MULTICAN_RX_FIFO_1 for KIT_AURIX_TC297_TFT

MULTICAN RX FIFO data transmission

AURIX™ TC2xx Microcontroller Training
V1.0.2

Scope of work

MULTICAN is used to exchange data using RX FIFO structure between two nodes, implemented in the same device using Loop-Back mode.

The CAN messages are sent from CAN node 0 over CAN bus (in case of Loop-Back mode all nodes can access the internal bus). CAN node 1 receives the transmitted messages and stores them inside of RX FIFO buffer structure. Once the RX FIFO buffer threshold value has been reached, an interrupt service request is generated. CPU 1 handles the interrupt service request and reads the received CAN messages from the RX FIFO buffer. The content of the received data will be compared to the content of the transmitted CAN messages and in case of success, an LED is turned on to confirm the message reception.

Introduction

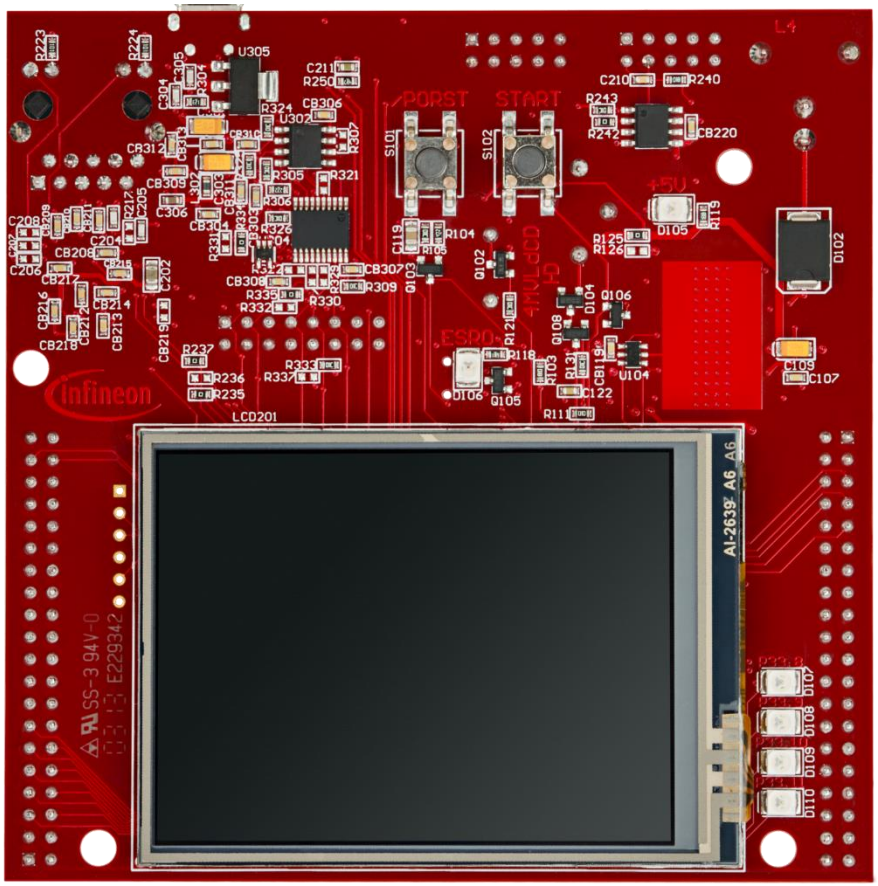
- The MultiCAN+ module provides a communication interface which is **fully compliant with CAN specification V2.0B (active)** and to **CAN FD ISO11898-1 DIS version 2014**, providing communication up to **1 Mbit/s in Classical CAN** (ISO 11898-1:2003(E)mode) and/or **CAN FD up to 5 Mbit/s** (dependent on frequency and nodes)

- The MultiCAN+ module consists of several **CAN nodes** (in case of AURIX™ TC29x 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

- All CAN nodes share a common set of 256 **message objects**. Each message object can be individually allocated to one of the CAN nodes. Besides serving as a **storage container for incoming and outgoing frames**, message objects can be combined to build **gateways** between the CAN nodes or to setup a **FIFO buffer**

