# How to Use CXPI Controller in Traveo II Family

## About this document

### Scope and purpose

This application note describes how to use the clock extension peripheral interface (CXPI) controller in Traveo™ II family MCU. The CXPI controller of Traveo II supports autonomous transfer of the CXPI frame to reduce CPU processing.

### Associated Part Family

Traveo II Family CYT2/CYT3/CYT4 Series

### Intended audience

This document is intended for anyone who uses the CXPI controller of the Traveo II family.

## Table of contents

# 1 Introduction

This application note describes how to use the CXPI controller in Traveo II family MCU.

To understand the described functionality and the terminologies used in this application note, see the “Clock extension peripheral interface (CXPI)” chapter of the **Architecture Technical Reference Manual (TRM)**.

This document is applicable to CYT2/CYT3/CYT4 series devices.

# 2 CXPI overview

## 2.1 CXPI network

CXPI protocol provides a low-speed, low-cost, and lightweight connection in automotive controls of simple devices like wipers, sensors, or switches. As an example, in **Figure 1**, the MCU with a CXPI controller would be the CXPI master node, whereas the devices attached to the CXPI network would be the CXPI slave nodes. The CXPI controller can control the devices, and get the status and confirmation from devices via the CXPI communication bus.

The CXPI protocol provides a better performance in communication than LIN protocol. This is because it can handle multiplexing between multiple devices in a more efficient manner by making the arbitration decision at the lower layer (hardware) rather than having higher layer (software) assistance.

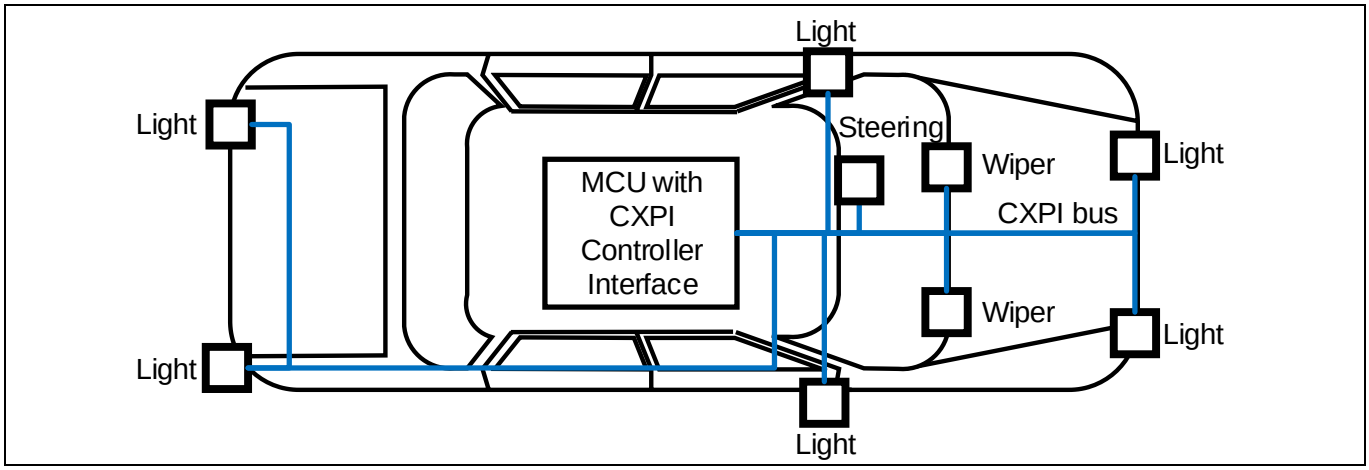


**Figure 1 Example of CXPI network in a vehicle**

**Table 1 CXPI protocol feature**

| Term | Description |
|---|---|
| Network layout | • Single master node and multiple slave nodes<br>• Data with clock is transmitted and received on a single communication bus<br>• Up to 16 nodes connected to a CXPI communication bus |
| Communication baud rate | Maximum baud rate of 20 kbps. |
| Network protocol | Collision Resolution supported Carrier Sense Multiple Access (CSMA/CR) |
| Bus access method | • Event Trigger method to support the responsiveness of slave node communication<br>• Polling method to support periodic schedules |

## 2.2 CXPI bus access method

The CXPI protocol is based on a single master and multiple slave communication systems and supports two methodologies about how and when data is transferred: event trigger method and polling method. In both communication methods, only the master node provides the clock to the communication bus; all slave nodes connected to the bus receive the clock from the communication bus and use it for communication processing. Only one of the two methods can be implemented in all nodes connected to the CXPI communication bus:

- Event trigger method: Each node can freely issue the “PID” field if idle state of the communication bus is detected. If several nodes transmit the PID field at the same time and the PID field collides on the communication bus, and non-destructive arbitration is performed, the highest-priority PID field is transmitted to the communication bus.
- Polling method: The master node can freely issue a “PID” field to request a response from slave nodes. The slave node can issue the “PID” field only after receiving PTYPE sent from the master node.

The difference between these methods is that the polling method requires the master node to control the network communication by issuing PTYPE field request, which allows all nodes to issue an event-driven PID field on the communication bus. In conclusion, polling method is suitable for a network which requires high-periodicity communications while event trigger method is suitable for networks that requires high responsivity to events.

In this application note, a message frame refers to the entire transmitted frame (including PID/FrameInfo/Data/CRC bytes in one message frame). The byte frame refers to the byte (in terms of PID byte, FrameInfo byte, data byte, and CRC byte individually).

## 2.3 Message frame format

The basic CXPI frame has normal and long frames. Due to two different bus communication methods, the basic CXPI gets an extension by an additional request field (event), usually described as the PTYPE field.

**Figure 2** illustrates the message frame format based on byte fields, each with a START and STOP bit and the least significant bit (LSb) first. The Inter Byte Space (IBS) defines the idle time between two bytes within a message frame. The Inter Frame Space (IFS) defines the period (minimum 20 Tbit) before a next frame can be transmitted by any node.

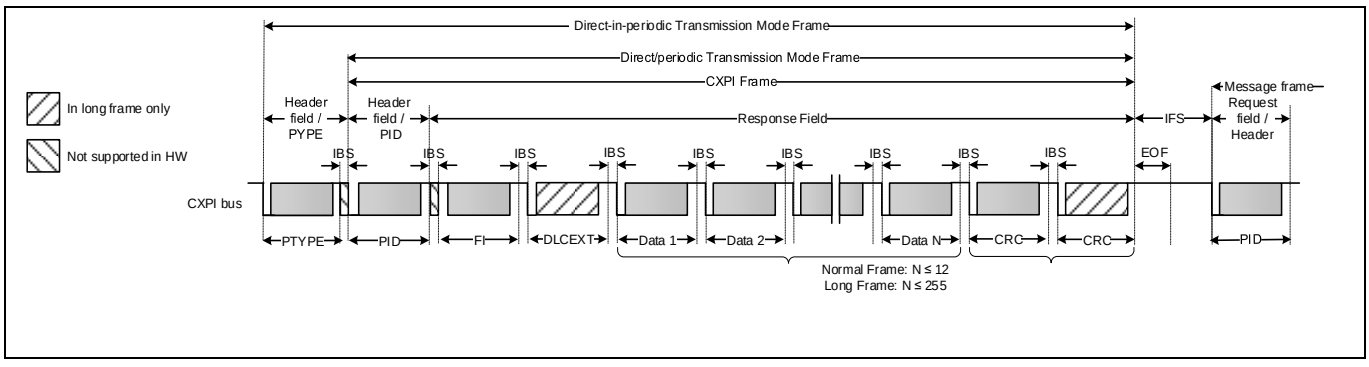


**Figure 2 CXPI message frame format**

The fields of the CXPI message frame format includes a PTYPE field (only in polling method), a PID field, a frame information field, a Data Length Code Extension (DLXECT, only for Long frame), a data field, and a CRC field.

### PTYPE field (only in polling method)

The 8-bit Protected Type field (PTYPE), only applicable in the polling method, corresponds to a PID field with the identifier value 0x00 (0x80 including parity bit). The master node sends a PTYPE byte to allow all slave nodes to send a request field for this time slot.

### Protected identifier (PID) field

The request field (header) consists of an 8-bit PID field, which contains a 7-bit frame identifier and a 1-bit odd parity over the frame identifier.

### Frame information (FI) field

As the first byte field of the response, the FI field provides information on Data Length Code (DLC), Network Management (NM), and a Frame Counter (CT).

### Data length code extension (only for long frame)

A long frame can have up to 255 data bytes. In this case, the DLC of the FI field must be set to 15. The DLCEXT field is present to indicate the number of data bytes in the message frames.

### Data field

The Data field can be transmitted by every node. In the normal frame, it is present when DLC > 0, and can be a maximum of 12 bytes long; in the long frame, the Data field is present when DLC_EXT > 0 and the maximum length is 255 bytes.

### Cyclic redundancy check (CRC) field

The end of a message frame consists of a CRC field and is accessible by the CRC register. The CRC length differs for normal and long frames. For the normal frame, an 8-bit CRC polynomial is computed over the PID, FI, and Data fields. For a long frame, a 16-bit CRC polynomial is calculated over the PID, FI, DLCEXT, and Data fields. The PTYPE field is not included in the CRC calculation.

# 3 CXPI controller in Traveo II

CXPI channels are part of a common CXPI module. Each channel has its own control, status registers, and interrupts. The total number of available CXPI channels depends on the device variant. For details, see the device **datasheet**.

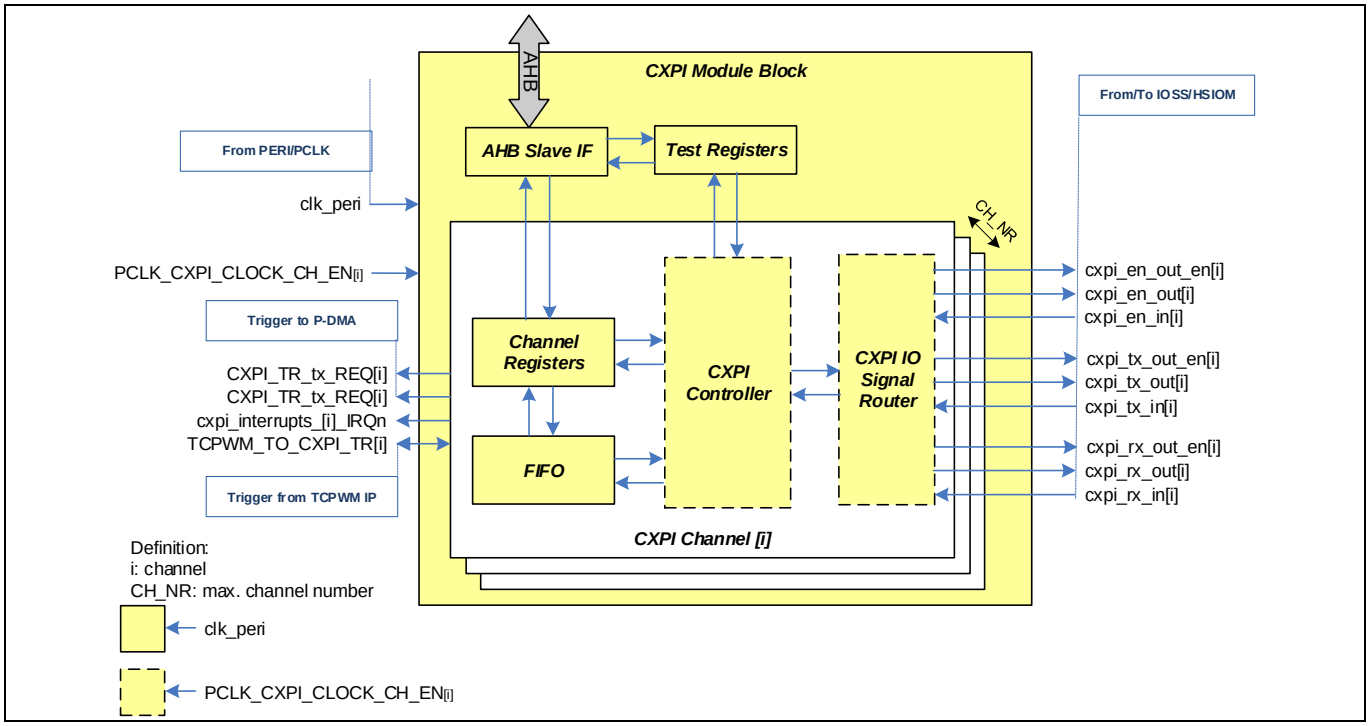


**Figure 3 CXPI block diagram**

## 3.1 Mode of operation

The CXPI controller in the Traveo II family MCU supports the CXPI communication protocol in two operation modes: NRZ mode and PWM mode.

NRZ mode: The CXPI controller interfaces an external transceiver chip that has PWM encoder/decoder logic.

PWM mode: The CXPI controller interfaces an external driver/receiver chip that level shifts the 3.3 V or 5 V signaling to signaling at CXPI bus voltage without changing the encoding of the signal.

**Master node**

- NRZ mode: When the CXPI channel is the master node (CTL0.MASTER = 1) and the CXPI transceiver performs the PWM bus signal encoding, the channel generates only the NRZ signal. Because the channel does not provide the CXPI clock signal, the clock must be generated by another module (for example, TCPWM) separately.
- PWM mode: The PWM mode must be selected for the CXPI channel to process PWM signals. The PWM encoding and decoding is done in the CXPI channel. Therefore, additional devices are not required to generate the clock on the CXPI bus.

**Slave node**

- NRZ mode: When the CXPI channel is a slave node (CTL0.MASTER = 0) and the CXPI transceiver performs the PWM bus signal encoding/decoding, the module must process NRZ signals.
- PWM mode: To process PWM signals directly, the CXPI module must be configured in the PWM mode.

## 3.2 Baud rate and sampling

### 3.2.1 Baud rate

The CXPI channel uses a fixed oversampling of 400. This means that the CXPI channel clock's frequency is 400 times the required CXPI interface frequency, i.e., baud rate of the CXPI channel. For details of sampling concept, see the "Clock Extension Peripheral Interface (CXPI)" chapter of the **Architecture TRM**.

The baud rate can be individually configured for each channel. The relationship between the target baud rate and the clock divider is shown in **Equation 1**.

$$CLK\_DIV = \frac{f_{clk\_peri}}{400 \times Baud\ rate} \quad \text{Equation 1}$$

Here,

- $f_{clk\_peri}$: Peripheral clock shown as PCLK_CXPI_CLOCK_CH_EN in Figure 3
- $Baud\ rate$: Target baud rate
- $CLK\_DIV$: Peripheral clock divider for dedicated CXPI channel

To achieve the target baud rate with the permitted relative tolerance of the nominal CXPI bit time, apply a fractional clock divider.

**Example:**

**Equation 2** shows an example of the divider setting value when the peripheral clock is 80 MHz and the target baud rate is 19.2 kbps (19.2 kHz).

$$CLK\_DIV = \frac{f_{clk\_peri}}{400 \times Baud\ rate} = \frac{80MHz}{400 \times 19.2kHz} = 10.41 \quad \text{Equation 2}$$

To have the nearest divider value, choose a 16.5-bit divider with an integer divide value of 10 and a fractional divider of 13, which has a divider value of

$$CLK\_DIV = 10 + \frac{13}{32} = 10.40625 \quad \text{Equation 3}$$

and generates

$$Baud\ rate = \frac{80MHz}{400 \times 10.40625} = 19.22kHz \quad \text{Equation 4}$$

Applying the fractional divider results in a relative bit time tolerance of 0.1% while applying an integer divider of 10 or 11 results in relative bit time tolerance of 4.2% and 5.3% respectively.

For details on the clock divider settings and the clock tree, see the Clock System chapter in the **Architecture TRM**.

### 3.2.2 Sampling

When transmitting, CXPI channels provide two registers, CTL1.T_LOW1 and CTL1.T_LOW0, to configure the low count of logic '1' and '0' respectively. CTL1.T_LOW1 and CTL1.T_LOW0 indicate the number of clocks per CXPI channel to drive a '0' at the CXPI bus before releasing it to indicate a logical '1' and a logical '0' respectively.

When receiving, the CXPI channel starts counting after the detection of the falling edges on the Rx signal and the register CTL1.T_OFFSET indicates the value of offset that is used for sampling the Rx signal.

As described before, the CXPI channel uses a fixed oversampling of 400, meaning 1 Tbit is equivalent to 400 samplings. To achieve the target width of transmission low level when outputting logical value '1' and '0' and the sampling point of the Rx signal, set the value of the registers CTL1.T_LOW1, CTL1.T_LOW0, and CTL1.T_OFFSET as shown in **Equation 5**, **Equation 6**, and **Equation 7**.

$$t_{low\_1} = \frac{\text{CTL1.T\_LOW1} + 1}{400} T_{bit} \quad \text{Equation 5}$$

$$t_{low\_0} = \frac{\text{CTL1.T\_LOW0} + 1}{400} T_{bit} \quad \text{Equation 6}$$

$$t_{rx\_sampling} = \frac{\text{CTL1.T\_OFFSET} + 1}{400} T_{bit} \quad \text{Equation 7}$$

Here,

- $t_{low\_1}$: Width of transmission low level when outputting logical value '1'
- $t_{low\_0}$: Width of transmission low level when outputting logical value '0'
- $t_{rx\_sampling}$: "cxpi_rx_in" sampling point

**Example:**

**Table 2** shows the sample values set for the registers CTL1.T_LOW1, CTL1.T_LOW0, and CTL1.T_OFFSET to satisfy the timing parameters shown in **Table 3**.

**Table 2 Timing parameters**

| Item | Value |
|---|---|
| $t_{low\_1}$ | $0.258T_{bit}$ |
| $t_{low\_0}$ | $0.445T_{bit}$ |
| $t_{rx\_sampling}$ | $t_{low\_1} + 0.04\text{T}_{\text{bit}}$ |

**Table 3 Values of CTL1.T_LOW1, CTL1.T_LOW0, and CTL1.T_OFFSET**

| Item | Value |
|---|---|
| CTL1.T_LOW1 | 102 |
| CTL1.T_LOW0 | 177 |
| CTL1.T_OFFSET | 118 |

## 3.3 CXPI message transmission commands and interrupt events

### 3.3.1 Message transfer

The CXPI controller supports different message types such as transmission and reception of header/response. Message transfer processing is done by command sequences. Every command is listed in the CMD register.

