set a ethernet filter with only VLAN 10 and the already configured filters should
be reset.
fcEthFilterConfig filterCfg = {0};
filterCfg.Flags.FlagVlanId = 1;
filterCfg.VlanId = 10;
fcError e = fcbEthSetFilterConfiguration(handle, ccIndex, false, filterCfg, true);
if (0 != e) {/* Error handling */}


// Set a ethernet combined filter with VLAN 20 and destination port 3333. The already
configured filters should be reset.
fcEthFilterConfig filterCfg = {0};
filterCfg.Flags.FlagVlanId = 1;
filterCfg.Flags.FlagDstPort = 1;
filterCfg.VlanId = 20;
filterCfg.DstPort = 3333;
e = fcbEthSetFilterConfiguration(handle, ccIndex, true, filterCfg, true);
if (0 != e) {/* Error handling */}
```

8.3 Transmit

8.3.1 fcbEthTransmit

This function writes the data bytes in an Ethernet communication controller of the FlexCard. The function returns immediately and does not wait for the data frame to arrive on the bus. The message should normally be transmitted as soon as possible. The messages appear on the bus in the order in which they were transmitted by the user.

```
fcError fcbEthTransmit(
      fcHandle hFlexCard,
      fcCC CC,
      const fcByte* pRawEthernetFrame,
      fcWord lengthOfEthernetFrame
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CC

[IN] Ethernet Communication Controller identifier.

pRawEthernetFrame

[IN] Ethernet raw frame to transmit. This includes the following fields (Layer 2 Ethernet frame without Frame check sequence (32-bit CRC):

MAC Destination	: 6 octets
MAC Source	: 6 octets
802.1Q tag (optional)	: 4 octets
Ethertype (Ethernet II) or length (IEEE 802.3)	: 2 octets
Payload	: 46‐1500 octets

lengthOfEthernetFrame

[IN] Length of the ethernet frame in number of octets.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

9 Self Synchronization API

The following section describes the data structures and features used for Self Sync functionality. To use these functions the FlexCard must have a firmware with exactly one FlexRay CC and the FlexCard must be licensed for FlexRay.

To also be able to test FlexRay nodes that do not take part actively in the synchronization process of a FlexRay network, the FlexCard provides the possibility to generate a second start-up/synchronization frame. Thus, the FlexCard synchronizes the FlexRay network independently. Self synchronization runs on the first Communication Controller.

If TxAcknowledgeEnable is set to disabled for the Self Synchronization feature, no fcPacketTypeTxAcknowledge is generated. No fcBase packet is generated for the Self Synchronization frame, also no fcPacketTypeFlexRayFrame.

If TxAcknowledgeEnable is set to enabled for the Self Synchronization feature, an fcPacketTypeTxAcknowledge with fcCC2 is generated. No fcPacketTypeFlexRayFrame is generated for the Self Synchronization frame.

Information

All enumerations, structures and functions in this chapter are initially supported for FlexCard Cyclone II (SE) devices by:

- FlexCard Windows API version S3V0-F.
- FlexCard Linux API version S2V0-F.
- FlexCard Xenomai API version S4V2-F.

Information

This additional API is also initially supported for:

- FlexCard PMC devices with only one FlexRay Communication Controller and the FlexCard API version S4V2-F.
- FlexCard PMC-II devices with only one FlexRay Communication Controller and the FlexCard API version S5V1-F.
- FlexCard USB-M devices with the FlexCard API version S6V2-F.

9.1 Configuration

9.1.1 fcbConfigureMessageBufferSelfSynchronization

This function configures up to 2 additional start-up/synchronization message buffers. Configuring message buffers is only allowed if the Communication Controller is in its configuration state, `fcStateConfig`. The message buffer needs to be defined as *fcMsgBufCfgTx*. The *SyncFrameIndicator* and *StartupFrameIndicator* need to be set, while *CycleCounterFilter* must be set to 0.

```
FcError fcbConfigureMessageBufferSelfSynchronization(
      fcHandle hFlexCard,
      fcDword* bufferId,
      fcMsgBufCfg cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId

[OUT] Message buffer identifier. If the configuration was successful, the message buffer identifier is greater than 0. This identifier will be required to transmit the content of the buffer.

Cfg

[IN] Message buffer configuration parameters

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

Only one additional start-up/synchronization ID can be defined. Therefore maximum 2 additional message buffers can be configured: *fcChannelA* and *fcChannelB* or *fcChannelBoth*. Before configuring the message buffers, it is necessary to set up the global communication parameters (cluster parameters). Calling **fcbFRSetCcConfiguration**, **fcbFRSetCcConfigurationChi** or **fcbFRSetCcConfigurationCANdb** will reset the additional start-up/synchronization frames.

See Also

fcMsgBufCfg, **fcMsgBufCfgTx**, **fcbReconfigureMessageBufferSelfSynchronization**, **fcbGetCcMessageBufferSelfSynchronization**, **fcbResetCcMessageBuffersSelfSynchronization**, **fcbFRSetMsgBufCfgMode**

Example

```
// The following code configures a self start-up/synch transmit buffer
fcMsgBufCfg cfg;
memset(&cfg, 0, sizeof(fcMsgBufCfg));

cfg.Type = fcMsgBufTx;
cfg.ChannelFilter = fcChannelA;
cfg.CycleCounterFilter = 0x0;       // sync frames must appear in every cycle

cfg.Tx.FrameId = 3;                 // unused slotId of static segment
cfg.Tx.PayloadLength = 2;
cfg.Tx.PayloadLengthMax = 127;
cfg.Tx.PayloadPreambleIndicator = 0;
cfg.Tx.SyncFrameIndicator = 1;      // mandatory to be set to 1
cfg.Tx.StartupFrameIndicator = 1;   // mandatory to be set to 1
cfg.Tx.TxAcknowledgeEnable= 1;
cfg.Tx.TransmissionMode = fcMsgBufTxSingleShot;

unsigned int bufIdx = 0;
fcError e=fcbConfigureMessageBufferSelfSynchronization(hFlexCard,&bufIdx,cfg);
```

9.1.2 fcbReconfigureMessageBufferSelfSynchronization

This function reconfigures the additional transmit message buffers for self start-up/synchronization. A reconfiguration is only allowed for message buffers which are already configured and if the Communication Controller is in its configuration state, `fcStateConfig`. The message buffer needs to be defined for a start-up/synchronization transmit frame. Therefore, it is mandatory to set the *SyncFrameIndicator* and *StartupFrameIndicator* to 1 and the *CycleCounterFilter* to 0.

```
FcError fcbReconfigureMessageBufferSelfSynchronization(
       fcHandle hFlexCard,
       fcDword bufferId,
       fcMsgBufCfg cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId
[IN] The identifier of the message buffer which should be reconfigured.
Cfg
[IN] Message buffer configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcMsgBufCfg, **fcMsgBufCfgTx**, **fcbConfigureMessageBufferSelfSynchronizatio**, **fcbGetCcMessageBufferSelfSynchronization**, **fcbResetCcMessageBuffersSelfSynchronization**, **fcbFRSetMsgBufCfgMode**

9.1.3 fcbReinitializeCcMessageBufferSelfSynchronization

This function re-initializes the message buffer configuration of the self-start-up synchronization Communication Controller. After calling this function the Communication Controller does not send old payload data. Re-initialization of message buffers is only allowed if the Communication Controller is in configuration state.

```
FcError fcbReinitializeCcMessageBufferSelfSynchronization(
      fcHandle hFlexCard
)
```

Parameters

hFlexCard
[IN] Handle to a FlexCard

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information

9.1.4 fcbGetCcMessageBufferSelfSynchronization

This function reads a specific message buffer configuration of the additional message buffers for self start-up/synchronization.

```
FcError fcbGetCcMessageBufferSelfSynchronization(
      fcHandle hFlexCard,
      fcDword bufferId,
      fcMsgBufCfg* cfg
)
```

Parameters

hFlexCard
[IN] Handle to a FlexCard
bufferId
[IN] The identifier of the additional start-up/sync message buffer to be read
cfg
[OUT] The configuration parameters of the specified message buffer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcMsgBufCfg, **fcMsgBufCfgTx**, **fcbConfigureMessageBufferSelfSynchronization**, **fcbReconfigureMessageBufferSelfSynchronization**, **fcbResetCcMessageBuffersSelfSynchronization**

Example

```
// Get all configured additional start-up/synchronization transmit
// message buffers
std::map<unsigned int, fcMsgBufCfg> Buffers;

// valid buffer indexes are only 1 and 2
for(unsigned int bufIdx = 1;bufIdx <=2; bufIdx++)
{
  fcMsgBufCfg cfg;

  // as long no error occurs we try to get each buffer
  fcError e=fcbGetCcMessageBufferSelfSynchronization(m_hFlexCard,bufIdx,&cfg);
  if (0 != e)
      continue;

  //and add it to our list
  Buffers[bufIdx] = cfg;
}
```

9.1.5 fcbResetCcMessageBuffersSelfSynchronization

This function resets the additional start-up/synchronization message buffers.

```
FcError fcbResetCcMessageBufferSelfSynchronization(
      fcHandle hFlexCard
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

9.2 Transmit

9.2.1 fcbTransmitSelfSynchronization

This function writes a data frame into a self start-up/synchronization transmit buffer of the FlexCard. The function returns immediately and does not wait for the data frame to arrive on the bus. The frame should normally be transmitted in the next cycle. If the transmit acknowledgment is activated, an acknowledge packet is generated as soon as the frame has been transmitted. This function should only be called when the FlexCard is in normal active state or when all message buffer configurations have been done. When the user transmits several times new data with the same buffer ID in a very short time periode, it may happen that data is overwritten that was not transmitted yet. If you experience that behavior, wait for the TxAck for the data you wanted to send.

```
FcError fcbTransmitSelfSynchronization(
      fcHandle hFlexCard,
      fcDword bufferId,
      fcWord payload[],
      fcByte payloadLength
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId

[IN] The id of the additional start-up/synchronization message buffer used for the transmission

payload

The payload data to be transmitted

payloadLength

The size of the payload data (number of 2-byte words)

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

The payload data has to be organized as follows: if Data0 is the first byte to transmit and Data1 the second byte to transmit, then the high byte (Bit 8 – 15) of payload[0] contains Data1, the low byte (Bit 0-7) of payload[0] contains Data0, etc.

Parameter `payload`	`payload[0]` (Word 0)		`payload[1]` (Word 1)		…
	High byte	Low byte	High byte	Low byte	…
FlexRay payload segment	Data 1	Data 0	Data 3	Data 2	…

Example

```
fcWord payload[fcPayloadMaximum];
payload[0] = 0x0001; // Update your payload data

fcError e = fcbTransmitSelfSynchronization(m_hFlexCard,bufferIdx,
             payload,payloadLength);
```

10 Trigger API

This chapter shows how to set up the two FlexCard trigger lines. FlexCard Cyclone II (SE) and FlexCard USB-M have 2 unidirectional triggers. One IN and one OUT line. Via the IN trigger line it has the ability to receive trigger events and forward them to the user application. The two trigger lines of the FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3 may be configured as IN or OUT. To configure and activate this feature, use the following structures and functions. The trigger event data (trigger IN) is received as **`fcTriggerExInfoPacket`** with the **`fcbReceive`** function.

The OUT trigger level depends on the FlexCard. Refer to the document Instructions for Use. It may be high or low active. The IN trigger does not detect the voltage level, but it detects either the rising or falling signal edge. This can be configured.

The following table lists the supported triggers during asynchronous and synchronous FlexRay monitoring.

Trigger	Supported in Asynch-Mode	Supported in Synch-Mode	Supported FlexCard
fcTriggerIn	OK	OK	FC Cyclone II (SE), FC USB-M
fcTriggerInOnSWPulse	OK	OK	FC Cyclone II (SE), FC USB-M
fcTriggerInOnSWTimer	OK	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnPulse	OK	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnTimeStampChanged	OK	OK	FC USB-M
fcTriggerOutOnCycle	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnSlotChX	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnSlotInCycleChX	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnCycleStart	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnErrorDetected	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnErrorX	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnStartupCompleted	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerOutOnStartDynamicSegment	-	OK	FC Cyclone II (SE), FC USB-M
fcTriggerPMCIn	OK	OK	FC PMC (II), FC PXIe3/PCIe3
fcTriggerPMCOutOnPulse	OK	OK	FC PMC (II), FC PXIe3/PCIe3
fcTriggerPMCOutOnTimeStampChanged	OK	OK	FC PMC (II), FC PXIe3/PCIe3
fcTriggerPMCOutOnErrorDetected	-	OK	FC PMC (II), FC PXIe3/PCIe3
fcTriggerPMCOutOnStartupCompleted	-	OK	FC PMC (II), FC PXIe3/PCIe3
fcTriggerPMCOutOnCycleStart	-	OK	FC PMC (II), FC PXIe3/PCIe3

Information

The IN trigger line of the FlexCard Cyclone II SE recognizes a trigger impuls only if it is longer than 50 us.

10.1 Enumerations

10.1.1 fcTriggerConditionEx

This enumeration defines the conditions available for a trigger configuration. It is used for the FlexCard Cyclone II (SE) and FlexCard USB-M. The conditions may be binary Ored.

```
Typedef enum fcTriggerConditionEx
{
      fcTriggerIn                              = 0x00000002,
      fcTriggerOutOnPulse                      = 0x00000004,
      fcTriggerInOnSWPulse                     = 0x00000008,
      fcTriggerInOnSWTimer                     = 0x00000010,
      fcTriggerOutOnCycle                     = 0x00000040,
      fcTriggerOutOnSlotChA                   = 0x00000080,
      fcTriggerOutOnSlotChB                   = 0x00000100,
      fcTriggerOutOnSlotInCycleChA            = 0x00000200,
      fcTriggerOutOnSlotInCycleChB            = 0x00000400,
      fcTriggerOutOnTimeStampChanged          = 0x00008000,
      fcTriggerOutOnCycleStart                = 0x00010000,
      fcTriggerOutOnErrorDetected             = 0x00020000,
      fcTriggerOutOnStartupCompleted          = 0x00040000,
      fcTriggerOutOnStartDynamicSegment       = 0x00080000,
      fcTriggerOutOnErrorSFBM                 = 0x00100000,
      fcTriggerOutOnErrorSFO                  = 0x00200000,
      fcTriggerOutOnErrorCCF                  = 0x00400000,
      fcTriggerOutOnErrorSBVA                 = 0x00800000,
      fcTriggerOutOnErrorPERR                 = 0x01000000,
      fcTriggerOutOnErrorEDA                  = 0x02000000,
      fcTriggerOutOnErrorLTVA                 = 0x04000000,
      fcTriggerOutOnErrorTABA                 = 0x08000000,
      fcTriggerOutOnErrorEDB                  = 0x10000000,
      fcTriggerOutOnErrorLTVB                 = 0x20000000,
      fcTriggerOutOnErrorTABB                 = 0x40000000,
      fcTriggerOutOnErrorSBVB                 = 0x80000000,
      fcTriggerOutOnErrorAll                  = 0xFFF00000,
}fcTriggerConditionEx;
```

Members

fcTriggerIn

IN: A trigger packet is generated as soon as the set edge (falling/rising) is detected on the input trigger line.

FcTriggerOutOnPulse

OUT: A signal is generated on the output trigger line as soon as the condition is set to the driver.

FcTriggerInOnSWPulse

IN: A trigger packet is generated as soon as the trigger function is called.

FcTriggerInOnSWTimer

IN: A trigger packet is generated by a set time interval.

FcTriggerOutOnCycle

OUT: A signal is generated on the output trigger line at each start of a set FlexRay cycle.

