MCMCAN_Filtering_1 for KIT_AURIX_TC397_TFT

MCMCAN acceptance filtering

AURIX™ TC3xx Microcontroller Training
V1.0.0

Scope of work

The initialization and configuration of several filter modes are used to illustrate different acceptance filtering options.

The CAN messages are sent from CAN node 0 to CAN node 1 using Loop-Back mode. Each transmitted CAN message contains a different message ID and based on the filter configuration, the message is either accepted or rejected by CAN node 1. Messages that passed acceptance filtering are stored in RX FIFOs 0 and 1 or dedicated RX buffer based on the filter configuration. Upon storing the messages, the interrupt service routine is called and the content of the received CAN message is read. Once the content of all the received messages is read, the received data is compared to the transmitted data. If all messages are received without any error detected, an LED is turned on to confirm successful message reception.

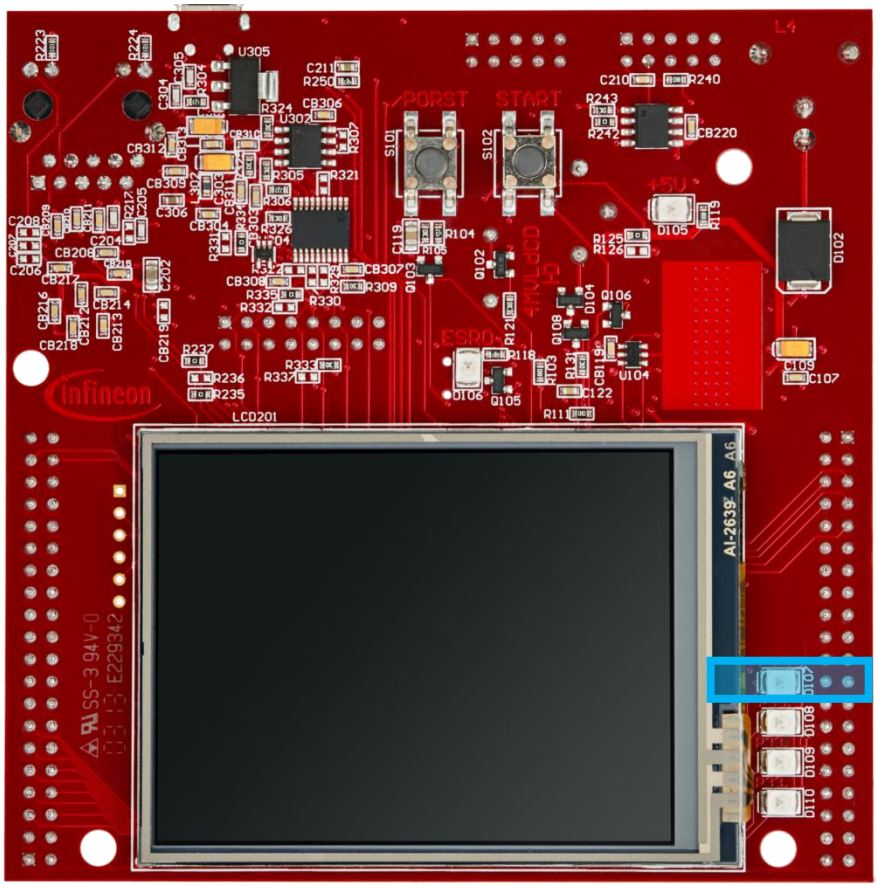
Introduction

- **MCMCAN** is the new CAN interface replacing MultiCAN+ module from the AURIX™ TC2xx family
- The MCMCAN module supports **Classical CAN** and **CAN FD** according to the **ISO 11898-1** standard and Time Triggered CAN (**TTCAN**) according to the **ISO 11898-4** standard
- The MCMCAN module consists of **M_CAN** as CAN nodes (in case of AURIX™ TC39x device, 4 nodes) which are **CAN FD capable**. Each CAN node communicates over two pins (TXD and RXD). Additionally, there is an internal **Loop-Back Mode** functionality available for test purposes
- A configurable **Message RAM** is used to store the messages to be transmitted or received. The message RAM is shared by all the CAN nodes within an MCMCAN module

Hardware setup

This code example has been developed for the board KIT_A2G_TC397_5V_TFT.

