CAN_LDD
Embedded Component User Guide

Contents

1 General Information

Component Level: Logical Device Driver

Category: Logical Device Drivers-Communication

This component provides services for serial communication using the CAN 2.0 A and B protocol developed by Robert Bosch 1991 standard ISO 11868.

These services are corresponding to a *data link layer* in ISO/OSI Network model and are designed for "fullCAN" chips. Generally there are independent message buffers which can be configured for standard or extended frames and also can have different *acceptance codes* for message filtering. Each message buffer is used both as transmit and receive buffer.

Guidelines for using this component:

1. Set important properties in the inspector window (communication baud rate, etc...)

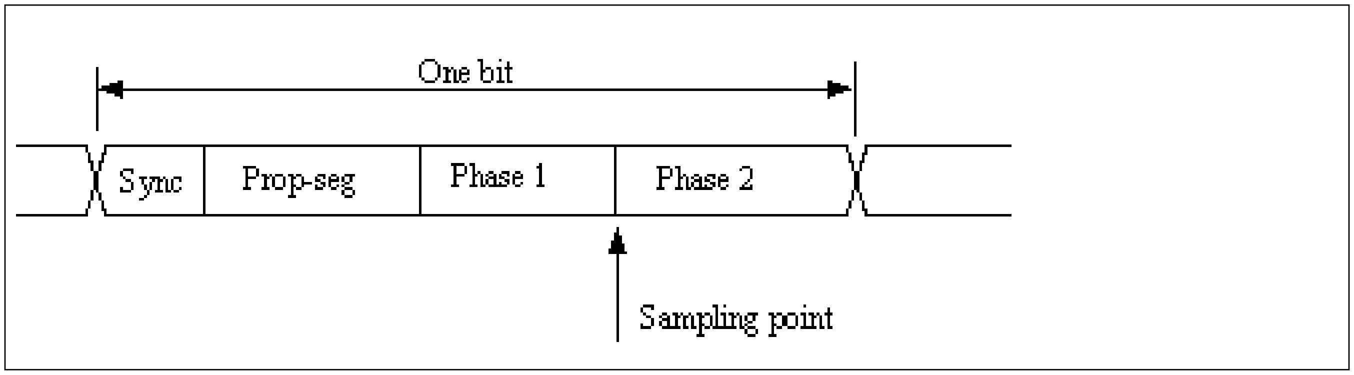
2. The operation of the component is started by the Enable method.

3. Data frames and remote frames are sent by the SendFrame method.

4. The communication is stopped by the Disable method. It is necessary to disable the communication before switching to a power saving mode.

Timing segments

- the Synchronization Segment
- the Propagation Segment
- the Phase Segment 1
- the Phase Segment 2



The length of segments that can be set in the timing group of the component properties.

2 Properties

This section describes component's properties. Properties are parameters of the component that influence the generated code. Please see the Processor Expert user manual for more details.

- **Component name** - Component name
- **CAN channel** - CAN channel
- **Interrupt service** - Component may or may not be implemented using interrupts. If this property is set to "Enabled", the component functionality (e.g. sending and receiving data) depends on the interrupt service and will not operate if the CPU interrupts are disabled. For details please refer to chapter "interrupt service in the component's generated code".

  The following items are available only if the group is enabled (the value is "Enabled"):

  - **Interrupt error** - Error interrupt
  - **Interrupt error priority** - Priority of interrupt
  - *Settings only if component supports interrupt service routine properties.*
    - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
  - *Settings only if compiler supports preserve interrupt preserve registers.*
    - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

      If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

  The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.

- **Interrupt bus off** - Bus-off interrupt
- **Interrupt bus off priority** - Priority of interrupt
- *Settings only if component supports interrupt service routine properties.*
    - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
- *Settings only if compiler supports preserve interrupt preserve registers.*
    - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

      If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

      The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.
- *Settings only if CAN supports global RxTx MB interrupt vector.*
    - **Interrupt message buffers** - Interrupt Or'ed Message Buffers
    - **Interrupt message buffers priority** - Or'ed Message buffers priority
    - *Settings only if component supports interrupt service routine properties.*
        - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
    - *Settings only if compiler supports preserve interrupt preserve registers.*
        - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

          If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

          The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.
- *Settings only if CAN supports RxTx warning interrupt vector.*
    - **Interrupt Tx warning** - Transmit Warning Interrupt
    - **Interrupt Tx warning priority** - Transmit Warning Interrupt priority
    - *Settings only if component supports interrupt service routine properties.*
        - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
    - *Settings only if compiler supports preserve interrupt preserve registers.*

                - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

                  If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

                  The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.

            - **Interrupt Rx warning** - Receive Warning Interrupt
            - **Interrupt Rx warning priority** - Receive Warning Interrupt priority
            - *Settings only if component supports interrupt service routine properties.*
                - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
            - *Settings only if compiler supports preserve interrupt preserve registers.*
                - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

                  If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

                  The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.
        - *Settings only if CAN supports WakeUp feature.*
            - **Interrupt wake up** - Wake Up interrupt
            - **Interrupt wake up priority** - Wake Up interrupt priority
            - *Settings only if component supports interrupt service routine properties.*
                - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
            - *Settings only if compiler supports preserve interrupt preserve registers.*
                - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

                  If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

                  The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.
    - **Settings** - Settings of the CAN
        - **Pins** - Pins.

- **Rx pin** - Rx pin.
  - **Rx pin** - Receive pin
  - **Rx pin signal** - Input signal name
- **Tx pin** - Tx pin.
  - **Tx pin** - Transmit pin
  - **Tx pin Signal** - Output signal name

- **Global acceptance mask** - Enable/Disable global/invidual buffer masking.