FcTriggerOutOnSlotChA

OUT: A signal is generated on the output trigger line at each start of a set FlexRay slot on channel A.

fcTriggerOutOnSlotChB

OUT: A signal is generated on the output trigger line at each start of a set FlexRay slot on channel B.

fcTriggerOutOnSlotInCycleChA

OUT: A signal is generated on the output trigger line at each start of a set slot in a set FlexRay cycle on channel A.

fcTriggerOutOnSlotInCycleChB

OUT: A signal is generated on the output trigger line at each start of a set slot in a set FlexRay cycle on channel B.

fcTriggerOutOnTimeStampChanged

OUT: A signal is generated on the output trigger line at each change of the internal FlexCard time stamp.

FcTriggerOutOnCycleStart

OUT: A signal is generated on the output trigger line at a FlexRay cycle start.

FcTriggerOutOnErrorDetected

OUT: A signal is generated on the output trigger line at a detected FlexRay error.

FcTriggerOutOnStartupCompleted

OUT: A signal is generated on the output trigger line at a completed FlexRay start-up.

FcTriggerOutOnStartDynamicSegment

OUT: A signal is generated on the output trigger line at the start of the FlexRay dynamic segment.

FcTriggerOutOnErrorSFBM

OUT: A signal is generated on the output trigger line at the FlexRay error SFBM (sync frame below minimum).

FcTriggerOutOnErrorSFO

OUT: A signal is generated on the output trigger line at the FlexRay error SFO (sync frame overflow).

FcTriggerOutOnErrorCCF

OUT: A signal is generated on the output trigger line at the FlexRay error CCF (clock correction failure).

FcTriggerOutOnErrorSBVA

OUT: A signal is generated on the output trigger line at the FlexRay error SBVA (slot boundary violation channel A).

fcTriggerOutOnErrorPERR

OUT: A signal is generated on the output trigger line at the FlexRay error PERR (parity error).

FcTriggerOutOnErrorEDA

OUT: A signal is generated on the output trigger line at the FlexRay error EDA (error detected on channel A).

fcTriggerOutOnErrorLTVA

OUT: A signal is generated on the output trigger line at the FlexRay error LTVA (latest transmit violation channel A).

fcTriggerOutOnErrorTABA

OUT: A signal is generated on the output trigger line at the FlexRay error TABA (transmission across boundary channel A).

fcTriggerOutOnErrorEDB

OUT: A signal is generated on the output trigger line at the FlexRay error EDB (error detected on channel B).

fcTriggerOutOnErrorLTVB

OUT: A signal is generated on the output trigger line at the FlexRay error LTVB (latest transmit violation channel B).

fcTriggerOutOnErrorTABB

OUT: A signal is generated on the output trigger line at the FlexRay error TABB (transmission across boundary channel B).

fcTriggerOutOnErrorSBVB

OUT: A signal is generated on the output trigger line at the FlexRay error SBVB (slot boundary violation channel B).

fcTriggerOutOnErrorAll

OUT: A signal is generated on the OUT trigger line at detected an error

See Also

fcbSetTrigger, **fcTriggerConfigurationEx**

Remarks

In the FlexRay monitoring mode DebugAsynchron only the conditions *fcTriggerIn*, *fcTriggerOutOnPulse*, *fcTriggerInOnSWTimer*, *fcTriggerInOnSWPulse* and *fcTriggerOutOnTimeStampChanged* can be used.

10.1.2 fcTriggerConditionPMC

This enumeration defines the conditions available for a trigger configuration of a FlexCard PMC (II). Please note that these conditions can not be OR-ed.

```
Typedef enum fcTriggerConditionPMC
{
      fcTriggerPMCNone = 0x00000000,
      fcTriggerPMCIn = 0x00000100,
      fcTriggerPMCOutOnPulse = 0x00001000,
      fcTriggerPMCOutOnTimeStampChanged = 0x00002000,
      fcTriggerPMCOutOnErrorDetected = 0x00010000,
      fcTriggerPMCOutOnStartupCompleted = 0x00020000,
      fcTriggerPMCOutOnCycleStart = 0x00100000,
} fcTriggerConditionPMC;
```

Members

fcTriggerPMCNone

None.

FcTriggerPMCIn

Input trigger. When a trigger impulse is detected a fcTriggerExInfoPacket packet is generated

FcTriggerPMCOutOnPulse

Output trigger. Impulse is generated as soon as the condition is set to the driver

FcTriggerPMCOutOnTimeStampChanged

Output trigger. Impulse is generated if the internal FlexCard time stamp was changed

FcTriggerPMCOutOnErrorDetected

Output trigger. Impulse is generated if a FlexRay error was detected

FcTriggerPMCOutOnStartupCompleted

Output trigger. Impulse is generated when the FlexRay startup was completed

FcTriggerPMCOutOnCycleStart

Output trigger. Impulse is generated at a FlexRay cycle start

See Also

fcbSetTrigger, **fcTriggerConditionEx**

Remarks

In the FlexRay monitoring mode DebugAsynchron only the conditions *fcTriggerPMCNone*, *fcTriggerPMCIn*, *fcTriggerPMCOutOnPulse* and *fcTriggerPMCOutOnTimeStampChanged* can be used.

10.2 Structures

10.2.1 fcTriggerConfigurationEx

This structure configures the triggers of the FlexCard. Using the parameter *Condition* the trigger is enabled. For *Condition* the enumeration *fcTriggerConditionEx* is used for FlexCard Cyclone II (SE) and FlexCard USB-M. To set more than one trigger condition the conditions available in *fcTriggerConditionEx* must be binary OR-ed. Setting *Condition* to zero resets all triggers. In case you add additional trigger conditions, they have to be binary OR-ed with the former ones. Otherwise the previous settings will be reset.

When you use a FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3, use *fcTriggerConditionPMC* as *Condition*. The conditions **cannot be OR-ed**. If you do it nevertheless, none of the conditions are set and an error message is returned. Set the parameter *TriggerLineToConfigure* to either 1 or 2. Set the parameter *TriggerGeneratingCC* to the CC index you wish to use.

Some conditions need additional parameters:

The condition *fcTriggerIn* demands to set the parameter *onEdge*.

The condition *fcTriggerInOnSWTimer* demands to set the parameter *onTimePeriod*.

The condition *fcTriggerOutOnCycle* demands to set the parameter *onCycle*.

The condition *fcTriggerOutOnSlotChA* demands to set the parameter *onSlotChA*.

The condition *fcTriggerOutOnSlotChB* demands to set the parameter *onSlotChB*.

The condition *fcTriggerOutOnSlotInCycleChA* demands to set the parameters *onSlotChA* and *onCycle*.

The condition *fcTriggerOutOnSlotInCycleChB* demands to set the parameters *onSlotChB* and *onCycle*.

The PMC conditions *fcTriggerPMCOutOnErrorDetected*, *fcTriggerPMCOutOnCycleStart* and *fcTriggerPMCOutOnStartupCompleted* demand to set the parameter *TriggerGeneratingCC*.

```
Typedef struct fcTriggerConfigurationEx
{
       fcDword Condition;
       fcDword onEdge;

       // For FlexCard Cyclone II (SE) / FlexCard USB-M
       fcDword onCycle;
       fcDword onSlotChA;
       fcDword onSlotChB;
       fcDword onTimePeriod;
       fcDword Reserved1[4];

       // For FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3:
       fcDword TriggerLineToConfigure;
       fcCC TriggerGeneratingCC;
       fcDword Reserved2[4];
} fcTriggerConfigurationEx;
```

Members

Condition

Set this parameter to 0 to reset all triggers. FlexCard Cyclone II (SE): Set it to an OR-ed combination of conditions available in *fcTriggerConditionEx*.

FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3: Set it to *fcTriggerConditionPMC*. Conditions **cannot** be OR-ed.

OnEdge

This parameter must be set when the condition *fcTriggerIn* is chosen.

Valid values are 0 = falling edge and 1 = rising edge. Setting the trigger edge for output triggers is NOT supported.

OnCycle

This parameter must be set when at least one of the conditions *fcTriggerOutOnCycle*, *fcTriggerOutOnSlotInCycleChA* and *fcTriggerOutOnSlotInCycleChB* are chosen.

Valid values range from 0 to 63.

OnSlotChA

This parameter must be set when at least on of the conditions *fcTriggerOutOnSlotChA* or *fcTriggerOutOnSlotInCycleChA* are chosen.

Valid values range from 1 to 2047.

OnSlotChB

This parameter must be set when at least one of the conditions *fcTriggerOutOnSlotChB* or *fcTriggerOutOnSlotInCycleChB* are chosen.

Valid values range from 1 to 2047.

OnTimePeriod

This parameter is only used in timer mode. The unit is millisecond. On Windows, the minimum granularity is 16 ms.

Reserved1[4]

Reserved.

TriggerLineToConfigure

For FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3:

This parameter sets the trigger line which should be configured.

Trigger line 1 and 2 are on the front panel. See the document Instructions for Use.

Trigger line 3 can be configured on FlexCard PMC-II. It is at the UserIo backplane on pin 16. Please make sure that the signals are within the defined range. See separate document “FlexCard PMC-II Trigger Extension Dio3”. Trigger line 3 is not available in every fpga image. If trigger line 3 is configured, but the feature is not available in the firmware, the error INVALID_PARAMETER is returned.

TriggerGeneratingCC

For FlexCard PMC, FlexCard PMC-II, FlexCard PXIe3, FlexCard PCIe3. This parameter must be set when CC dependent trigger conditions are set. Valid values range from fcCC1 to fcCC4

Reserved2[4]

Reserved.

See Also

fcbSetTrigger, **fcTriggerConditionEx**, **fcTriggerConditionPMC**

10.3 fcbSetTrigger

This function configures and starts/stops triggers. For further information, refer to the structure fcTriggerConfigurationEx.

```
FcError fcbSetTrigger(
      fcHandle hFlexCard,
      fcTriggerConfigurationEx cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

cfg

[IN] The **trigger configuration**

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcTriggerConfigurationEx, **fcTriggerConditionEx**

Example for FlexCard Cyclone II (SE) or FlexCard USB-M

```
// Generate a pulse at the beginning of a detected FlexRay error and on
// FlexRay cycle 3
fcTriggerConfigurationEx triggerCfg;
memset(&triggerCfg, 0, sizeof(fcTriggerConfigurationEx));
triggerCfg.Condition = 0;
triggerCfg.Condition |= (fcDword)fcTriggerOutOnErrorDetected;
triggerCfg.Condition |= (fcDword)fcTriggerOutOnCycle;
triggerCfg.onCycle = 3;
// Generate a trigger packet all 1000 milliseconds
triggerCfg.Condition |= (fcDword)fcTriggerInOnSWTimer;
triggerCfg.onTimePeriod = 1000;

fcError e = fcbSetTrigger(hFlexCard,triggerCfg);
```

Example for FlexCard PMC (II)

```
// Generate a pulse on trigger line 1 when the Communication Controller 2
// completed its FlexRay start-up
fcTriggerConfigurationEx triggerCfg;
memset(&triggerCfg, 0, sizeof(fcTriggerConfigurationEx));
triggerCfg.Condition = fcTriggerPMCOutOnStartupCompleted;
triggerCfg.TriggerLineToConfigure = 1;
triggerCfg.TriggerGeneratingCC = fcCC2;

fcError e = fcbSetTrigger(hFlexCard,triggerCfg);

// Generate a trigger packet when a pulse on trigger line 2 is detected
triggerCfg.Condition = fcTriggerPMCIn;
triggerCfg.TriggerLineToConfigure = 2;

e = fcbSetTrigger(hFlexCard,triggerCfg);
```

11 Termination API

This chapter shows how to configure the on-board termination of the FlexCard PMC and FlexCard PMC-II.

11.1 Enumerations

11.1.1 fcBusChannel

This enumeration defines the bus channels available on the card.

```
Typedef enum fcBusChannel
{
      fcBusChannel1 = 1,
      fcBusChannel2 = 2,
      fcBusChannel3 = 3,
      fcBusChannel4 = 4,
      fcBusChannel5 = 5,
      fcBusChannel6 = 6,
      fcBusChannel7 = 7,
      fcBusChannel8 = 8,
} fcBusChannel;
```

Members

fcBusChannel1
Identifies bus channel 1.

FcBusChannel2
Identifies bus channel 2.

FcBusChannel3
Identifies bus channel 3.

FcBusChannel4
Identifies bus channel 4.

FcBusChannel5
Identifies bus channel 5.

FcBusChannel6
Identifies bus channel 6.

FcBusChannel7
Identifies bus channel 7.

FcBusChannel8
Identifies bus channel 8.

See Also

fcbSetBusTerminationCc, **fcbGetBusTerminationCc**, **fcbSetBusTermination**, **fcbGetBusTermination**

11.2 fcbSetBusTerminationCc

This function sets the bus termination individually for each Communication Controller channel.

```
FcError fcbSetBusTerminationCc(
      fcHandle hFlexCard,
      fcBusType BusType,
      fcCC CC,
      fcChannel Channel
      fcBool bTermination
)
```

Parameters

hFlexCard
[IN] Handle to a FlexCard

BusType
[IN] The bus type termination.

CC
[IN] Index of the bus type Communication Controller.

Channel
[IN] FlexRay channel(s) that shall be terminated. This parameter needs only to be set for *fcBusTypeFlexRay*.

BTermination
[IN] Set the value <> 0 to enable the bus termination. Set the value 0 to disable the bus termination.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

The termination will not be switched off by the driver automatically if the application closes the device or the driver will be unloaded. So, the bus will not be disturbed by termination loss in case the user application fails.

See Also

fcbGetBusTerminationCc

ℹ	**Information**
	This function is initially supported by FlexCard API version S6V1-F.

11.3 fcbGetBusTerminationCc

This function reads the bus termination individually for each Communication Controller channel.

```
FcError fcbGetBusTerminationCc(
      fcHandle hFlexCard,
      fcBusType BusType,
      fcCC CC,
      fcChannel Channel
      fcBool* pbTermination
)
```

Parameters

hFlexCard
[IN] Handle to a FlexCard

BusType
[IN] The bus type termination.

CC
[IN] Index of the bus type Communication Controller.

Channel

[IN] States the FlexRay channel(s) for which the termination status is read. This parameter needs only be set for *fcBusTypeFlexRay*.

PbTermination

[OUT] The current bus termination. The value 0 indicates a disabled termination. A value <> 0 indicates an enabled termination.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbSetBusTerminationCc

ℹ	Information
	This function is initially supported by FlexCard API version S6V1-F.

11.4 fcbSetBusTermination

This function sets the bus termination individually for each hardware bus channel.

```
FcError fcbSetBusTermination(
       fcHandle hFlexCard,
       fcBusChannel BusChannel,
       fcBusType BusType,
       fcBool bTermination
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

BusChannel

[IN] The bus channel describes the channel at which the termination should be switched on or off.

BusType

[IN] The bus type describes which bus protocol/transceiver is used for the channel. Different bus protocols/transceivers demand different bus terminations.

BTermination

[IN] This parameter enables or disables the bus termination

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbGetBusTermination

Example

```
fcBusChannel busChannel = fcBusChannel3;
fcBusType busType = fcBusTypeFlexRay;
bool bTerm = true; // enable termination

// set FlexRay termination on bus channel 3
fcError e = fcbSetBusTermination(m_hFlexCard,busChannel,busType,bTerm);
```

Remarks

The termination will not be switched off by the driver automatically if the application closes the device or the driver will be unloaded. So, the bus will not be disturbed by termination loss in case the user application fails.

The bus channels for a FlexCard PMC (II) are named channel1 to channel 8 as shown in the figures below. Please note that the bus type (FlexRay or CAN) of channel 3 and 4 for a FlexCard PMC/PCI need to be set by dip switches as described in the FlexCard PMC (II) instructions for use.