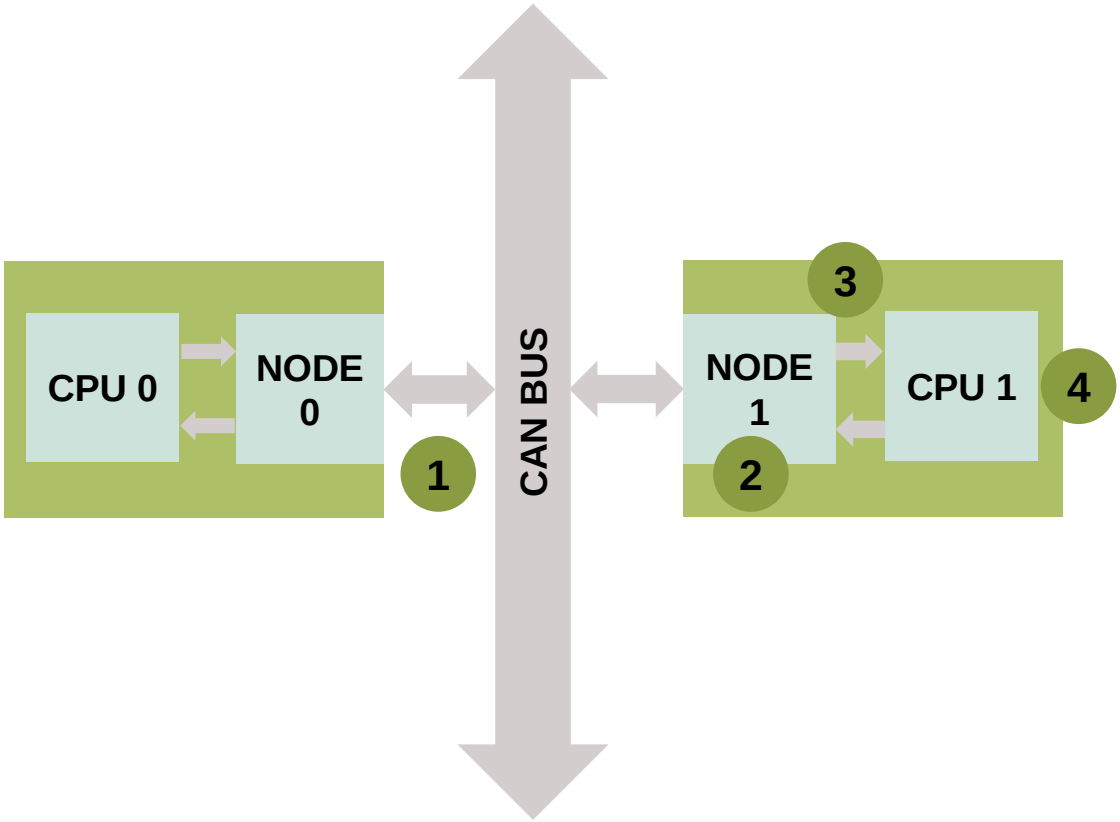
Hardware setup

This code example has been developed for the board KIT_AURIX_TC297_TFT_BC-Step.



Implementation

Application use case covered by this example:



Implementation

Application use case short description:

- The internal Loop-Back mode allows the implementation of the previously shown application use case by using 2 CAN nodes on an AURIX™ TC29x device
- In case of high CPU load, it might be difficult to process a series of CAN frames in time. This might happen if multiple messages are received or must be transmitted in short time. Therefore, a FIFO buffer structure is available to avoid loss of incoming messages and to minimize the setup time for outgoing messages. The FIFO structure can also be used to automate the reception or transmission of a series of CAN messages and to generate a single message interrupt when the whole CAN frame series is transmitted/received

- The application use case is implemented as follows:
  1. CPU 0 instructs the CAN node 0 to send several CAN messages (the number of CAN messages is defined by the user) over the CAN bus as fast as possible (no synchronization between the transmit and receive processes is used)
  2. CAN node 1 receives the messages and stores them inside of the RX FIFO buffer structure
  3. Once the RX FIFO buffer threshold value has been reached, an interrupt service request is generated
  4. CPU 1 handles the interrupt service request and reads the received CAN messages from the RX FIFO buffer

Implementation

The application code can be separated into four segments between two CPU cores:

- **CPU0 core:**
  - Initialization of the MultiCAN+ module together with the accompanying node and message objects, implemented in the ***initMultican()*** function
  - Transmission of the configured CAN messages, implemented in the ***transmitCanMessages()*** function

- **CPU1 core:**
  - 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
  - Verification of the received CAN messages, implemented in the ***verifyCanMessages()*** function

An additional interrupt service routine (ISR) is serviced by CPU1 core:

- On overflow interrupt, the ISR reads the received CAN messages and, in case of no errors, increments the counter to indicate the number of successfully received CAN messages (realized by ***canIsrOverflowHandler()*** function)

Implementation

MultiCAN+ module initialization

MultiCAN+ module initialization is performed in three phases:

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

- A default CAN node configuration is loaded into the configuration structure by using the function ***IfxMultican_Can_Node_initConfig()***. Initialization of the CAN nodes (0 and 1) with the different CAN node ID values and definition of Loop-Back Mode usage for both nodes is done with the function ***IfxMultican_Can_Node_init()***

- A default CAN message object configuration is loaded into the configuration structure by using the function ***IfxMultican_Can_MsgObj_initConfig()***.
  Then, the configuration is customized as needed and finally it is applied with the ***IfxMultican_Can_MsgObj_init()*** function

All functions used for the MultiCAN+ module initialization are declared in the iLLD header ***IfxMultican_Can.h***.