In this example, LED D107 is used.



Implementation

- The MCMCAN module supports acceptance filtering in hardware by configuring two sets of acceptance filters, one for standard identifiers and one for extended identifiers. These filters can be assigned to the dedicated RX Buffers or to RX FIFOs 0,1

- The main features of the acceptance filtering are as follows:
  - Each filter element can be configured as:
    - Range filter [from – to]
    - Filter for one or two dedicated message IDs
    - Classic bit mask filter
  - Each filter element is configurable for acceptance or rejection filtering
  - Each filter element can be enabled / disabled individually
  - Filters are checked sequentially, execution stops with the first matching filter element

- In the User Manual, the flow for standard Message ID (11-bit Identifier) and extended Message ID (29-bit Identifier) filtering is described by figures

Implementation

- To demonstrate acceptance filtering mechanisms, the following example of the potential application use case is given:

  - Global filter requirements:
    - Reject remote frames with standard IDs
    - Reject remote frames with extended IDs
    - Accept non-matching messages with standard IDs and store them to RX FIFO 1
    - Reject non-matching messages with extended IDs

  - Standard ID filter requirements:
    - Reject messages with standard IDs in range [0x17-0x19]
    - Accept messages with standard IDs in range [0x14-0x1A] and store them in RX FIFO 0
    - Accept messages with standard IDs 0x184 or 0x187 and store them in RX FIFO 1
    - Accept messages with standard ID 0x189 and store them in RX FIFO
    - Accept messages according to standard ID 0x200 and mask 0x39F and store them in RX FIFO 0
    - Reject messages according to standard ID 0x201 and mask 0x39F
    - Accept messages with standard ID 0x325 and store them in dedicated RX buffer at index 0x2
    - Accept messages with standard ID 0x326 and store them in dedicated RX buffer at index 0x5

Implementation

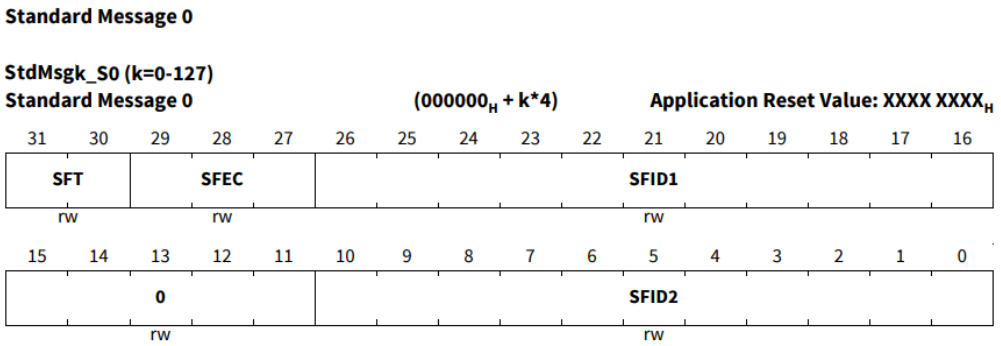
- Extended ID filter requirements:
  - Accept messages with extended ID 0x12222222 and store them in dedicated RX buffer at index 0x0
  - Accept messages with extended IDs in range [0x19999999-0x1BBBBBBB] and store them in RX FIFO 1
  - Accept messages with extended ID 0x1FFABCDE and store them in RX FIFO 0
  - Accept messages with extended ID 0x16666666 and store them in RX FIFO 1

› Based on the given requirements for this example, we can define the following global, standard ID, and extended ID filter elements configuration:

g_mcmcan.canNodeConfig.filterConfig.rejectRemoteFramesWithStandardId = TRUE;
g_mcmcan.canNodeConfig.filterConfig.rejectRemoteFramesWithExtendedId = TRUE;
g_mcmcan.canNodeConfig.filterConfig.standardFilterForNonMatchingFrames = IfxCan_NonMatchingFrame_acceptToRxFifo1;
g_mcmcan.canNodeConfig.filterConfig.extendedFilterForNonMatchingFrames = IfxCan_NonMatchingFrame_reject;