  The following items are available only if the group is enabled (the value is "yes"):

  - *Settings only for FlexCAN periphery*
    - **Acceptance mask for buffer 0 .. n** - Acceptance Mask used to mask the filter fields of all Rx MBs, excluding MBs 14-15. This mask is used for standard and extended frames.
    - **Acceptance mask for buffer 14** - Mask for message filtering of the received frames and is valid for message buffers 14. Logical 0 in this mask means that the corresponding incoming ID bit is "don't care". This mask is used for standard and extended frames.
    - **Acceptance mask for buffer 15** - Mask for message filtering of the received frames and is valid for message buffers 14. Logical 0 in this mask means that the corresponding incoming ID bit is "don't care". This mask is used for standard and extended frames.
- **Receiver FIFO** - Enable/Disable receiver FIFO feature.

  The following items are available only if the group is enabled (the value is "Enabled"):
- *Settings only for support message buffers*
  - **Message buffers** - FlexCAN message buffer configuration.

    One Item of the list looks like:

    Buffer0 - Buffer

    - **Buffer type** - Configuration of the given message buffer. Each buffer can be configured as receive or transmit buffer.

      The following items are available only if the group is enabled (the value is "Receive"):

      - *Settings only for FlexCAN periphery*
        - **Accept frames** - Specifies what frame types (standard/extended) will be accepted during reception process.
        - **Message ID** - Init value of the buffer ID. ID of each buffer is used by FlexCAN during receiving process.
      - *Settings only if CAN supports invidual acceptance mask feature.*
        - **Invidual Acceptance Mask** - The item enables/disables invidual acceptance mask. Invidual acceptance mask settings is controlled by "Global Acceptance Mask" property.

          The following items are available only if the group is enabled (the value is "Enabled"):

          - **Acceptance Mask** - Acceptance Mask of each buffer, it is used by FlexCAN during receiving process.
- *Settings only if CAN supports abort transmission mode feature.*
  - **Abort transmission mode** - It enables the Tx abort feature. This feature guarantees a safe procedure for aborting a pending transmission, so that no frame is sent in the CAN bus without notification.
- *Settings only if CAN supports remote request storing feature.*

  - **Remote request storing** - If remote response frame is generated an automatic remote response frame is generated if a message buffer with CODE=0b1010 is found with the same ID. Otherwise remote request frame is submitted to a matching process and stored in the corresponding message buffer in the same fashion of a data frame

- *Settings only if CAN supports Entire Frame Arbitration Field Comparison feature.*
  - **Entire frame arbitration field comparison** - Controls the comparison of IDE and RTR bits within Rx Mailboxes filters with their corresponding bits in the incoming frame by the matching process. This bit does not affect matching for Rx FIFO.
- *Settings only if CAN supports local priority feature.*
  - **Local priority** - It controls whether the local priority feature is enabled or not. It is used to expand the ID used during the arbitration process. With this expanded ID concept, the arbitration process is done based on the full 32-bit word, but the actual transmitted ID still has 11-bit for standard frames and 29-bit for extended frames.
- *Settings only if CAN supports self reception feature.*
  - **Self reception** - Defines whether FlexCAN is allowed to receive frames transmitted by itself. If disabled frames transmitted by the module will not be stored in any MB, regardless if the MB is programmed with an ID that matches the transmitted frame, and no interrupt flag or interrupt signal will be generated due to the frame reception.
- **Timer synchronization** - This property enables a mechanism to reset, or clear, the free-running Timer each time a message is received in the message buffer 0.
- *Settings only for FlexCAN periphery*
  - **Lowest buffer transmitted first** - This property defines the transmit-first scheme.
- **Loop mode** - When this bit is set, the CAN module performs an internal loop back which can be used for self test operation. The bit stream output of the transmitter is fed back to the receiver internally. The RXCAN input pin is ignored and the TXCAN output goes to the recessive state (logic ‘1’). The CAN module behaves as it does normally when transmitting and treats its own transmitted message as a message received from a remote node. In this state, the CAN module ignores the bit sent during the ACK slot in the CAN frame Acknowledge field to ensure proper reception of its own message. Both transmit and receive interrupts are generated.
- **Bus off recovery mode** - The property configures the bus-off recovery mode. See also BusOffRecoveryRequest method.

  There are 2 options:
  - Automatic: Automatic bus-off recovery
  - User: Bus-off recovery upon user request
- **Listen only mode** - This property configures the CAN module as a bus monitor. When this property is set to "yes", all valid CAN messages with matching ID are received, but no acknowledgement or error frames are sent out. In addition, the error counters are frozen. Listen Only Mode supports applications which require "hot plugging" or throughput analysis. The CAN module is unable to transmit any messages, when listen only mode is active.
- *Settings only if CAN supports WakeUp feature.*
  - **Wake up** - Enables wakeup function.

    The following items are available only if the group is enabled (the value is "Enabled"):
- *Settings only if CAN support Memory error detection and correction*
  - **Memory error detection and correction** - Settings of memory error detection and correction (ECC).

    The following items are available only if the group is enabled (the value is "Enabled"):

        - **Non-Correctable errors interrupt from host** - Enables the interrupt in case of non-correctable errors detected in memory reads issued by the host (CPU).
        - **Non-Correctable errors interrupt from CAN** - Enables the interrupt in case of non-correctable errors detected in memory reads issued by the FlexCAN internal processes.
        - **Correctable errors interrupt** - Enables the interrupt in case of correctable errors detected in memory reads issued by the host or FlexCAN internal processes.
        - **Host access error injection** - Enables the injection of errors only in memory reads issued by the host (CPU).
        - **FlexCAN access error injection** - Enables the injection of errors only in memory reads issued by the FlexCAN internal processes.
        - **Extend error injection** - Extends the error injection on FlexCAN accesses larger than 32-bit up to 64-bit word accessed by FlexCAN internal process.
        - **Error report update** - Enables the update of the error report registers.
        - **Non-Correctable errors in CAN in freeze mode** - Determines the response when a non-correctable error is detected in a memory read performed by FlexCAN internal processes.
        - **Inject error to address** - This vlue specifies the address where error will be injected in physical FlexCAN RAM memory.