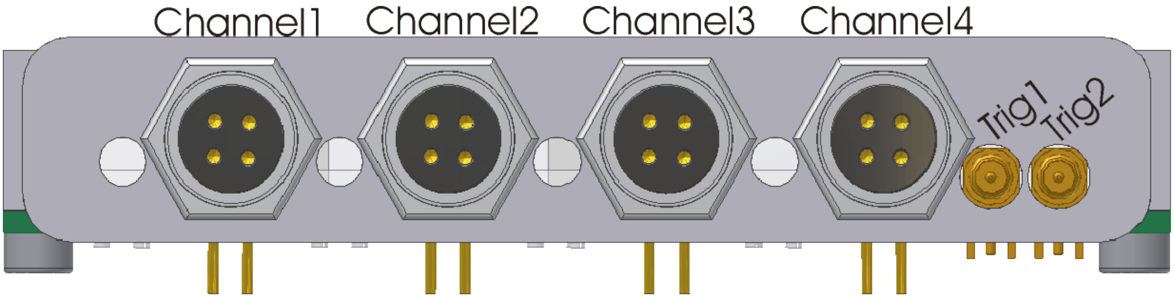


Figure 13: FlexCard PMC front panel

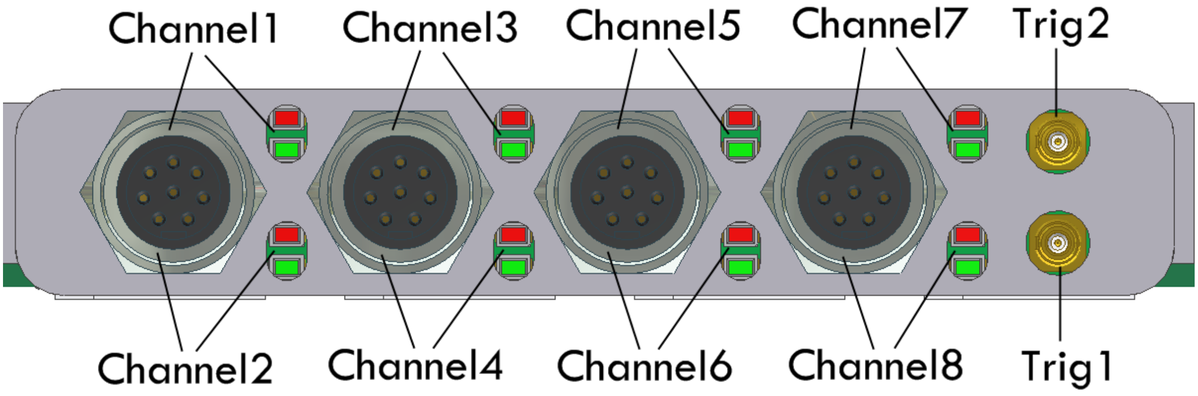


Figure 14: FlexCard PMC-II front panel

11.5 fcbGetBusTermination

This function reads the current bus termination individually for each hardware bus channel.

```
FcError fcbGetBusTermination(
       fcHandle hFlexCard,
       fcBusChannel BusChannel,
       fcBusType BusType,
       fcBool* pbTermination
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

BusChannel

[IN] The bus channel of the termination.

BusType

[IN] The bus type describes which bus termination type has be to be checked. Currently only FlexRay bus terminations are available.

PbTermination

[OUT] The current bus termination. The value 0 indicates a disabled termination. A value <> 0 indicates an enabled termination. The caller must provide memory for this parameter.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbSetBusTermination

12 Firmware API

This chapter shows how to get information about the firmware of a FlexCard and switch to another firmware slot. The functions only work on FlexCard PMC-II and FlexCard USB-M. The FlexCard USB-M offers 2 firmware slots while the FlexCard PMC-II offers 8.

12.1 Structures

12.1.1 fcFWInfo

This struct defines the information that is read from the hardware. Only used for the FlexCard PMC-II application firmware images. The information is read with *fcbFWGetImageInfo*.

```
Typedef struct fcFWInfo
{
      fcNumberCC NumberOfCCs;
      fcVersionNumber FwVersion;
      fcDword bIsActive : 1;
      fcDword Reserved[3];
} fcFWInfo;
```

Members

NumberOfCCs

Contains the number of available CCs that are supported by the firmware. Note that the CCs only work if they are licensed.

FwVersion

Contains the version number of the firmware.

bIsActive

Shows whether the stated firmware index is currently running on the FlexCard PMC-II.

Reserved

Data is reserved for future use.

See Also

fcbFWGetImageInfo, **fcNumberCC**, **fcVersionNumber**

ℹ	Information
	This function is initially supported by FlexCard API version S6V1-F.

12.2 fcbFWGetImageInfo

This function reads information about a firmware slot. If supported by the FlexCard, it has a number of firmware slots and each slot can hold a firmware (also called user image). A user image can be activated by **fcbFWSelectImage**. The firmware slot update or activation on a FlexCard PMC-II take effect after a complete shut down of the PC. On a FlexCard USB-M, the changes take effect after reconnecting it. Whether an image slot is active at the moment can be checked with the flag *bIsActive* in the struct **fcFWInfo**.

```
FcError fcbFWGetImageInfo (
       fcHandle hFlexCard,
       fcDword index,
       fcFWInfo* pFWInfo
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

`index`

[IN] The firmware slot whose information should be read.

`PFWInfo`

[OUT] Pointer to the image information struct.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcFWInfo`, **`fcbFWSelectImage`**

	Information
ℹ	This function is initially supported by FlexCard API version S6V1-F.

12.3 fcbFWSelectImage

This function selects a firmware slot. If supported by the FlexCard, it has a number of firmware slots and each slot can hold a firmware (also called user image). The firmware slot update or activation on a FlexCard PMC-II takes effect after a complete shut down of the PC. On a FlexCard USB-M, the changes take effect after reconnecting it. Information about an image slot can be read with **`fcbFWGetImageInfo`**.

```
FcError fcbFWSelectImage(
       fcHandle hFlexCard,
       fcDword index
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

`index`

[IN] The firmware slot that should be activated.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcbFWGetImageInfo`

	Information
ℹ	This function is initially supported by FlexCard API version S6V1-F.

13 Additional Linux API

There are additional functions available in the FlexCard Linux driver.

ℹ	Information
	This chapter refers to the new FlexCard Linux driver! For the old FlexCard Linux driver (S5V3), please refer to the API Documentation for S5V3.

13.1 Integration

For a detailed description of the installation process, please refer to the text file *Read_Me.txt* which is included in the *tar.gz* archive.

To use the additional Linux API, please include the header file *fcBaseLinux.h* in your application.

After a successful installation please check the correct device initialization with *cat /proc/flexcard* for the FlexCard PMC-2 or with *cat /proc/flexcard_ng* for the FlexCard PXIe3/PCIe3.

ℹ	Information
	For the FlexCard PMC-2 use the driver *flexcard* and library *libfcBase.so.* For the FlexCard PXIe3 and FlexCard PCIe3 use the driver *flexcard_ng* and library *libfcBase_ng.so*.

13.2 Event

13.2.1 fcbSetEventHandleSemaphore

This function registers an event handle (as semaphore) for a specific notification type. *HEvent* must be an unnamed POSIX semaphore from type (sem_t).

```
fcError fcbSetEventHandleSemaphore(
       fcHandle hFlexCard,
       fcCC CC,
       fcHandle hEvent,
       fcNotificationType type
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

hEvent

[IN] Event handle to be registered to signal when a new cycle starts, or a timer interval has elapsed depending on the given type.

Type

[IN] The notification type for which the event must be registered.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcNotificationType**

Remarks

The table below gives an overview of the **`fcNotificationType`** which are CC specific and which is not.

CC specific fcNotificationType	CC global fcNotificationType
fcNotificationTypeFRCycleStarted fcNotificationTypeFRWakeup fcNotificationTypeFRCcTimer	fcNotificationTypeTimer

14 Additional Xenomai API

There is a difference in the event handling between the FlexCard Xenomai driver and the other drivers for the FlexCard products. Instead of **`fcbSetEventHandleV2`** function, the **`fcbWaitForEventV2`** function should be used.

Please note that if the fcbOpen call took place in non-real-time context, fcbClose must be issued within non-real-time as well. Otherwise, the call to fcbClose will fail.

⚠	⚠ **CAUTION**
	The FlexCard **Xenomai driver** version S6V5-F does **not** support the FlexCard Cyclone II (SE) officially. Please do not use these cards with enabled device interrupts. This may lead to system freeze.

14.1 Integration

For a detailed description of the installation process, please refer to the text file *Read_Me.txt* which is included in the zip archive.

After a successful installation please check the correct device with 'cat/proc/xeno_flexcard'. All installed devices must be shown with versions and irq info. Please compare the irq info with used irqs (cat/proc/interrupts). Make sure no non real time device shares an irq with a FlexCard.

To use the additional Xenomai API, please include the header file *fcBaseXENOMAI.h* in your application.

14.2 Structures

14.2.1 fcFROffsetSynchronization

This structure describes the configuration of a FlexRay offset synchronization.

```
Typedef struct fcFROffsetSynchronization
{
    fcBool activate;
    fcCC masterCc;
    fcCC slaveCc;
    fcBool relative;
    fcDword cycleOffset;
    fcDword macrotickOffset;
    fcBool resync;
    fcDword resyncMacrotickOffset;
    fcDword reserved[6];
} fcFROffsetSynchronization;
```

Members

`activate`

Activate the firmware offset synchronization between a master and a slave FlexRay network.

`MasterCc`

The master CC to which the slave will be synchronized with a precise time delay. Currently must be fcCC1!

`SlaveCc`

The slave CC which follows the master with a precise time delay. Currently must be fcCC2!

`Relative`

When the bit relative is activated, the firmware synchronizes the master and the slave CC but disregards the cycle number.

`CycleOffset`

The cycle offset value.

`MacrotickOffset`

The macrotick offset value.

Resync

When the bit resync is activated, the firmware stops to establish an offset synchronization after a certain amount of macroticks passed. This duration is defined with the variable resyncMacrotickOffset. The application has to call fcbFRMonitoringStop and then fcbFRMonitoringStart in order to retry the offset synchronization. This is useful when the target and actual value are far apart, and the firmware would take very long to bring them together.

ResyncMacrotickOffset

Value in macroticks in which the firmware tries to establish an offset synchronization.

Reserved[6]

Data is reserved for future use.

14.3 Event

14.3.1 fcbWaitForEventV2

This function makes a safe real time I/O-Control that blocks the user process in kernel-space, until an event of the given type occurs, or the event does not appear within the specified amount of time. The driver's kernel interrupt service routine then unblocks, and the program routine continues. You don't need to set a handle with **fcbSetEventHandle (Obsolete)** or **fcbSetEventHandleV2**.

```
FcError fcbWaitForEventV2(
       fcHandle hFlexCard,
       fcCC CC,
       fcNotificationType type,
       fcDword nTimeout
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

type

[IN] The notification type for which event has to be waited for.

NTimeout

[IN] The maximum amount of time in usec to wait for the event.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCC, **fcNotificationType**

Remarks

The table below gives an overview of the **fcNotificationType** which are CC specific and which is not.

CC specific fcNotificationType	CC global fcNotificationType
fcNotificationTypeFRCycleStarted fcNotificationTypeFRWakeup fcNotificationTypeFRCcTimer	fcNotificationTypeTimer

14.4 Initialization

14.4.1 fcbFRSetOffsetSynchronization

This function synchronizes two FlexRay networks with a defined time delay. The master always comes first, and the slave follows. This feature may be used to route frames from the master to the slave CC in the same FlexRay cycle number. Make sure that the time offset is big enough to route the frames. However, when you make the time offset too large, routing frames the other way (from the slave to the master) will lead to a bigger time offset. The user must make a trade-off here.

The following figure shows a FlexRay offset synchronization with an in-cycle offset of half a cycle and a cycle offset of 0. This makes it possible to receive ID 3, manipulate the value and send it on the slave network in the same cycle number.

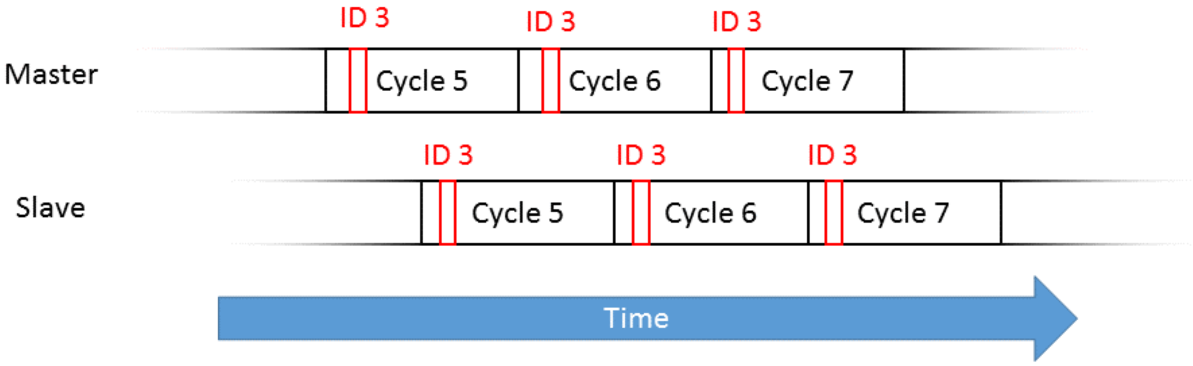


Figure 15: Example for a FlexRay offset synchronization

The user must set the parameters pExternOffsetCorrection and pExternRateCorrection via fcbFRSetCcConfigurationChi or fcbFRSetCcConfiguration. These parameters control how big an external offset correction is. Valid values for these parameters are 0 to 7 microticks.

VExternOffsetControl and vExternRateControl can have three values:

- 0 (no offset control)
- 2 (add the correction value)
- 3 (substract the correction value)

The user must set those values to 0. The FlexCard sets these values. This way it controls the time offset between the master and the slave.

POffsetCorrectionOut and pRateCorrectionOut hold the maximum allowed offset correction values.

- pOffsetCorrectionOut: 5 to 15266 microticks
- pRateCorrectionOut: 2 to 1923 microticks

Those parameters should be set high enough.

Example: pExternOffsetCorrection is set to 7 microticks. When the defined time offset between the master and the slave is not reached yet, the FlexCard control adds 7 microticks offset each cycle.

Calling fcbFRSetOffsetSynchronization is only allowed when the CCs are not running. The offset synchronization is only possible if `fcMonitoringNormal` is used on both CCs and the slave CC sends a startup/sync frame.

When the offset synchronization is activated, the startup/sync frame from the slave CC should not come immediately after a different sync frame. E.g., when a communication partner uses frame Id 5 as sync frame, the slave CC should use frame Id 7 or higher with startup/sync bit.

Please note that a firmware with FlexRay offset synchronization is required.

This function is experimental.

```
FcError fcbFRSetOffsetSynchronization(
            fcHandle hFlexCard,
            fcFROffsetSynchronization offsetSync
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

offsetSync

[IN] Configures the offset synchronization.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
// precondition: valid flexcard handle exists

// add code here to call fcbFRSetCcConfiguration or
// fcbFRSetCcConfigurationChi for fcCC1 and fcCC2.