**Commands supporting header field transmission/reception:**

- CMD.RX_HEADER (C.RXH): Enables PTYPE and normal PID field reception
- CMD.TX_HEADER (C.TXH): Requests PTYPE and normal PID field transmission

**Commands supporting response field transmission/reception:**

- CMD.RX_RESPONSE (C.RXR): Enables response field reception
- CMD.TX_RESPONSE (C.TXR): Requests response field transmission

**Command supporting IFS check:**

- CMD.IFS_WAIT (C.IFS): Directs HW to check bus idleness before transmitting or receiving the header

The use case for the combination of commands to transmit/receive message frames is shown in **Table 4** with abbreviation of commands shown along with these commands.

**Table 4 Combination of commands [1]**

| Transfer method | Master/Slave configuration | No. | Transaction | Setting for {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} | Description |
|---|---|---|---|---|---|
| Event trigger | Master/slave | 1 | Transmit both header (PID) and response | {1, 1, 1, 1, 1} | Master/slave in event trigger method uses this combination to transmit both header (PID) and response.<br>In this case, HW will check for IFS before transmitting the header and response.<br>CMD.RX_HEADER and CMD.RX_RESPONSE are set to '1' to anticipate receiving the PID and response while checking the duration of bus idleness and receiving the response if arbitration is lost. |
| | | 2 | Transmit header (PID) and receive response | {1, 1, 1, 0, 1} | Master/slave in event trigger method uses this combination to transmit the header (PID).<br>CMD.RX_HEADER is set to '1' to anticipate receiving the PID while checking the duration of bus idleness. |

[1] See the "Clock Extension Peripheral Interface (CXPI)" chapter of the **Architecture TRM** for details on the priority of these commands.

| Transfer method | Master/Slave configuration | No. | Transaction | Setting for {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} | Description |
|---|---|---|---|---|---|
| | | 3 | Receive both header (PID) and response | {0, 0, 1, 0, 1} | Enables header and response reception immediately. |
| | | 4 | | {1, 0, 1, 0, 1} | Setting CMD.IFS_WAIT = '1' can make HW wait for CLT0.IFS before enabling the header and response reception. |
| | | 5 | Transmit response only | {0, 0, 0, 0, 1} | Master/slave in event trigger method uses this combination to transmit the response after receiving then relevant PID. |
| Polling | Master | 6 | Transmit PTYPE and receiving header (PID) and response | {1, 1, 1, 0, 1} | Master in polling method uses this combination to transmit PTYPE and then receive the PID and response. |
| | | 7 | Transmit both header (PID) and response | {1 or 0, 1, 0, 1, 1} | Master in polling method uses this combination to transmit both header (PID) and response.<br>Setting CMD.IFS = '1' makes sure that IFS is checked before transmitting the header; however, this is not mandatory.<br>CMD.IFS is set to '0' if the master transmits the PID after transmitting PTYPE. |
| | | 8 | Transmit header (PID) and receiving response | {1 or 0, 1, 0, 0, 1} | Master in polling method uses this combination to transmit the PID and receive response.<br>CMD.IFS is set to '0' if the master transmits the PID after transmitting PTYPE. |
| | | 9 | Transmit response only | {0, 0, 0, 0, 1} | Master in polling method uses this combination to transmit the response. |
| | Slave (ensure CTL0.RXPIDZERO_CHECK_EN=1) [2] | 10 | Transmit header (PID) and response | {0, 1, 0, 1, 1} | Slave in polling method uses this combination to transmit both PID and response.<br>CMD.RX_RESPONSE is set to '1' to anticipate the receiving response after arbitration is lost. |

[2] CTL0.RXPIDZERO_CHECK_EN = '1' makes HW (slave) does not clear CMD.RX_HEADER and HW is able to continue receiving header coming after PTYPE.

| Transfer method | Master/Slave configuration | No. | Transaction | Setting for {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} | Description |
|---|---|---|---|---|---|
| | | 11 | Receive header (PTYPE and PID) and response | {0, 0, 1, 0, 1} | Slave in polling method uses this combination to receive PTYPE/PID and response. |
| | | 12 | Transmit response only | {0, 0, 0, 0, 1} | Slave in polling method uses this combination to transmit the response. |

### 3.3.2 Interrupt events

Each CXPI channel has a dedicated set of interrupt registers: INTR, INTR_SET, INTR_MASK, and INTR_MASKED. In general, INTR registers are the interrupt event logging registers that are set by HW and cleared by SW (as part of the interrupt service handler).

Software can set the INTR_SET register for testing purpose. Writing '1' into a bit of this register will set the corresponding bit of INTR register.

The INTR_MASK register is used to mask interrupt sources. Only the interrupt sources with their masks enabled, meaning the corresponding bit in the INTR_MASK register is set to '1', can trigger the interrupt.

The INTR_MASKED register reflects the bit-wise logical AND of INTR and INTR_MASK. This register allows SW to read the status of all mask-enabled interrupt causes with a single load operation.

These interrupt causes are grouped as either functional interrupts or error reporting interrupts. The following are the functional interrupt causes:

**Table 5 Interrupt cause and description**

| Interrupt cause | Description |
|---|---|
| TX_HEADER_DONE | Transmission of frame header is completed. |
| TX_RESPONSE_DONE | Transmission of frame response is completed. |
| TX_WAKEUP_DONE | Transmission of wake up is done. |
| TX_FIFO_TRIGGER | TX FIFO's number of used slot is less than TRIGGER_LEVEL. |
| RX_HEADER_DONE | Reception of frame header is completed and will be set only if the reception of response is completed. |
| RX_HEADER_PID_DONE | This bit is set when the reception of the frame header is completed without waiting for the completion of the reception of the response. |
| RX_RESPONSE_DONE | Reception of the frame response is completed. |
| RX_WAKEUP_DETECT | Falling edge of the RX pin in Sleep mode is detected. |
| RX_FIFO_TRIGGER | RX FIFO's number of used slot is greater than TRIGGER_LEVEL. |
| TXRX_COMPLETE | HW sets this field to '1', when the message frame ends after End of Frame (EOF) is confirmed and TX/RX_DATA_LENGTH_ERROR = 0. |
| TX_HEADER_ARB_LOST | HW sets this field to '1', when it detects arbitration is lost after the number of retries has exceeded the maximum allowed retries defined in CTL2.RETRY. |

| Interrupt cause | Description |
|---|---|
| TIMEOUT | HW sets this field to ‘1’, when the transmitted/received IBS within a message frame exceeds CTL2.TIMEOUT_LENGTH. |

**Table 6 Error-reporting interrupts and description**

| Error-reporting interrupts | Description |
|---|---|
| TX_BIT_ERROR | HW sets this field to ‘1’, when a transmitted “cxpi_tx_out” value does not match with a received “cxpi_rx_in” value. |
| RX_CRC_ERROR | HW sets this field to ‘1’, when the received CRC does not match with the CRC computed from header and response. |
| RX_HEADER_PARITY_ERROR | HW sets this field to ‘1’, when the received PID field or PType field has a parity error. |
| RX_DATA_LENGTH_ERROR | HW sets this field to ‘1’, when the received message frame’s data fields are not equal to the value specified in DLC (for normal frame) or DLCEXT (for long frame).<br>If the received message frame’s data fields are greater than the value specified in DLC, it may result in the RX_CRC_ERROR error before RX_DATA_LENGTH_ERROR. |
| TX_DATA_LENGTH_ERROR | HW sets this field to ‘1’, when the transmitted message frame’s data fields are not equal to the value specified in DLC (for normal frame) or DLCEXT (for long frame). |
| RX_OVERFLOW_ERROR | HW sets this field to ‘1’, when the RX data is overwritten by HW before the SW reads from it. |
| TX_OVERFLOW_ERROR | HW sets this field to ‘1’, when the TX data is overwritten by SW before the HW reads from it to transmit on to the CXPI bus. |
| RX_UNDERFLOW_ERROR | HW sets this field to ‘1’, when RX FIFO is empty and SW reads from it. |
| TX_UNDERFLOW_ERROR | HW sets this field to ‘1’, when TX FIFO is empty and HW reads from it. |
| RX_FRAME_ERROR | HW sets this field to ‘1’, when the stop bit of a byte frame is incorrect. |
| TX_FRAME_ERROR | HW sets this field to ‘1’, when the stop bit of a byte frame is incorrect. |

## 3.4 PID arbitration

An arbitration is done to avoid collisions between different nodes during PID field transmission. The arbitration loss is determined through a mismatch between the transmitted header and the received header.

The CXPI controller provides an automated retransmission feature, that is, the command for requesting header transmission (CMD.TX_HEADER) is not cleared after losing arbitration but retained as '1' to trigger the retransmission. The register field (CTL2.RETRY) predefines the maximum number of PID retransmissions, whereas STATUS.RETRIES_COUNT shows the number of retries. When the number of retransmissions exceeds CTL2.RETRY, the flag INTR.TX_HEADER_ARB_LOST is set.

The following scenario explains the operation of software and hardware at arbitration loss:

- Software sets {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'1', '1', '1', '1', '1'} to request both header and response transmission or sets {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'1', '1', '1', '0', '1'} to request header transmission and enable response reception.
- Hardware transmits header and it detects arbitration loss. Hardware stops transmitting and receives the "winning" header. Hardware notifies software that it has received PID by setting INTR.RX_HEADER_PID_DONE = '1'. If the number of header transmission does not exceed CTL2.RETRY, CMD.TX_HEADER is not cleared but kept as '1', else, hardware notifies software by setting INTR.TX_HEADER_ARB_LOST flag and clears the TX_HEADER and TX_RESPONSE command (if TX_RESPONSE is set).
  CMD.TX_RESPONSE = 1 has higher priority over CMD.RX_RESPONSE = 1 when arbitration is "won". However, if arbitration is "lost", CMD.RX_RESPONSE has higher priority over CMD.TX_RESPONSE.

*Note: There is a possibility that hardware receives a header while waiting for IFS. In this case, CMD.TX_HEADER is cleared by hardware and INTR.TX_HEADER_ARB_LOST is not set. Software can check these two bits to distinguish with arbitration lost case.*

- Software reads the received header and determines the next step.
  - If the received PID indicates that the node that lost arbitration should receive the response. CMD.RX_RESPONSE is set in advance, so software does nothing but waits for INTR.RX_RESPONSE_DONE flag. After that, if the arbitration lost count does not exceed the maximum number of retries defined by CTL2.RETRY, hardware will continue to transmit the previous header.
    The transmission happens after fulfilling IFS only if software sets IFS_WAIT.
  - If the received PID indicates that the node that lost arbitration should transmit the response, software needs to clear TX FIFO and CMD.TX_HEADER (to stop pending retry for arbitration lost). Software needs to prepare response in TX FIFO and set CMD.TX_RESPONSE = 1. In addition, software needs to clear CMD.RX_RESPONSE for hardware to service the CMD.TX_RESPONSE command. Hardware will transmit the response and notify software after it completes the transmission by setting INTR.TX_RESPONSE_DONE = 1.
    In this case, hardware will not attempt any retry for the previous header because software has already cancelled the transaction. Software needs to reprogram the previous message frame if it needs hardware to re-attempt. STATUS.RETRIES_COUNT is also reset when software clears CMD.TX_HEADER.

# 4 Example of CXPI controller operation

This section shows an example of software implementation for the CXPI controller in Traveo II family using Sample Driver Library (SDL). The code snippets in this application note are part of the SDL. See **7 Other references**.

SDL has a configuration part and a driver part. The configuration part mainly configures the parameter values for the desired operation. The driver part configures each register based on the parameter values in the configuration part.

## 4.1 CXPI setup

In this example, software handles PID re-transmission after arbitration is lost.

Additionally, this example will focus on using of command sequences to perform transmission and reception of the message frame. The use case is simplified so that there is no transmission or reception of the frame whose data length is larger than 16 bytes.

In a long frame, up to 255 data bytes are transferred/received. To avoid additional CPU access to the FIFO buffers, every CXPI channel is connected to the P-DMA with trigger signal lines for both FIFO buffers. See the “Clock Extension Peripheral Interface (CXPI)” chapter of the **Architecture TRM** for details.

(1) Initialize the clock: See **3.2.1 Baud rate** for example of clock divider setting and see the related chapter of the **Architecture TRM** for details on clock setting.

(2) Port settings: Enable and configure the I/O ports used for CXPI communication. Enable external CXPI transceiver or driver/receiver before starting the CXPI communication.

(3) System interrupt settings: Map the CXPI system interrupt source to the available external CPU interrupt.

(4) CXPI controller initialization: See **4.2 CXPI controller initialization** for an example of CXPI Controller initialization flow.

(5) Enable the CXPI channel: Configure CTL0.ENABLE to ‘1’ to enable the CXPI channel.

(6) Move the CXPI controller to normal mode. It is assumed that the CXPI bus is in active state (wakeup and sleep modes are not discussed in this application note). For details on the power modes of the CXPI controller, see the “Clock Extension Peripheral Interface (CXPI)” chapter of the **Architecture TRM**.

(7) Start message transmission/reception. See **4.3 Message frame transmission/reception** for an example of the CXPI message frame transmission/reception flow.

**Example of CXPI controller operation**

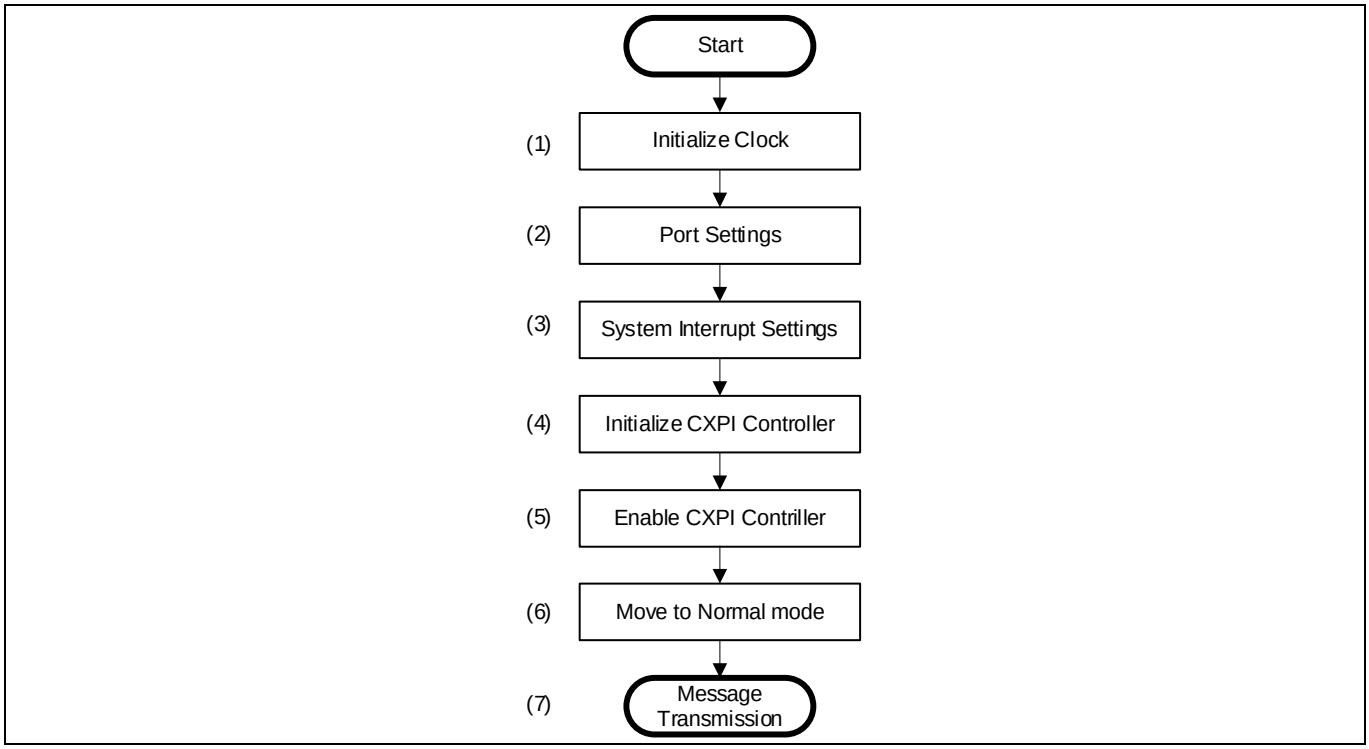


**Figure 4 Example of the CXPI setup flow**

### 4.1.1 Use case

- CXPI channel: 3
- Interrupt number: 2
- Interrupt priority: 1

### 4.1.2 Configuration

**Table 7 CXPI setup functions in SDL**

| Function | | Description | Value |
|---|---|---|---|
| Cy_CXPI_Enable() | | Enable the CXPI channel | – |
| | stc_CXPI_CH_t* | Specifies the CXPI channel instance | CXPI_CH_INSTANCE = CY_CXPICH3_TYPE |
| Cy_CXPI_Init() | | | |
| | stc_CXPI_CH_t* | Specifies the CXPI channel instance | CXPI_CH_INSTANCE = CY_CXPICH3_TYPE |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | &cxpiCtx |
| Cy_CXPI_SetMode() | | Changes the CXPI channel mode | – |
| | stc_CXPI_CH_t* | Specifies the CXPI channel instance | CXPI_CH_INSTANCE = CY_CXPICH3_TYPE |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | &cxpiCtx |
| | mode | Specifies the CXPI channel mode<br>CXPI_SLEEP_MODE: Sleep mode<br>CXPI_STANDBY_MODE: Standby mode<br>CXPI_NORMAL_MODE: Normal mode | CXPI_NORMAL_MODE |

## 4.1.3 Example code for CXPI setup

### Code Listing 1 Example of CXPI setup in configuration part

```
#define CXPI_CH_INSTANCE CY_CXPICH3_TYPE
:
int main(void)
{
    SystemInit();

    __enable_irq();

    /* Enable CM4. CY_CORTEX_M4_APPL_ADDR is calculated in linker script, check it in case of problems. */
    Cy_SysEnableApplCore(CY_CORTEX_M4_APPL_ADDR);

    /* Initialize CXPI clock */
    CxpiClockInit();

    /* Initialize CXPI port */
    CxpiPortInit();

    /* Initialize CXPI interrupt request */
    CxpiIrqInit();

    /* Initialize CXPI channel */
    Cy_CXPI_Init(CXPI_CH_INSTANCE, &stcCxpiConfig,&cxpiCtx);
    Cy_CXPI_Enable(CXPI_CH_INSTANCE);

    Cy_CXPI_SetMode(CXPI_CH_INSTANCE, &cxpiCtx, CXPI_STANDBY_MODE);
    Cy_CXPI_SetMode(CXPI_CH_INSTANCE, &cxpiCtx, CXPI_NORMAL_MODE);

    /* Initialize schedule */
    CxpiSchedulerInit();

    for(;;) {
    }
}
```

(1) Initialize Clock

(2) Port Settings

(3) System Interrupt Settings (See **Code Listing 2**)

(4) Initialize CXPI Controller (See **Code Listing 6**)

(5) Enable CXPI Channel (See **Code Listing 3**)

(6) Move to Normal mode (See **Code Listing 4**)

### Code Listing 2 Example of system interrupt setting in configuration part

```
static void CxpiIrqInit(void)
{
   /* Setup the IRQ */
   cy_stc_sysint_irq_t irq_cxpi_cfg = {
        .sysIntSrc = CXPI_CH_IRQ, .intIdx = CPUIntIdx2_IRQn, .isEnabled = true,
   };
   Cy_SysInt_InitIRQ(&irq_cxpi_cfg);
   Cy_SysInt_SetSystemIrqVector(irq_cxpi_cfg.sysIntSrc, CxpiInterruptHandler);
   NVIC_SetPriority(CPUIntIdx2_IRQn, 1);
   NVIC_ClearPendingIRQ(CPUIntIdx2_IRQn);
   NVIC_EnableIRQ(CPUIntIdx2_IRQn);
   :
}
```

System Interrupt Setting

### Example of CXPI controller operation

**Code Listing 3** demonstrates an example program of enabling CXPI channel in driver part.

The following description will help you understand the register notation of the driver part of SDL:

- pstcCXPI->un***CTL0***.stcField.u1***ENABLED*** is the CXPIx_CHy_CTL0.ENABLED register mentioned in the **Registers TRM**. Other registers are also described in the same manner. “x” signifies the CXPI instance and “y” signifies the channel number of CXPI instance.