          There are 15 options:

            - Message Buffers: Address in RAM: 0x0080
            - RXIMRs: Address in RAM: 0x0880
            - RXFIR_0: Address in RAM: 0x0A80
            - RXFIR_1: Address in RAM: 0x0A84
            - RXFIR_2: Address in RAM: 0x0A88
            - RXFIR_3: Address in RAM: 0x0A8C
            - RXFIR_4: Address in RAM: 0x0A90
            - RXFIR_5: Address in RAM: 0x0A94
            - RXMGMASK: Address in RAM: 0x0AA0
            - RXFGMASK: Address in RAM: 0x0AA4
            - RX14MASK: Address in RAM: 0x0AA8
            - RX15MASK: Address in RAM: 0x0AAC
            - SMB_TX: Address in RAM: 0x0AB0
            - SMB_RX0: Address in RAM: 0x0AC0
            - SMB_RX1: Address in RAM: 0x0AD0

- **Timing** - Timing of the CAN
    - **CAN timing calculator** - CAN timing calculator allows to set the communication speed and then calculates all necessary timing properties. More details describes AN1798 - CAN Bit Timing Requirements.
    - *Settings only for FlexCAN periphery*
        - **Propagation segment** - The length of the propagation segment (number of time quanta).

          There are 8 options:

            - 1: 1 time quanta

    - 2: 2 time quanta
    - 3: 3 time quanta
    - 4: 4 time quanta
    - 5: 5 time quanta
    - 6: 6 time quanta
    - 7: 7 time quanta
    - 8: 8 time quanta

- **Time segment 1** - The length of the time segment 1 (number of time quanta).

  There are 16 options:

  - 1: 1 time quanta
  - 2: 2 time quanta
  - 3: 3 time quanta
  - 4: 4 time quanta
  - 5: 5 time quanta
  - 6: 6 time quanta
  - 7: 7 time quanta
  - 8: 8 time quanta
  - 9: 9 time quanta
  - 10: 10 time quanta
  - 11: 11 time quanta
  - 12: 12 time quanta
  - 13: 13 time quanta
  - 14: 14 time quanta
  - 15: 15 time quanta
  - 16: 16 time quanta

- **Time segment 2** - The length of the time segment 2 (number of time quanta).

  There are 8 options:

  - 1: 1 time quanta
  - 2: 2 time quanta
  - 3: 3 time quanta
  - 4: 4 time quanta
  - 5: 5 time quanta
  - 6: 6 time quanta
  - 7: 7 time quanta
  - 8: 8 time quanta

- **Resync jump width** - Resynchronization jump width (number of time quanta).

  There are 4 options:

- 1: 1 time quanta
- 2: 2 time quanta
- 3: 3 time quanta
- 4: 4 time quanta
- **Time quanta per bit** - Number of time quanta per bit. The formula used to calculate the value is: Time quanta per bit = (Propagation segment) + (Time segment 1) + (Time segment 2) + 1.

  Note: The propagation segment property may not be available on all CAN devices. It depends on HW implementation of the CAN module.

  This property is for information only.
- **Samples per bit** - This bit determines the number of serial bus samples to be taken per bit time.
- **Bit rate** - Communication clock rate.

- **Initialization** - Initialization of the CAN
  - **Enabled in init. code** - Enabled in initialization code
  -
    - **Auto initialization** - Automated initialization of the component. The component Init method is automatically called from CPU component initialization function PE_low_level_init(). In this mode, the constant <ComponentName>_DeviceData is defined in component header file and it can be used as a device data structure pointer that can be passed as a first parameter to all component methods.
  - **Event mask** - This group defines initialization event mask value.
    - **OnFreeTxBuffer** - Specifies if OnFreeTxBuffer event is enabled in initialization code.
    - **OnFullRxBuffer** - Specifies if OnFullRxBuffer event is enabled in initialization code.
    - **OnTransmitWarning** - Specifies if OnTransmitWarning event is enabled in initialization code.
    - **OnReceiveWarning** - Specifies if OnReceiveWarning event is enabled in initialization code.
    - **OnBusOff** - Specifies if OnBusOff event is enabled in initialization code.
    - **OnWakeUp** - Specifies if OnWakeUp event is enabled in initialization code.
    - **OnError** - Specifies if OnError event is enabled in initialization code.
-
  - **CPU clock/configuration selection** - Settings for the CPU clock configurations: specifies whether the component is supported or not.

    For details about speed modes please refer to page Speed Modes Support.
    - **Clock configuration 0** - The component is enabled/disabled in the clock configuration 0.
    - **Clock configuration 1** - The component is enabled/disabled in the clock configuration 1.
    - **Clock configuration 2** - The component is enabled/disabled in the clock configuration 2.
    - **Clock configuration 3** - The component is enabled/disabled in the clock configuration 3.
    - **Clock configuration 4** - The component is enabled/disabled in the clock configuration 4.
    - **Clock configuration 5** - The component is enabled/disabled in the clock configuration 5.
    - **Clock configuration 6** - The component is enabled/disabled in the clock configuration 6.
    - **Clock configuration 7** - The component is enabled/disabled in the clock configuration 7.

3 Methods

This section describes component's methods. Methods are user-callable functions/subroutines intended for the component runtime control. Please see the Processor Expert user manual for more details.

3.1 Init

Initializes the device. Allocates memory for the device data structure, allocates interrupt vectors and sets interrupt priority, sets pin routing, sets timing, etc. If the "Enable in init. code" is set to "yes" value then the device is also enabled(see the description of the Enable() method). In this case the Enable() method is not necessary and needn't to be generated.

Prototype

```
LDD_TDeviceData* Init(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr: Pointer to LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer will be passed as an event or callback parameter.

Return value

- *Return value:LDD_TDeviceData** - Pointer to the dynamically allocated private structure or NULL if there was an error.

3.2 Deinit

Deinitializes the device and frees the device data structure memory.

Prototype

```
void Deinit(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.3 Enable

Enables the device, starts the transmitting and receiving.

Prototype

```
LDD_TError Enable(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

3.4 Disable

Disables the device, stops the transmitting and receiving.

Prototype

```
LDD_TError Disable(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

3.5 SetEventMask

Enables/Disables events. This method is available if the interrupt service/event property is enabled and at least one event is enabled. Pair method to GetEventMask().