Due to the multiple message objects used in this application use case, the following slides cover the configuration of each message object.

Implementation

Message objects configuration

- Source standard message object (**MO0**):
  - ***canMsgObjConfig.msgObjId = 0*** – defines the message object ID
  - ***canMsgObjConfig.messageId = 0x444*** – defines the CAN message ID used during arbitration phase
  - ***canMsgObjConfig.frame = IfxMultican_Frame_transmit*** – defines the message object as a transmit message object

  Source standard message object (**MO0**) is assigned to **CAN Node 0**.

- RX FIFO object (**MO1** as FIFO base object and **MO2-MO4** as the FIFO slave objects):
  - ***canMsgObjConfig.msgObjId = 1*** – defines the message object ID
  - ***canMsgObjConfig.messageId = 0x444*** – defines the CAN message ID used during arbitration phase (shares the same ID with message object 0 (**MO0**))
  - ***canMsgObjConfig.msgObjCount = 3*** – defines the size of the structure (a value bigger than 1 implies the usage of the FIFO structure: **MO2-MO4**)
  - ***canMsgObjConfig.frame = IfxMultican_Frame_receive*** – defines the message object as a receive message object (RX FIFO in this case)
  - ***canMsgObjConfig.firstSlaveObjId = 2*** – defines the first slave object ID as the next message object after RX FIFO base object

Implementation

- Additional configuration of RX FIFO object needs to be done to support overflow interrupt because the default configuration and the consequent initialization of the message object does not support it.

- This is done in four steps:
  - Get the pointer to the RX FIFO base object by calling ***IfxMultican_MsgObj_getPointer()*** function
  - Enable the overflow interrupt generation by using the function ***IfxMultican_MsgObj_setOverflowInterrupt()***
  - In case the RX FIFO base object is used, the transmit interrupt node becomes active once the overflow event occurs. To connect the transmit interrupt node with the overflow event, the ***IfxMultican_MsgObj_setTransmitInterruptNodePointer()*** function is used
  - Finally, SET pointer should be placed at the first slave message object via ***IfxMultican_MsgObj_setSelectObjectPointer()*** function (once the **whole** RX FIFO buffer is filled, the overflow interrupt will be triggered)

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

Implementation

Transmission of CAN messages

Before the CAN messages are transmitted, a number of CAN messages needs to be initialized. The user can change the number of CAN messages by modifying ***NUMBER_OF_CAN_MESSAGES*** macro value. The TX messages (messages that will be transmitted) are initialized with the combination of predefined content and the current CAN message value. The RX messages (messages where the received CAN message will be stored) are initialized with invalid ID, data, and length value. After successful CAN transmission the values are replaced with the valid content.

- Initialization of both TX and RX messages is done by using ***IfxMultican_Message_init()***
- A CAN message is transmitted by using ***IfxMultican_Can_MsgObj_sendMessage()***. A CAN message is continuously transmitted as long as the returned status is ***IfxMultican_Status_notSentBusy*** (this status occurs if there is a pending transmit request)

The function ***IfxMultican_Message_init()*** is declared in the iLLD header ***IfxMultican.h*** while the ***IfxMultican_Can_MsgObj_sendMessage()*** function is declared in the iLLD header ***IfxMultican_Can.h***.

Implementation

Verification of CAN messages

After successful reception of all expected CAN messages, several checks are performed:

1. Message ID check (check that the received message ID matches the transmitted one)
2. Message length check (check that the received message length matches the transmitted one)
3. Expected valid data check (check that the received data matches with the expected one)
4. Invalid data check (check that the invalid data has not been modified)

Two macros define the range of expected valid and invalid data checks:

- ***NUMBER_OF_CAN_MESSAGES*** (total number of CAN messages to be transmitted)
- ***NUMBER_OF_RECEIVED_MESSAGES*** (expected number of received CAN messages by the RX FIFO object read by the ISR)