See *cyip_cxpi.h* under *hdr/rev_x/ip* for more information on the union and structure representation of registers.

**Code Listing 3 Example of enabling the CXPI channel in driver part**

```
cy_en_cxpi_status_t Cy_CXPI_Enable(volatile stc_CXPI_CH_t* pstcCXPI)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
         return CY_CXPI_BAD_PARAM;
    }
    /** Enable the CXPI channel
    **/
    pstcCXPI->unCTL0.stcField.u1ENABLED = 1ul;
    // Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_RX_HEADER);
    // Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_ALL_ERROR_MASK);
    Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_WAKEUP_DETECT | CXPI_INTR_ALL_ERROR_MASK);
    return ret;
}
```

Enable CXPI Channel

**Code Listing 4 Example for changing the CXPI channel mode in driver part**

```
cy_en_cxpi_status_t Cy_CXPI_SetMode(volatile stc_CXPI_CH_t *pstcCXPI, stc_CXPI_context_t *cxpiCtx, uint32_t mode)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    switch(mode) {
    case CXPI_SLEEP_MODE:
        Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_SLEEP);
        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_WAKEUP_DETECT);
        break;
    case CXPI_STANDBY_MODE:
        Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_WAKE_TO_STANDBY);
        break;
    case CXPI_NORMAL_MODE:
        Cy_CXPI_SetCmd(pstcCXPI,0x0);
        Cy_CXPI_EnableMessageFrameReception(pstcCXPI, cxpiCtx, 20);
        break;
     }
     return ret;
}
```

Move to Normal mode

## 4.2 CXPI controller initialization

Do the following:

(1) Disable the CXPI channel.
Configure CTL0.ENABLE to '0' to disable the CXPI channel before performing initialization.

(2) Set master/slave and Mode of operation (NRZ or PWM).
Configure CTL0.MODE to '0' for NRZ mode, to '1' for PWM mode.
Configure CTL0.MASTER to '0' for slave mode, to '1' for master mode.

(3) Turn Automatic Transceiver Handling ON/OFF:
Configure CTL0.AUTO_EN to '0' to turn it OFF and to '1' to turn it ON.

(4) Turn Receive PID Zero check ON/OFF.
Configure CLT0.RXPIDZERO_CHECK_EN to '1' only for slave mode in Polling method.
For other modes, set CLT0.RXPIDZERO_CHECK_EN to '0'.

(5) Turn RX filtering ON/OFF.
Configure CTL0.FILTER_EN to '0' to turn it OFF, to '1' to turn it ON.

(6) Set IFS length and IBS length.
Configure CTL0.IFS to desired Inter Frame Space in bit periods. Values lesser than 10 are not allowed.
Configure CTL0.IBS to desired Inter Byte Space in bit periods. Values greater than 9 are invalid per specification of CXPI protocol.

(7) Turn TX Abort for bit error detection ON/OFF.
Specifies the behavior on a detected bit error during header or response transmission by setting CTL0.BIT_ERROR_IGNORE to:

- '0': Message transfer is aborted.
- '1': Message transfer is NOT aborted.

(8) Define the PWM pulse (only for PWM mode).
Configure CTL1.T_LOW1 and CTL2.T_LOW2 to define PWM pulse corresponding to logic '1' and logic '0'. See **3.2.2 Sampling** for example of settings.

(9) Configure sample point.
Configure CTL1.T_OFFSET to configure RX sample point. See **3.2.2 Sampling** for example of settings.

(10) Define TX wake-up pulse length.
Configure CTL2.T_WAKEUP_LENGTH to specify the wake-up pulse low period in Tbits that is transmitted during Standby mode.

(11) Select the Time-out.
Configure CTL2.TIMEOUT_LENGTH to specify the number of Tbits to exceed timeout between frame bytes within a message frame. CXPI spec states that the maximum allowed inter byte space (IBS) is 9Tbits.
Configure CTL2.TIMEOUT_SEL to one of following value:

- '0' - Timeout check is disabled. HW clears the timeout counter.
- '1' - Timeout check is enabled and HW refers to CTL2.TIMEOUT_LENGTH as the number of Tbits allowed between the header and response.
- '2' - Timeout check is enabled to check header-header, header-response, and header-header-response within a message frame to be space within the CTL2.TIMEOUT_LENGTH bit time.

**Example of CXPI controller operation**

(12) Set the number of retries after arbitration loss to CLT2.RETRY.
In this example, software handles PID re-transmission after arbitration loss; therefore, CLT2.RETRY is set to '0'.

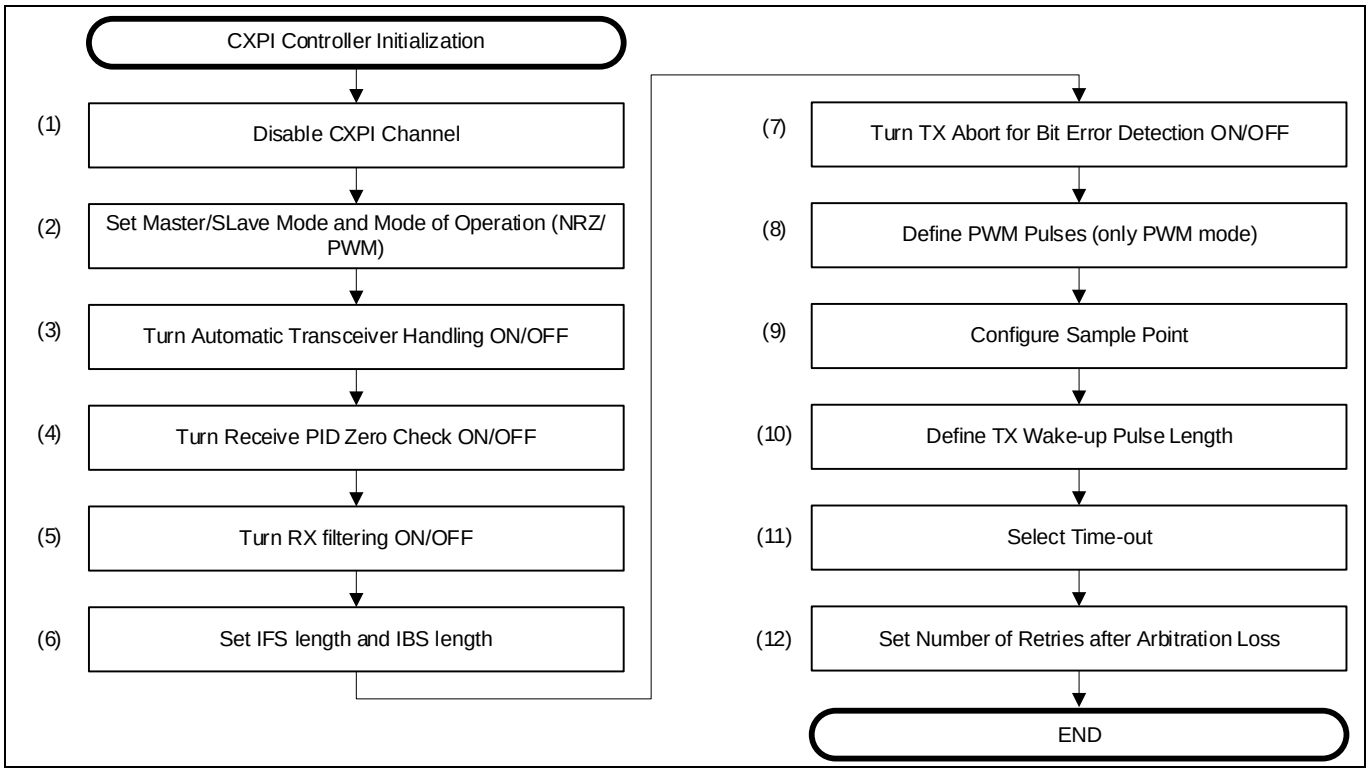


**Figure 5 Example of CXPI controller initialization flow**

### 4.2.1 Use case

- Operation mode: PWM mode
- Master/slave mode: Master mode
- Automatic transceiver handling: ON
- Receive PID zero check: OFF
- IFS length: 19-bit periods
- IBS length: 3-bit periods
- Tx abort for bit error: OFF
- CTL1.T_LOW1: 99
- CTL1.T_LOW0: 180
- CTL1.T_OFFSET: 120
- TIMEOUT_LENGTH: 9 Tbits
- Timeout check: Enabled. HW refers to TIMEOUT_LENGTH as the number of Tbits allowed between the header and response.

### 4.2.2 Configuration

**Table 8 CXPI initialization parameters in SDL**

| Parameters | | Description | Value |
|---|---|---|---|
| .stcCxpiChCtl0Config | | Configuration for CXPI controls | – |
| | .enMode | Selects the operation mode:<br>CXPI_NRZ_Mode: NRZ mode<br>CXPI_PWM_Mode: PWM mode | CXPI_PWM_Mode |
| | .bCxpi Transceiver AutoEnable | Selects automatic transceiver handling:<br>True: Enable<br>False: Disable | true |
| | .bRxPIDZero AutoCheck Enable | Selects receive PID zero check enable:<br>True: If received PID[6:0] = 0 and PID[7] = 1'b1,<br>HW does not clear CMD.RX_HEADER and will anticipate receiving header again.<br>False: No action if received PID[6:0] = 0 and PID[7] = 1'b1. | false |
| | .u8IFS | Sets IFS in bit periods | 19ul |
| | .u8IBS | Sets IBS in bit periods | 3ul |
| | .enBitError Ignore | Selects the behavior on a detected bit error during header or response transmission:<br>MESSAGE_TRANFER_ABORTED: Message transfer is aborted.<br>MESSAGE_TRANFER_NOT_ABORTED: Message transfer is NOT aborted. | MESSAGE_TRANFER_NOT_ABORTED |
| | .enCxpi MasterMode | Selects master mode or slave mode:<br>CXPI_MASTER_MODE: Master mode<br>CXPI_SLAVE_MODE: Slave mode | CXPI_MASTER_MODE |
| .stcCxpiChCtl1Config | | Configuration for timing controls at data link layer | – |
| | .u16TLow1 | Sets low count for logic 1 | 99 |
| | .u16TLow0 | Sets low count for logic 0 | 180 |
| | .u16TOffset | Sets the value of offset that is used for sampling the Rx | 120 |
| .stcCxpiChCtl2Config | | Configuration for timeout | – |
| | .u8Timeout Length | Specifies the number of Tbits to exceed the timeout between frame bytes within a message frame | 9u |
| | .enTimeoutSel | Selects the timeout check behavior.<br>CxpiTimeoutDisabled: Timeout check disabled<br>CxpiTimeoutCheckHdrRes: Timeout check enabled. HW will refer to TIMEOUT_LENGTH | CxpiTimeoutCheckHdrRes |

**Example of CXPI controller operation**

| Parameters | | Description | Value |
|---|---|---|---|
| | | as number of Tbits allowed between header and response.<br>CxpiTimeoutCheckAllByte: Timeout check enabled to check header-header, header-response, and header-header-response within a message frame to be space within TIMEOUT_LENGTH bit time. | |
| .numberOfAcceptedPID | | Number of accepted PID | DIM(AcceptPIDList) |
| .pAcceptedPIDList | | Accepted PID list | AcceptPIDList[3] = {0x6D, 0x7D, 0x67} |
| .maximumRetries | | Maximum number of retries after arbitration lost | 1 |
| .pNotifyTrasmitted WakeupPulse | | Sets the notification handler address for each event.<br>No handling if set to NULL. | NULL |
| .pNotifyRXFallingEdge Detected | | | NULL |
| .pNotifyError | | | &CxpiNotifyError |
| .pNotifySentPTYPE | | | NULL |
| .pNotifyReceivedPTYPE | | | NULL |
| .pNotifyReceivedPID | | | &CxpiNotifyReceivedPID |
| .pNotifyReceived Response | | | &CxpipNotifyReceivedResponse |
| .pNotifyArbitrationLost | | | &CxpipNotifyArbitrationLost |

## Example of CXPI controller operation

### Code Listing 5 Example of CXPI initialization in configuration part

```
uint8_t AcceptPIDList[3] = {0x6D /* receive PID then transmit responseList[0]*/,
                            0x7D /* receive PID then transmit responseList[1]*/,
                            0x67 /* receive both PID and response*/};
:
stc_cxpi_config_t stcCxpiConfig = {
    .stcCxpiChCtl0Config =
        {
            .enMode = CXPI_PWM_Mode,
            .bCxpiTransceiverAutoEnable = true,
            .bRxPIDZeroAutoCheckEnable = false,
            .u8IFS = 19ul,
            .u8IBS = 3ul,
            .enBitErrorIgnore = MESSAGE_TRANFER_NOT_ABORTED,
            .enCxpiMasterMode = CXPI_MASTER_MODE,
        },
    .stcCxpiChCtl1Config =
        {
            .u16TLow1 = 99,
            .u16TLow0 = 180,
            .u16TOffset = 120,
        },
    .stcCxpiChCtl2Config =
        {
            .u8TimeoutLength = 9ul,
            .enTimeoutSel = CxpiTimeoutCheckHdrRes,
        },
    .numberOfAcceptedPID = DIM(AcceptPIDList),
    .pAcceptedPIDList = AcceptPIDList,
    .maximumRetries = 1,
    .pNotifyTrasmittedWakeupPulse = NULL,
    .pNotifyRXFallingEdgeDetected = NULL,
    .pNotifyError = &CxpiNotifyError,
    .pNotifySentPTYPE = NULL,
    .pNotifyReceivedPTYPE = NULL,
    .pNotifyReceivedPID = &CxpiNotifyReceivedPID,
    .pNotifyReceivedResponse = &CxpipNotifyReceivedResponse,
    .pNotifyArbitrationLost = &CxpipNotifyArbitrationLost,
};
```

Accepted PID list

PWM mode

Automatic transceiver handling: ON

Receive PID zero check: OFF

IFS and IBS setting

Message transfer is NOT aborted.

Master mode

Configuration for timing controls at Data link layer.

Timeout length setting.

Configuration for timeout check behavior

Specifies the number of accepted PID

Specifies the accepted PID list

Specifies the maximum number of retries after arbitration lost.

Configuration for each notification handler.
No handling, if set to NULL.

## 4.2.3 Example code

The following description will help you understand the register notation of the driver part of SDL:

- pstcCXPI->un***CTL0***.u32Register is the CXPIx_CHy_CTL0 register mentioned in the **Registers TRM**. Other registers are also described in the same manner. “x” signifies the CXPI instance and “y” signifies the channel number of CXPI instance.
- Performance improvement measures
  For register setting performance improvement, the SDL writes a complete 32-bit data to the register. Each bit field is generated in advance in a bit writable buffer and written to the register as the final 32-bit data.

```
unInitCtl1.stcField.u9T_LOW1 = pstcConfig->stcCxpiChCtl1Config.u16TLow1;
unInitCtl1.stcField.u9T_LOW0 = pstcConfig->stcCxpiChCtl1Config.u16TLow0;
unInitCtl1.stcField.u9T_OFFSET = pstcConfig->stcCxpiChCtl1Config.u16TOffset;
pstcCXPI->unCTL1.u32Register = unInitCtl1.u32Register;
```

1. Generate 32-bit data on the buffer.

2. Write to register as complete 32-bit data.

See *cyip_cxpi.h* under *hdr/rev_x/ip* for more information on the union and structure representation of registers.

**Code Listing 6 Example of CXPI initialization in driver part**

```
Cy_CXPI_Init(volatile stc_CXPI_CH_t* pstcCXPI, const stc_cxpi_config_t* pstcConfig, stc_CXPI_context_t* pCxpiCtx)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if((NULL == pstcCXPI) || (NULL == pstcConfig)) {
        ret = CY_CXPI_BAD_PARAM;
    } else if((pstcConfig->stcCxpiChCtl1Config.u16TLow1 > CXPI_T_LOW1_MAX) ||
        (pstcConfig->stcCxpiChCtl1Config.u16TLow0 > CXPI_T_LOW0_MAX) ||
        (pstcConfig->stcCxpiChCtl1Config.u16TOffset > CXPI_T_OFFSET_MAX) ||
        (pstcConfig->stcCxpiChCtl2Config.enRetriesNumber > CXPI_RETRY_MAX) ||
        (pstcConfig->stcCxpiChCtl2Config.u8TimeoutLength > CXPI_TIMEOUT_MAX)) {
        ret = CY_CXPI_BAD_PARAM;
    } else {
        pstcCXPI->unCTL0.stcField.u1ENABLED = 0ul;

        un_CXPI_CH_CTL0_t unInitCtl0 = pstcCXPI->unCTL0;
        un_CXPI_CH_CTL1_t unInitCtl1 = pstcCXPI->unCTL1;
        un_CXPI_CH_CTL2_t unInitCtl2 = pstcCXPI->unCTL2;

        unInitCtl0.stcField.u1MODE = pstcConfig->stcCxpiChCtl0Config.enMode;
        unInitCtl0.stcField.u1MASTER = pstcConfig->stcCxpiChCtl0Config.enCxpiMasterMode;
        unInitCtl0.stcField.u1AUTO_EN = pstcConfig->stcCxpiChCtl0Config.bCxpiTransceiverAutoEnable;
        unInitCtl0.stcField.u1RXPIDZERO_CHECK_EN = pstcConfig->stcCxpiChCtl0Config.bRxPIDZeroAutoCheckEnable;
        unInitCtl0.stcField.u1FILTER_EN = pstcConfig->stcCxpiChCtl0Config.bFilterEnable;
        unInitCtl0.stcField.u5IFS = pstcConfig->stcCxpiChCtl0Config.u8IFS;
        unInitCtl0.stcField.u4IBS = pstcConfig->stcCxpiChCtl0Config.u8IBS;
        unInitCtl0.stcField.u1BIT_ERROR_IGNORE = pstcConfig->stcCxpiChCtl0Config.enBitErrorIgnore;

        pstcCXPI->unCTL0.u32Register = unInitCtl0.u32Register;

        if(pstcConfig->stcCxpiChCtl0Config.enMode == CXPI_PWM_Mode){
            unInitCtl1.stcField.u9T_LOW1 = pstcConfig->stcCxpiChCtl1Config.u16TLow1;
            unInitCtl1.stcField.u9T_LOW0 = pstcConfig->stcCxpiChCtl1Config.u16TLow0;
            unInitCtl1.stcField.u9T_OFFSET = pstcConfig->stcCxpiChCtl1Config.u16TOffset;
            pstcCXPI->unCTL1.u32Register = unInitCtl1.u32Register;
        }
```

Check if configuration parameter values are valid.