Prototype

```
LDD_TError SetEventMask(LDD_TDeviceData *DeviceDataPtr, LDD_TEventMask EventMask)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *EventMask:LDD_TEventMask* - Mask of events to enable.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - Device is disabled

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_MASK - Event mask not valid

3.6 GetEventMask

Returns current event mask. This method is available if the interrupt service/event property is enabled and at least one event is enabled.

Prototype

```
LDD_TEventMask GetEventMask(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TEventMask* - Current event mask.

3.7 SetAcceptanceMask

Sets the message acceptance mask. The acceptance mask is used for message filtering of incoming frames.

Prototype

```
 LDD_TError SetAcceptanceMask(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TAccMask AccMask)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index for Acceptance Mask Message Buffer.
- *AccMask:LDD_CAN_TAccMask* - 11-bit Mask could be selected for a standard frame or 29-bit for an extended frame.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of buffer index is out of range.

  ERR_NOTAVAIL - Acceptance mask isn't possible change when CAN component is running, component must be disabled or switched to freeze mode.

3.8 GetAcceptanceMask

Returns the message acceptance mask. The acceptance mask is used for message filtering of incoming frames.

Prototype

```
 LDD_TError GetAcceptanceMask(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TAccMask *AccMask)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index for Acceptance Mask Message Buffer.
- *AccMask: Pointer to LDD_CAN_TAccMask* - Pointer to returned acceptance mask for message buffer[Index].

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_PARAM_RANGE - Value of buffer index is out of range.

3.9 SetRxBufferID

Sets the ID of the receive message buffer specified by the parameter BufferIdx.

Prototype

```
 LDD_TError SetRxBufferID(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TMessageID MessageID)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Rx message buffer.
- *MessageID:LDD_CAN_TMessageID* - 11-bit Mask could be selected for a standard frame or 29-bit for an extended frame.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user.

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of message buffer index is out of range.

  ERR_PARAM_INDEX - Index of message buffer is not for data receive.

3.10 GetRxBufferID

Returns the ID of the receive message buffer specified by the parameter BufferIdx.

Prototype

```
 LDD_TError GetRxBufferID(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TMessageID *MessageID)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Rx message buffer.
- *MessageID: Pointer to LDD_CAN_TMessageID* - Pointer to the return messageID.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user.

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of message buffer index is out of range.

  ERR_PARAM_INDEX - Index of message buffer is not for data receive.

3.11 SetRxBufferState

Sets the message buffer code for the appropriate message buffer.

Prototype

```
 LDD_TError SetRxBufferState(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TRxBufferState BufferCode)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Rx message buffer.
- *BufferCode:LDD_CAN_TRxBufferState* - Message buffer code for the appropriate message buffer.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user.

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of buffer index is out of range.

  ERR_PARAM_INDEX - Index of message buffer is not for data receive.

  ERR_PARAM_VALUE - Value of message buffer code is fail.

3.12 GetRxBufferState

Sets the message buffer code for the appropriate message buffer.

Prototype

```
 LDD_TError GetRxBufferState(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TRxBufferState *BufferCode)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Rx message buffer.
- *BufferCode: Pointer to LDD_CAN_TRxBufferState* - Pointer to return message buffer code for the appropriate message buffer.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user.

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of buffer index is out of range.

  ERR_PARAM_INDEX - Index of message buffer is not for data receive.

3.13 SendFrame

Sends a frame via the CAN device. This method allow to specify CAN buffer number, message ID, data to be sent, frame type and whether the message will be sent after the request comes.

Prototype

```
 LDD_TError SendFrame(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TFrame *Frame)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Tx message buffer.
- *Frame: Pointer to LDD_CAN_TFrame* - Pointer to the CAN frame to send.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of buffer index is out of range.

  ERR_PARAM_INDEX - Index of message buffer is not for transmit.

  ERR_PARAM_LENGTH - Number of data in the frame is greater than MaxDataLength.

  ERR_PARAM_ATTRIBUTE_SET - Frame type isn't supported.

  ERR_PARAM_VALUE - Value of Tx priority is fail.

  ERR_BUSY - CAN module is busy.

3.14 ReadFrame

Reads a frame from the CAN device. The user is informed about CAN reception through OnFullRxBuffer event or GetStateRX method.

Prototype

```
 LDD_TError ReadFrame(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx,
LDD_CAN_TFrame *Frame)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Rx message buffer.
- *Frame: Pointer to LDD_CAN_TFrame* - Pointer to the received CAN frame.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_PARAM_RANGE - Value of buffer index is out of range.

  ERR_PARAM_INDEX - Index of message buffer is not for receive.

  ERR_BUSY - CAN module is busy.

  ERR_RXEMPTY - The receive buffer is empty.

  ERR_OVERRUN - The receive buffer is overrun.

3.15 GetTxFrameState

Returns the complete status of the transmission buffer.

Prototype

```
 bool GetTxFrameState(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Tx message buffer.

Return value

- *Return value:bool* - Return value:

  true - frame is completely transmitted.

  false - frame isn't completely transmitted.

3.16 GetRxFrameState

Returns the complete status of the reception buffer.

Prototype

```
bool GetRxFrameState(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TMBIndex BufferIdx)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Index of the Rx message buffer.

Return value

- *Return value:bool* - Return value:

  true - frame is completely received.

  false - frame isn't completely received.

3.17 GetTransmitErrorCounter

Returns a value of the transmission error counter. This method is available only if method SendFrame is enabled.

Prototype

```
LDD_CAN_TErrorCounter GetTransmitErrorCounter(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_CAN_TErrorCounter* - The value of the transmission error counter.

3.18 GetReceiveErrorCounter

Returns a value of the reception error counter.

Prototype

```
LDD_CAN_TErrorCounter GetReceiveErrorCounter(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_CAN_TErrorCounter* - The value of the reception error counter.