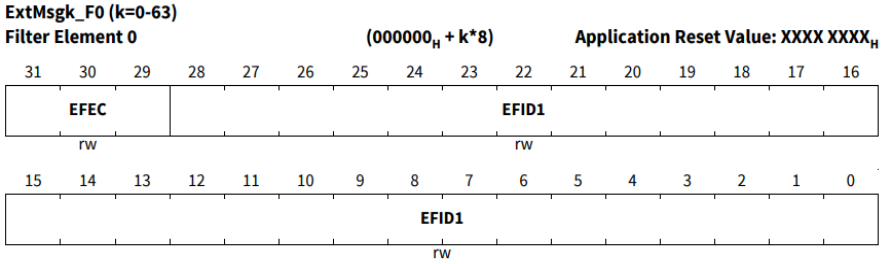
Implementation

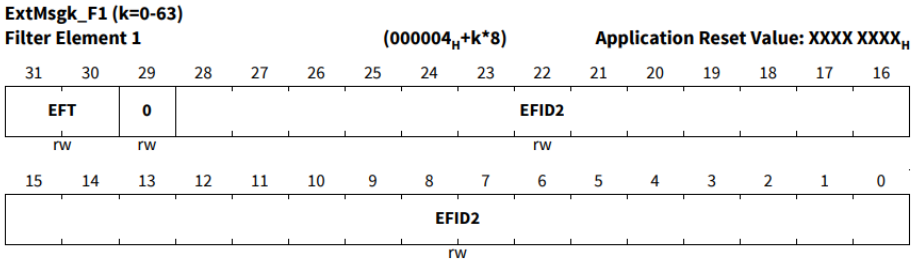
- The standard ID filter elements configuration:
  - SFT: Standard Filter Type
  - SFEC: Standard Filter Element Configuration
  - SFID1: Standard Filter ID 1
  - SFID2: Standard Filter ID 2



Filter Element #	StdMsgk_S0(k=0-127)			
	SFT	SFEC	SFID1	SFID2
0	Range filter	Reject	0x017	0x019
1	Range filter	Store in RX FIFO 0	0x014	0x01A
2	Dual ID filter	Store in RX FIFO 1	0x184	0x187
3	Dual ID filter	Store in RX FIFO 0	0x189	0x189
4	Classic filter	Store in RX FIFO 0	0x200	0x39F
5	Classic filter	Reject	0x201	0x39F
6	Not applicable	Store in RX Buffer	0x325	0x02 (dedicated RX buffer index)
7	Not applicable	Store in RX Buffer	0x326	0x05 (dedicated RX buffer index)

Implementation





- The extended ID filter elements configuration:
  - EFT: Extended Filter Type
  - EFEC: Extended Filter Element Configuration
  - EFID1: Extended Filter ID 1
  - EFID2: Extended Filter ID 2

Filter Element #	ExtMsgk_F0/1(k=0-63)			
	EFT	EFEC	EFID1	EFID2
0	Not applicable	Store in RX Buffer	0x12222222	0x00 (dedicated RX buffer index)
1	Range filter	Store in RX FIFO 1	0x19999999	0x1BBBBBBB
2	Dual ID filter	Store in RX FIFO 0	0x1FFABCDE	0x1FFABCDE
3	Not applicable	Store in RX FIFO 1	0x16666666	0x16666666

Implementation

- To validate the different acceptance filtering options, a set of 25 CAN messages is transmitted. All the CAN messages are acknowledged by the CAN node 1, but based on the acceptance filter configuration certain received CAN messages are accepted and stored while others are rejected accordingly

CAN message #	Message ID	Data Length (in bytes / DLC value)	Accept/ Reject	Matching Filter Element	Storage Destination
0	0x014	8 / 0x8	Accept	Standard ID filter element #1	RX FIFO 0
1	0x015	8 / 0x8	Accept	Standard ID filter element #1	RX FIFO 0
2	0x016	32 / 0xD	Accept	Standard ID filter element #1	RX FIFO 0
3	0x017	64 / 0xF	Reject	Standard ID filter element #0	Not applicable
4	0x018	12 / 0x9	Reject	Standard ID filter element #0	Not applicable