(1) Disable CXPI Channel

(2) Configure operation mode

(3) Configure Automatic Transceiver Handling

(4) Configure Receive PID Zero Check

(5) Configure RX filtering

(6) IFS and IBS setting

(7) Select behavior for bit error detection ON/OFF

(8) Define PWM pulses

(9) Configure sample point

## Example of CXPI controller operation

### Code Listing 6 Example of CXPI initialization in driver part

```
        unInitCtl2.stcField.u6T_WAKEUP_LENGTH = pstcConfig->stcCxpiChCtl2Config.u8TWakeUpLength;
        unInitCtl2.stcField.u4TIMEOUT_LENGTH = pstcConfig->stcCxpiChCtl2Config.u8TimeoutLength;
        unInitCtl2.stcField.u2TIMEOUT_SEL = pstcConfig->stcCxpiChCtl2Config.enTimeoutSel;
        unInitCtl2.stcField.u2RETRY = 0; //retransmission is fully controlled by software

        pstcCXPI->unCTL2.u32Register = unInitCtl2.u32Register;

        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
        pCxpiCtx->arbState = CXPI_ARB_PENDING;
        pCxpiCtx->retriesCount = 0;

        pCxpiCtx->pAcceptedPIDList = pstcConfig->pAcceptedPIDList;
        pCxpiCtx->numberOfAcceptedPID = pstcConfig->numberOfAcceptedPID;
        pCxpiCtx->maximumRetries = pstcConfig->maximumRetries;

        pCxpiCtx->pNotifyTrasmittedWakeupPulse = pstcConfig->pNotifyTrasmittedWakeupPulse;
        pCxpiCtx->pNotifyRXFallingEdgeDetected = pstcConfig->pNotifyRXFallingEdgeDetected;
        pCxpiCtx->pNotifyError = pstcConfig->pNotifyError;
        pCxpiCtx->pNotifySentPTYPE = pstcConfig->pNotifySentPTYPE;
        pCxpiCtx->pNotifyReceivedPTYPE = pstcConfig->pNotifyReceivedPTYPE;
        pCxpiCtx->pNotifyReceivedPID = pstcConfig->pNotifyReceivedPID;
        pCxpiCtx->pNotifyReceivedResponse = pstcConfig->pNotifyReceivedResponse;
        pCxpiCtx->pNotifyArbitrationLost = pstcConfig->pNotifyArbitrationLost;

        Cy_CXPI_SetInterruptMask(pstcCXPI, 0x0ul);
    }
    return ret;
}
```

(10) Define TX wake-up pulse length

(11) Select timeout check behavior

(12) Set number of retries after arbitration loss

## 4.3 Message frame transmission/reception

In this example, two state variables, cxpi_state and arbitration_state, are used to manage the communication state and arbitration state. Setting of the command sequence is managed by two state variables.

The “cxpi_state” communication state variable has one of the following values:

- CXPI_STATE_TXRX_PENDING: Message frame transmission/reception pending state. This state indicates that the message frame reception has been enabled, the node is waiting for coming message frame, or user requested a message frame transmission.
- CXPI_STATE_TX_PTYPE: This state indicates that the master node is transmitting PTYPE field. This state is available only for the master node in polling method.
- CXPI_STATE_TX_HDR_TX_RESP: This state indicates that master/slave node is transmitting both header and response.
- CXPI_STATE_TX_HDR_RX_RESP: This state indicates that master/slave node is transmitting header and receiving response.
- CXPI_STATE_RX_HDR_TX_RESP: This state indicates that master/slave node received a relevant PID and is sending response.
- CXPI_STATE_RX_HDR_RX_RESP: This state indicates that master/slave node received a relevant PID and continues receiving response.

The “arbitration_state” arbitration state variable has one of the following values:

- CXPI_ARB_PENDING: Arbitration pending state. No arbitration loss is detected.
- CXPI_ARB_LOST_TX_RESP: Master/slave node attempted to send both header and response but lost the arbitration.
- CXPI_ARB_LOST_RX_RESP: Master/slave node attempted to send a header to receive response but lost the arbitration.

### Example of CXPI controller operation

**Figure 6** shows the transition of “cxpi_state”.

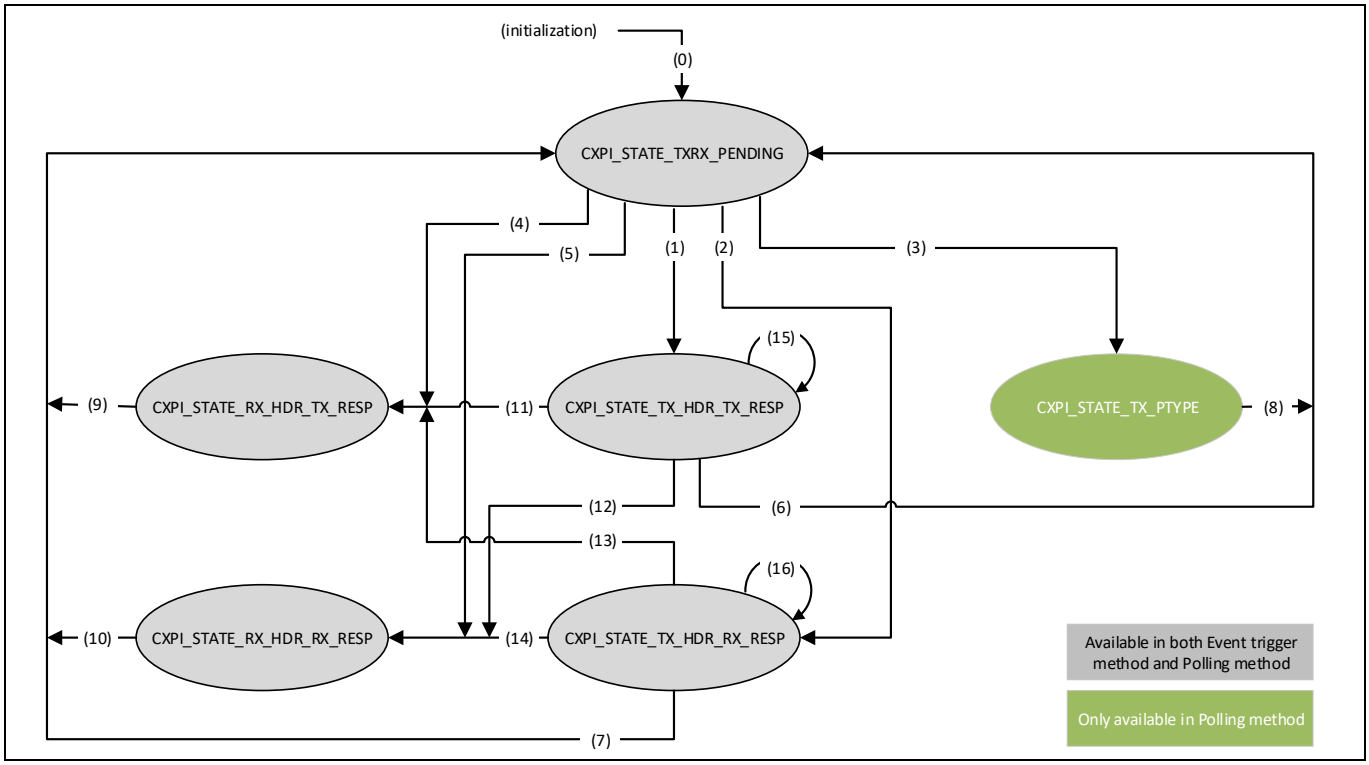


**Figure 6 Transition of “cxpi_state”**

**Table 9 Condition for transition of communication state**

| No | Present state | Next state | Conditions | Command sequence issued in transition (No. in Table 4) |
|---|---|---|---|---|
| (0) | (after initialization) | CXPI_STATE_ TXRX_PENDING | User enables message frame reception. | No.4 or No.11<br>Not required for master in Polling method |
| (1) | CXPI_STATE_TXRX_ PENDING | CXPI_STATE_TX_ HDR_TX_RESP | User requests both header and response transmission. | No.1, No.7, or No.11 |
| (2) | CXPI_STATE_TXRX_ PENDING | CXPI_STATE_TX_ HDR_RX_RESP | User requests both header transmission and enables response reception. | No.2, No.8 |
| (3) | CXPI_STATE_TXRX_ PENDING | CXPI_STATE_TX_ PTYPE | User requests PTYPE field transmission. | No.6 |
| (4) | CXPI_STATE_TXRX_ PENDING | CXPI_STATE_RX_ HDR_TX_RESP | Master/slave node received a header and the received header indicates that this node should transmit/receive response. | No.5, No.9, or No.12 |
| (5) | CXPI_STATE_TXRX_ PENDING | CXPI_STATE_RX_ HDR_RX_RESP | | Not required |
| (6) | CXPI_STATE_TX_ HDR_TX_RESP | CXPI_STATE_ TXRX_PENDING | Message frame reception or transmission is done (even in | For all nodes in Event Trigger method: |

**Example of CXPI controller operation**

| No | Present state | Next state | Conditions | Command sequence issued in transition (No. in Table 4) |
|---|---|---|---|---|
| (7) | CXPI_STATE_TX_HDR_RX_RESP | CXPI_STATE_TXRX_PENDING | error case). User enabled message frame reception and this node reverts to transmission/reception pending state. | • No.3, if there is no error<br>• No.4, if an error occurred<br>• For slave node in Polling Method: No.11 |
| (8) | CXPI_STATE_TX_PTYPE | CXPI_STATE_TXRX_PENDING | Master in polling method finished transmitting PTYPE. | Not required |
| (9) | CXPI_STATE_RX_HDR_TX_RESP | CXPI_STATE_TXRX_PENDING | Message frame reception or transmission is completed (even in error case). User enable message frame reception. This node reverts to transmission/reception pending state. | For all nodes in Event Trigger method:<br>• No.3, if there is no error<br>• No.4, if error an occurred<br>• For slave node in Polling Method: No.11 |
| (10) | CXPI_STATE_RX_HDR_RX_RESP | CXPI_STATE_TXRX_PENDING | | |
| (11) | CXPI_STATE_TX_HDR_TX_RESP | CXPI_STATE_RX_HDR_TX_RESP | This node attempted to transmit header but received a header while waiting for IFS or lost the arbitration.<br>The received header indicates that this node should transmit/receive response. | No.5 |
| (12) | CXPI_STATE_TX_HDR_TX_RESP | CXPI_STATE_RX_HDR_RX_RESP | | Not required |
| (13) | CXPI_STATE_TX_HDR_RX_RESP | CXPI_STATE_RX_HDR_TX_RESP | | No.5 |
| (14) | CXPI_STATE_TX_HDR_RX_RESP | CXPI_STATE_RX_HDR_RX_RESP | | Not required |
| (15) | CXPI_STATE_TX_HDR_TX_RESP | CXPI_STATE_TX_HDR_TX_RESP | This node attempted to transmit header but lost the arbitration and received an irrelevant PID. After that, the node retries to transmit the header and transmit/receive response. | No.1 |
| (16) | CXPI_STATE_TX_HDR_RX_RESP | CXPI_STATE_TX_HDR_RX_RESP | This node attempted to transmit header but lost the arbitration and received an irrelevant PID. After that, the node retries to transmit the header and transmit/receive response. | No.2 |

### 4.3.1 Enable message frame reception

**Figure 7** shows an example for enabling message frame reception.

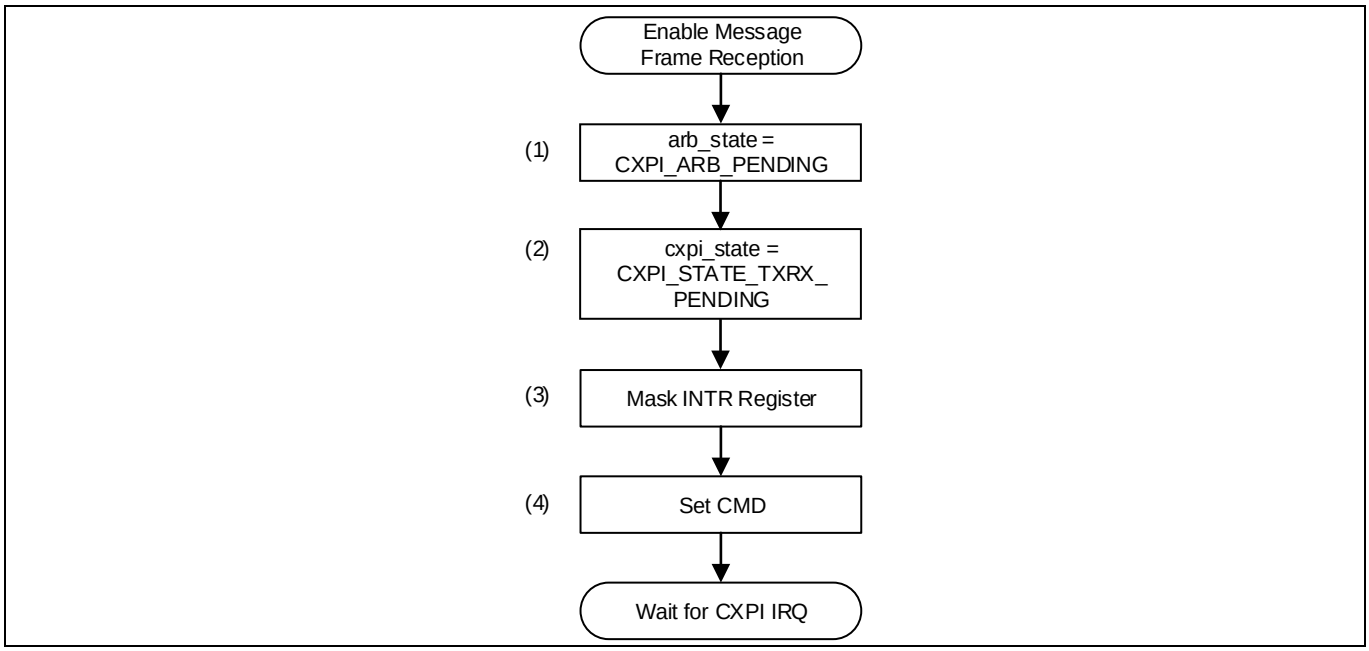


**Figure 7 Message frame reception enable flowchart**

(1) Initialize the arbitration state variable
Set arb_state = CXPI_ARB_PENDING

(2) Initialize the communication state variable
Set cxpi_state = CXPI_STATE_TXRX_PENDING

(3) Mask the INTR register by setting INTR_MASK:
INTR_MASK.RX_HEADER_PID_DONE = '1'
INTR_MASK.RX_RESPONSE_DONE = '1'
INTR_MASK.TXRX_COMPLETE = '1'

(4) Set CMD (enable header/frame reception)
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'0', '0', '1', '0', '1'} to enable header/frame reception.
If the slave node joins the CXPI bus in the middle, configure CMD.IFS_WAIT to '1' and CLT0.IFS = '20' to make hardware check for bus idleness before enabling header/frame reception.

After message reception is enabled, SW waits for occurrence of a CXPI interrupt. The CXPI interrupt handler is explained in **4.3.3 CXPI interrupt handle**.