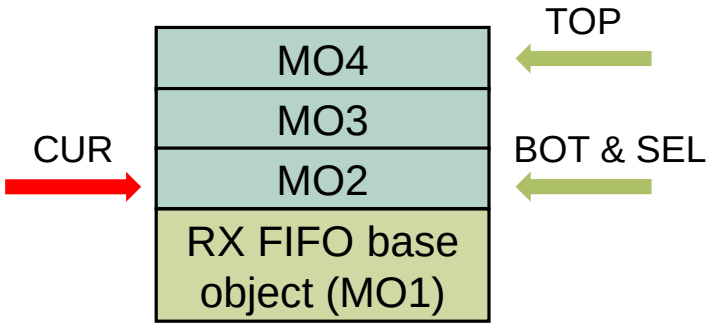
NUMBER_OF_RECEIVED_MESSAGES = RX_FIFO_SIZE * (NUMBER_OF_CAN_MESSAGES / RX_FIFO_SIZE)

These values do not need to match and the impact of these two macros on CAN message verification is shown in the following example.

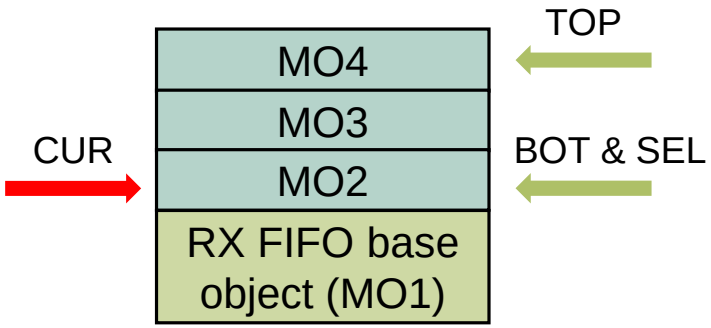
Implementation

Verification of CAN messages

Let us assume the following scenario. ***NUMBER_OF_CAN_MESSAGES*** is **9**, ***RX_FIFO_SIZE*** is 3 which defines the ***NUMBER_OF_RECEIVED_MESSAGES*** to be equal to **9** as well based on the formula on the previous slide.
If RX FIFO base object is defined as message object 1, then the message objects 2 to 4 are slave message objects and the following pointer values are valid: BOT = 2, TOP = 4, CUR = 2, and SEL was set to the same value as first slave object (SEL = 2).



Initial state after initialization



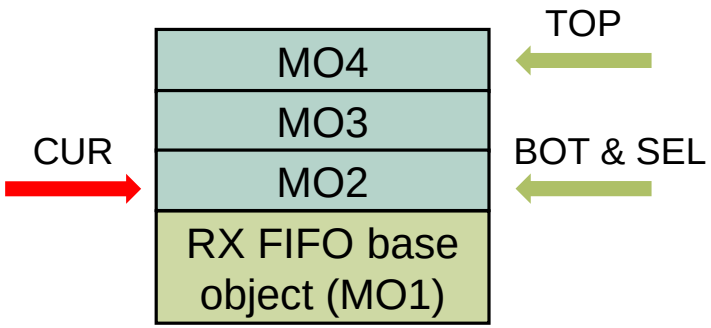
State after 9 CAN messages are received

After 3 CAN messages are received, an overflow interrupt will be triggered (while the CUR pointer equals SEL pointer value) and all the messages located in the RX FIFO object are read and stored in RX message structures. After 6 and finally, after 9 CAN messages are received, same behavior can be observed.

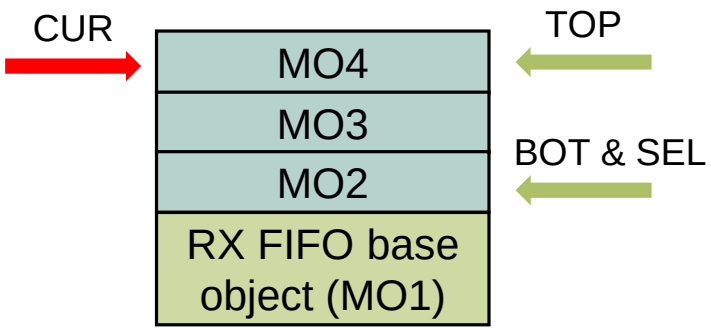
Implementation

Verification of CAN messages

Consider a slightly different scenario. ***NUMBER_OF_CAN_MESSAGES*** is **11**, ***RX_FIFO_SIZE*** is 3 which defines the ***NUMBER_OF_RECEIVED_MESSAGES*** to be equal to **9**.
If RX FIFO base object is defined as message object 1, then the message objects 2 to 4 are slave message objects and the following pointer values are valid: BOT = 2, TOP = 4, CUR = 2, and SEL was set to the same value as first slave object (SEL = 2).