Implementation

CAN message #	Message ID	Data Length (in bytes / DLC value)	Accept/ Reject	Matching Filter Element	Storage Destination
5	0x019	16 / 0xA	Reject	Standard ID filter element #0	Not applicable
6	0x01A	32 / 0xB	Accept	Standard ID filter element #1	RX FIFO 0
7	0x184	4 / 0x4	Accept	Standard ID filter element #2	RX FIFO 1
8	0x187	8 / 0x8	Accept	Standard ID filter element #2	RX FIFO 1
9	0x189	8 / 0x8	Accept	Standard ID filter element #3	RX FIFO 0
10	0x200	8 / 0x8	Accept	Standard ID filter element #4	RX FIFO 0
11	0x201	8 / 0x8	Reject	Standard ID filter element #5	Not applicable

Implementation

CAN message #	Message ID	Data Length (in bytes / DLC value)	Accept/ Reject	Matching Filter Element	Storage Destination
12	0x220	8 / 0x8	Accept	Standard ID filter element #4	RX FIFO 0
13	0x221	8 / 0x8	Reject	Standard ID filter element #5	Not applicable
14	0x240	4 / 0x4	Accept	Standard ID filter element #4	RX FIFO 0
15	0x241	64 / 0xF	Reject	Standard ID filter element #5	Not applicable
16	0x260	12 / 0x9	Accept	Standard ID filter element #4	RX FIFO 0
17	0x261	8 / 0x8	Reject	Standard ID filter element #5	Not applicable
18	0x325	4 / 0x4	Accept	Standard ID filter element #6	RX Buffer #2

Implementation

CAN message #	Message ID	Data Length (in bytes / DLC value)	Accept/ Reject	Matching Filter Element	Storage Destination
19	0x326	32 / 0xD	Accept	Standard ID filter element #7	RX Buffer #5
20	0x327	8 / 0x8	Accept	Global filter configuration	RX FIFO 1
21	0x12222222	8 / 0x8	Accept	Extended ID filter element #0	RX Buffer #0
22	0x19999999	8 / 0x8	Accept	Extended ID filter element #1	RX FIFO 1
23	0x1FFABCDE	32 / 0xD	Accept	Extended ID filter element #2	RX FIFO 0
24	0x16666666	8 / 0x8	Accept	Extended ID filter element #3	RX FIFO 1

Implementation

Application code can be separated into four segments:

- Initialization of the MCMCAN module with the accompanying node and filter elements initialization, implemented in the ***initMCMCAN()*** function
- Initialization of the port pin connected to the LED (D107 on the board). The LED is used to verify the success of a CAN message reception. This is done inside the ***initLed()*** function
- Transmission of the configured CAN messages, implemented in the ***transmitCanMessage()*** function
- Verification of the received CAN messages, implemented in the ***verifyCanMessage()*** function

The additional Interrupt Service Routines (ISRs) are implemented:

- On dedicated RX buffer interrupt, the ISR reads the received CAN message (implemented by ***canIsrRxBufferHandler()*** function)
- On RX FIFO 0 interrupt, the ISR reads the received CAN message (implemented by ***canIsrRxFifo0Handler()*** function)
- On RX FIFO 1 interrupt, the ISR reads the received CAN message (implemented by ***canIsrRxFifo1Handler()*** function)

Implementation

MCMCAN module initialization

Module initialization:

- A default CAN module configuration is loaded into the configuration structure by using the function ***IfxCan_Can_initModuleConfig()***.
  Afterwards, the initialization of the CAN module with the user configuration is done with the function ***IfxCan_Can_initModule()***

Source node initialization:

- A default CAN node configuration is loaded into the configuration structure by using the function ***IfxCan_Can_initNodeConfig()***. Source node is configured as CAN node 0 operating in the Loop-Back Mode. CAN node 0 is set to CAN FD long + fast frame mode and the dedicated TX buffer is used to transmit the CAN messages. CAN node 0 is initialized by calling ***IfxCan_Can_initNode()*** function

Implementation

MCMCAN module initialization

Destination node initialization:

- A default CAN node configuration is loaded into the configuration structure by using the function ***IfxCan_Can_initNodeConfig()***. Destination node is configured as CAN node 1 operating in the Loop-Back Mode. CAN node 1 is set to CAN FD long + fast frame mode and the dedicated RX buffers, RX FIFO 0 and RX FIFO 1 are used to receive the CAN messages. In case the message is accepted by the acceptance filtering, different interrupt service routines are triggered based on the received message storage destination. CAN node 1 is initialized by calling ***IfxCan_Can_initNode()*** function. Finally, standard and extended ID filter elements are initialized based on the configuration given at the slide 8 and 9. This is achieved by calling ***IfxCan_Can_setStandardFilter()*** and ***IfxCan_Can_setExtendedFilter()*** functions.

- All functions used for the MCMCAN module and node initialization are declared in the iLLD header ***IfxCan_Can.h***.

Implementation

Initialization of a pin connected to the LED

An LED is used to verify the success of a CAN message reception. Before using the LED, the port pin to which the LED is connected must be configured.

- First step is to set the port pin to level “HIGH”; this keeps the LED (low-level active) turned off as a default state (***IfxPort_setPinHigh()*** function)
- Second step is to set the port pin to push-pull output mode with the ***IfxPort_setPinModeOutput()*** function
- Finally, the pad driver strength is defined through the function ***IfxPort_setPinPadDriver()***

All functions are declared in the iLLD header ***IfxPort.h***.

Implementation

Transmission of CAN messages

Before a CAN message is transmitted, TX message needs to be initialized with the default configuration (***IfxCan_Can_initMessage()*** function). Default configuration is then modified based on the current filter use case (see the slides 10-13 for more details). Complete TX message data content (data content that is transmitted) is firstly invalidated (***memset()*** function) and then initialized with the combination of current data payload word and current filter use case, using the following format:

bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	0	0	0	***g_currentFilterUseCase*** range: 0 - 24					0	0	0	0	0	0	0	0
bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0	0	0	0	0	0	0	0	0	0	0	0	***currentDataPayloadWord*** range: 0 - 15			

Implementation

Transmission of CAN messages

The RX message (container where the received CAN message is stored) is initialized with the default configuration (after successful acceptance filtering, the values are replaced with the valid content). Additionally, RX message parameters such as "***messageID***", "***dataLenghtCode***", "***frameMode***", and the RX message data content need to be invalidated (***memset()*** function).

After the complete initialization of the TX and RX messages and the message data content, the TX message is transmitted. A CAN message is transmitted by using the ***IfxCan_Can_sendMessage()*** function. A CAN message is continuously transmitted as long as the returned status is ***IfxCan_Status_notSentBusy*** (this status occurs if there is a pending transmit request). The following transmission is delayed by 1 ms using ***wait()*** function allowing the destination node to perform acceptance filtering and to store the received message before next transmission.

The functions ***IfxCan_Can_initMessage()*** and ***IfxCan_Can_sendMessage()*** are declared in the iLLD header ***IfxCan_Can.h***. The function ***memset()*** is declared in the standard C library header ***string.h***. The function ***wait()*** is declared in the iLLD header ***Bsp.h***.

Implementation

Verification of CAN messages

After successful reception of each CAN message, several checks are performed. The expected result of each check depends if the message is expected to be accepted or rejected:

1. Message ID check (check that the received message ID matches the transmitted one). Verifies that both standard and extended IDs have been received
2. Message length check (check that the received message length matches the transmitted one). The check is covering both classical CAN and CAN FD frame sizes
3. Frame mode check (check that the received FD Format (FDF) and Bit Rate Switching (BRS) bit field values match with the expected ones)
4. Storage destination check (check that the received message storage destination matches with the expected one)
5. Expected valid data check (check that the received data matches with the expected one). Both classical CAN and CAN FD data content is covered
6. Invalid data check (check that the invalid data has not been modified with the CAN transmission)

If no error has been observed, the ***g_status*** variable holds ***CanCommunicationStatus_Success*** value upon returning from the ***verifyCanMessage()*** function.