**Example of CXPI controller operation**

### 4.3.1.1 Use case

- Interrupt:
  - PID reception done (RX_HEADER_PID_DONE)
  - Response reception done (RX_RESPONSE_DONE)
  - Message frame ends (TXRX_COMPLETE)
- Command:
  - Receive header (RX_HEADER)
  - Receive response (RX_RESPONSE)
  - Wait for IFS (IFS_WAIT)

### 4.3.1.2 Configuration

**Table 10 Enabling message frame reception functions in SDL**

| Function | | Description | Value |
|---|---|---|---|
| Cy_CXPI_SetInterruptMask() | | Setup interrupt source to be accepted | – |
| | pstcCXPI | Specifies the pointer to CXPI instance register area | pstcCXPI |
| | mask | The mask with the OR of the interrupt source to be accepted:<br>CXPI_INTR_TX_HEADER_DONE: Header transmission done<br>CXPI_INTR_TX_RESPONSE_DONE: Response transmission done<br>CXPI_INTR_TX_WAKEUP_DONE: Wakeup transmission done<br>CXPI_INTR_TX_FIFO_TRIGGER: Transmission FIFO trigger level shortfall<br>CXPI_INTR_RX_HEADER_DONE: Header reception done<br>CXPI_INTR_RX_RESPONSE_DONE: Response reception done<br>CXPI_INTR_RX_WAKEUP_DETECT: Receiver wakeup detect<br>CXPI_INTR_RX_FIFO_TRIGGER: Reception FIFO trigger level exceeded<br>CXPI_INTR_RX_HEADER_PID_DONE: PID reception done<br>CXPI_INTR_TXRX_COMPLETE: Frame message ends<br>CXPI_INTR_TIMEOUT: Time out<br>CXPI_INTR_TX_HEADER_ARB_LOST: Transmitter arbitration loss<br>CXPI_INTR_TX_BIT_ERROR: Transmitter bit error | CXPI_INTR_RX_HEADER_PID_DONE \|<br>CXPI_INTR_TXRX_COMPLETE \|<br>CXPI_INTR_RX_RESPONSE_DONE \|<br>CXPI_INTR_TIMEOUT \|<br>CXPI_INTR_ALL_ERROR_MASK |

**Example of CXPI controller operation**

| Function | | Description | Value |
|---|---|---|---|
| | | CXPI_INTR_RX_CRC_ERROR: Receiver response CRC error<br>CXPI_INTR_RX_HEADER_PARITY_ERROR: Receiver PID parity error<br>CXPI_INTR_RX_DATA_LENGTH_ERROR: Receiver data length error<br>CXPI_INTR_TX_DATA_LENGTH_ERROR: Transmitter data length error<br>CXPI_INTR_RX_OVERFLOW_ERROR: Reception overflow error<br>CXPI_INTR_TX_OVERFLOW_ERROR: Transmission overflow error<br>CXPI_INTR_RX_UNDERFLOW_ERROR: Reception underflow error<br>CXPI_INTR_TX_UNDERFLOW_ERROR: Transmission underflow error<br>CXPI_INTR_RX_FRAME_ERROR: Receiver frame error<br>CXPI_INTR_TX_FRAME_ERROR: Transmitter frame error<br>CXPI_INTR_ALL_ERROR_MASK: CXPI all error | |
| Cy_CXPI_SetCmd() | | Setup CXPI operation command | – |
| | pstcCXPI | Specifies the pointer to CXPI instance register area | pstcCXPI |
| | command | Specifies the required operation command:<br>CXPI_CMD_TX_HEADER: Transmit header only<br>CXPI_CMD_RX_HEADER: Receive header only<br>CXPI_CMD_TX_HEADER_RX_HEADER: Transmit header and receive header<br>CXPI_CMD_TX_RESPONSE: Transmit response<br>CXPI_CMD_RX_RESPONSE: Receive response<br>CXPI_CMD_TX_HEADER_TX_RESPONSE: Transmit header and transmit response<br>CXPI_CMD_TX_HEADER_RX_RESPONSE: Transmit header and receive response<br>CXPI_CMD_RX_HEADER_RX_RESPONSE: Receive header and receive response<br>CXPI_CM_TX_WAKEUP: Wakeup frame<br>CXPI_CMD_SLEEP: Sleep mode<br>CXPI_CMD_WAKE_TO_STANDBY: Wake up from sleep mode to standby mode<br>CXPI_CMD_IFS_WAIT: Wait for IFS | CXPI_CMD_IFS_WAIT \| CXPI_CMD_RX_HEADER_RX_ RESPONSE |

### 4.3.1.3 Example code

**Code Listing 7 Example of message frame reception enable in driver part**

```
void Cy_CXPI_EnableMessageFrameReception(volatile stc_CXPI_CH_t *pstcCXPI, stc_CXPI_context_t *cxpiCtx,
uint32_t IFSwait){
    uint32_t command = 0ul;
    cxpiCtx->arbState = CXPI_ARB_PENDING;
    cxpiCtx->state = CXPI_STATE_TXRX_PENDING;

    Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_TXRX_COMPLETE |
    CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TIMEOUT | CXPI_INTR_ALL_ERROR_MASK);

    command = CXPI_CMD_RX_HEADER_RX_RESPONSE;
    if(IFSwait > 0ul){
        pstcCXPI->unCTL0.stcField.u5IFS = IFSwait;
        command |= CXPI_CMD_IFS_WAIT;
    }
    Cy_CXPI_SetCmd(pstcCXPI, command);
}
```

(1) Initialize arbitration state

(2) Initialize communication state variable

(3) Mask INTR register (**Code Listing 8**)

(4) Set CMD (**Code Listing 9**)

**Code Listing 8 Example of message frame reception enable in driver part**

```
cy_en_cxpi_status_t Cy_CXPI_SetInterruptMask(volatile stc_CXPI_CH_t* pstcCXPI, uint32_t mask)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    pstcCXPI->unINTR_MASK.u32Register = mask;
    return ret;
}
```

(3) Set INTR_MASK register

**Code Listing 9 Example of message frame reception enable in driver part**

```
cy_en_cxpi_status_t Cy_CXPI_SetCmd(volatile stc_CXPI_CH_t* pstcCXPI, uint32_t command)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    pstcCXPI->unCMD.u32Register = command;
    return ret;
}
```

(4) Set CMD register

### 4.3.2 Request message frame transmission

**Figure 8** shows an example of request message frame transmission.

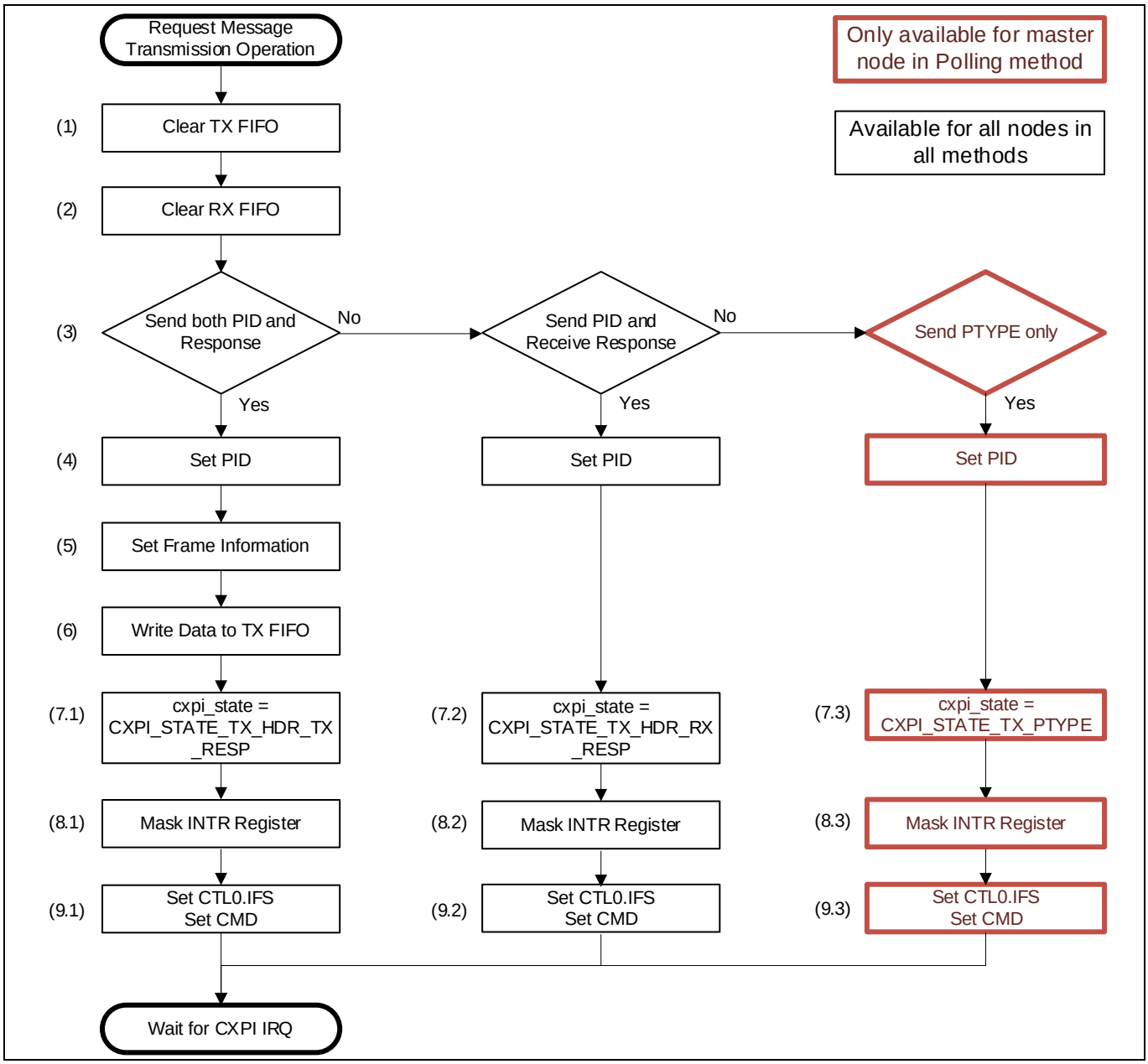


**Figure 8 Flowchart for requesting message frame transmission**

(1) Clear the TX FIFO in advance
Configure TX_FIFO_CTL.CLEAR to '1' and followed by '0' to perform clearing the TX FIFO.

(2) Clear the RX FIFO in advance
Configure RX_FIFO_CTL.CLEAR to '1' and followed by '0' to perform clearing the RX FIFO.

**Example of CXPI controller operation**

(3) Set the PID/PTYPE field of the header by setting TXPID_FI.PID as listed in **Table 11**.

**Table 11 PID/PTYPE field setting**

| Register | Bit | Description |
|---|---|---|
| TXPID_FI | PID[6:0] | PID to be transmitted<br>This field can be used to transmit the PTYPE field because the hardware handles both PID and PTYPE the same way, i.e., setting TXPID_FI.PID[6:0]= 0 to transmit PTYPE field. |
| | PID[7] | Odd parity bit. Calculated by the hardware.<br>PID[7] = ! (ID[6] ^ ID[5] ^ ID[4] ^ ID[3] ^ ID[2] ^ ID[1] ^ ID[0]) |

(4) Set the Frame Information field of the response by setting TXPID_FI.FI. as listed in **Table 12**.

**Table 12 Frame information setting**

| Register | Bit | Description |
|---|---|---|
| TXPID_FI | FI [7:4] | Denotes the data length count (DLC). |
| | FI [3:2] | Denotes Network Management.<br>FI[3] - wakeup.ind<br>FI[2] - sleep.ind |
| | FI [1:0] | denotes CT |

(5) Set the Data Length Code Extension field for long frame by writing the data length to TXPID_FI.DLCEXT. This field is valid only if TXPID_FI.FI[7:4] = 4'b1111. The value specified in this field will be the new data length count. Valid values are 0–255.

(6) Write each byte of data to TX_FIFO_WR.DATA[7:0]. Hardware shadows over the write data to TX FIFO after SW performs a write to this field.

(7) Modify the value of communication state variable 'cxpi_state' depending on the requested transmission.

(8) Set the INTR_MASK register to enable the event interrupt according to the requested transmission:

(8.1) CXPI_STATE_TX_HDR_TX_RESP case:
INTR_MASK.TX_HEADER_DONE = '1'
INTR_MASK.RX_HEADER_PID_DONE = '1'
INTR_MASK.TX_RESPONSE_DONE = '1'
INTR_MASK.RX_RESPONSE_DONE = '1'
INTR_MASK.TXRX_COMPLETE = '1'

(8.2) CXPI_STATE_TX_HDR_RX_RESP case:
INTR_MASK.TX_HEADER_DONE = '1'
INTR_MASK.RX_HEADER_PID_DONE = '1'
INTR_MASK.RX_RESPONSE_DONE = '1'
INTR_MASK.TXRX_COMPLETE = '1'

(8.3) CXPI_STATE_TX_PTYPE case:
INTR_MASK.TX_HEADER_DONE = '1'
INTR_MASK.RX_HEADER_PID_DONE = '1'
INTR_MASK.RX_RESPONSE_DONE = '1'
INTR_MASK.TXRX_COMPLETE = '1'

**Example of CXPI controller operation**

In all cases, enable necessary error reporting interrupts by configuring the corresponding bits in INTR_MASK to '1'.

(9) Set the command sequence according to state according to the requested transmission.

(9.1) CXPI_STATE_TX_HDR_TX_RESP case:
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'0' or '1', '1', '1', '1', '1'}

(9.2) CXPI_STATE_TX_HDR_RX_RESP case:
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'0' or '1', '1', '1', '1', '0', '1'}

(9.3) CXPI_STATE_TX_PTYPE case:
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'0' or '1', '1', '1', '1', '0', '1'}

*Note:* *Set CMD.IFS_WAIT = '0' if IFS checking is not required, i.e., transmitting the header after transmitting or receiving PTYPE.*
*If IFS checking is required, before setting the CMD register, set CLT0.IFS to the number of '1' logic bits that needs to be fulfilled before transmitting the header.*

After requesting message transmission, SW waits for occurrence of a CXPI interrupt. The CXPI interrupt handler is explained in **4.3.3 CXPI interrupt handle**.

### 4.3.2.1 Use case

**Table 13 Use case of requesting message frame transmission**

| Use case | Send both PID and response | Send PID and receive response |
|---|---|---|
| CXPI channel | 3 | 3 |
| Frame ID | 0x4A or 0x4F | 0x0A or 0x0F |
| Response data length | 0x4A: 8 bytes, 0x4F: 16 bytes | – |
| State | CXPI_STATE_TX_HDR_TX_RESP | CXPI_STATE_TX_HDR_RX_RESP |
| Interrupt | Header transmission done (CXPI_INTR_TX_HEADER_DONE)<br>PID reception done (CXPI_INTR_RX_HEADER_PID_DONE)<br>Response transmission done (CXPI_INTR_TX_RESPONSE_DONE)<br>Response reception done (CXPI_INTR_RX_RESPONSE_DONE)<br>Frame message ends (CXPI_INTR_TXRX_COMPLETE) | Header transmission done (CXPI_INTR_TX_HEADER_DONE)<br>PID reception done (CXPI_INTR_RX_HEADER_PID_DONE)<br>Response reception done (CXPI_INTR_RX_RESPONSE_DONE)<br>Frame message ends (CXPI_INTR_TXRX_COMPLETE) |
| Command | Transmit header (CXPI_CMD_TX_HEADER)<br>Receive header (CXPI_CMD_RX_HEADER)<br>Transmit response (CXPI_CMD_TX_RESPONSE)<br>Receive response (CXPI_CMD_RX_RESPONSE)<br>Wait for IFS (CXPI_CMD_IFS_WAIT) | Transmit header (CXPI_CMD_TX_HEADER)<br>Receive header (CXPI_CMD_RX_HEADER)<br>Receive response (CXPI_CMD_RX_RESPONSE)<br>Wait for IFS (CXPI_CMD_IFS_WAIT) |

### 4.3.2.2 Configuration

**Table 14 Requesting message frame transmission functions in SDL**

| Function | | Description | Value |
|---|---|---|---|
| Cy_CXPI_SendPID() | | PID transmission and response transmission/reception | – |
| | stc_CXPI_CH_t* | Specifies the CXPI channel instance | CXPI_CH_INSTANCE = CY_CXPICH3_TYPE |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | &cxpiCtx |
| | frameID | Specifies the Frame ID that will be sent | scheduledFrameCtx[scheduleIdx].id (See **Code Listing 10**) |
| | withResponse | Selects response transmission:<br>True: Send response after finishing transmitting PID<br>False: No response | CXPI_STATE_TX_HDR_TX_RESP case: true<br>Other cases: false |
| | pResponse | Specifies the pointer to the response which is sent after transmitting PID, if "withResponse" is true. | &scheduledFrameCtx[scheduleIdx].response<br>(See **Code Listing 10**) |
| | IFSwait | Selects IFS checking:<br>True: Wait for IFS<br>False: No wait for IFS | true |
| Cy_CXPI_SetCmd() | | Sets up the CXPI operation command:<br>See **Table 10**. | See **Code Listing 11**. |
| Cy_CXPI_SetInterruptMask() | | Sets up the interrupt source to be accepted:<br>See **Table 10**. | See **Code Listing 11**. |
| Cy_CXPI_ClearTxFIFO() | | Clears TxFIFO | – |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | pstcCXPI |
| Cy_CXPI_ClearRxFIFO() | | Clears RxFIFO | – |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | pstcCXPI |
| Cy_CXPI_SetPID() | | Sets PID | – |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | pstcCXPI |
| | id | Specifies the Frame ID that will be sent | frameID |
| Cy_CXPI_SetFI() | | Sets the frame information | – |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | pstcCXPI |
| | fi | Specifies the frame information | pResponse->frameIF<br>(See **Code Listing 10**) |
| Cy_CXPI_SendPTYPE() | | Sends PTYPE | – |

**Example of CXPI controller operation**

| Function | | Description | Value |
|---|---|---|---|
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | – |
| | cxpiCtx | Pointer to the context structure | – |

### 4.3.2.3 Example code

**Code Listing 10 Example of requesting message frame transmission in configuration part**

```
cxpi_frame_context_t scheduledFrameCtx[] = {
    {
        .id = 0x0A,
    },
    {
        .id = 0x4A,
        .response =
        {.frameIF =
            {
                .datalength = 8ul,
                .SleepInd = 0x1ul,
                .WakeupInd = 0x1ul,
                .counter = 0x03ul,
            },
        }
    },
    {
        .id = 0x0F,
    },
    {
        .id = 0x4F,
        .response =
        {.frameIF =
            {
                .datalength = 16ul,
                .SleepInd = 0x1ul,
                .WakeupInd = 0x1ul,
                .counter = 0x03ul,
            },
        }
    },
};
:
cxpi_response_t reponseList[] = {
    {.frameIF =
            {
                .datalength = 8ul,
                .SleepInd = 0x1ul,
                .WakeupInd = 0x1ul,
                .counter = 0x03ul,
            },
        .dataBuffer = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 }
     },
     {.frameIF =
            {
                .datalength = 24ul,
                .SleepInd = 0x1ul,
                .WakeupInd = 0x1ul,
                .counter = 0x03ul,
            },
        .dataBuffer = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                        0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
                        0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28 }
     },

};
:
void CxpiScheduleHandler(void)
{
    static uint32_t scheduleIdx = 0ul;
    switch (scheduledFrameCtx[scheduleIdx].id){
```

## Example of CXPI controller operation

### Code Listing 10 Example of requesting message frame transmission in configuration part

```
        case 0x0A:
            Cy_CXPI_SendPID(CXPI_CH_INSTANCE, &cxpiCtx, scheduledFrameCtx[scheduleIdx].id,
                    false,&scheduledFrameCtx[scheduleIdx].response, true);
            break;
        case 0x4A:
            Cy_CXPI_SendPID(CXPI_CH_INSTANCE, &cxpiCtx, scheduledFrameCtx[scheduleIdx].id,
                    true,&scheduledFrameCtx[scheduleIdx].response, true);
            break;
        case 0x0F:
            Cy_CXPI_SendPID(CXPI_CH_INSTANCE, &cxpiCtx, scheduledFrameCtx[scheduleIdx].id,
                    false,&scheduledFrameCtx[scheduleIdx].response, true);
            break;
        case 0x4F:
            Cy_CXPI_SendPID(CXPI_CH_INSTANCE, &cxpiCtx, scheduledFrameCtx[scheduleIdx].id,
                    true,&scheduledFrameCtx[scheduleIdx].response, true);
            break;
    }
    scheduleIdx = (scheduleIdx + 1) % DIM(scheduledFrameCtx);
}
```