// add code here to configure the FlexRay message buffers

fcFROffsetSynchronization offsetSync;
memset(&offsetSync, 0, sizeof(offsetSync));

offsetSync.activate = 1;
offsetSync.masterCc = fcCC1;
offsetSync.slaveCc = fcCC2;
offsetSync.relative = 0;
offsetSync.cycleOffset = 3;
offsetSync.macrotickOffset = 20;
offsetSync.resync = 0;
offsetSync.resyncMacrotickOffset = 0;

// create a time offset so that the slave CC runs 3
// cycles and 20 macroticks later than the master CC.
fcError e = fcbFRSetOffsetSynchronization(hFlexCard, offsetSync);
if (0 == e)
{
      // add code here to start monitoring on fcCC1 and fcCC2
}
else
{
      // error handling …
}
```

14.5 Obsolete

14.5.1 fcbWaitForEvent (Obsolete)

	Information
ℹ	This function is obsolete. Please use **`fcbWaitForEventV2`** instead.

This function makes a safe real time I/O-Control that blocks the user process in kernel-space, until an event of the given type occurs, or the event does not appear within the specified amount of time. The driver's kernel interrupt service routine then unblocks, and the program routine continues. You don't need to set a handle with **`fcbSetEventHandle (Obsolete)`**.

```
FcError fcbWaitForEvent(
       fcHandle hFlexCard,
       fcNotificationType hEvent,
       fcDword nTimeout
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

hEvent

[IN] The notification type for which event must be waited for.

NTimeout

[IN] The maximum amount of time in usec to wait for the event.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcNotificationType

Remarks

The table below gives an overview of the fcNotificationType which are CC specific and which is not. To use the CC specific fcNotificationType, the CC index has to be set.

CC specific fcNotificationType	CC global fcNotificationType
fcNotificationTypeCycleStarted fcNotificationTypeWakeup	fcNotificationTypeTimer

15 Additional VxWorks API

The VxWorks driver provides additional functionality for the FlexCard PMC. Please note that there are also some fcBase API functions and type definitions which were changed or are not supported by the VxWorks driver. The VxWorks driver only supports multiple CC indexes by using the function `fcbSetCcIndex`. The FRxxx functions are not supported.

To use the driver in a user application, the header files *fcPmcDrv.h*, *fcBaseTypesVxWorks.h* and *fcbBaseVxWorks.h* must be included in that order.

15.1 Integration

For a detailed description of the installation process, please refer to the text file *Read_Me.txt* which is included in the zip archive.

	Information
ℹ	The FlexCard **VxWorks driver** version S1V2-F supports only the FlexCard PMC firmware version S1V2-F. The FlexCard **VxWorks driver** version S2V1-F supports only the FlexCard PMC firmware version S5V2-F with 2 FR controllers. Other driver and versions are not compatible.

15.1.1 fcDrvInit

This function initializes the FlexCard PMC VxWorks driver.

```
STATUS fcDrvInit()
```

Return values

If the function succeeds, the return value is (OK). If the value is (ERROR) the driver could not be initialized.

See Also

`fcDrvExit`

15.1.2 fcDrvExit

This function finalizes the FlexCard PMC VxWorks driver.

```
STATUS fcDrvExit()
```

Return values

If the function succeeds, the return value is (OK). If the value is (ERROR) the driver could not be finalized.

See Also

`fcDrvInit`

15.2 Restrictions / Changes

15.2.1 Not Supported Type Definitions

The VxWorks driver does not support the following type definitions:

```
fcFreeMemory
fcTriggerCondition (Obsolete)
fcTriggerType (Obsolete)
```

```
fcTriggerMode (Obsolete)
fcTriggerCfgHardware (Obsolete)
fcTriggerCfgSoftware (Obsolete)
fcTriggerCfg (Obsolete)
fcTriggerInfoPacket (Obsolete)
fcTriggerConditionEx
```

15.2.2 Changed Type Definitions

15.2.2.1 fcVersion

This structure provides version information about the FlexCard hardware and software components.

```
Typedef struct fcVersion
{
       fcVersionNumber    DeviceDriver;
       fcVersionNumber    Firmware;
       fcVersionNumber    Hardware;
       fcCCType           CCType;
       fcVersionNumber    CC;
       fcVersionNumber    BusGuardian;
       fcVersionNumber    Protocol;
       fcDword            Serial;
       fcFlexCardDeviceId DeviceId;
       fcDword            Reserved[3];
} fcVersion;
```

Members

DeviceDriver

Device driver version

Firmware

Firmware (gateway software) version

Hardware

FlexCard hardware version

CCType

Communication controller type

CC

Communication controller module version

BusGuardian

Bus Guardian version

Protocol

FlexRay Protocol version

Serial

FlexCard serial number. A zero value indicates a non-valid FlexCard serial number.

DeviceId

Device identifier to detect the FlexCard type (FlexCard Cyclone II, FlexCard Cyclone II SE or FlexCard PMC)

Reserved[3]

Reserved for internal purpose

See Also

fcInfo, fcbGetEnumFlexCards (Obsolete), fcFlexCardDeviceId

15.2.2.2 fcTriggerConfigurationEx

This structure is used for the configuration of a trigger. By using the parameter *Condition* several triggers can be enabled. The trigger conditions of the FlexCard PMC are defined in the enumeration **fcTriggerConditionPMC**. The **conditions cannot be combined (OR-ed)**. If it is done, none of the conditions will be set and an error message will be returned.

The conditions fcTriggerPMCOutOnErrorDetected, fcTriggerPMCOutOnCycleStart and fcTriggerPMCOutOnStartupCompleted demand to set the parameter *TriggerGeneratingCC*. Please note the FlexCard trigger lines are not hard defined as IN or OUT trigger lines. Therefore, a valid value has always to be set for the parameter *TriggerLineToConfigure*.

```
Typedef struct fcTriggerConfigurationEx
{
      fcDword Condition;
      fcDword onEdge;
      fcDword TriggerLineToConfigure;
      fcCC    TriggerGeneratingCC;
      fcDword Reserved[4];
} fcTriggerConfigurationEx;
```

Members

Condition

This parameter can either be set to 0 (fcTriggerPMCNone) to reset the trigger or to any condition available in fcTriggerConditionPMC.

OnEdge

This parameter must be set when the condition fcTriggerPMCIn is chosen. Valid values are 0 = falling edge and 1 = rising edge.

TriggerLineToConfigure

This parameter sets the trigger line which should be configured. Valid values range from 1 to 2.

TriggerGeneratingCC

This parameter has to be set when a CC dependent trigger condition was set. Valid values range from fcCC1 to fcCC2.

Reserved[4]

Reserved Dwords for possible later use.

See Also

fcbSetTrigger, **fcTriggerConditionPMC**

15.2.2.3 fcNotificationType

This enumeration defines different notification types. These types are used in the functions **fcbSetEventHandle.** and **fcbSetNotificationTypeCount** to specify on which kind of event the application has to be notified.

```
Typedef enum fcNotificationType
{
      fcNotificationTypeCycleStarted      = 1,
      fcNotificationTypeTimer             = 2,
      fcNotificationTypeWakeup            = 3,
      fcNotificationTypeRxCount           = 4,
      fcNotificationTypeTxCount           = 5,
      fcNotificationTypeInfoCount         = 6,
      fcNotificationTypeErrorCount        = 7,
      fcNotificationTypeStatusCount       = 8,
      fcNotificationTypeTriggerCount      = 9,
      fcNotificationTypeNMVCount          = 10,
      fcNotificationTypeNotificationCount = 11,
      fcNotificationTypeCcTimer           = 12,
} fcNotificationType;
```

Members

fcNotificationTypeCycleStarted

Used to notify that a new cycle has started, and that probably new data has been received.

FcNotificationTypeTimer

Used to notify that the timer interval has elapsed. This notification requires the internal timer of the FlexCard to be enabled (See **fcbSetTimer**).

FcNotificationTypeWakeup

Used to notify that one of the transceivers has received a wake-up event (only if one of the transceivers was in sleep mode).

FcNotificationTypeRxCount

Used to notify that the configured amount of FlexRay frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeTxCount

Used to notify that the configured amount of TxAcknowledge frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeInfoCount

Used to notify that the configured amount of info frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeErrorCount

Used to notify that the configured amount of error frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeStatusCount

Used to notify that the configured amount of status frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeTriggerCount

Used to notify that the configured amount of trigger frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeNMVCount

Used to notify that the configured amount of network management vector frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeNotificationCount

Used to notify that the configured amount of notification frames has been received. This notification can be configured (See **fcbSetNotificationTypeCount**).

FcNotificationTypeCcTimer

Used to notify that the configured CC timer macrotick offset has elapsed.

See Also

fcbMonitoringStart, **fcbSetEventHandle**, **fcbSetNotificationTypeCount**, **fcbSetTimer**, **fcbSetCcTimerConfig (Obsolete)**

15.2.2.4 fcTriggerExInfoPacket

This structure provides information about a trigger packet.

```
Typedef struct fcTriggerExInfoPacket
{
      fcDword Condition;
      fcDword TimeStamp;
      fcDword SequenceCount;
      fcDword Edge;
      fcDword TriggerLine;
      fcDword Reserved[4];
} fcTriggerExInfoPacket;
```

Members

Condition

The fulfilled condition which has caused the trigger packet generation.

TimeStamp

The FlexCard time stamp (1 µs resolution). Indicates the time at which the packet was generated.

SequenceCount

Sequence count for each signal.

Edge

The edge on which the trigger was signalled.

TriggerLine

The trigger line which detected a trigger signal.

Reserved[4]

Reserved for future use.

See Also

fcPacket

15.2.2.5 fcPacketType

This enumeration contains the different packet types.

```
Typedef enum fcPacketType
{
      fcPacketTypeInfo            = 1,
      fcPacketTypeFlexRayFrame    = 2,
      fcPacketTypeError           = 3,
      fcPacketTypeStatus          = 4,
      fcPacketTypeTxAcknowledge   = 6,
      fcPacketTypeNMVector        = 7,
      fcPacketTypeNotification    = 8,
      fcPacketTypeTriggerEx       = 9,
} fcPacketType;
```

Members

fcPacketTypeInfo

Frame is an info packet.

FcPacketTypeFlexRayFrame

Frame is a FlexRay frame.

FcPacketTypeError

Frame is an error packet.

FcPacketTypeStatus

Frame is a status packet.

FcPacketTypeTxAcknowledge

Frame is a transmit acknowledge packet.

FcPacketTypeNMVector

Frame is a network management vector packet.

FcPacketTypeNotification

Frame is a notification packet.

FcPacketTypeTriggerEx

Frame is a trigger packet.

See Also

fcPacket, **fcInfoPacket**, **fcFlexRayFrame**, **fcTxAcknowledgePacket**, **fcErrorPacket**, **fcStatusPacket**, **fcNMVectorPacket**, **fcNotificationPacket**, **fcTriggerExInfoPacket**

15.2.2.6 fcPacket

This structure provides information about a packet.

```
Typedef struct fcPacket
{
      fcPacketType Type;
      union
      {
            fcFlexRayFrame*         FlexRayFrame;
            fcInfoPacket*           InfoPacket;
            fcErrorPacket*          ErrorPacket;
            fcStatusPacket*         StatusPacket;
            fcTriggerExInfoPacket*  TriggerExPacket;
            fcTxAcknowledgePacket*  TxAcknowledgePacket;
            fcNMVectorPacket*       NMVectorPacket;
            fcNotificationPacket*   NotificationPacket;
      };
      fcPacket* pNextPacket;
} fcPacket;
```

Members

Type

Type of packet.

FlexRayFrame

Pointer to the packet data. The content depends on the type of packet.

InfoPacket

Pointer to the packet data. The content depends on the type of packet.

ErrorPacket

Pointer to the packet data. The content depends on the type of packet.

StatusPacket

Pointer to the packet data. The content depends on the type of packet.

TriggerExPacket

Pointer to the packet data. The content depends on the type of packet.

TxAcknowledgePacket

Pointer to the packet data. The content depends on the type of packet.

NMVectorPacket

Pointer to the packet data. The content depends on the type of packet.

NotificationPacket

Pointer to the packet data. The content depends on the type of packet.

PNextPacket

Pointer to the next packet. If the pointer is NULL, there are no more packets available.

See Also

fcPacketType, fcInfoPacket, fcFlexRayFrame, fcTxAcknowledgePacket, fcErrorPacket, fcStatusPacket, fcNMVectorPacket, fcNotificationPacket, fcTriggerExInfoPacket

15.2.2.7 fcState

This enumeration defines the possible Communication Controller POC states (FlexRay Protocol Specification: Vpoc!State). For more details about Communication Controller POC states, please refer to [3].

```
Typedef enum fcState
{
        fcStateUnknown = 0,
        fcStateDefaultConfig,
        fcStateReady,
        fcStateNormalActive,
        fcStateNormalPassive,
        fcStateHalt,
        fcStateMonitorMode,
        fcStateConfig,

        fcStateWakeupStandby,
        fcStateWakeupListen,
        fcStateWakeupSend,
        fcStateWakeupDetect,

        fcStateStartupPrepare,
        fcStateColdstartListen,
        fcStateColdstartCollisionResolution,
        fcStateColdstartConsistencyCheck,
        fcStateColdstartGap,
        fcStateColdstartJoin,
        fcStateIntegrationColdstartCheck,
        fcStateIntegrationListen,
        fcStateIntegrationConsistencyCheck,
        fcStateInitializeSchedule,
        fcStateAbortStartup,
        fcStateStartupSuccess,
}fcState;
```

Members

fcStateUnknown

Communication controller state is not known.

FcStateDefaultConfig

Communication controller is in DEFAULT_CONFIG state.

FcStateReady
: Communication controller is in READY state.

FcStateNormalActive
: Communication controller is in NORMAL_ACTIVE state.

FcStateNormalPassive
: Communication controller is in NORMAL_PASSIVE state.

FcStateHalt
: Communication controller is in HALT state.

FcStateMonitorMode
: Communication controller is in MONITORMODE state

fcStateConfig
: Communication controller is in CONFIG state.

FcStateWakeupStandby
: Communication controller is in WAKEUP_STANDBY state.

FcStateWakeupListen
: Communication controller is in WAKEUP_LISTEN state.

FcStateWakeupSend
: Communication controller is in WAKEUP_SEND state.

FcStateWakeupDetect
: Communication controller is in WAKEUP_DETECT state.

FcStateStartupPrepare
: Communication controller is in STARTUP_PREPARE state.

FcStateColdstartListen
: Communication controller is in COLDSTART_LISTEN state.

FcStateColdstartCollisionResolution
: Communication controller is in COLDSTART_COLLISION_RESOLUTION state.

FcStateColdstartConsistencyCheck
: Communication controller is in COLDSTART_CONSISTENCY_CHECK state.

FcStateColdstartGap
: Communication controller is in COLDSTART_GAP state.

FcStateColdstartJoin
: Communication controller is in COLDSTART_JOIN state.

FcStateIntegrationColdstartCheck
: Communication controller is in INTEGRATION_COLDSTART_CHECK state.

FcStateIntegrationListen
: Communication controller is in INTEGRATION_LISTEN state.

FcStateIntegrationConsistencyCheck
: Communication controller is in INTEGRATION_CONSISTENCY_CHECK state.

FcStateInitializeSchedule
: Communication controller is in INITIALIZE_SCHEDULE state.

FcStateAbortStartup
: Communication controller is in ABORT_STARTUP state.

FcStateStartupSuccess
: Communication controller is in STARTUP_SUCCESS state.