Implementation

Interrupt Service Routines (ISRs)

The ISRs are triggered by the successful CAN message reception. Based on the received message storage destination, different ISRs are triggered, but they all share the same functionality:

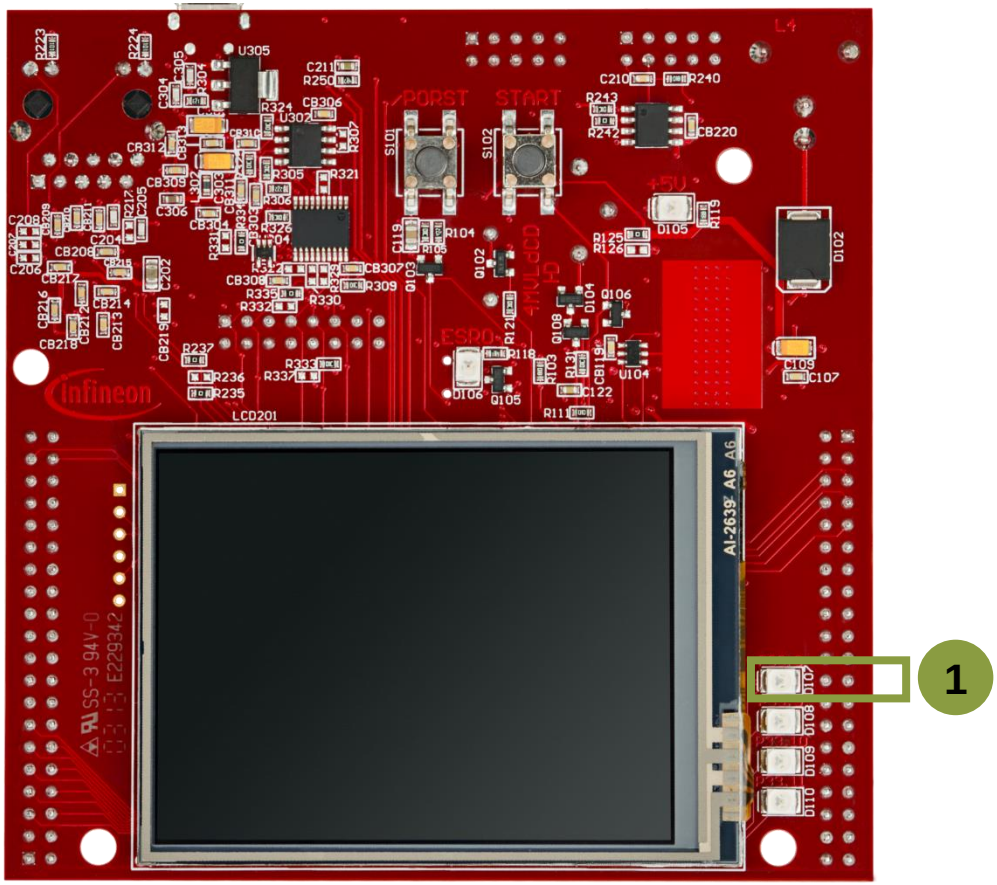
- Clear the pending interrupt flag by using ***IfxCan_Node_clearInterruptFlag()*** function and read the received CAN message with the ***IfxCan_Can_readMessage()*** function

The functions ***IfxCan_Node_clearInterruptFlag()*** is declared in the iLLD header ***IfxCan.h*** while the function ***IfxCan_Can_readMessage()*** is declared in the iLLD header ***IfxCan_Can.h***.

Run and Test

After code compilation and flashing the device, observe the following behavior:

- Check that the LED D107 (1) is turned on (all expected CAN messages have been successfully received and all checks have been passed)



References

- AURIX™ Development Studio is available online:
- https://www.infineon.com/aurixdevelopmentstudio
- Use the *„Import...“* function to get access to more code examples.

- More code examples can be found on the GIT repository:
- https://github.com/Infineon/AURIX_code_examples

- For additional trainings, visit our webpage:
- https://www.infineon.com/aurix-expert-training

- For questions and support, use the AURIX™ Forum:
- https://www.infineonforums.com/forums/13-Aurix-Forum

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-03
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Email: erratum@infineon.com

Document reference
MCMCAN_Filtering_1_KIT_TC397_TFT

IMPORTANT NOTICE
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie") .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.