Request for PID transmission and response reception

Request for both PID and response transmission

Request for PID transmission and response reception

Request for both PID and response transmission

### Code Listing 11 Example of PID transmission and response transmission/reception in driver part

```
cy_en_cxpi_status_t Cy_CXPI_SendPID(volatile stc_CXPI_CH_t* pstcCXPI,
    stc_CXPI_context_t* cxpiCtx,
    uint8_t frameID,
    bool withResponse,
    cxpi_response_t* pResponse,
    bool IFSwait)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    if(cxpiCtx->state != CXPI_STATE_TXRX_PENDING) {
        return CY_CXPI_BAD_PARAM;
    }
    Cy_CXPI_ClearTxFIFO(pstcCXPI);
    Cy_CXPI_ClearRxFIFO(pstcCXPI);

    if(!withResponse) {
        Cy_CXPI_SetPID(pstcCXPI, frameID);
        cxpiCtx->state = CXPI_STATE_TX_HDR_RX_RESP;
        cxpiCtx->receivedFrame.id = frameID;
        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_TX_HEADER_DONE | CXPI_INTR_RX_HEADER_PID_DONE |
CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TXRX_COMPLETE | CXPI_INTR_TIMEOUT |  CXPI_INTR_ALL_ERROR_MASK);
        if(IFSwait) {
            Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_IFS_WAIT | CXPI_CMD_TX_HEADER_RX_RESPONSE |
CXPI_CMD_RX_HEADER);
        } else {
            Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_TX_HEADER_RX_RESPONSE | CXPI_CMD_RX_HEADER);
        }
    } else {
        Cy_CXPI_SetPID(pstcCXPI, frameID);
        Cy_CXPI_SetFI(pstcCXPI, pResponse->frameIF);

        uint32_t k = 0;
        for(k = 0; k < pResponse->frameIF.datalength; k++) {
            Cy_CXPI_WriteTxFIFO(pstcCXPI, pResponse->dataBuffer[k]);
        }
        cxpiCtx->txBuf = pResponse->dataBuffer;
        cxpiCtx->txBuffCount = pResponse->frameIF.datalength;

        cxpiCtx->state = CXPI_STATE_TX_HDR_TX_RESP;
        cxpiCtx->transmittingFrame.id = frameID;
        cxpiCtx->pTransmittingFrameResponse = pResponse;

        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_TX_HEADER_DONE | CXPI_INTR_RX_HEADER_PID_DONE |
CXPI_INTR_TX_RESPONSE_DONE | CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TXRX_COMPLETE | CXPI_INTR_TIMEOUT |
CXPI_INTR_ALL_ERROR_MASK);
```

(1) Clear TX FIFO (See **Code Listing 14**)

(2) Clear RX FIFO (See **Code Listing 12**

(3) Send PID and Receive Response

(4) Set PID (See **Code Listing 13**)

(7.2) cxpi_state = CXPI_STATE_TX_HDR_RX_RESP

(8.2) Mask INTR register (**Code Listing 8**)

(9.2) Set CMD (**Code Listing 9**)

(3) Send both PID and Response

(4) Set PID (See **Code Listing 13**)

(5) Set frame information (**Code Listing 15**)

(6) Write data to TX FIFO

(7.1) cxpi_state = CXPI_STATE_TX_HDR_TX_RESP

(8.1) Mask INTR register (**Code Listing 8**)

## Example of CXPI controller operation

### Code Listing 11 Example of PID transmission and response transmission/reception in driver part

```
        if(IFSwait) {
            Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_IFS_WAIT | CXPI_CMD_TX_HEADER_TX_RESPONSE |
CXPI_CMD_RX_HEADER_RX_RESPONSE);
        } else {
            Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_TX_HEADER_TX_RESPONSE | CXPI_CMD_RX_HEADER_RX_RESPONSE);
        }
    }
    return ret;
}
```

(9.1) Set CMD (**Code Listing 9**)

### Code Listing 12 Example of clearing the Rx FIFO in driver part

```
cy_en_cxpi_status_t Cy_CXPI_ClearRxFIFO(volatile stc_CXPI_CH_t* pstcCXPI)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    pstcCXPI->unRX_FIFO_CTL.stcField.u1CLEAR = 1ul;
    while(pstcCXPI->unRX_FIFO_STATUS.stcField.u5USED != 0ul) {
    };
    pstcCXPI->unRX_FIFO_CTL.stcField.u1CLEAR = 0ul;
    return ret;
}
```

Clear RX FIFO

### Code Listing 13 Example of PID setting for header transmission in driver part

```
cy_en_cxpi_status_t Cy_CXPI_SetPID(volatile stc_CXPI_CH_t* pstcCXPI, uint8_t id)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    pstcCXPI->unTXPID_FI.stcField.u8PID = id;
    return ret;
}
```

Set PID

### Code Listing 14 Example of clearing the Tx FIFO in driver part

```
cy_en_cxpi_status_t Cy_CXPI_ClearTxFIFO(volatile stc_CXPI_CH_t* pstcCXPI)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    pstcCXPI->unTX_FIFO_CTL.stcField.u1CLEAR = 1ul;
    while(pstcCXPI->unTX_FIFO_STATUS.stcField.u5USED != 0ul) {
    };
    pstcCXPI->unTX_FIFO_CTL.stcField.u1CLEAR = 0ul;
    return ret;
}
```

Clear TX FIFO

## Example of CXPI controller operation

### Code Listing 15 Example of setting the frame information in driver part

```
cy_en_cxpi_status_t Cy_CXPI_SetFI(volatile stc_CXPI_CH_t* pstcCXPI, stc_cxpi_frame_info_t fi)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if((NULL == pstcCXPI) || (fi.counter > CXPI_COUNTER_MAX)) {
        return CY_CXPI_BAD_PARAM;
    }
    uint8_t temp = 0u;
    if(fi.datalength <= 12u) {
        temp = temp | ((fi.datalength & 0x0fu) << 4);
        temp = temp | ((fi.WakeupInd & 0x01u) << 3);
        temp = temp | ((fi.SleepInd & 0x01u) << 2);
        temp = temp | (fi.counter & 0x03u);
        pstcCXPI->unTXPID_FI.stcField.u8FI = temp;
        // pstcCXPI->unTXPID_FI.stcField.u8DLCEXT = 0u;
    } else {
        temp = temp | ((0x0fu) << 4);
        temp = temp | ((fi.WakeupInd & 0x01u) << 3);
        temp = temp | ((fi.SleepInd & 0x01u) << 2);
        temp = temp | (fi.counter & 0x03u);
        pstcCXPI->unTXPID_FI.stcField.u8FI = temp;
        pstcCXPI->unTXPID_FI.stcField.u8DLCEXT = fi.datalength;
    }
    return ret;
}
```

Set Frame Information

### Code Listing 16 Example of sending PTYPE only in driver part

```
cy_en_cxpi_status_t Cy_CXPI_SendPTYPE(volatile stc_CXPI_CH_t* pstcCXPI, stc_CXPI_context_t* cxpiCtx)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    cxpiCtx->state = CXPI_STATE_TX_PTYPE;
    Cy_CXPI_SetPID(pstcCXPI, CXPI_PTYPE);
    Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_TX_HEADER_DONE |
            CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TXRX_COMPLETE | CXPI_INTR_TIMEOUT |
CXPI_INTR_TXRX_COMPLETE | CXPI_INTR_ALL_ERROR_MASK);
    Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_IFS_WAIT | CXPI_CMD_TX_HEADER_RX_HEADER | CXPI_CMD_RX_RESPONSE);
    return ret;
}
```

(7.3) cxpi_state = CXPI_STATE_TX_PTYPE

(8.3) Mask INTR register (**Code Listing 8**)

(9.3) Set CMD (**Code Listing 9**)

## 4.3.3 CXPI interrupt handle

This section explains the implementation of CXPI interrupt handling (see **Figure 9**).

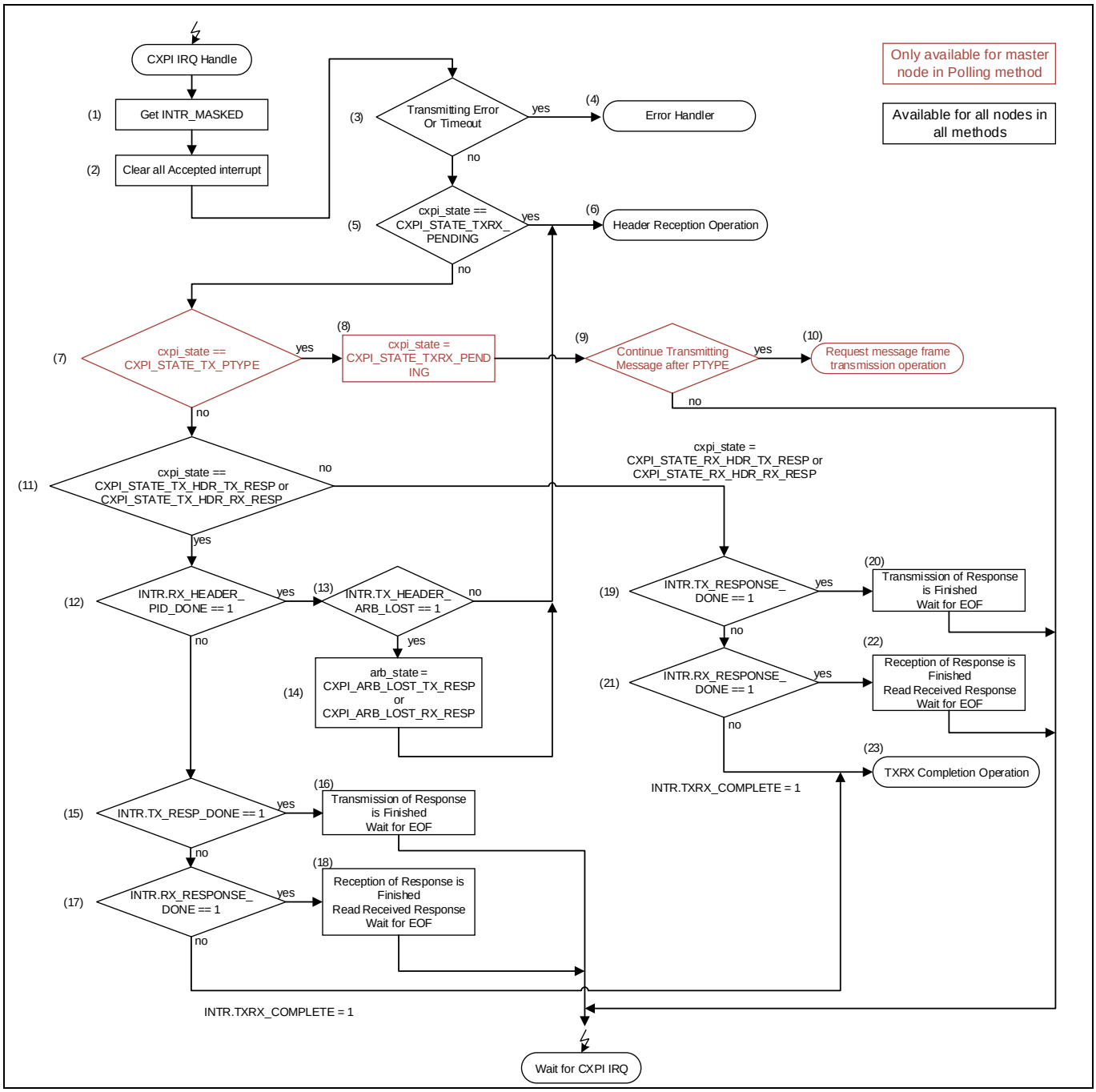


**Figure 9 CXPI interrupt handle flowchart**

(1) Acquire the interrupt information from INTR_MASKED register

(2) Clear all accepted interrupts

(3) Check if error occurred. If yes, go to (4)

(4) Handle error. See section **4.3.6 Error handler**.

**Example of CXPI controller operation**

(5) If the current state is CXPI_STATE_TXRX_PENDING, go to (6) to perform header reception operation. Else, go to (7).

(6) Header reception operation. See **4.3.4 Header reception operation** for the software flowchart.

(7) If the current state is CXPI_STATE_TX_PTYPE, go to (8); else, go to (11).

(8) PTYPE is transmitted properly, set communication state to CXPI_STATE_TXRX_PENDING.

(9) If the master continues transmitting message frame after PTYPE transmission, go to (10) to request message frame transmission. If not, the master stays in transmission/reception pending state.

(10) Request message frame transmission. See **4.3.2 Request message frame transmission** for the software flowchart.

(11) If the current state is CXPI_STATE_TX_HDR_TX_RESP or CXPI_STATE_TX_HDR_RX_RESP, go to (12); else, go to (19).

(12) Check if INTR.RX_HEADER_PID_DONE flag is set. If yes, go to (13); else, go to (15).

(13) Check if INTR.TX_HEADER_ARB_LOST flag is set. If yes, this node has lost arbitration when attempting to transmit the header. Clear INTR.TX_HEADER_ARB_LOST and go to (14). If not, go to (6) to perform header reception operation.

(14) Change the arbitration state variable to indicate that arbitration is lost.
If arbitration is lost when transmitting both header and response, set arb_state = CXPI_ARB_LOST_TX_RESP.
If arbitration is lost when transmitting header and receiving response, set arb_state = CXPI_ARB_LOST_RX_RESP.
Then, go to (6) to perform the header reception operation.

(15) Check if INTR.TX_RESPONSE_DONE flag is set. If yes, go to (16); else, go to (17).

(16) Response transmission is done. Wait for hardware to confirm EOF after frame transmission, i.e., wait for INTR.TXRX_COMPLETE flag to be set by hardware.

(17) Check if INTR.RX_RESPONSE_DONE flag is set. If yes, go to (18); else, go to (23).

(18) Response reception is done. Read received response from the Rx FIFO and wait for hardware to confirm EOF after frame reception, i.e., wait for INTR.TXRX_COMPLETE flag to be set by hardware.

(19) Check if INTR.TX_RESPONSE_DONE flag is set. If yes, go to (20) else, go to (21).

(20) Response transmission is done. Wait for hardware to confirm EOF after frame response transmission, i.e., wait for INTR.TXRX_COMPLETE flag to be set by hardware.

(21) Check if INTR.RX_RESPONSE_DONE flag is set. If yes, go to (22); else, go to (23).

(22) Response reception is done. Read received response in RX FIFO and wait for hardware to confirm EOF after frame reception, i.e. wait for INTR.TXRX_COMPLETE flag to be set by hardware.

(23) Hardware confirmed EOF after frame ends without data length error. See **4.3.5 TXRX completion operation** for example of software flowchart.

**Example of CXPI controller operation**

### 4.3.3.1 Use case

This section explains an example of CXPI interrupt handle using the use case shown in **4.3.1.1 Use case** and **4.3.2.1 Use case**.

### 4.3.3.2 Configuration

**Table 15 CXPI interrupt handle functions**

| Function | | Description | Value |
|---|---|---|---|
| Cy_CXPI_Interrupt() | | Interrupt function for the CXPI channel | – |
| | stc_CXPI_CH_t* | Specifies the CXPI channel instance | CXPI_CH_INSTANCE = CY_CXPICH3_TYPE |
| | pstcCXPI | Specifies the pointer to CXPI instance register area | &cxpiCtx |
| Cy_CXPI_GetInterrupt MaskedStatus() | | Returns the status of all mask enabled interrupt causes | – |
| | pstcCXPI | Specifies the pointer to CXPI instance register area | pstcCXPI |
| | status | Reflects a bitwise AND between the INTR and INTR_MASK registers | – |
| Cy_CXPI_ClearInterrupt() | | Clears interrupt sources in the interrupt request register | – |
| | pstcCXPI | Specifies the pointer to CXPI instance register area | pstcCXPI |
| | status | Reflects the status of unmasked interrupt | – |

### 4.3.3.3 Example code

**Code Listing 17 Example of CXPI interrupt handle in configuration part**

```
void CxpiInterruptHandler(void)
{
  Cy_CXPI_Interrupt(CXPI_CH_INSTANCE, &cxpiCtx);
}
```

CXPI Interrupt Handle (See **Code Listing 18**)

**Code Listing 18 Example of CXPI interrupt handle in driver part**

```
cy_en_cxpi_status_t Cy_CXPI_Interrupt(volatile stc_CXPI_CH_t* pstcCXPI, stc_CXPI_context_t* pCxpiCtx)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    uint32_t maskedStatus = 0;
    Cy_CXPI_GetInterruptMaskedStatus(pstcCXPI, &maskedStatus);
    /* Clear accepted interrupt */
    Cy_CXPI_ClearInterrupt(pstcCXPI, maskedStatus);

    if((CXPI_INTR_TX_WAKEUP_DONE & maskedStatus) != 0ul) {
        /* Wakeup pulse transmitted */
        if(pCxpiCtx->pNotifyTrasmittedWakeupPulse != NULL) {
            pCxpiCtx->pNotifyTrasmittedWakeupPulse();
        }
    }
```

(1) Get INTR_MASKED (See **Code Listing 19**)

(2) Clear all accepted interrupt (See **Code Listing 20**)