Initial state after initialization



State after 11 CAN messages are received

After 9 CAN messages are received, an overflow interrupt will be triggered and all the messages located in the RX FIFO object are read and stored in RX message structures. However, no additional overflow interrupt will be triggered after two additional CAN messages are received while the CUR pointer value is not equal to SEL pointer value. The additional CAN messages will be read by CPU only if any additional CAN message is received (CUR == SEL).

Implementation

Verification of CAN messages

Finally, in case of successful verification of the CAN messages, the LED is turned on (***IfxPort_setPinLow()***) to indicate the correctness of the received messages and consequently the correctness of the CAN transmission.

The function ***IfxPort_setPinLow()*** function is declared in iLLD header ***IfxPort.h***.

Interrupt Service Routine (ISR)

An ISR is triggered once the RX FIFO buffer threshold value has been reached.

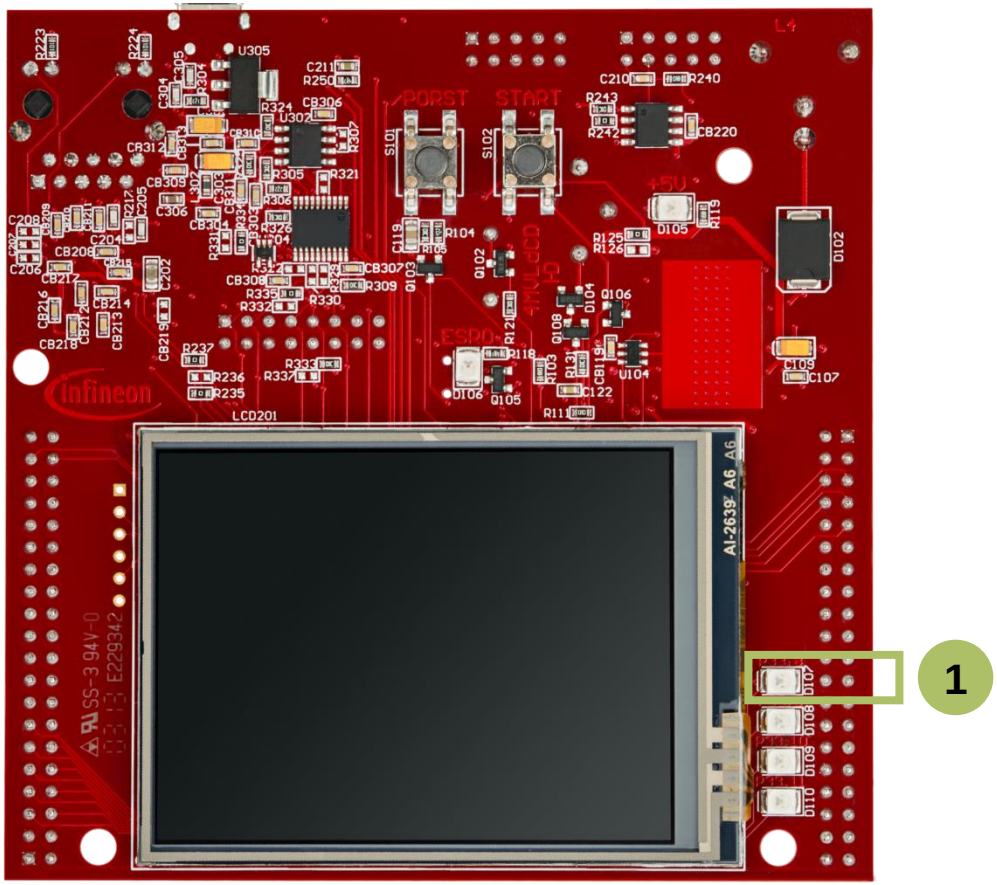
- The RX ISR reads the received CAN messages with the ***IfxMultican_Can_MsgObj_readMessage()*** function. If a non-erroneous return status is present, then a local variable ***numOfReceivedMessages*** is incremented. This variable is used as a counter to indicate the number of successfully received and read CAN messages

The function is declared in the iLLD header ***IfxMultican_Can.h***.

Run and Test

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

- Check that the LED (1) is turned on (correct CAN messages content has been 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

Revision history

Revision	Description of change
V1.0.2	Update of version to be in line with the code example’s version
V1.0.1	Update of version to be in line with the code example’s version
V1.0.0	Initial version

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

Edition 2021-06
Published by
Infineon Technologies AG
81726 Munich, Germany

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

Document reference
MULTICAN_RX_FIFO_1_KIT_TC297_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.