3.19 GetError

Returns value of error mask, e.g. LDD_CAN_BIT0_ERROR

Prototype

```
LDD_TError GetError(LDD_TDeviceData *DeviceDataPtr, LDD_CAN_TErrorMask *ErrorMaskPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *ErrorMaskPtr: Pointer to LDD_CAN_TErrorMask* - Pointer to a variable where errors value mask will be stored.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - Device is disabled

  ERR_SPEED - This device does not work in the active clock configuration

3.20 Main

This method is available only for polling mode. If interrupt service is disabled this method replaces the CAN interrupt handlers. This method should be called if SendFrame/ReadFrame was invoked before in order to run the reception/ transmition.

Prototype

```
void Main(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.21 RequestBusOffRecovery

Request to CAN module recover after a Bus Off, this methos is enabled if Recovery mode is set to User.

Prototype

```
LDD_TError RequestBusOffRecovery(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_DISABLED - This component is disabled by user.

ERR_SPEED - This device does not work in the active clock configuration.

3.22 GetStats

Returns device receive/transmit statistics since the device initialization or since the statistical information has been cleared.

Prototype

```
LDD_CAN_TStats GetStats(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_CAN_TStats* - Device receive/transmit statistics.

3.23 ClearStats

Clears the device receive/transmit statistics.

Prototype

```
void ClearStats(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.24 ConnectPin

This method reconnects the requested pin associated with the selected peripheral in this component. This method is available only for CPU derivatives and peripherals that support runtime pin sharing with other internal on-chip peripherals.

Prototype

```
LDD_TError ConnectPin(LDD_TDeviceData *DeviceDataPtr, LDD_TPinMask PinMask)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *PinMask:LDD_TPinMask* - Mask for the requested pins. The peripheral pins are reconnected according to this mask.

  Possible parameters:

  'ComponentName'_TX_PIN - Transmitter pin

  'ComponentName'_RX_PIN - Receiver pin

Return value

- *Return value:LDD_TError* - Error code, possible codes:

ERR_OK - OK

ERR_SPEED - This device does not work in the active clock configuration

ERR_PARAM_MASK - Invalid PinMask value.

3.25 SetOperationMode

This method requests to change the component's operation mode. Upon a request to change the operation mode, the component will finish a pending job first and then notify a caller that an operation mode has been changed. When no job is pending (ERR_OK), the component changes an operation mode immediately and notify a caller about this change.

Prototype

```
 LDD_TError SetOperationMode(LDD_TDeviceData *DeviceDataPtr, LDD_TDriverOperationMode
OperationMode, LDD_TCallback ModeChangeCallback, LDD_TCallbackParam
*ModeChangeCallbackParamPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *OperationMode:LDD_TDriverOperationMode* - Requested driver operation mode.
- *ModeChangeCallback:LDD_TCallback* - Callback to notify the upper layer once a mode has been changed.
- *ModeChangeCallbackParamPtr: Pointer to LDD_TCallbackParam* - Pointer to callback parameter to notify the upper layer once a mode has been changed.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - The change operation mode request has been accepted, callback will notify an application as soon as the mode is changed.

  ERR_SPEED - The component does not work in the active clock configuration.

  ERR_DISABLED - This component is disabled by user.

  ERR_PARAM_MODE - Invalid operation mode.

  ERR_BUSY - Job is running and the driver can't detect job end by itself. The approximate end of the job can be detected by method GetDriverState. The real transmission/reception finishes later. It depends on component settings (data width, timing, size of buffer, etc.).

3.26 GetDriverState

This method returns the current driver status.

Prototype

```
 LDD_TDriverState GetDriverState(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TDriverState* - The current driver status mask.

  Following status masks defined in PE_Types.h can be used to check the current driver status.

  PE_LDD_DRIVER_DISABLED_IN_CLOCK_CONFIGURATION - 1 - Driver is disabled in the current mode; 0 - Driver is enabled in the current mode.

  PE_LDD_DRIVER_DISABLED_BY_USER - 1 - Driver is disabled by the user; 0 - Driver is enabled by the user.

  PE_LDD_DRIVER_BUSY - 1 - Driver is the BUSY state (Tx or Rx frame is in message buffer); 0 - Driver is in the IDLE state.

4 Events

This section describes component's events. Events are call-back functions called when an important event occurs. For more general information on events, please see the Processor Expert user manual.

4.1 OnFreeTxBuffer

This event is called when the buffer is empty after a successful transmit of a message. This event is available only if method SendFrame is enabled.

Prototype

```
void OnFreeTxBuffer(LDD_TUserData *UserDataPtr, LDD_CAN_TMBIndex BufferIdx)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Receive message buffer index.

4.2 OnFullRxBuffer

This event is called when the buffer is full after a successful receive a message. This event is available only if method ReadFrame or SetRxBufferState is enabled.

Prototype

```
void OnFullRxBuffer(LDD_TUserData *UserDataPtr, LDD_CAN_TMBIndex BufferIdx)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.
- *BufferIdx:LDD_CAN_TMBIndex* - Transmit buffer index.

4.3 OnTransmitWarning

This event is called when the CAN controller goes into warning status due to the transmit error counter exceeding 96 and neither an error status nor a BusOff status are present. This event is available only if method SendFrame is enabled.

Prototype

```
void OnTransmitWarning(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.

4.4 OnReceiveWarning

This event is called when the CAN controller goes into a warning status due to the receive error counter exceeding 96 and neither an error status nor a BusOff status are present. The event is available only if Interrupt service/event is enabled.

Prototype

```
void OnReceiveWarning(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.

4.5 OnBusOff

This event is called when the node status becomes bus-off. The event is available only if Interrupt service/event is enabled.

Prototype

```
void OnBusOff(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.

4.6 OnWakeUp

This event is called when a wakeup is detected. To enable this event, property Wakeup has to be enabled first.

Prototype

```
void OnWakeUp(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.

4.7 OnError

This event is called when a channel error (not the error returned by a given method) occurs.