See Also

fcbGetCcState, **fcbMonitoringStart**

15.2.2.8 fcFlexRayFrame

This structure is equivalent to the FlexRay frame described in the FlexRay specification [3].

```
Typedef struct fcFlexRayFrame
{
    fcDword ID : 11;
    fcDword STARTUP : 1;
    fcDword SYNC : 1;
    fcDword NF : 1;
    fcDword PP : 1;
    fcDword R : 1;
    fcDword PayloadLength : 7;
    fcDword CycleCount : 6;
    fcDword HeaderCRC : 11;
    fcWord* pData;

    fcChannel Channel;
    fcDword ValidFrame : 1;
    fcDword SyntaxError : 1;
    fcDword ContentError : 1;
    fcDword SlotBoundaryViolation : 1;
    fcDword AsyncMode : 1;
    fcDword FrameCRC : 24;

    fcDword TimeStamp;
    fcCC    CC;
} fcFlexRayFrame;
```

Members

ID

The frame id defines the slot in which the frame was transmitted.
(FlexRay Protocol Specification: Vrf!Header!FrameID)

STARTUP

Indicates if the frame is a start-up frame (=1) or not (=0)
(FlexRay Protocol Specification: Vrf!Header!SuFIndicator)

SYNC

Indicates if the frame is a sync frame (=1) or not (=0)
(FlexRay Protocol Specification: Vrf!Header!SyFIndicator)

NF

Set to 0, the null frame indicator indicates that *pData* contains no valid data. Set to 1, it indicates that *pData* contains valid data.
(FlexRay Protocol Specification: Vrf!Header!NFIndicator)

PP

The payload preamble indicator indicates whether an optional vector is contained within the payload segment of the frame transmitted. In the static segment, it indicates the presence of a network management vector at the beginning of the payload. In the dynamic segment it indicates the presence of a message id at the beginning of the payload, (FlexRay Protocol Specification: Vrf!Header!PPIndicator).

R

Reserved Bit (FlexRay Protocol Specification: Vrf!Header!Reserved)

PayloadLength

Defines the number of 16-bit words contained in *pData*
(FlexRay Protocol Specification: Vrf!Header!Length)

CycleCount

The cycle in which the frame was received. (FlexRay Protocol Specification: Vrf!Header!CycleCount)

HeaderCRC

The header CRC containing the cyclic redundancy check code is computed over the sync frame indicator, the start-up frame indicator, the frame id and the payload length (FlexRay Protocol Specification: Vrf!Header!HeaderCRC)

pData

The pointer to the payload data. The payload is given in 16-bit words.
(FlexRay Protocol Specification: Vrf!Payload)

Channel

The channel (A or B) on which the frame was received. (FlexRay Protocol Specification: Vrf!Channel)

ValidFrame

If a valid frame was received, this parameter is set to 1 (FlexRay Protocol Specification: Vss!ValidFrameA or Vss!ValidFrameB depends on Channel - Table 9-2: Slot status interpretation)

SyntaxError

If a syntax error was observed, this parameter is set to 1 (frame is syntactically incorrect). (FlexRay Protocol Specification: Vss!SyntaxErrorA or Vss!SyntaxErrorB depends on Channel)

ContentError

If a content error was observed, this parameter is set to 1 (frame is semantically incorrect). (FlexRay Protocol Specification: Vss!ContentErrorA or Vss!ContentErrorB depends on Channel)

SlotBoundaryViolation

If a slot boundary violation was observed, this parameter is set to 1 (FlexRay Protocol Specification: Vss!BviolationA or Vss!BviolationB depends on Channel)

AsyncMode

If the packet was generated by the asynchronous debug mode, this parameter is set to 1.

FrameCRC

If the packet was generated by the asynchronous debug mode, the FrameCRC contains the cyclic redundancy check code is computed over complete frame. In synchronous monitoring mode, this parameter is not set.

TimeStamp

The FlexCard time stamp (1 µs resolution). The timestamp marks the begin of the reception of the frame.

CC

The FlexCard CC which created this packet.

	Information
ℹ	The payload length is a multiple of 16-bit words. The payload data is also given in 16-bit words.

	Information
ℹ	Members *AsyncMode* and *FrameCRC* are initially supported by FlexCard VxWorks driver S2V1-F.

15.2.2.9 fcTxAcknowledgePacket

This structure provides information about a transmit acknowledge packet. Transmit acknowledge packets are used to inform the user when a frame is transmitted.

```
Typedef struct fcTxAcknowledgePacket
{
    fcDword BufferId;
    fcDword TimeStamp;
    fcDword CycleCount;

    fcDword ID : 11;
    fcDword STARTUP : 1;
    fcDword SYNC : 1;
    fcDword NF : 1;
    fcDword PP : 1;
    fcDword R : 1;
    fcDword PayloadLength : 7;
    fcDword ValidFrame : 1;
    fcDword SyntaxError : 1;
    fcDword ContentError : 1;
    fcDword HeaderCRC : 11;
    fcWord* pData;
    fcChannel Channel;
    fcCC CC;
} fcTxAcknowledgePacket;
```

Members

BufferId

The buffer id used to transmit the frame (equivalent to the buffer id returned by the function **fcbFRConfigureMessageBuffer**).

TimeStamp

The FlexCard time stamp (1 µs resolution). The timestamp marks the beginning of the transmission of the frame.

CycleCount

Indicates the cycle in which the frame was transmitted. (FlexRay Protocol Specification: Vtf!Header!CycleCount)

ID

The frame id defines the slot in which the frame was transmitted.

STARTUP

Indicates if the frame was a start-up frame (=1) or not (=0)

SYNC

Indicates if the frame was a sync frame (=1) or not (=0)

NF

Set to 0, the null frame indicator indicates that *pData* contains no valid data. Set to 1, it indicates that *pData* contains valid data.

PP

The payload preamble indicator indicates whether an optional vector is contained within the payload segment of the frame transmitted. In the static segment, it indicates the presence of a network management vector at the beginning of the payload. In the dynamic segment it indicates the presence of a message id at the beginning of the payload.

R

Reserved Bit

PayloadLength

Defines the number of 16-bit words contain in *pData*

ValidFrame

If a valid frame was received, this parameter is set to 1 (FlexRay Protocol Specification: Vss!ValidFrameA or Vss!ValidFrameB depends on Channel - Table 9-2: Slot status interpretation)

SyntaxError

If a syntax error was observed, this parameter is set to 1 (frame is syntactically incorrect). (FlexRay Protocol Specification: Vss!SyntaxErrorA or Vss!SyntaxErrorB depends on Channel)

ContentError

If a content error was observed, this parameter is set to 1 (frame is semantically incorrect). (FlexRay Protocol Specification: Vss!ContentErrorA or Vss!ContentErrorB depends on Channel)

HeaderCRC

The header CRC contains the cyclic redundancy check code is computed over the sync frame indicator, the start-up frame indicator, the frame id and the payload length.

PData

The pointer to the payload data. The payload is given in 16-bit words.

Channel

The channel (A or B) on which the frame was transmitted.
(FlexRay Protocol Specification: Vrf!Channel)

CC

The FlexCard CC which created this packet.

	Information
ℹ	Members *ValidFrame*, *SyntaxError*, *ContentError* and *HeaderCRC* are initially supported by FlexCard VxWorks driver S2V1-F.

15.2.3 Not Supported Functions

The VxWorks driver doesn't support the following functions:

```
fcGetErrorText
fcFreeMemory
fcbCanDbCcConfiguration (Obsolete)
fcbTrigger (Obsolete)
fcbGetEnumFlexCardsV2 (Obsolete)
```

15.2.4 Changed Functions

15.2.4.1 fcbMonitoringStart

This function is used to start the monitoring of a FlexRay bus. Once called, the function changes the Communication Controller state from configuration state to normal active state (if the cluster integration succeeds). The current Communication Controller state can be read using the function **fcbGetCcState (Obsolete)**. If the FlexCard is synchronized with the cluster the function **fcbGetCcState (Obsolete)** will return the value *fcStateNormalActive*. Please note, that if an event for the event counter (for the several packet type) is registered with **fcbSetEventHandle**, this function activates the corresponding hardware interrupts and the application is notified if this event occurred.

```
FcError fcbMonitoringStart(
       fcHandle hFlexCard,
       fcMonitoringModes mode,
       bool restartTimestamps,
       bool enableCycleStartEvents
       bool enableColdstart,
       bool enableWakeup
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

Mode

[IN] The monitoring mode. See **fcMonitoringModes** for details.

RestartTimestamps

[IN] Set this parameter to false to restart the measurement without resetting the FlexCard timestamp. Set it to true to start the measurement from the beginning. The timestamps have microsecond resolution.

EnableCycleStartEvents

[IN] Set this parameter to true to enable the cycle start events in order that at the beginning of every cycle the event `fcNotificationTypeCycleStarted` is signalled.

EnableColdstart

[IN] Set this parameter to true to allow the FlexCard to initialize the cluster communication, otherwise the coldstart inhibit mode is active. This feature can not be used in the monitoring modes `fcMonitoringDebug` and `fcMonitoringDebugAsynchron`.

EnableWakeup

[IN] Set this parameter to true to transmit a wake-up pattern to the configured wake-up channel (FlexRay Protocol Specification: pWakeupChannel). A cluster wake-up must precede the communication start-up to ensure that all nodes in a cluster are awake. The minimum requirement for a cluster wake-up is that all bus drivers are supplied with power. This feature can not be used in the monitoring modes `fcMonitoringDebug` and `fcMonitoringDebugAsynchron`.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

After the monitoring has started, the user should check if the integration in the cluster was successful: **`fcbGetCcState (Obsolete)`** should return the state `fcStateNormalActive`.

	Information
ℹ	After the monitoring has successfully started, the receive process has to be started as soon as possible to avoid an overflow (error packet `fcErrFlexcardOverflow` is received). Once an overflow occurred, no more packets can be received. The monitoring has to be stopped and started again.

See Also

`fcbMonitoringStop`, **`fcbGetCcState (Obsolete)`**, **`fcMonitoringModes`**, **`fcbSetEventHandle`**

15.2.4.2 fcbMonitoringStop

This function stops the FlexRay bus measurement. The Communication Controller is set back in its configuration state.

Please note, that if an event for the event counter (for the several packet types) is registered with **`fcbSetEventHandle, this function`** deactivates the corresponding hardware interrupts and the application is not notified if this event occurred.

```
FcError fcbMonitoringStop(
       fcHandle hFlexCard
)
```

Parameters

hFlexCard

[IN] Handle to FlexCard

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcbMonitoringStart`

15.2.4.3 fcbSetEventHandle

This function registers an event handle for a specific notification type. The event handling is based on binary semaphores.

```
FcError fcbSetEventHandle(
       fcHandle hFlexCard,
       fcHandle hEvent,
       fcNotificationType type
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

hEvent

[IN] Event handle to be registered. This value depends on the given *type*. Set this parameter to NULL to deregister the event handle for the given type.

Type

[IN] The notification type for which the event has to be registered.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcNotificationType

Example

See Example in **fcbSetNotificationTypeCount**

15.2.4.4 fcbReceive

This function reads all available packets from the FlexCard memory into a memory block allocated by the fcBase API during the initialization phase in **fcbOpen**. The frames are stored into a linked list. The memory allocated by this function is released by the **fcbClose** function. Please note, that every function call from **fcbReceive** overwrites the old frames in the memory block. The size of the memory block can be configured with **fcbSetReceiveMemorySize**.

```
FcError fcbReceive(
       fcHandle hFlexCard,
       fcPacket** pPacket
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

pPacket

[OUT] Address of the `fcPacket` object pointer. The memory for this structure and its content is allocated by the fcBase API. Packets are available if the return code is 0 and *pPacket* is not a null pointer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Example

```
fcPacket* pPackets = NULL;
fcError e = fcbReceive(m_hFlexCard, &pPackets);
if (0 == e)
{
      fcPacket* pCurrentPacket = pPacket;
      while (NULL != pCurrentPacket)
      {
            switch (pCurrentPacket->Type)
            {
            case fcPacketTypeInfo:
            {
                  fcInfoPacket* pFrame = pCurrentPacket->InfoPacket;
                  printf(“[fcPacketTypeInfo] CC: %d TimeStamp: %f Cycle: %d”,
                        pFrame->CC + 1,
                        (float) pFrame->TimeStamp * 0.000001,
                        pFrame->CurrentCycle);
                  printf(“ Rate Correction: %d”, pFrame->RateCorrection);
                  printf(“ Offset Correction: %d”, pFrame->OffsetCorrection);
                  printf(“ Clock Correction Failed Counter: %d”,
                        pFrame->ClockCorrectionFailedCounter);
                  printf(“ Passive to Active Count: %d”,
                        pFrame->PassiveToActiveCount);
                  printf(“\n”);
                  break;
            }

            case fcPacketTypeFlexRayFrame:
            {
                  fcFlexRayFrame* pFrame = pCurrentPacket->FlexRayFrame;
                  printf(“[fcPacketTypeFlexRayFrame] CC: %d TimeStamp: %f “
                        “Cycle: %d ID: %d Channel: %d PayloadLength: %d”,
                        pFrame->CC + 1,
                        (float) pFrame->TimeStamp * 0.000001,
                        pFrame->CycleCount,
                        pFrame->ID,
                        pFrame->Channel,
                        pFrame->PayloadLength);

                  for (int I = 0; i < pFrame->PayloadLength; i++)
                  {
                        printf(“ %04X”, pFrame->pData[i]);
                  }

                  if (pFrame->PP) printf(“ PP”);
                  if (pFrame->NF) printf(“ NF”);
                  if (pFrame->SYNC) printf(“ SYNC”);
                  if (pFrame->STARTUP) printf(“ STARTUP”);
                  if (pFrame->SyntaxError) printf(“ SyntaxError”);
                  if (pFrame->ContentError) printf(“ ContentError”);
                  if (pFrame->ValidFrame) printf(“ ValidFrame”);
                  if (pFrame->SlotBoundaryViolation)
                        printf(“ SlotBoundaryViolation”);
                  printf(“\n”);
                  break;
            }

            case fcPacketTypeError:
                  printf(“[fcPacketTypeError]\n”);
                  break;

            case fcPacketTypeStatus:
                  printf(“[fcPacketTypeStatus]\n”);
                  break;

            case fcPacketTypeTriggerEx:
                  printf(“[fcPacketTypeTriggerEx]\n”);
                  break;
            case fcPacketTypeTxAcknowledge:
                  printf(“[fcPacketTypeTxAcknowledge]\n”);
                  break;

            case fcPacketTypeNMVector:
                  printf(“[fcPacketTypeNMVector]\n”);
                  break;
            }

            pCurrentPacket = pCurrentPacket->pNextPacket;
      }
}
```

15.3 Configuration

15.3.1 fcbSetPacketGeneration

This function allows to dis- or enable the generation of a packet type. It is designed to reduce the number of packets, which will be generated by the FlexCard.