## Example of CXPI controller operation

### Code Listing 18 Example of CXPI interrupt handle in driver part

```
    if((CXPI_INTR_RX_WAKEUP_DETECT & maskedStatus) != 0ul) {
        if(pCxpiCtx->pNotifyRXFallingEdgeDetected != NULL) {
            pCxpiCtx->pNotifyRXFallingEdgeDetected();
        }
    }

    if((CXPI_INTR_ALL_ERROR_MASK & maskedStatus) != 0ul) {
        /* There are some error */
        /* Handle error if needed */
        Cy_CXPI_SetCmd(pstcCXPI, 0x0);

        pCxpiCtx->retriesCount = 0ul;
        pCxpiCtx->arbState = CXPI_ARB_PENDING;
        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;

        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_TXRX_COMPLETE |
        CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TIMEOUT | CXPI_INTR_ALL_ERROR_MASK);
        pstcCXPI->unCTL0.stcField.u5IFS = 20u;
        Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_RX_HEADER_RX_RESPONSE | CXPI_CMD_IFS_WAIT);
        if(pCxpiCtx->pNotifyError != NULL) {
            (pCxpiCtx->pNotifyError)(CXPI_INTR_ALL_ERROR_MASK & maskedStatus);
        }
    } else if((CXPI_INTR_TIMEOUT & maskedStatus) != 0ul){
        /* Timeout */
        Cy_CXPI_ClearTxFIFO(pstcCXPI);
        Cy_CXPI_ClearRxFIFO(pstcCXPI);

        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_TXRX_COMPLETE |
        CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TIMEOUT | CXPI_INTR_ALL_ERROR_MASK);
        pCxpiCtx->arbState = CXPI_ARB_PENDING;
        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
        Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_RX_HEADER_RX_RESPONSE);

    } else {
        if (pCxpiCtx->state == CXPI_STATE_TXRX_PENDING){
            //Received header
            HeaderReceptionOperation(pstcCXPI, pCxpiCtx);
        } else if (pCxpiCtx->state == CXPI_STATE_TX_PTYPE){
            //PTYPE is sent
            pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
            //notify user
            if (pCxpiCtx->pNotifySentPTYPE != NULL){
                pCxpiCtx->pNotifySentPTYPE();
            }
        } else if ((pCxpiCtx->state == CXPI_STATE_TX_HDR_TX_RESP) || (pCxpiCtx->state ==
CXPI_STATE_TX_HDR_RX_RESP)){
            if((CXPI_INTR_RX_HEADER_PID_DONE & maskedStatus) != 0ul){
                // received header while attempting to transmit header
                // check if arbitration lost
                if (pstcCXPI->unINTR.stcField.u1TX_HEADER_ARB_LOST == 1ul){
                    //arbitration lost
                    //clear INTR.TX_HEADER_ARB_LOST
                    pstcCXPI->unINTR.stcField.u1TX_HEADER_ARB_LOST = 1ul;
                    //set arbitration state
                    if(pCxpiCtx->state == CXPI_STATE_TX_HDR_RX_RESP) {
                        pCxpiCtx->arbState = CXPI_ARB_LOST_RX_RESP;
                    } else if(pCxpiCtx->state == CXPI_STATE_TX_HDR_TX_RESP) {
                        pCxpiCtx->arbState = CXPI_ARB_LOST_TX_RESP;
                    }
                }
                HeaderReceptionOperation(pstcCXPI, pCxpiCtx);
            } else if ((CXPI_INTR_TX_RESPONSE_DONE & maskedStatus) != 0ul){
                // response transmission is done
                // do nothing but wait for INTR.TXRX_COMPLETE
            } else if((CXPI_INTR_RX_RESPONSE_DONE & maskedStatus) != 0ul) {
                // response received
                RXResponseDoneHandler(pstcCXPI, pCxpiCtx);
            } else if((CXPI_INTR_TXRX_COMPLETE & maskedStatus) != 0ul) {
                TXRXCompletionOperation(pstcCXPI, pCxpiCtx);
            }

        } else if ((pCxpiCtx->state == CXPI_STATE_RX_HDR_TX_RESP) || (pCxpiCtx->state ==
CXPI_STATE_RX_HDR_RX_RESP)){
            if ((CXPI_INTR_TX_RESPONSE_DONE & maskedStatus) != 0ul){
```

(3) Transmitting error

(4) Error Handler

(3) Timeout

(5) cxpi_state = CXPI_STATE_TXRX_PENDING

(6) Header Reception Operation (See **Code Listing 21**)

(7) cxpi_state = CXPI_STATE_TX_PTYPE

(8) cxpi_state = CXPI_STATE_TXRX_PENDING

(9) Continue transmitting message after PTYPE

(10) Request message frame transmission

(11) cxpi_state = CXPI_STATE_TX_HDR_TX_RESP or CXPI_STATE_TX_HDR_RX_RESP

(12) INTR.RX_HEADER_PID_DONE == 1

(13) INTR.TX_HEADER_ARB_LOST == 1

(14) arb_state = CXPI_ARB_LOST_TX_RESP or CXPI_ARB_LOST_RX_RESP

(15) INTR.TX_RESP_DONE == 1

(16) Transmission of response is finished. Wait for EOF

(17) INTR.RX_RESP_DONE == 1

(18) Reception of response is finished. Read received response. Wait for EOF.

(19) INTR.TX_RESPONSE_DONE == 1

## Example of CXPI controller operation

### Code Listing 18 Example of CXPI interrupt handle in driver part

```
                // response transmission is done
                // do nothing but wait for INTR.TXRX_COMPLETE
            } else if((CXPI_INTR_RX_RESPONSE_DONE & maskedStatus) != 0ul) {
                // response received
                RXResponseDoneHandler(pstcCXPI, pCxpiCtx);
            } else if((CXPI_INTR_TXRX_COMPLETE & maskedStatus) != 0ul) {
                TXRXCompletionOperation(pstcCXPI, pCxpiCtx);
            }
        }
    }
    return ret;
}
```

(20) Transmission of response is finished. Wait for EOF

(21) INTR.RX_RESPONSE_DONE == 1

(22) Reception of response is finished. Read received response. Wait for EOF.

(23) TXRX completion operation (See **Code Listing 24**)

### Code Listing 19 Example for getting the status of all mask-enabled interrupt causes in the driver part

```
cy_en_cxpi_status_t Cy_CXPI_GetInterruptMaskedStatus(volatile stc_CXPI_CH_t* pstcCXPI, uint32_t*
status)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    *status = pstcCXPI->unINTR_MASKED.u32Register;
    return ret;
}
```

Get INTR_MASKED

### Code Listing 20 Example of clearing all accepted interrupts in driver part

```
cy_en_cxpi_status_t Cy_CXPI_ClearInterrupt(volatile stc_CXPI_CH_t* pstcCXPI, uint32_t mask)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    pstcCXPI->unINTR.u32Register = mask;
    return ret;
}
```

Clear all accepted interrupt in the interrupt request register

### 4.3.4 Header reception operation

**Figure 10** shows an example flowchart for header reception.

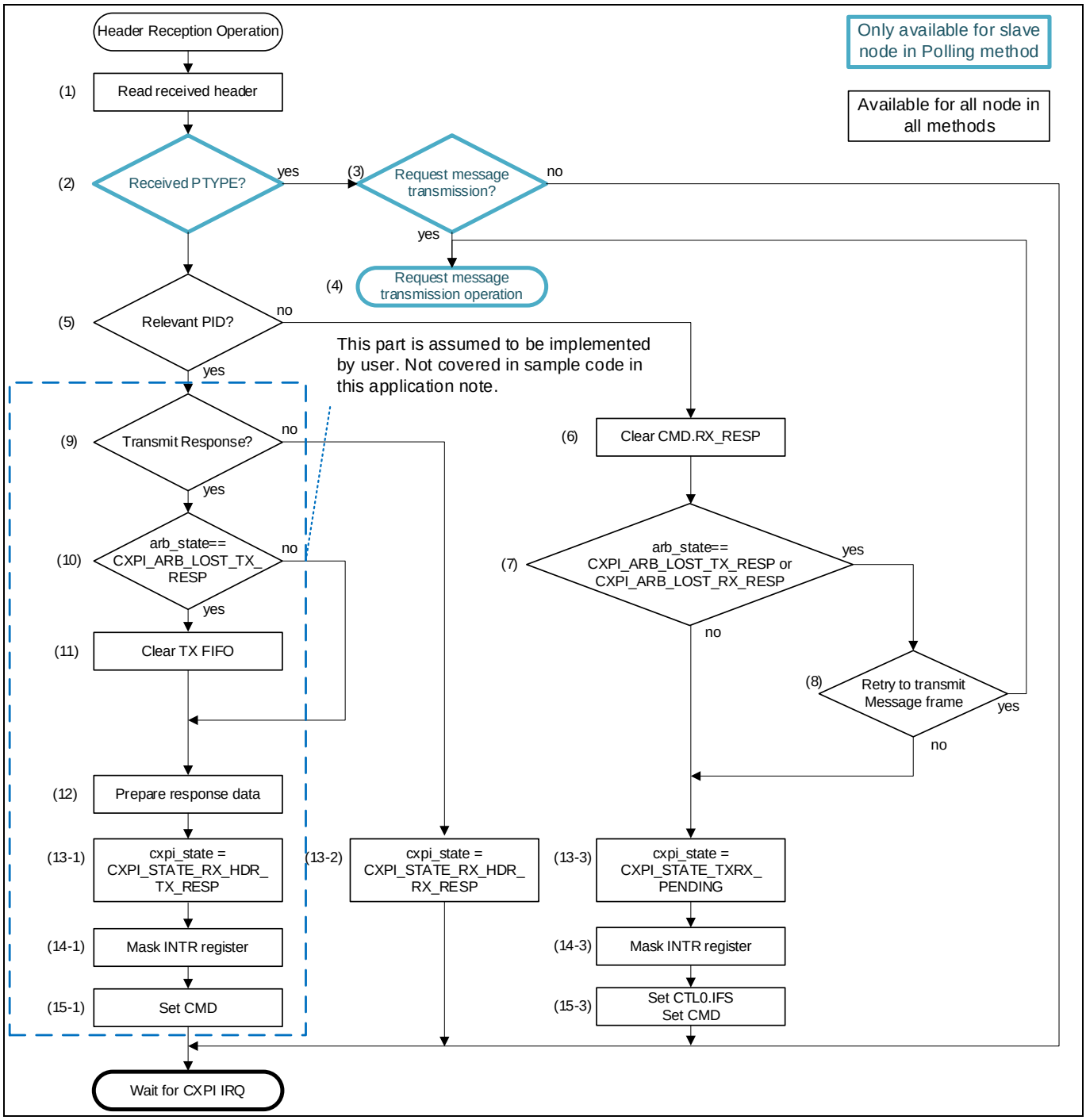


**Figure 10 Header reception flowchart**

**Example of CXPI controller operation**

(1) Read the received header by reading the RXPID_FI.PID field:
RXPID_FI.PID[6:0] shows received PID/PTYPE.
RXPID_FI.PID[7] shows received odd parity bit.

(2) If PTYPE is received, go to (3); else if normal PID is received, go to (5).

(3) If there is a request message for frame transmission, go to (4); if no, continue waiting for header/response.

(4) Request message frame transmission. The flowchart is shown in **4.3.2 Request message frame transmission**.
Note that IFS checking is not required before transmitting the PID after receiving PTYPE.

(5) Check whether the received PID is relevant.
If yes, go to (8) to check whether the transmission of response is needed.
If no, go to (6) to clear CMD.RX_RESP.

(6) Clear CMD.RX_RESP to stop receiving response.

(7) Check whether PID arbitration is lost by checking the arbitration state variable.
If yes, go to (8) to request message frame re-transmission.
If no, go to (13–3) to move the hardware back to transmission/reception pending state.

(8) At this stage, arbitration is lost and the “won” PID is received, but the received PID can be ignored. Check if PID that lost arbitration needs to be re-transmitted. If yes, go to (4) to request message frame transmission; else, go to (13–3).

(9) If the received PID indicates that this node should transmit response, go to (10); else, go to (13–2).

(10) Check whether PID arbitration is lost during the attempt to transmit both header and response. If yes, CMD.TX_RESPONSE is still set to ‘1’ and Tx FIFO is still filled by old response data; therefore, go to (11). If no, go to (12).

(11) Clear TX FIFO:
Configure TX_FIFO_CTL.CLEAR to ‘1’ and followed by ‘0’ to perform clearing TX FIFO.

(12) Prepare the response to be transmitted by writing each byte of response to TX_FIFO_WR.DATA[7:0].

(13) Modify the value of the communication state variable ‘cxpi_state’ depending on the requested transmission.

(14) Set the INTR_MASK register to enable the event interrupt according to the transmission:

(14-1) CXPI_STATE_RX_HDR_TX_RESP
INTR_MASK.TX_RESPONSE_DONE = ‘1’
INTR_MASK.TXRX_COMPLETE = ‘1’

(14-3) CXPI_STATE_TXRX_PENDING case
INTR_MASK.RX_HEADER_PID_DONE = ‘1’
INTR_MASK.RX_RESPONSE_DONE = ‘1’
INTR_MASK.TXRX_COMPLETE = ‘1’

**Example of CXPI controller operation**

(15) Set the command sequence according to the state of the requested transmission:

(15-1) CXPI_STATE_RX_HDR_TX_RESP
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {‘0’, ‘0’, ‘0’, ‘1’, ‘0’}.

(15-3) CXPI_STATE_TXRX_PENDING
Configure CLT0.IFS = ‘20’.
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {‘1’, ‘0’, ‘1’, ‘0’, ‘1’} to enable header/frame reception.
CMD.IFS_WAIT is set to ‘1’to make the hardware wait for 20Tbit of logic ‘1’ before enabling header/frame reception.

### 4.3.4.1 Use case

This section explains an example of header reception operation using the use case shown in **4.3.1.1 Use case** and **4.3.2.1 Use case**.

### 4.3.4.2 Configuration

**Table 16 CXPI header reception operation functions in SDL**

| Function | | Description | Value |
|---|---|---|---|
| Cy_CXPI_GetPID() | | Gets the header protected identifier (PID) for header reception operation | – |
| | pstcCXPI | Specifies the pointer to the CXPI instance register area | pstcCXPI |
| | id | Received frame ID | – |
| | parity | Parity bit of the received frame ID | – |

### 4.3.4.3 Example code

**Code Listing 21 Example of header reception operation in driver part**

```
void HeaderReceptionOperation(volatile stc_CXPI_CH_t* pstcCXPI, stc_CXPI_context_t* pCxpiCtx){
    bool respond = false;
    bool retransmit = false;
    /* PID received */
    /* Check received PID and correspond if need */
    uint8_t id, parity;
    Cy_CXPI_GetPID(pstcCXPI, &id, &parity);

    if(id == CXPI_PTYPE) {
        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
        if(pCxpiCtx->pNotifyReceivedPTYPE != NULL) {
            pCxpiCtx->pNotifyReceivedPTYPE();
        }
    } else {
        // read received id
        pCxpiCtx->receivedFrame.id = id;
        /* Check if this node should respond to the received PID */
        for (uint32_t tempIdx = 0; tempIdx < pCxpiCtx->numberOfAcceptedPID; tempIdx++){
            if (*(pCxpiCtx->pAcceptedPIDList + tempIdx) == id){
                respond = true;
            }
        }
        if (needRespond){
            /* Should receive or transmit response */
            pCxpiCtx->state = CXPI_STATE_RX_HDR_RX_RESP;
            if(pCxpiCtx->pNotifyReceivedPID != NULL) {
                pCxpiCtx->pNotifyReceivedPID(id);
            }
```

(1) Read received header (See **Code Listing 22**)
(2) Received PTYPE?
(3) Request message transmission?
(4) Request message transmission operation (See **Code Listing 23**)
(5) Relevant PID?
(13-2) cxpi_state = CXPI_STATE_RX_HDR_RX_RESP

## Example of CXPI controller operation

### Code Listing 21 Example of header reception operation in driver part

```
        } else {
            /* Ignore the received PID */
            /* Clear CMD.RX_RESPONSE to stop receiving response */
            pstcCXPI->unCMD.stcField.u1RX_RESPONSE = 0;
            /* Check if arbitration lost and retransmission is needed or not */
            if ((pCxpiCtx->arbState == CXPI_ARB_LOST_TX_RESP) || (pCxpiCtx->arbState ==
CXPI_ARB_LOST_RX_RESP)){
                pCxpiCtx->retriesCount++;
                if (pCxpiCtx->retriesCount <= pCxpiCtx->maximumRetries){
                    retransmit = true;
                } else {
                    pCxpiCtx->retriesCount = 0;
                }
            }
            if (retransmit){
                /* Need to perform retransmission because of arbitration loss */
                /* Move machine state back to CXPI_STATE_TXRX_PENDING */
                pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
                pstcCXPI->unCTL0.stcField.u5IFS = 20ul;
                if(pCxpiCtx->arbState == CXPI_ARB_LOST_TX_RESP) {
                    Cy_CXPI_SendPID(pstcCXPI, pCxpiCtx, pCxpiCtx->transmittingFrame.id, true, pCxpiCtx-
>pTransmittingFrameResponse, true);
                } else if(pCxpiCtx->arbState == CXPI_ARB_LOST_RX_RESP) {
                    Cy_CXPI_SendPID(pstcCXPI, pCxpiCtx, pCxpiCtx->transmittingFrame.id, false, NULL,
true);
                }
            } else {
            /* Enable message frame reception and move machine state back to CXPI_STATE_TXRX_PENDING */
                pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
                Cy_CXPI_SetTriggerLevelRxFIFO(pstcCXPI, (CXPI_RX_FIFO_SIZE / 2u));
                pstcCXPI->unCTL0.stcField.u5IFS = 20ul;
                Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE |
                CXPI_INTR_TXRX_COMPLETE | CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_RX_FIFO_TRIGGER |
                CXPI_INTR_TIMEOUT | CXPI_INTR_ALL_ERROR_MASK);
                Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_IFS_WAIT|CXPI_CMD_RX_HEADER_RX_RESPONSE);

                if ((pCxpiCtx->arbState == CXPI_ARB_LOST_TX_RESP) || (pCxpiCtx->arbState ==
CXPI_ARB_LOST_RX_RESP)){
                    pCxpiCtx->arbState = CXPI_ARB_PENDING;
                    if(pCxpiCtx->pNotifyArbitrationLost != NULL) {
                        pCxpiCtx->pNotifyArbitrationLost();
                    }
                }
            }
        }
    }
}
```

(6) Clear CMD.RX_RESP

(7) CMD.RX_RESParb_state == CXPI_ARB_LOST_TX_RESP or CXPI_ARB_LOST_RX_RESP

(8) Retry to transmit message frame?

(13–3) cxpi_state = CXPI_STATE_TXRX_PENDING

(14–3) Mask INTR (**Code Listing 8**)

(15–3) Set CMD (**Code Listing 9**)

### Code Listing 22 Example of getting the header PID in driver part

```
cy_en_cxpi_status_t Cy_CXPI_GetPID(volatile stc_CXPI_CH_t* pstcCXPI, uint8_t* id, uint8_t* parity)
{
    cy_en_cxpi_status_t ret = CY_CXPI_SUCCESS;
    if(NULL == pstcCXPI) {
        return CY_CXPI_BAD_PARAM;
    }
    uint8_t temp = pstcCXPI->unRXPID_FI.stcField.u8PID;
    *id = temp & 0x7F;
    *parity = temp >> 7;
    return ret;
}
```

Read received header PID

**Example of CXPI controller operation**

**Code Listing 23 Example of requesting message transmission operation for slave node in polling method**

```
void CxpiNotifyReceivedPTYPE(void){
    if (slave_tx_hdr_tx_resp){
        Cy_CXPI_SendPID(CXPI_CH_INSTANCE, &cxpiCtx, eventFrameCtx[0].id,
true,&eventFrameCtx[0].response, false);
        slave_tx_hdr_tx_resp = false;
    } else if (slave_tx_hdr_rx_resp){
        Cy_CXPI_SendPID(CXPI_CH_INSTANCE, &cxpiCtx, eventFrameCtx[1].id, false, NULL, false);
        slave_tx_hdr_rx_resp = false;
    }
}
```

Request message transmission operation (See **Code Listing 11**)

## 4.3.5 TXRX completion operation

**Figure 11** shows an example flowchart for TXRX completion, indicating a finished message transfer independently if the transfer had an error.