Prototype

```
void OnError(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer is passed as the parameter of Init method.

5 Types and Constants

This section contains definitions of user types and constants. User types are derived from basic types and they are designed for usage in the driver interface. They are declared in the generated code.

Type Definitions

- **LDD_TPinMask** : user definition

  Bit mask of pins that need to be connected

- **LDD_CAN_TMBIndex** : user definition

  Type specifying the message buffer index variable.

- **LDD_CAN_TAccMask** : user definition

  Type specifying the acceptance mask variable.

- **LDD_CAN_TMessageID** : user definition

  Type specifying the ID mask variable.

- **LDD_CAN_TErrorCounter** : user definition

  Type specifying the error counter variable.

- **LDD_CAN_TErrorMask** : user definition

Error mask type.

- **LDD_CAN_TModuleMode** : user definition

  Type specifying the CAN module mode.

- **LDD_CAN_TIDType** = enum { CAN_LDD_STANDARD_ID, CAN_LDD_EXTENDED_ID }

- **LDD_CAN_TFrame** = struct { CAN frame buffer descriptor.

  LDD_CAN_TMessageID MessageID; – *Message ID*

  LDD_CAN_TFrameType FrameType; – *Type of the frame LDD_CAN_DATA_FRAME_STD/ LDD_CAN_DATA_FRAME_EXT/LDD_CAN_REMOTE_FRAME/LDD_CAN_RESPONSE_FRAME*

  uint8_t* Data; – *Message data buffer*

  uint8_t Length; – *Message length*

  uint16_t TimeStamp; – *Message time stamp*

  uint8_t LocPriority; – *Local Priority Tx Buffer*

  }

- **LDD_CAN_TStats** = struct { Device receive/transmit/Error statistics.

  uint32_t TxFrames; – *Transmitted frame counter*

  uint32_t TxWarnings; – *Transmission warning counter*

  uint32_t RxFrames; – *Received frame counter*

  uint32_t RxWarnings; – *Reception warning counter*

  uint32_t BusOffs; – *Bus off counter*

  uint32_t Wakeups; – *Wakeup counter*

  uint32_t Bit0Errors; – *Bit0 error counter*

  uint32_t Bit1Errors; – *Bit1 error counter*

  uint32_t AckErrors; – *ACK error counter*

  uint32_t CrcErrors; – *CRC error counter*

  uint32_t FormErrors; – *Message form error counter*

  uint32_t BitStuffErrors; – *Bit stuff error counter*

  uint32_t Errors; – *Error counter*

  }

- **LDD_CAN_TFrameType** = enum { LDD_CAN_DATA_FRAME, LDD_CAN_REMOTE_FRAME, LDD_CAN_RESPONSE_FRAME } Type specifying the CAN frame type.

- **LDD_CAN_TIDHitFilterIndex** : user definition

Identifier acceptance filter hit indicator index type.

- **LDD_CAN_IDAcceptanceFilterMode** = enum { LDD_CAN_ONE_32BIT_FILTER, LDD_CAN_TWO_16BIT_FILTERS, LDD_CAN_FOUR_8BIT_FILTERS, LDD_CAN_FILTER_CLOSED }

- **LDD_CAN_TElementIndex** : user definition

  Type specifying the filter table mask element index

- **LDD_CAN_TAccCode** : user definition

  Type specifying the acceptance code variable.

- **LDD_CAN_TBufferMask** : user definition

  Message buffer mask type

- **LDD_CAN_TRxBufferState** = enum { LDD_CAN_MB_RX_NOT_ACTIVE, LDD_CAN_MB_RX_FULL, LDD_CAN_MB_RX_EMPTY, LDD_CAN_MB_RX_OVERRUN, LDD_CAN_MB_RX_BUSY, LDD_CAN_MB_RX_RANSWER } Type specifying the Rx message buffer state.

- **LDD_CAN_TRunModeMask** : user definition

  Type specifying the run mode variable.

Constants

- **LDD_CAN_ON_FULL_RXBUFFER** - OnFullRxBuffer event mask.
- **LDD_CAN_ON_FREE_TXBUFFER** - OnFreeTxBuffer event mask.
- **LDD_CAN_ON_BUSOFF** - OnBussOff event mask.
- **LDD_CAN_ON_TXWARNING** - OnTxWarning event mask.
- **LDD_CAN_ON_RXWARNING** - OnRxWarning event mask.
- **LDD_CAN_ON_ERROR** - OnError event mask.
- **LDD_CAN_ON_WAKEUP** - WakeUp event mask.
- **LDD_CAN_MESSAGE_ID_EXT** - Mask value specifying extended ID.

- **LDD_CAN_BIT0_ERROR** - Bit0 error detect mask
- **LDD_CAN_BIT1_ERROR** - Bit1 error detect mask
- **LDD_CAN_ACK_ERROR** - Acknowledge error detect mask
- **LDD_CAN_CRC_ERROR** - Cyclic redundancy check error detect mask
- **LDD_CAN_FORM_ERROR** - Message form error detect mask
- **LDD_CAN_STUFFING_ERROR** - Bit stuff error detect mask
- **LDD_CAN_RX_OVERRUN_ERROR** - Receiver overrun detect error mask
- **LDD_CAN_RX_PIN** - Receiver pin mask
- **LDD_CAN_TX_PIN** - Transmitter pin mask
- **LDD_CAN_RUN_MODE** - Run mode mask
- **LDD_CAN_FREEZE_MODE** - Freeze (initialization) mode mask
- **LDD_CAN_SLEEP_MODE** - Sleep mode mask

6 Typical usage

This section contains examples of a typical usage of the component in user code. For general information please see the section Component Code Typical Usage in Processor Expert user manual.

Examples of typical settings and usage of CAN_LDD component

1. Sending data frame with interrupt service
2. Receiving data frame with interrupt service
3. Sending data frame without interrupt service (polling)
4. Receiving data frame without interrupt service (polling)

Sending data frame with interrupt service

The following example demonstrates sending a data frame with standard ID and extended ID.

OnFreeTxBuffer event is called when a frame is successfully transmitted and returns buffer index (in this case 0), which was send.

Required component setup :

- *Interrupt service* : Enabled
- *Message buffers* : 1, Buffer0 - Buffer type: Transmit

- *Bit rate* : 100kbit/s
- *Loop mode* : no
- *Enabled in init. code* : yes
- Methods to enable : SendFrame
- Events to enable : OnFreeTxBuffer

Content of ProcessorExpert.c:

```
volatile bool DataFrameTxFlg;
LDD_TDeviceData *MyCANPtr;
LDD_TError Error;
LDD_CAN_TFrame Frame;
uint8_t OutData[4] = {0x00U, 0x01U, 0x02U, 0x03U};                /* Initialization of output data
buffer */

void main(void)
{
  . . .
  MyCANPtr = CAN2_Init(NULL);                                     /* Initialization of CAN2 component
*/

  Frame.MessageID = 0x123U;                                       /* Set Tx ID value - standard */
  Frame.FrameType = LDD_CAN_DATA_FRAME;                           /* Specyfying type of Tx frame - Data
frame */
  Frame.Length = sizeof(OutData);                                 /* Set number of bytes in data frame -
4B */
  Frame.Data = OutData;                                           /* Set pointer to OutData buffer */
  DataFrameTxFlg = FALSE;                                         /* Initialization of DataFrameTxFlg */
  Error = CAN2_SendFrame(MyCANPtr, 0U, &Frame);                   /* Sends the data frame over buffer 0
*/
  while (!DataFrameTxFlg) {                                       /* Wait until data frame is
transmitted */
  }
  . . .
  Frame.MessageID = (0x123456U | LDD_CAN_MESSAGE_ID_EXT);         /* Set Tx ID value - extended */
  Frame.FrameType = LDD_CAN_DATA_FRAME;                           /* Specyfying type of Tx frame - Data
frame */
  Frame.Length = sizeof(OutData);                                 /* Set number of bytes in data frame -
4B */
  Frame.Data = OutData;                                           /* Set pointer to OutData buffer */
  DataFrameTxFlg = FALSE;                                         /* Clear DataFrameTxFlg */
  Error = CAN2_SendFrame(MyCANPtr, 0U, &Frame);                   /* Sends the data frame over buffer 0
*/
  while (!DataFrameTxFlg) {                                       /* Wait until data frame is
transmitted */
  }
  . . .
  for(;;) {}
}
```

Content of Event.c:

```
extern volatile bool DataFrameTxFlg;
void CAN2_OnFreeTxBuffer(LDD_TUserData *UserDataPtr, LDD_CAN_TMBIndex BufferIdx)
{
  DataFrameTxFlg = TRUE; /* Set DataFrameTxFlg flag */
}
```

Receiving data frame with interrupt service

The following example demonstrates receiving a frame with standard ID (Initialization over component) and extended ID, what is sets in run time.

OnFullRxBuffer event is called when a frame is successfully received and returns buffer index (in this case 0), where are data stored.

Typical usage

Required component setup :

- *Interrupt service* : Enabled
- *Global Acceptance Mask* : yes
- *Acceptance mask for buffer 0..n* : 1FFFFFFF
- *Message buffers* : 1, Buffer0 - Buffer type: Receive
- *Accept frames* : Standard
- *Message ID* : 7FF
- *Bit rate* : 100kbit/s
- *Loop mode* : no
- *Enabled in init. code* : yes
- Methods to enable : ReadFrame,SetRxBufferID
- Events to enable : OnFullRxBuffer

Content of ProcessorExpert.c:

```
volatile bool DataFrameRxFlg = FALSE;
LDD_TDeviceData *MyCANPtr;
LDD_TError Error;
LDD_CAN_TFrame Frame;
uint8_t InpData[8];

void main(void)
{
  . . .
  MyCANPtr = CAN2_Init(NULL);                                      /* Initialization of CAN2 component
*/

  while (!DataFrameRxFlg) {                                        /* Wait until data frame is received */
  }
  Frame.Data = InpData;                                            /* Set pointer to InpData buffer */
  Error = CAN2_ReadFrame(MyCANPtr, 0U, &Frame);                    /* Reads a data frame from buffer 0
and fills Frame structure */
  /*
    Frame.MessageID => Contains ID value. if((Frame.MessageID & LDD_CAN_MESSAGE_ID_EXT)!=0) then
extended ID, else standard ID
    Frame.FrameType => Type of Rx frame, e.g. LDD_CAN_DATA_FRAME
    Frame.Length => Number of Rx bytes in Rx frame
    InpData[]    => Contains Rx data bytes
  */
  . . .
  DataFrameRxFlg = FALSE;                                          /* Clear DataFrameRxFlg */
  TError = CAN2_SetRxBufferID(MyCANPtr, 0U, (0x123456U|LDD_CAN_MESSAGE_ID_EXT)); /* Set new value of the
Rx ID for buffer 0 - extended type */
  while (!DataFrameRxFlg) {                                        /* Wait until data frame is received */
  }
  Frame.Data = InpData;                                            /* Set pointer to InpData buffer */
  Error = CAN2_ReadFrame(MyCANPtr, 0U, &Frame);                    /* Reads a data frame from buffer 0
and fills Frame structure */
  /*
    Frame.MessageID => Contains ID value. if((Frame.MessageID & LDD_CAN_MESSAGE_ID_EXT)!=0) then
extended ID, else standard ID
    Frame.FrameType => Type of Rx frame, e.g. LDD_CAN_DATA_FRAME
    Frame.Length => Number of Rx bytes in Rx frame
    InpData[]    => Contains Rx data bytes
  */
  . . .
  for(;;) {}
}
```

Content of Event.c:

```
extern volatile bool DataFrameRxFlg;
```

```
void CAN2_OnFullRxBuffer(LDD_TUserData *UserDataPtr, LDD_CAN_TMBIndex BufferIdx)
{
  DataFrameRxFlg = TRUE; /* Set DataFrameRxFlg flag */
}
```

Sending data frame without interrupt service (polling)

The following example demonstrates sending a data frame with standard ID and extended ID.

OnFreeTxBuffer event is called when a frame is successfully transmitted and returns buffer index (in this case 0), which was send.

Required component setup :

- *Interrupt service* : Disabled
- *Message buffers* : 1, Buffer0 - Buffer type: Transmit
- *Bit rate* : 100kbit/s
- *Loop mode* : no
- *Enabled in init. code* : yes
- Methods to enable : SendFrame
- Events to enable : OnFreeTxBuffer

Content of ProcessorExpert.c:

```
volatile bool DataFrameTxFlg;
LDD_TDeviceData *MyCANPtr;
LDD_TError Error;
LDD_CAN_TFrame Frame;
uint8_t OutData[4] = {0x00U, 0x01U, 0x02U, 0x03U};                /* Initialization of output data
buffer */

void main(void)
{
  . . .
  MyCANPtr = CAN2_Init(NULL);                                     /* Initialization of CAN2 component
*/

  Frame.MessageID = 0x123U;                                       /* Set Tx ID value - standard */
  Frame.FrameType = LDD_CAN_DATA_FRAME;                           /* Specyfying type of Tx frame - Data
frame */
  Frame.Length = sizeof(OutData);                                 /* Set number of bytes in data frame -
4B */
  Frame.Data = OutData;                                           /* Set pointer to OutData buffer */
  DataFrameTxFlg = FALSE;                                         /* Initialization of DataFrameTxFlg */
  Error = CAN2_SendFrame(MyCANPtr, 0U, &Frame);                   /* Sends the data frame over buffer 0
*/
  while (!DataFrameTxFlg) {                                       /* Wait until data frame is
transmitted */
    CAN2_Main(MyCANPtr);
  }
  . . .
  Frame.MessageID = (0x123456U | LDD_CAN_MESSAGE_ID_EXT);         /* Set Tx ID value - extended */
  Frame.FrameType = LDD_CAN_DATA_FRAME;                           /* Specyfying type of Tx frame - Data
frame */
  Frame.Length = sizeof(OutData);                                 /* Set number of bytes in data frame -
4B */
  Frame.Data = OutData;                                           /* Set pointer to OutData buffer */
  DataFrameTxFlg = FALSE;                                         /* Clear DataFrameTxFlg */
  Error = CAN2_SendFrame(MyCANPtr, 0U, &Frame);                   /* Sends the data frame over buffer 0
*/
  while (!DataFrameTxFlg) {                                       /* Wait until data frame is
transmitted */
    CAN2_Main(MyCANPtr);
  }
  . . .
  for(;;) {}
}
```

Content of Event.c:

```
extern volatile bool DataFrameTxFlg;
void CAN2_OnFreeTxBuffer(LDD_TUserData *UserDataPtr, LDD_CAN_TMBIndex BufferIdx)
{
  DataFrameTxFlg = TRUE; /* Set DataFrameTxFlg flag */
}
```

Receiving data frame without interrupt service (polling)

The following example demonstrates receiving a frame with standard ID (Initialization over component) and extended ID, what is sets in run time.

OnFullRxBuffer event is called when a frame is successfully received and returns buffer index (in this case 0), where are data stored.

Required component setup :

- *Interrupt service* : Disabled
- *Global Acceptance Mask* : yes
- *Acceptance mask for buffer 0..n* : 1FFFFFFF
- *Message buffers* : 1, Buffer0 - Buffer type: Receive
- *Accept frames* : Standard
- *Message ID* : 7FF
- *Bit rate* : 100kbit/s
- *Loop mode* : no
- *Enabled in init. code* : yes
- Methods to enable : ReadFrame,SetRxBufferID
- Events to enable : OnFullRxBuffer

Content of ProcessorExpert.c:

```
volatile bool DataFrameRxFlg = FALSE;
LDD_TDeviceData *MyCANPtr;
LDD_TError Error;
LDD_CAN_TFrame Frame;
uint8_t InpData[8];

void main(void)
{
  . . .
  MyCANPtr = CAN2_Init(NULL);                                   /* Initialization of CAN2 component
*/

  while (!DataFrameRxFlg) {                                     /* Wait until data frame is received */
    CAN2_Main(MyCANPtr);
  }
  Frame.Data = InpData;                                         /* Set pointer to InpData buffer */
  Error = CAN2_ReadFrame(MyCANPtr, 0U, &Frame);                 /* Reads a data frame from buffer 0
and fills Frame structure */
  /*
    Frame.MessageID => Contains ID value. if((Frame.MessageID & LDD_CAN_MESSAGE_ID_EXT)!=0) then
extended ID, else standard ID
    Frame.FrameType => Type of Rx frame, e.g. LDD_CAN_DATA_FRAME
    Frame.Length => Number of Rx bytes in Rx frame
    InpData[]    => Contains Rx data bytes
  */
  . . .
  DataFrameRxFlg = FALSE;                                        /* Clear DataFrameRxFlg */
  Error = CAN2_SetRxBufferID(MyCANPtr, 0U, (0x123456U|LDD_CAN_MESSAGE_ID_EXT)); /* Set the new value of
the Rx ID for buffer 0 - extended type */
  while (!DataFrameRxFlg) {                                      /* Wait until data frame is received */
    CAN2_Main(MyCANPtr);
  }
  Frame.Data = InpData;                                          /* Set pointer to InpData buffer */
  Error = CAN2_ReadFrame(MyCANPtr, 0U, &Frame);                  /* Reads a data frame from buffer 0
and fills Frame structure */
  /*
    Frame.MessageID => Contains ID value. if((Frame.MessageID & LDD_CAN_MESSAGE_ID_EXT)!=0) then
extended ID, else standard ID
    Frame.FrameType => Type of Rx frame, e.g. LDD_CAN_DATA_FRAME
    Frame.Length => Number of Rx bytes in Rx frame
    InpData[]    => Contains Rx data bytes
  */
  . . .
  for(;;) {}
}
```

Content of Event.c:

```
extern volatile bool DataFrameRxFlg;
void CAN2_OnFullRxBuffer(LDD_TUserData *UserDataPtr, LDD_CAN_TMBIndex BufferIdx)
{
  DataFrameRxFlg = TRUE; /* Set DataFrameRxFlg flag */
}
```

Document Number N/A
Revision 1, 12/2013