```
fcError fcbSetPacketGeneration(
      fcHandle hFlexCard,
      fcPacketType type,
      bool bEnable
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

type

[IN] The packet type.

bEnable

[IN] Set to true to enable the generation and to false to disable it.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbReceive, **fcPacketType**

15.3.2 fcbSetReceiveMemorySize

This function allows configuring the size of memory, where **fcbReceive** will store all received frames. This function must be called before you open a connection to the FlexCard. During the 217ehavior217ation phase (in **fcbOpen**) the amount of memory is dynamically allocated. Closing the connection (by **fcbClose**) releases the memory automatically.

```
fcError fcbSetReceiveMemorySize(
      fcDword size;
)
```

Parameters

size

[IN] The size of memory. The default value is 128 KB and it is recommended to set *size* in a range from 20 KB to 70 MB. Other values than the recommended values are ignored, and *size* will be set to default.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcbReceive`, **`fcbOpen`**, **`fcbClose`**

15.4 Event

15.4.1 fcbSetNotificationTypeCount

This function allows configuring the event counter for the several packet types. *Count* represents the number of packets (of a dedicated packet type) which need to be received to initiate an event of the chosen notification packet type.

```
fcError fcbSetNotificationTypeCount(
      fcHandle hFlexCard,
      fcNotificationType type,
      fcByte count
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

`type`

[IN] The notification type for which the configuration has to be used. The notification types *`fcNotificationTypeCycleStarted`*, *`fcNotificationTypeWakeup`*, *`fcNotificationTypeTimer`* and *`fcNotificationTypeCcTimer`* are not supported.

`Count`

[IN] The value represents the number of packets (of a dedicated packet type) which need to be received to initiate an event of the chosen notification packet type. Valid values range from 1 to 255.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcNotificationType`, **`fcbSetEventHandle`**, **`fcbMonitoringStart`**, **`fcbMonitoringStop`**

Example

```
fcPacket* pPackets = NULL;
SEM_ID semInfoCount = NULL;
semInfoCount = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
assert (NULL != semInfoCount);

fcError e = fcbSetEventHandle(m_hFlexCard, (void *) semInfoCount, \
      fcNotificationTypeInfoCount);
if (0 == e)
{
      // Configure the Info packet event counter
      fcbSetNotificationTypeCount(m_hFlexCard,fcNotificationTypeInfoCount, 2);

      // Start monitoring and wait for the event forever
      fcbMonitoringStart(m_hFlexCard, fcMonitoringNormal, 1, 0, 0, 0);
      semTake(semInfoCount, WAIT_FOREVER);

      // Min. 2 Info packets can be received now
      e = fcbReceive(m_hFlexCard, &pPackets);
      if (0 == e)
      { /* Process packets */ }
}
```

16 Obsolete

16.1 fcInfo (Obsolete)

	Information
ℹ	This structure is obsolete. Please use **fcInfoHwSw** instead.

This structure provides information about the components and the identifier of a FlexCard. If more than one FlexCard is detected on the system, the **fcbGetEnumFlexCards (Obsolete)** function returns a linked list of this structure. If a connection to a FlexCard is already opened, this FlexCard does not appear in this list.

```
Typedef struct fcInfo
{
        fcDword FlexCardId;
        fcVersion Version;
        struct fcInfo* pNext;
} fcInfo;
```

Members

FlexCardId

Unique number used to identify a FlexCard. This id is required to open a connection to the FlexCard.

Version

Version information about hardware and software components of the FlexCard.

pNext

Pointer to the next available FlexCard. If no other FlexCard exists, pNext is a null pointer.

See Also

fcVersion (Obsolete), **fcbGetEnumFlexCards (Obsolete)**

16.2 fcInfoV2 (Obsolete)

	Information
ℹ	This structure is obsolete. Please use **fcInfoHwSw** instead.

This structure provides information about the components, the identifier and the current device state of a FlexCard. If more than one FlexCard is detected on the system, the **fcbGetEnumFlexCardsV2 (Obsolete)** function returns a linked list of this structure.

```
Typedef struct fcInfoV2
{
        fcDword FlexCardId;
        fcVersion Version;
        fcDword Busy;
        fcDword Reserved;
        struct fcInfoV2* pNext;
} fcInfoV2;
```

Members

FlexCardId

Unique number used to identify a FlexCard. This id is required to open a connection to the FlexCard.

Version

Version information about hardware and software components of the FlexCard.

Busy

The current device state. A value <> 0 indicates a connection to this FlexCard is already opened.

Reserved

Reserved for future use.

pNext

Pointer to the next available FlexCard. If no other FlexCard exists, `pNext` is a null pointer.

See Also

fcVersion (Obsolete), fcbGetEnumFlexCardsV2 (Obsolete)

16.3 fcVersion (Obsolete)

	Information
ℹ	This structure is obsolete. Please use **fcInfoHw** and **fcInfoSw** instead.

This structure provides version information about the FlexCard hardware and software components.

```
Typedef struct fcVersion
{
       fcVersionNumber BaseDll;
       fcVersionNumber DeviceDriver;
       fcVersionNumber Firmware;
       fcVersionNumber Hardware;
       fcCCType CCType;
       fcVersionNumber CC;
       fcVersionNumber BusGuardian;
       fcVersionNumber Protocol;
       fcDword Serial;
       fcFlexCardDeviceId DeviceId;
       fcVersionCC* pVersionCC;
       fcDword Reserved[2];
}fcVersion;
```

Members

BaseDll

DLL Base Version

DeviceDriver

Device driver version

Firmware

Firmware (gateway software) version

Hardware

FlexCard hardware version

CCType

Communication controller type

CC

Communication controller module version

BusGuardian

Bus Guardian version

Protocol

FlexRay Protocol version

Serial

FlexCard serial number. A zero value indicates a non-valid FlexCard serial number.

DeviceId

Device identifier to detect the FlexCard type (FlexCard Cyclone II, FlexCard Cyclone II SE or FlexCard PMC).

pVersionCC

Pointer to version information about the available Communication Controllers.

Reserved

Reserved for internal purpose

See Also

fcFlexCardDeviceId, **fcInfo (Obsolete)**, **fcInfoV2 (Obsolete)**, **fcbGetEnumFlexCards (Obsolete)**, **fcbGetEnumFlexCardsV2 (Obsolete)**

16.4 fcbGetEnumFlexCards (Obsolete)

Information

This function is obsolete. Please use **fcbGetEnumFlexCardsV3** instead.

This function returns a linked list of the unused FlexCards found on the system. To free the memory, which was allocated by the function, please use the function **fcFreeMemory** with type *fcMemoryTypeInfo.*

```
fcError fcbGetEnumFlexCards(
      fcInfo** pInfo
)
```

Parameters

pInfo

[OUT] linked list of **fcInfo (Obsolete)** objects

Return values

If the function succeeds, the return value is 0. If the function fails, the content of pInfo is not valid. The error code NULL_PARAMETER is returned if pInfo parameter is a null pointer. If the memory allocation fails, the error code MEMORY_ALLOCATION_FAILED is returned.

Remarks

If a connection to a FlexCard is already opened, this FlexCard does not appear in this list. If the function succeeds, there will always be one valid **fcInfo (Obsolete)** structure regardless if there is a FlexCard in the system or not. This functionality is given to provide version information about the DLL / library. The version information concerning the hardware is only valid if the identifier (***pInfo->FlexCardId***) is not 0.

Information

This function allocates memory for use. To prevent memory leaks you have to free it up by calling the function **fcFreeMemory** with the type *fcMemoryTypeInfo*.

From FlexCard API version S2V0-F on it is possible to use four FlexCards in one PC. With FlexCard API versions up to S2V0-F it isn't possible to use two FlexCards in one PC at the same time. That means that only the first inserted FlexCard can be used. The second one doesn't appear in the list of available FlexCards.

From FlexCard API version S6V1-F on it is possible to use eight FlexCards in one PC.

See Also

fcInfo (Obsolete),

16.5 fcbGetEnumFlexCardsV2 (Obsolete)

Information

This function is obsolete. Please use **fcbGetEnumFlexCardsV3** instead.

This function returns a linked list of the installed FlexCards found on the system. To free the memory, which was allocated by this function, please use the function **fcFreeMemory** with type *fcMemoryTypeInfoV2.*

```
fcError fcbGetEnumFlexCardsV2(
      fcInfoV2** pInfoV2
)
```

Parameters

pInfoV2

[OUT] linked list of **fcInfoV2 (Obsolete)** objects

Return values

If the function succeeds, the return value is 0. If the function fails, the content of pInfoV2 is not valid. The error code NULL_PARAMETER is returned if pInfoV2 parameter is a null pointer. If the memory allocation fails, the error code MEMORY_ALLOCATION_FAILED is returned.

Remarks

If the function succeeds, there will always be one valid **fcInfoV2 (Obsolete)** structure regardless if there is a FlexCard in the system or not. This functionality is given to provide version information about the DLL / library. The version information concerning the hardware is only valid if the identifier (***pInfoV2->FlexCardId***) is not 0.

	Information
ⓘ	This function allocates memory for use. To prevent memory leaks you have to free it up by calling the function **fcFreeMemory** with the type *fcMemoryTypeInfoV2*.

See Also

fcInfoV2 (Obsolete)

16.6 fcbMonitoringStart (Obsolete)

	Information
ⓘ	This function is obsolete. Please use **fcbFRMonitoringStart** instead.

This function is used to start the monitoring of a FlexRay bus. Once called, the function changes the Communication Controller state from configuration state to normal active state (if the cluster integration succeeds). The current Communication Controller state can be read using the function **fcbGetCcState (Obsolete)**. If the FlexCard is synchronized with the cluster the function **fcbGetCcState (Obsolete)** will return the value *fcStateNormalActive*.

```
fcError fcbMonitoringStart(
      fcHandle hFlexCard,
      fcMonitoringModes mode,
      fcBool restartTimestamps,
      fcBool enableCycleStartEvents
      fcBool enableColdstart,
      fcBool enableWakeup
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

Mode

[IN] The monitoring mode. Not every monitoring mode is supported by all Communication Controllers. See **fcMonitoringModes** for details.

restartTimestamps

[IN] Set this parameter to 0 to restart the measurement without resetting the FlexCard timestamp. Set it to <> 0 to start the measurement from the beginning. The timestamps have microsecond resolution.

enableCycleStartEvents

[IN] Set this parameter to <> 0 to enable the cycle start events in order that at the beginning of every cycle the event *fcNotificationTypeCycleStarted* is signalled.

enableColdstart

[IN] Set this parameter to <> 0 to allow the FlexCard to initialize the cluster communication, otherwise the coldstart inhibit mode is active. This feature can not be used in the monitoring modes fcMonitoringDebug and fcMonitoringDebugAsynchron.

enableWakeup

[IN] Set this parameter to <> 0 to transmit a wake-up pattern on the configured wake-up channel (FlexRay Protocol Specification: pWakeupChannel). A cluster wake-up must precede the communication start-up to ensure that all nodes in a cluster are awake. The minimum requirement for a cluster wake-up is that all bus drivers are supplied with power. This feature can not be used in the monitoring modes *fcMonitoringDebug* and *fcMonitoringDebugAsynchron*.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

After the monitoring has started, the user should check if the integration in the cluster was successful: **fcbGetCcState (Obsolete)** should return the state fcStateNormalActive.

	Information
ⓘ	After the monitoring has successfully started, the receive process has to be started as soon as possible to avoid an overflow (error packet fcErrFlexcardOverflow is received). Once an overflow occurred, no more packets can be received. The monitoring has to be stopped and started again.

See Also

fcbMonitoringStop (Obsolete), **fcbGetCcState (Obsolete)**, **fcMonitoringModes**, **fcbSetEventHandle (Obsolete)**

16.7 fcbMonitoringStop (Obsolete)

	Information
ⓘ	This function is obsolete. Please use **fcbFRMonitoringStop** instead.

This function stops the FlexRay bus measurement. The Communication Controller is set back in its configuration state.

```
fcError fcbMonitoringStop(
       fcHandle hFlexCard
)
```

Parameters

hFlexCard

[IN] Handle to FlexCard

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbMonitoringStart (Obsolete)

16.8 fcbGetCcState (Obsolete)

	Information
ℹ	This function is obsolete. Please use **fcbFRGetCcState** instead.

This function returns the current Communication Controller POC state. For a description of possible states, refer to the enumeration **fcState**. This function should be used to check if the integration into a FlexRay cluster has succeeded.

```
fcError fcbGetCcState(
      fcHandle hFlexCard,
      fcState* pState
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

pState

[OUT] Current Communication Controller state

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See

fcbMonitoringStart (Obsolete), **fcbMonitoringStop (Obsolete)**

16.9 fcbSetTransceiverState (Obsolete)

	Information
ℹ	This function is obsolete. Please use **fcbFRSetTransceiverState** instead.

This function sets the transceiver mode individually for each channel.

```
fcError fcbSetTransceiverState (
      fcHandle hFlexCard,
      fcTransceiverState stateChannelA,
      fcTransceiverState stateChannelB
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

stateChannelA

[IN] The new transceiver state for channel A

stateChannelB

[IN] The new transceiver state for channel B

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

If one of the transceivers is in the sleep mode and the transceiver detects a wake-up event, the notification event *fcNotificationTypeWakeup* is fired once only.

See

fcTransceiverState, **fcbMonitoringStart (Obsolete)**, **fcbGetTransceiverState (Obsolete)**

16.10 fcbGetTransceiverState (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRGetTransceiverState** instead.

This function gets the transceiver state individually for each channel.

```
fcError fcbGetTransceiverState (
       fcHandle hFlexCard,
       fcTransceiverState* stateChannelA,
       fcTransceiverState* stateChannelB
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

stateChannelA

[OUT] The current transceiver state for channel A

stateChannelB

[OUT] The current transceiver state for channel B

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

If one of the transceiver is in the sleep mode and the transceiver detects a wake-up event, the notification event *fcNotificationTypeWakeup* is fired once only.

See

fcTransceiverState, fcbMonitoringStart (Obsolete), **fcbSetTransceiverState (Obsolete)**

16.11 fcbSetEventHandle (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbSetEventHandleV2** or **fcbSetEventHandleSemaphore** instead.

This function registers an event handle for a specific notification type.

```
fcError fcbSetEventHandle(
      fcHandle hFlexCard,
      fcHandle hEvent,
      fcNotificationType type
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

hEvent

[IN] Event handle to be registered to signal when a new cycle starts, a timer interval has elapsed or the FlexCard receive buffer reaches a specific filling level depending on the given *type*.

Type

[IN] The notification type for which the event must be registered.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcNotificationType

16.12 fcbTransmit (Obsolete)

	Information
ℹ	This function is obsolete. Please use **fcbFRTransmit** instead.

This function writes a data frame into a Communication Controller transmit buffer of the FlexCard. The frame should normally be transmitted in the next cycle. If the transmit acknowledgment is activated, an acknowledge packet is generated as soon as the frame has been transmitted. This function should only be called when the FlexCard is in normal active state or when all message buffer configurations have been done.

```
fcError fcbTransmit(
      fcHandle hFlexCard,
      fcDword bufferId,
      fcWord payload[],
      fcByte payloadLength
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId

[IN] The id of the message buffer used for the transmission

payload

The payload data to be transmitted

payloadLength

The size of the payload data (number of 2-byte words)

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

The transmission may fail, if the buffer is currently in use (**fcGetErrorCode** returns MSG_BUFFER_BUSY). In that case retry later.

Remarks

The payload data has to be organized as follows: if Data0 is the first byte to transmit and Data1 the second byte to transmit, then the high byte (Bit 8 – 15) of payload[0] contains Data1, the low byte (Bit 0-7) of payload[0] contains Data0, etc.

Parameter `payload`	`payload[0]` (Word 0)		`payload[1]` (Word 1)		...
	High byte	Low byte	High byte	Low byte	...
FlexRay payload segment	Data 1	Data 0	Data 3	Data 2	...

16.13 fcbTransmitSymbol (Obsolete)

	Information
ℹ	This function is obsolete. Please use **`fcbFRTransmitSymbol`** instead.

This function transmits a symbol in the symbol window segment. It can only be called if the Communication Controller is in the POC state NORMAL_ACTIVE. For a list of available symbols to be transmitted, see the enumeration fcSymbolType.

```
fcError fcbTransmitSymbol(
      fcHandle hFlexCard,
      fcSymbolType symbol
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

symbol

[IN] Type of symbol to transmit

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

16.14 fcbSetCcRegister (Obsolete)

	Information
ℹ	This function is obsolete. Please use **`fcbFRSetCcRegister`** instead.