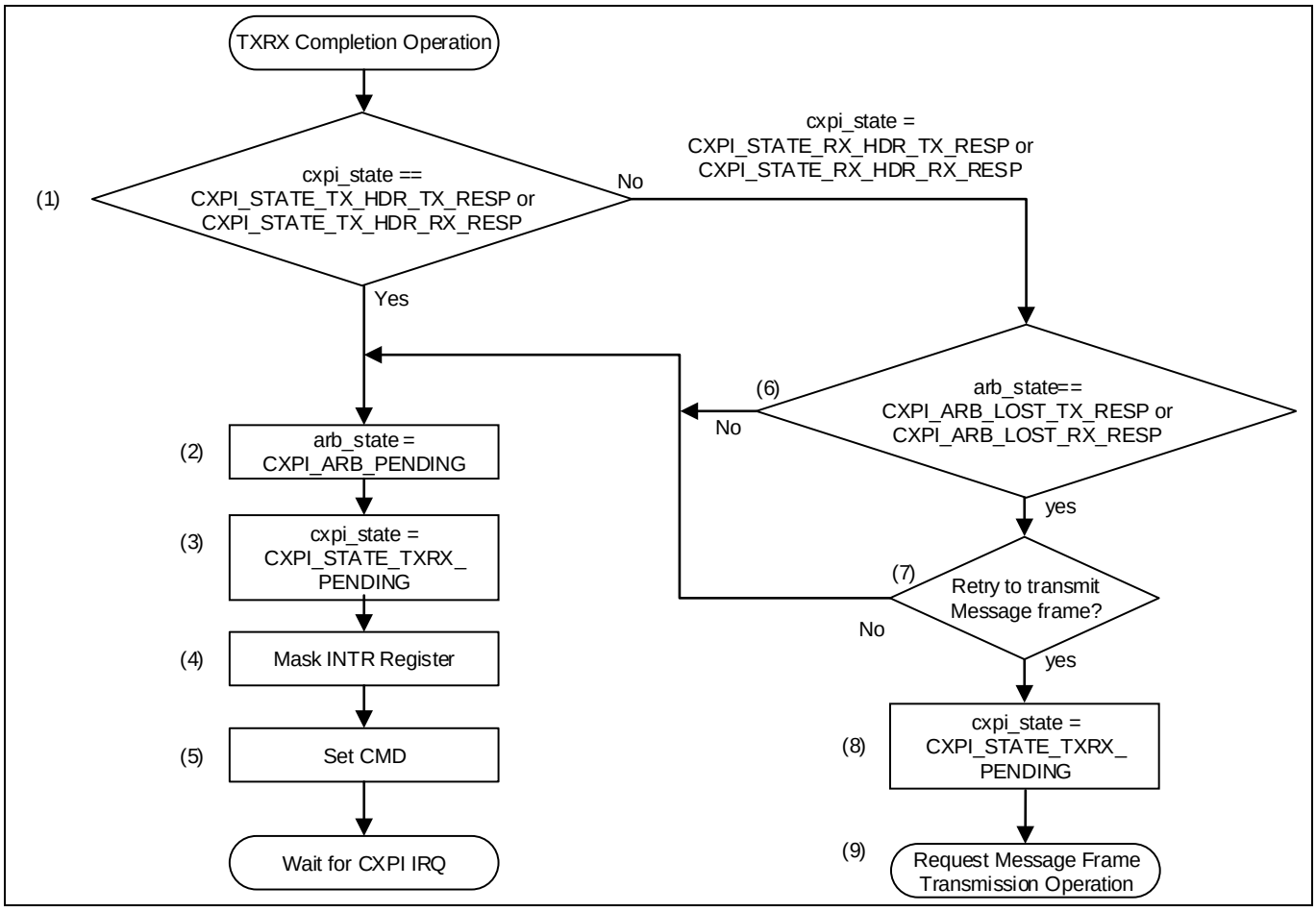


**Figure 11 TXRX completion flowchart**

(1) If the current state is CXPI_STATE_TX_HDR_TX_RESP or CXPI_STATE_TX_HDR_RX_RESP, go to (2). Else, the current state is CXPI_STATE_RX_HDR_TX_RESP or CXPI_STATE_RX_HDR_RX_RESP, go to (6).

(2) Reset arb_state to CXPI_ARB_PENDING.

(3) Reset cxpi_state to CXPI_STATE_TXRX_PENDING.

**Example of CXPI controller operation**

(4) Mask the INTR register:
INTR_MASK.RX_HEADER_PID_DONE = '1'
INTR_MASK.RX_RESPONSE_DONE = '1'
INTR_MASK.TXRX_COMPLETE = '1'

(5) Set the CMD register:
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'0', '0', '1', '0', '1'} to enable header/frame reception.

(6) Check whether arbitration is lost by checking arb_state. If yes, go to (8); else, go to (2).

(7) Arbitration lost. Check whether message frame retransmission is required. If yes, go to (9) to request message frame transmission. If not, go to (2).

(8) Set cxpi_state to CXPI_STATE_TXRX_PENDING.

(9) Request message frame transmission. See **4.3.2 Request message frame transmission** for the software flowchart.

### 4.3.5.1 Use case

This section explains an example of TXRX completion operation using the use case shown in **4.3.1.1 Use case** and **4.3.2.1 Use case**.

### 4.3.5.2 Configuration

The functions of the driver part in SDL for TXRX completion operation are same as **Table 10**.

### 4.3.5.3 Example code

**Code Listing 24 Example of TXRX completion in driver part**

```
void TXRXCompletionOperation(volatile stc_CXPI_CH_t* pstcCXPI, stc_CXPI_context_t* pCxpiCtx){
    bool retransmit = false;
    /* Clear pending command */
    Cy_CXPI_SetCmd(pstcCXPI, 0ul);

    if((pCxpiCtx->state == CXPI_STATE_TX_HDR_TX_RESP) || (pCxpiCtx->state ==
CXPI_STATE_TX_HDR_RX_RESP)){
        retransmit = false;
    } else{
        if ((pCxpiCtx->arbState == CXPI_ARB_LOST_TX_RESP) || (pCxpiCtx->arbState ==
CXPI_ARB_LOST_RX_RESP)){
            pCxpiCtx->retriesCount++;
            if (pCxpiCtx->retriesCount <= pCxpiCtx->maximumRetries){
                retransmit = true;
            } else {
                retransmit = false;
                if(pCxpiCtx->pNotifyArbitrationLost != NULL) {
                    pCxpiCtx->pNotifyArbitrationLost();
                }
            }
        } else{
            retransmit = false;
        }
    }

    if((pCxpiCtx->state == CXPI_STATE_TX_HDR_RX_RESP) || (pCxpiCtx->state ==
CXPI_STATE_RX_HDR_RX_RESP)){
        if(pCxpiCtx->pNotifyReceivedResponse != NULL) {
            pCxpiCtx->pNotifyReceivedResponse(&pCxpiCtx->receivedFrame);
        }
    }

    if(retransmit == false){
        pCxpiCtx->retriesCount = 0ul;
        pCxpiCtx->arbState = CXPI_ARB_PENDING;
```

(1) cxpi_state = CXPI_STATE_TX_HDR_TX_RESP or CXPI_STATE_TX_HDR_RX_RESP

(6) arb_state = CXPI_ARB_LOST_TX_RESP or CXPI_ARB_LOST_RX_RESP

(7) Retry to transmit message frame?

(2) arb_state = CXPI_ARB_PENDING

### Example of CXPI controller operation

**Code Listing 24 Example of TXRX completion in driver part**

```
        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_TXRX_COMPLETE |
            CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TIMEOUT | CXPI_INTR_ALL_ERROR_MASK);
        Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_RX_HEADER_RX_RESPONSE);
    } else{
        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;
        pstcCXPI->unCTL0.stcField.u5IFS = 10ul;
        if(pCxpiCtx->arbState == CXPI_ARB_LOST_TX_RESP) {
            Cy_CXPI_SendPID(pstcCXPI, pCxpiCtx, pCxpiCtx->transmittingFrame.id, true, pCxpiCtx-
>pTransmittingFrameResponse, true);
        } else if(pCxpiCtx->arbState == CXPI_ARB_LOST_RX_RESP) {
            Cy_CXPI_SendPID(pstcCXPI, pCxpiCtx, pCxpiCtx->transmittingFrame.id, false, NULL, true);
        }
    }
}
```

(3) cxpi_state = CXPI_STATE_TXRX_PENDING

(4) Mask INTR register (**Code Listing 8**)

(5) Set CMD (**Code Listing 9**)

(8) cxpi_state = CXPI_STATE_TXRX_RENDING

(9) Request message frame transmission operation (See **Code Listing 11**)

## 4.3.6 Error handler

**Figure 12** shows an example of error handling implementation.

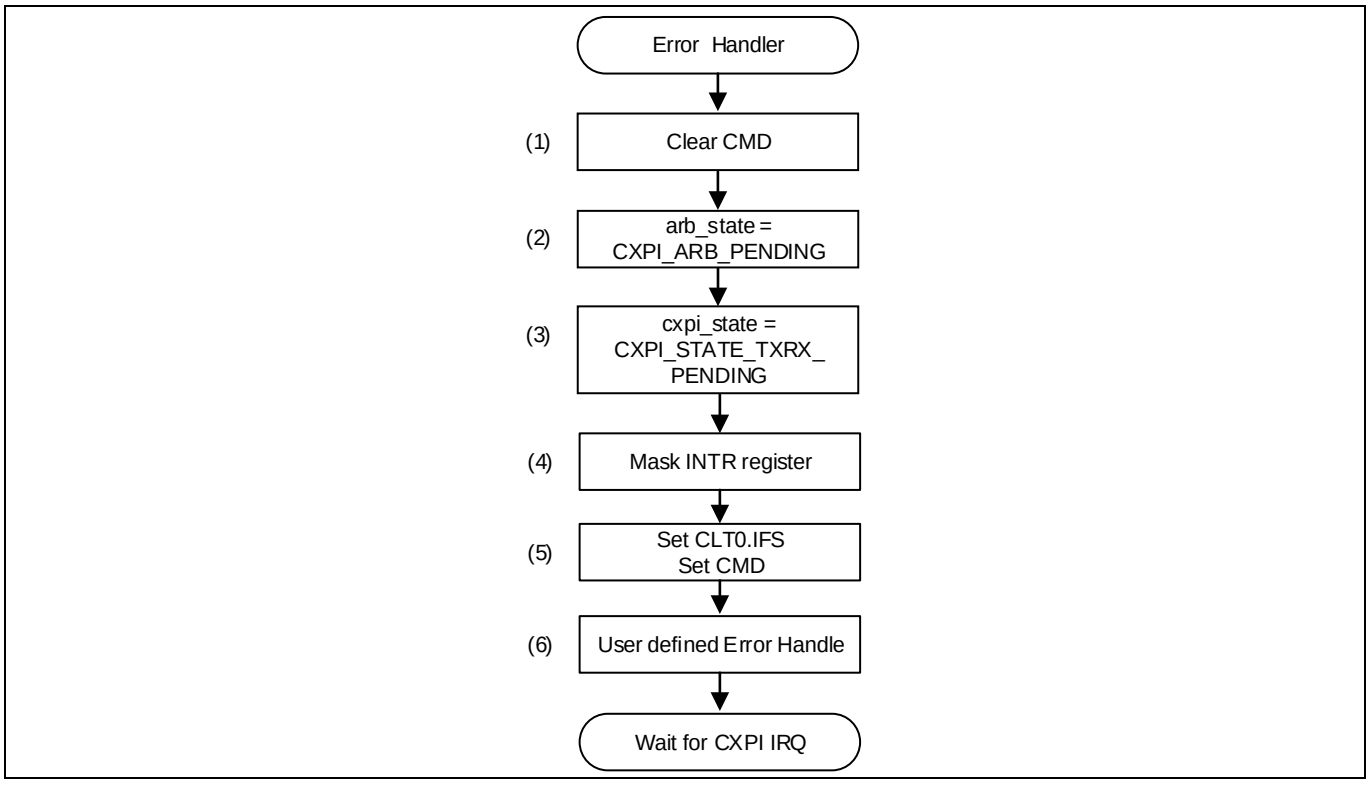


**Figure 12 Flowchart for error handling**

(1) Clear the CMD register to stop requesting transmission/reception.

(2) Reset the arbitration state variable:
arb_state = CXPI_ARB_PENDING

(3) Reset the communication state variable:
cxpi_state = CXPI_STATE_TXRX_PENDING

**Example of CXPI controller operation**

(4) Mask the INTR register by setting INTR_MASK:
INTR_MASK.RX_HEADER_PID_DONE = '1'
INTR_MASK.RX_RESPONSE_DONE = '1'
INTR_MASK.TXRX_COMPLETE = '1'

(5) Set CLT0.IFS, and then set then CMD register to enable header/response reception:
Configure {C.IFS, C.TXH, C.RXH, C.TXR, C.RXR} = {'0', '0', '1', '0', '1'} to enable header/frame reception.

(6) Execute the error handler defined by user.

### 4.3.6.1 Use case

This section explains an example of error handling using the use case shown in **4.3.1.1 Use case** and **4.3.2.1 Use case**.

### 4.3.6.2 Configuration

The parameters of the driver part in SDL for error handling are the same as in **Table 10**.

### 4.3.6.3 Example code

**Code Listing 25 Example of error handling in driver part**

```
cy_en_cxpi_status_t Cy_CXPI_Interrupt(volatile stc_CXPI_CH_t* pstcCXPI, stc_CXPI_context_t* pCxpiCtx)
{
:
    if((CXPI_INTR_ALL_ERROR_MASK & maskedStatus) != 0ul) {
        /* There are some error */
        /* Handle error if needed */
        Cy_CXPI_SetCmd(pstcCXPI, 0x0);

        pCxpiCtx->retriesCount = 0ul;
        pCxpiCtx->arbState = CXPI_ARB_PENDING;
        pCxpiCtx->state = CXPI_STATE_TXRX_PENDING;

        Cy_CXPI_SetInterruptMask(pstcCXPI, CXPI_INTR_RX_HEADER_PID_DONE | CXPI_INTR_TXRX_COMPLETE |
        CXPI_INTR_RX_RESPONSE_DONE | CXPI_INTR_TIMEOUT | CXPI_INTR_ALL_ERROR_MASK);
        pstcCXPI->unCTL0.stcField.u5IFS = 20ul;
        Cy_CXPI_SetCmd(pstcCXPI, CXPI_CMD_RX_HEADER_RX_RESPONSE | CXPI_CMD_IFS_WAIT);
        :
        }
:
}
```

(1) Clear CMD (**Code Listing 9**)
(2) arb_state = CXPI_ARB_PENDING
(3) cxpi_state = CXPI_STATE_TXRX_PENDING
(4) Mask INTR (**Code Listing 8**)
(5) Set CLT0.IFS
Set CMD (**Code Listing 9**)
(6) Execute the error handler defined by user

# 5 Glossary

**Table 17 Glossary**

| Terms | Description |
|---|---|
| AUTOSAR | AUTomotive Open System ARchitecture |
| CRC | Cyclic Redundancy Check field |
| CXPI | Clock Extension Peripheral Interface |
| GPIO | General Purpose Input/Output |
| Header | Consists of PID field and PTYPE field transmitted by the master and the slave. PTYPE can only be transmitted by the master in polling method. See the CXPI Message Frame Format section in the CXPI chapter of the **Architecture TRM** for details. |
| IRQ | Interrupt ReQuest |
| NRZ mode | Non-Return-to-Zero mode. CXPI controller interfacing with external transceiver chip that has PWM encoder/decoder logic |
| PERI clock | PERipheral Interconnect clock |
| PID | Protected Identifier field |
| PWM mode | CXPI controller interfacing with external driver chip that level shifts the 3.3 V or 5 V signaling to 12 V CXPI signaling without changing the encoding of the signal |
| PTYPE | The 8-bit Protected Type field (PTYPE), only applicable in the Polling Method, corresponds to a PID field with the identifier value 0x00 (0x80 included parity bit). |
| Response | Consists of frame information, data fields and CRC field, transmitted by the master and the slave. See the CXPI Message Frame Format section in the CXPI chapter of the **Architecture TRM** for details. |

# 6 Related documents

The following are the Traveo II family series datasheets and Technical Reference Manuals. Contact **Technical Support** to obtain these documents.

- Device datasheet
  - CYT2B9 Datasheet 32-Bit Arm® Cortex®-M4F Microcontroller Traveo™ II Family
  - CYT3DL Datasheet 32-Bit Arm® Cortex®-M7 Microcontroller Traveo™ II Family
  - CYT4DN Datasheet 32-Bit Arm® Cortex®-M7 Microcontroller Traveo™ II Family
- Body controller entry family
  - Traveo™ II Automotive Body Controller Entry Family Architecture Technical Reference Manual (TRM)
  - Traveo™ II Automotive Body Controller Entry Registers Technical Reference Manual (TRM) for CYT2B9
- Cluster 2D family
  - Traveo™ II Automotive Cluster 2D Family Architecture Technical Reference Manual (TRM)
  - Traveo™ II Automotive Cluster 2D Registers Technical Reference Manual (TRM) for CYT3DL
  - Traveo™ II Automotive Cluster 2D Registers Technical Reference Manual (TRM) for CYT4DN

# 7 Other references

A sample driver library (SDL) including startup as sample software to access various peripherals is provided. SDL also serves as a reference to customers for drivers that are not covered by official AUTOSAR products. The SDL cannot be used for production purposes because it does not qualify to automotive standards. Code snippets in this application note are part of the SDL. Contact **Technical Support** to obtain the SDL.

# Revision history

| Document version | Date of release | Description of changes |
|---|---|---|
| ** | 2019-07-10 | New application note. |
| *A | 2019-11-08 | Added target parts number (CYT4D series) |
| *B | 2020-06-25 | Changed target parts number (CYT2/CYT3/CYT4 series) |
| *C | 2021-05-25 | Added code examples using SDL.<br>Updated to Infineon template.<br>Completing Sunset Review. |

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

**Edition 2021-05-25**
**Published by**
**Infineon Technologies AG**
**81726 Munich, Germany**



**Do you have a question about this document?**
**Go to www.cypress.com/support**

**Document reference**
**002-24283 Rev. *C**

**IMPORTANT NOTICE**
The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. 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) with respect to any and all information given in this application note.

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.