This function writes a value in a given register of the Communication Controller. Not every register can be written (e.g. the registers belonging to the message buffer configuration or some interrupt settings).

```
fcError fcbSetCcRegister(
      fcHandle hFlexCard,
      fcDword address,
      fcDword value
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

address

[IN] Address of the CC register to be written

value

[IN] The value to be written

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information. If the register can not be written the error code `REGISTER_NOT_WRITEABLE` is returned.

Remarks

For a register description, refer to the specification of the corresponding Communication Controller. Modifying one of the following registers will reset message buffers with their default settings (FIFO receive buffers). The user's message buffers configuration will not be valid anymore.
Bosch E-Ray: MHDC (0x0098) and GTUC7 (0x00B8)

	Information
ℹ	Not all registers of a Communication Controller can be set. The base API will modify some parameters so that the operation of the FlexCard is guaranteed (e.g. interrupt settings). Access is denied to all registers which are used for message buffer configuration.

See Also

`fcbGetCcRegister (Obsolete)`

16.15 fcbGetCcRegister (Obsolete)

	Information
ℹ	This function is obsolete. Please use **`fcbFRGetCcRegister`** instead.

This function reads and returns the content of a given register of the Communication Controller.

```
fcError fcbGetCcRegister(
       fcHandle hFlexCard,
       fcDword  address,
       fcDword* pValue
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

`address`

[IN] Address of the CC register to be read.

`pValue`

[OUT] The content of the desired CC register.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information. If the register cannot be read the error code `REGISTER_NOT_READABLE` is returned.

Remarks

Not every register can be read. For a register description, refer to the specification of the corresponding Communication Controller.

See Also

`fcbSetCcRegister (Obsolete)`

16.16 fcbChiCcConfiguration (Obsolete)

Information

This function is obsolete. Please use **`fcbFRSetCcConfigurationChi`** instead.

This function configures the Communication Controller of the FlexCard with a FlexConfig compatible configuration string (CHI File). The configuration string contains the global FlexRay parameter and/or the message buffer configuration. The payload data for transmit message buffers are not set by this function. Before the configuration of the Communication Controller starts, all message buffers are reset to their default settings (FIFO buffer).

```
fcError fcbChiCcConfiguration(
       fcHandle hFlexCard,
       const char* szChi
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard.

`szChi`

[IN] Pointer to null-terminated CHI content string (refer to the CHI string example section).

Please note: Do not use the CHI file name here, but the content of the CHI file as parameter value.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Information

Internally, the function uses the function **`fcbSetCcRegister (Obsolete)`**; therefore the same restrictions as for writing registers exist.

See Also

`fcbSetCcRegister (Obsolete)`

16.17 fcbCanDbCcConfiguration (Obsolete)

Information

This function is obsolete. Please use **`fcbFRSetCcConfigurationCANdb`** instead.

This function configures the FlexRay Communication Controller of the FlexCard with a proprietary string. The configuration string contains the global FlexRay parameter and/or the message buffer configuration. Before the configuration of the Communication Controller starts, all message buffers are reset to their default settings (FIFO buffer). Configuring the CAN CC with a CANdb file is not supported by the FlexCard driver.

```
fcError fcbCanDbCcConfiguration(
       fcHandle hFlexCard,
       const char* szCanDb
)
```

Parameters

`hFlexCard`

[IN] Handle to a FlexCard

szCanDb

[IN] Pointer to null-terminated proprietary string

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

This function is only available in the Windows FlexCard driver. The FlexCard Linux and Xenomai drivers don’t support this function.

	Information
ℹ	Internally, the function uses the **`fcbSetCcRegister (Obsolete)`** function; therefore the same restrictions as for writing a register exist.

16.18 fcbConfigureMessageBuffer (Obsolete)

	Information
ℹ	This function is obsolete. Please use **`fcbFRConfigureMessageBuffer`** instead.

This function configures transmit, receive and FIFO message buffers of the Communication Controller. Configuring message buffers is only allowed if the Communication Controller is in its configuration state, *fcStateConfig*.

```
fcError fcbConfigureMessageBuffer(
      fcHandle hFlexCard,
      fcDword* bufferId,
      fcMsgBufCfg cfg
);
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId

[OUT] Message buffer identifier. If the configuration was successful the message buffer identifier is greater than 0. This identifier will be required to transmit the content of the buffer (in the case of a transmit buffer).

Cfg

[IN] Message buffer configuration parameters

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

Before configuring the message buffers, it is necessary to set up the global communication parameters (cluster parameters). Internally the FlexCard uses the FIFO buffers as receive buffers, therefore we recommend using FIFO message buffers as much as possible.

See Also

`fcMsgBufCfg`, **`fcMsgBufCfgTx`**, **`fcMsgBufCfgRx`**, **`fcMsgBufCfgFifo`**

16.19 fcbReconfigureMessageBuffer (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRReconfigureMessageBuffer** instead.

This function reconfigures transmit, receive and FIFO message buffers of the Communication Controller. A reconfiguration is only allowed for message buffers which are already configured. This function is available in all states of the CC. Not all configuration settings can be modified in monitoring state. Refer to the documentation of the message buffer structures for further details.

```
fcError fcbReconfigureMessageBuffer(
      fcHandle hFlexCard,
      fcDword bufferId,
      fcMsgBufCfg cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId

[IN] The identifier of the message buffer which should be reconfigured.

Cfg

[IN] Message buffer configuration parameters.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcMsgBufCfg, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**, **fcMsgBufCfgFifo**, **fcbConfigureMessageBuffer (Obsolete)**, **fcbGetCcMessageBuffer (Obsolete)**

16.20 fcbGetCcMessageBuffer (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRGetMessageBuffer** instead.

This function reads a specific message buffer configuration.

```
fcError fcbGetCcMessageBuffer(
      fcHandle hFlexCard,
      fcDword bufferId,
      fcMsgBufCfg* cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

bufferId

[IN] The identifier of the message buffer to be read

cfg

[OUT] The configuration parameters of the specified message buffer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

The buffer with id 1 is always a FIFO message buffer.

See Also

fcMsgBufCfg, **fcMsgBufCfgTx**, **fcMsgBufCfgRx**, **fcMsgBufCfgFifo**, **fcbConfigureMessageBuffer (Obsolete)**

16.21 fcbResetCcMessageBuffer (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRResetMessageBuffers** instead.

This function resets the Communication Controller message buffers. After calling this function, all message buffers are configured as receive FIFO – with maximal payload (depends on the Communication Controller).

```
fcError fcbResetCcMessageBuffer(
      fcHandle hFlexCard
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

16.22 fcbFilter (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRSetSoftwareAcceptanceFilter** or **fcbFRSetHardwareAcceptanceFilter** instead.

This function configures the frame ids accepted by the device driver. Only the ids which are in the filter list are forwarded to the user application, all other frames are rejected. To accept all frames set the parameters *pData* to NULL and nSize to zero or configure a single frame id of zero.

```
fcError fcbFilter(
      fcHandle hFlexCard,
      fcChannel channel,
      fcDword* pData,
      fcDword size
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

channel

[IN] FlexCard channel(s) concerned by the filter *pData*

[IN] Pointer to a fcDword array containing the ids accepted by the device driver. Each element (fcDword) contains one frame identifier.

fcDword	fcDword
ID x	ID y

size

[IN] Number of ids in the array

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

16.23 fcbSetCcTimerConfig (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRSetCcTimerConfig** instead.

This function configures the Communication Controller timer interrupt. To get a notification when the Communication Controller timer interval elapsed, an event of type *fcNotificationTypeCcTimer* has to be registered by the function **fcbSetEventHandle (Obsolete)**. Additionally, the Communication Controller timer can be enabled / disabled by this function.

```
fcError fcbSetCcTimerConfig(
       fcHandle hFlexCard,
       fcCcTimerCfg cfg,
       fcBool bEnable
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

Cfg

[IN] The Communication Controller timer configuration.

bEnable

[IN] Set to <> 0 to enable the CC timer, and to 0 to disable it.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbSetEventHandle (Obsolete), **fcCcTimerCfg**, **fcbGetCcTimerConfig (Obsolete)**

16.24 fcbGetCcTimerConfig (Obsolete)

ℹ	Information
	This function is obsolete. Please use **fcbFRGetCcTimerConfig** instead.

This function reads the Communication Controller timer configuration.

```
fcError fcbGetCcTimerConfig(
      fcHandle hFlexCard,
      fcCcTimerCfg* pCfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

pCfg

[OUT] The configuration parameters of the CC timer.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCcTimerCfg, **fcbSetCcTimerConfig (Obsolete)**

16.25 fcbCalculateMacrotickOffset (Obsolete)

	Information
ℹ	This function is obsolete. Please use **fcbFRCalculateMacrotickOffset** instead.

This function calculates the macrotick offset for a specific cycle position in a FlexRay cycle.

```
fcError fcbCalculateMacrotickOffset(
      fcHandle hFlexCard,
      fcCyclePos CyclePosition,
      fcDword SlotOrMiniSlotId,
      fcDword* pValue
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard.

CyclePosition

[IN] The cycle position of type **fcCyclePos**.

SlotOrMiniSlotId

[IN] This parameter is used for a cycle position of *fcCyclePosStaticSlot* and *fcCyclePosDynamicMiniSlot* to calculate the macrotick offset for a static slot or a dynamic mini slot id.

pValue

[OUT] The macrotick offset value.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcCyclePos, fcCcTimerCfg, **fcbSetCcTimerConfig (Obsolete)**

16.25.1 Trigger Configuration (Obsolete)

	Information
	This configuration is obsolete. Please see **`fcTriggerConfigurationEx`** instead.

If the FlexCard is equipped with a trigger interface, the FlexCard has the ability to receive trigger events and forward them to the user application. This feature allows e.g. a synchronization of different bus 236ehavior. To configure and activate this feature, use the following structures and functions. The trigger event data is received as **`fcTriggerInfoPacket (Obsolete)`** with the **`fcbReceive`** function.

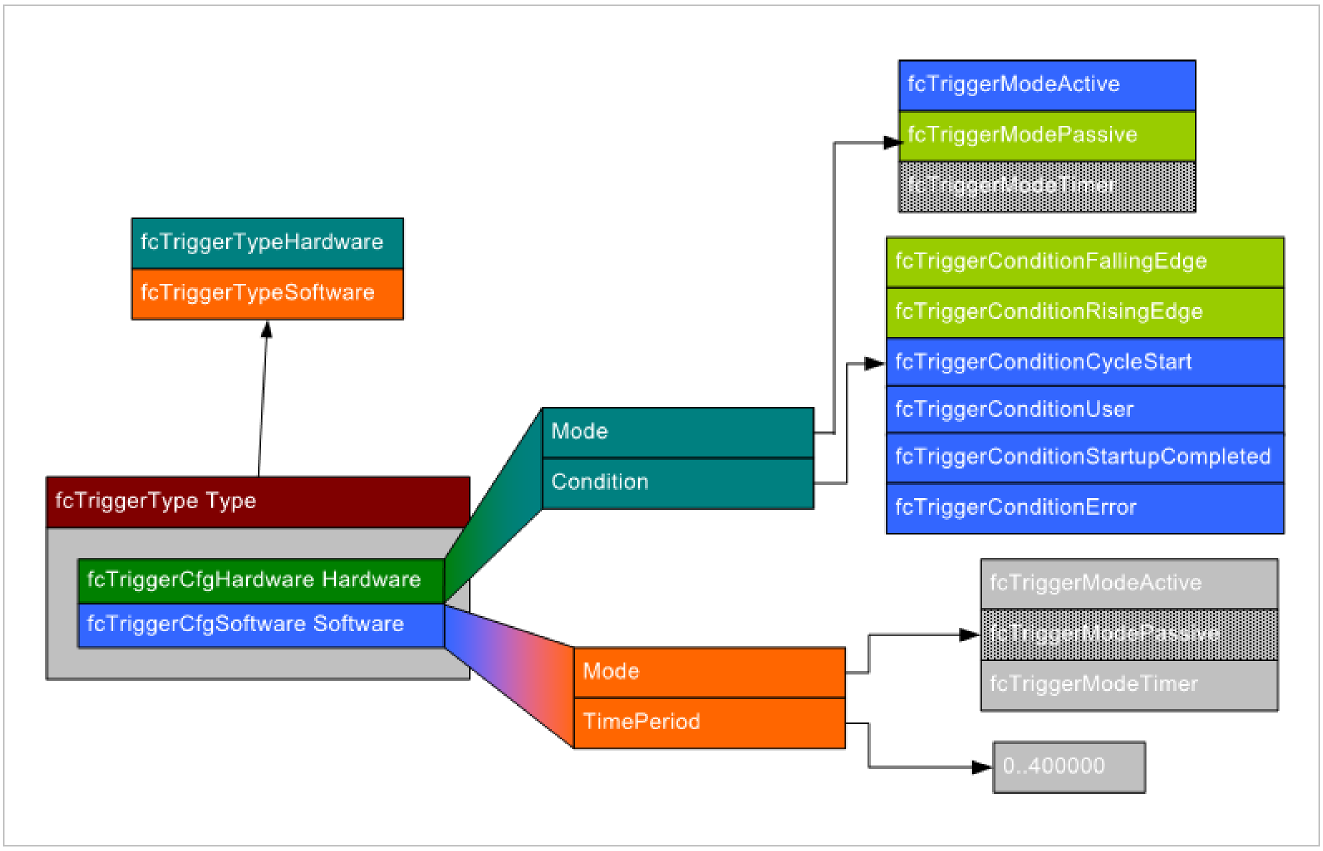


Figure 16: Overview obsolete structure fcbTriggerCfg

16.26 Typedefinitions (Obsolete)

16.26.1 fcTriggerCfgHardware (Obsolete)

This structure configures the hardware trigger. In the passive mode, the FlexCard waits for trigger events on its input line and generates a **`fcTriggerInfoPacket (Obsolete)`** object each time a trigger event is received. In this mode, the parameter *`Condition`* specifies on which condition the input signal will be recognized as a trigger event. In the active mode, the FlexCard generates a pulse on its output line when a trigger event is236onsultad. In this mode, the parameter *`Condition`* specifies on which condition a pulse will be generated by the FlexCard. For information about the pin assignment of the input and output line, refer to the user manual of the FlexCard.

```
Typedef struct fcTriggerCfgHardware
{
      fcTriggerMode Mode;
      fcTriggerCondition Condition;
}fcTriggerCfgHardware;
```

Members

Mode

Set the trigger mode (active or passive mode). The hardware trigger does not support the timer mode.

fcTriggerCondition

Depending on the mode, the following conditions can be used:

- Passive mode:
    - Falling edge (Trigger packet is generated on falling edge of the input signal)
    - Rising edge (Trigger packet is generated on rising edge of the input signal)
- Active mode:
    - Cycle start (A pulse is generated on the output line when a new cycle starts)
    - User (A pulse is generated on the output line when the user is calling the function fcbTrigger)
    - Error (A pulse is generated on the output line when an error occurred)
    - Start-up completed (A pulse is generated on the output line when the start-up was completed)

See Also

fcTriggerCfg (Obsolete), fcTriggerCondition (Obsolete), fcTriggerMode (Obsolete)

16.26.2 fcTriggerCfgSoftware (Obsolete)

This structure configures the software trigger. In active mode an **fcTriggerInfoPacket (Obsolete)** object is generated each time the function **fcbTrigger (Obsolete)** is called. In the timer mode an **fcTriggerInfoPacket (Obsolete)** object is generated every *TimePeriod* millisecond. A zero *TimePeriod* means that no **fcTriggerInfoPacket (Obsolete)** will be generated.

```
Typedef struct fcTriggerCfgSoftware
{
      fcTriggerMode Mode;
      fcDword  TimePeriod;
}fcTriggerCfgSoftware;
```

Members

Mode

Set the trigger mode (active or timer mode). The software trigger does not support the passive mode.

TimePeriod

This parameter is only used in timer mode. Every *TimePeriod* milliseconds (range: 0 – 400000) a trigger packet will be generated.

See Also

fcTriggerCfg (Obsolete), fcTriggerMode (Obsolete)

16.26.3 fcTriggerCfg (Obsolete)

This structure is used for the configuration of a trigger. Only one trigger at a time (hardware or software) can be used and the conditions cannot be combined.

```
Typedef struct fcTriggerCfg
{
    fcTriggerType Type;
    union
    {
        fcTriggerCfgHardware Hardware;
        fcTriggerCfgSoftware Software;
    };
}fcTriggerCfg;
```

Members

Type

Type of trigger (hardware or software)

Hardware

Configuration of hardware trigger

Software

Configuration of software trigger

See Also

fcTriggerType (Obsolete), fcTriggerCfgHardware (Obsolete), fcTriggerCfgSoftware (Obsolete), fcbTrigger (Obsolete)

16.26.4 fcTriggerInfoPacket (Obsolete)

This structure provides information about a trigger packet.

```
Typedef struct fcTriggerInfoPacket
{
    fcTriggerType Type;
    fcTriggerCondition Condition;
    fcDword TimeStamp;
    fcDword SequenceCount;
    fcQuad PerformanceCounter;

}fcTriggerInfoPacket;
```

Members

Type

Type of trigger info packet

Condition

The fulfilled condition which has caused the trigger packet generation

TimeStamp

The FlexCard time stamp (1 µs resolution). Indicates the time at which the packet was generated.

SequenceCount

Sequence count for each signal

PerformanceCounter

Variable that receives the current performance-counter value. This value is only valid for software triggers (*fcTriggerTypeSoftware*).

See Also

fcPacket, fcTriggerType (Obsolete), fcTriggerCondition (Obsolete)

16.27 Enumerations (Obsolete)

16.27.1 fcTriggerCondition (Obsolete)

This enumeration defines the conditions available for a trigger configuration.

```
Typedef enum fcTriggerCondition
{
      fcTriggerConditionFallingEdge          = 1,
      fcTriggerConditionRisingEdge           = 2,
      fcTriggerConditionCycleStart           = 3,
      fcTriggerConditionUser                 = 4,
      fcTriggerConditionErrorDetected        = 5,
      fcTriggerConditionStartupCompleted     = 6,
      fcTriggerConditionTimer                = 7,
}fcTriggerEdge;
```

Members

fcTriggerConditionFallingEdge
Passive mode condition: input trigger is detected on falling edge

fcTriggerConditionRisingEdge
Passive mode condition: input trigger is detected on rising edge

fcTriggerConditionCycleStart
Active mode condition*:* output trigger is set on start of a new FlexRay cycle

fcTriggerConditionUser
Active mode condition: output trigger is set by the user

fcTriggerConditionErrorDetected
Active mode condition: output trigger is set if an error was detected

fcTriggerConditionStartupCompleted
Active mode condition: output trigger is set when the start-up was completed

fcTriggerConditionTimer
Timer mode condition: Internal trigger is set by the software timer (neither input nor output trigger signal is used)

See Also

fcTriggerCfgHardware (Obsolete)

16.27.2 fcTriggerType (Obsolete)

This enumeration defines the different trigger types.

```
Typedef enum fcTriggerType
{
    fcTriggerTypeHardware   = 1,
    fcTriggerTypeSoftware   = 2,
} fcTriggerType;
```

Members

fcTriggerTypeHardware
Hardware trigger

fcTriggerTypeSoftware
Software trigger

See Also

fcTriggerCfg (Obsolete)

16.27.3 fcTriggerMode (Obsolete)

This enumeration defines the different trigger modes.

```
Typedef enum fcTriggerMode
{
    fcTriggerModeActive  = 1,
    fcTriggerModePassive = 2,
    fcTriggerModeTimer   = 3,
}fcTriggerMode;
```

Members

fcTriggerModeActive

Active mode: triggered by FlexCard or by user

fcTriggerModePassive

Passive mode: triggered by external hardware

fcTriggerModeTimer

Timer mode: triggered by software timer.

See Also

fcTriggerCfgHardware (Obsolete), fcTriggerCfgSoftware (Obsolete)

16.28 fcbTrigger (Obsolete)

This function configures and starts/stops a trigger. For further information, refer to the structures **fcTriggerCfgSoftware** and **fcTriggerCfgHardware**.

```
fcError fcbTrigger(
      fcHandle hFlexCard,
      fcBool enable,
      fcTriggerCfg cfg
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

enable

[IN] Set to <> 0 to enable the trigger, and to 0 to disable it.

Cfg

[IN] The trigger configuration

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcTriggerCfg (Obsolete)

16.29 fcbSetCcIndex (Obsolete)

	Information
ℹ	This function is obsolete. Please use the functions in chapter 5 and 6 instead and specify the Communication Controller as parameter.

This function sets the FlexRay Communication Controller index. Following functions refer to the Communication Controller that was set. This function was only available on FlexCard PMC.

```
fcError fcbSetCcIndex (
      fcHandle hFlexCard,
      fcCC     CCIndex
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CCIndex

[IN] The FlexRay Communication Controller to be set.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

`fcCC`

Remarks

The table below gives an overview of the functions which are CC specific.

CC specific functions	CC global functions
`fcbMonitoringStart (Obsolete)`	`fcGetErrorCode`
`fcbMonitoringStop (Obsolete)`	`fcGetErrorType`
`fcbGetCcState (Obsolete)`	`fcGetErrorText`
`fcbSetTransceiverState (Obsolete)`	`fcFreeMemory`
`fcbGetTransceiverState (Obsolete)`	`fcbGetEnumFlexCards (Obsolete)`
`fcbSetCcRegister (Obsolete)`	`fcbOpen`
`fcbGetCcRegister (Obsolete)`	`fcbClose`
`fcbChiCcConfiguration (Obsolete)`	`fcbSetTrigger`
`fcbCanDbCcConfiguration (Obsolete)`	`fcbSetTimer`
`fcbConfigureMessageBuffer (Obsolete)`	`fcbNotificationPacket`
`fcbReconfigureMessageBuffer (Obsolete)`	`fcbReceive`
`fcbGetCcMessageBuffer (Obsolete)`	`fcbSetBusTermination`
`fcbResetCcMessageBuffer (Obsolete)`	`fcbGetBusTermination`
`fcbFilter (Obsolete)`	`fcbGetEnumFlexCardsV2 (Obsolete)`
`fcbSetEventHandle (Obsolete)`	
`fcbTransmit (Obsolete)`	
`fcbTransmitSymbol (Obsolete)`	
`fcbSetCcTimerConfig (Obsolete)`	
`fcbGetCcTimerConfig (Obsolete)`	
`fcbCalculateMacrotickOffset (Obsolete)`	

16.30 fcbGetCcIndex (Obsolete)

Information

This function is obsolete. Please use the functions in chapter 5 and 6 instead and specify the Communication Controller as parameter.

This function reads the index of the set FlexRay Communication Controller. Communication controller dependent functions refer to this Communication Controller only. This function was only available on FlexCard PMC.

```
fcError fcbGetCcIndex (
        fcHandle hFlexCard,
        fcCC *   pCCIndex
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

pCCIndex

[OUT] The FlexRay Communication Controller which is currently set.

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

See Also

fcbSetCcIndex (Obsolete), fcCC

16.31 fcbFRSetCcConfigurationCANdb (Obsolete)

ℹ	Information
	This function is obsolete.

This function configures the FlexRay Communication Controller of the FlexCard with a proprietary string. The configuration string contains the global FlexRay parameter and/or the message buffer configuration. Before the configuration of the Communication Controller starts, all message buffers are reset to their default settings (FIFO buffer). Configuring the CAN CC with a CANdb file is not supported by the FlexCard driver.

```
fcError fcbFRSetCcConfigurationCANdb(
        fcHandle hFlexCard,
        fcCC CC,
        const char* szCanDb
)
```

Parameters

hFlexCard

[IN] Handle to a FlexCard

CC

[IN] Communication controller index

szCanDb

[IN] Pointer to null-terminated proprietary string

Return values

If the function succeeds, the return value is 0. If the value is <> 0, use the functions described in the section Error Handling to get extended error information.

Remarks

This function is only available in the Windows FlexCard driver. The FlexCard Linux and Xenomai drivers don’t support this function.

ℹ	Information
	Internally, the function uses the **fcbFRSetCcRegister** function; therefore the same restrictions as for writing a register exist.

17 Power Management

17.1 Windows

When the PC enters standby or hibernation, the current FlexCard measurement is stopped automatically. After standby the handle to the driver is not valid anymore. On resume, developers should call **`fcbClose, fcbGetEnumFlexCardsV3, fcbOpen`** to initialize the FlexCard again.

Applications have the possibility to react to standby or resume with the Windows message `WM_POWERBROADCAST` containing the events `PBT_APMSUSPEND` and `PBT_APMRESUMESUSPEND`.

Developers should inform the user under Windows Vista and later that standby and hibernation stops the current monitoring. Users have the possibility to deactivate the automatic standby in the control panel.

Developers may consider deactivating idle recognition with the Windows command **`SetThreadExecutionState()`** to prohibit automatic stand-by. However, manual switching to stand-by can not be prevented under Windows Vista and later.

17.2 Linux

ℹ	**Information**
	Please note: Under Linux, Power Management is not supported. Please deactivate kernel power management options to avoid undefined behavior with the FlexCard Linux and Xenomai driver.

18 Tracing

18.1 Overview

The tracing module allows the user to get more information about the *fcBase.dll* (Windows only) activity (e.g. in the case of an error).

The tracing consists of three parts:

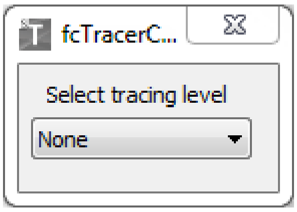
- The tracing module inside the fcBase dynamic link library. This module will send the trace messages to a debugger for displaying (using the windows function `OutputDebugString`).
- The tracing control application to choose the tracing level.
- A debug output viewer (e.g. DebugView from SysInternals) to view the trace messages. If you are debugging your own application, the messages appear normally in the debug output window of your IDE.

The followings tracing levels are available:

- Debug: all trace messages will be shown.
- Info: info and warning messages will be shown.
- Warn: only warning messages will be shown.
- Error: only error messages will be shown.
- Fatal: only fatal error message will be shown.
- None: tracing messages will not be generated.

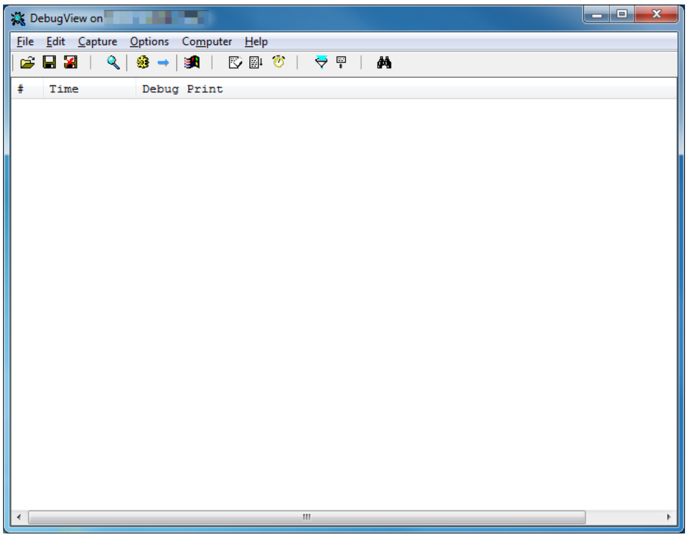
To use the tracing the following steps are required:

Step 1
Start the tracing control application (*fcTracerControl.exe*)

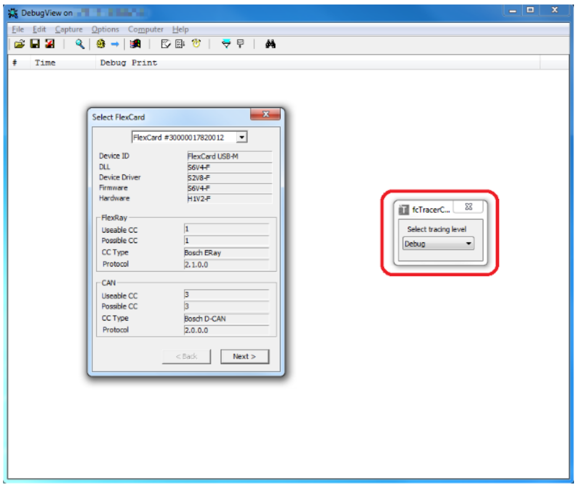


Select tracing level
None

Step 2
Start the debug output viewer *(DebugView.exe*)



Step 3
Start your application. In our case we use the demo application *(fcDemo.exe)*. Now, the tracing level should be selectable.



FlexCard #300000178200 12

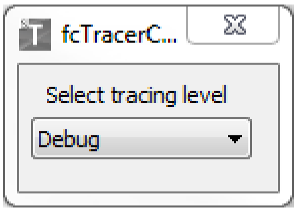
Device ID FlexCard USB-M
DLL S6V4-F
Device Driver S2V8-F
Firmware S6V4-F
Hardware H1V2-F

FlexRay
Useable CC 1
Possible CC 1
CC Type Bosch ERay
Protocol 2.1.0.0

CAN
Useable CC 3
Possible CC 3
CC Type Bosch D-CAN
Protocol 2.0.0.0

Select tracing level
Debug

Step 4
Activate the tracing by choosing a tracing level different of None (e.g. Debug). Use your application and view the trace messages.



Select tracing level
Debug

18.2 Limitation

The tracing module inside the fcBase DLL will update the new tracing level only by calling the following functions:

```
fcbGetEnumFlexCards (Obsolete)
fcbGetEnumFlexCardsV2 (Obsolete)
fcbGetEnumFlexCardsV3
fcbOpen
fcbClose
```

That means a level modification by the tracing control application will only be passed to the tracing module inside the *fcBase.dll* if one of the above functions is called.

This limitation ensures that performance critical functions such as fcbReceive or fcbTransmit are not delayed.

19 Appendix

19.1 Bibliography

[1] FlexCard Cyclone II (SE) Instruction for Use (3-0009-0T01-D01)
[2] MSDN: Dynamic-Link Library Search Order
[3] FlexRay Protocol Specification V2.1 Rev. A
[4] FlexRay Electrical Physical Layer Specification V2.1 Rev. A
[5] Bosch E-Ray FlexRay IP-Module User's Manual
[6] CAN Specification 2.0 Part A (Base frame format)
[7] CAN Specification 2.0 Part B (Base and extended frame format)
[8] FlexConfig_RBS_UserManual (3-0016-0Q01-D04)

19.2 Abbreviations

Abbreviations	Definition
API	Programming Interface
DLL	Dynamic Link Library
IDE	Integrated Development Environment
PDF	Portable Document Format
SYS	System device driver
MFC	Microsoft Foundation Class
CC	Communication controller
PMC	PCI Mezzazine Card
LKM	Loadable kernel module (for Linux OS)
LIB	Library (shared object file)
USB	Universal Serial Bus
PCB	Printed Circuit Board

19.3 Glossary

Term	Description
INF File	A text-based file containing information required by the system to install a device's software components
MFC	C++ Application framework for programming in Microsoft Windows
Qt	C++ Application framework for programming platform independent applications
Cluster	Network topology
CHI	File that configures a Communication Controller
CANdb	File that configures a Communication Controller

19.4 List of Figures

19.5 Index

STAR ELECTRONICS GmbH & Co. KG
A Company of the STAR COOPERATION Group
Jahnstraße 86
73037 Göppingen
Germany
Phone: +49 (0)7031 6288-5656
Info@star-cooperation.com
www.star-cooperation.com/ee-solutions