UART Connection Orientated Communication Protocol

This document describes the software of the UART communication protocol for the EIVE PLOC project.

Content

[1. Introduction 5](#_Toc43395811)

[1.1. S 5](#_Toc43395812)

[1.2. Overview 5](#_Toc43395813)

[2. Packet Structure 6](#_Toc43395814)

[2.1. Header 6](#_Toc43395815)

[2.1.1. Subsystem-ID 6](#_Toc43395816)

[2.1.2. CRC-8 7](#_Toc43395817)

[2.1.3. Data Size 8](#_Toc43395818)

[2.1.4. Flags 9](#_Toc43395819)

[3. Flow Charts 10](#_Toc43395820)

[3.1. Receiving data 10](#_Toc43395821)

[3.2. Transmitting Data 11](#_Toc43395823)

[4. Functions Overview 12](#_Toc43395824)

[4.1. Header *UART\_EIVE\_Protocol* for sending protocol 12](#_Toc43395825)

[int UART\_Send\_Data(uint8\_t ID, uint8\_t \*databytes, int dataLength); 12](#_Toc43395826)

[4.2. Header UART\_EIVE\_Protocol for receiving protocol 14](#_Toc43395827)

[4.3. Header UART\_IO 18](#_Toc43395828)

[int Initialize\_UART\_Device(u16 DeviceID); 18](#_Toc43395829)

[4.4. Used functions provided by Xilinx (Ref: xxx) 19](#_Toc43395830)

# **Introduction**

# **S**

# **Overview**

# **Packet Structure**

The FIFO of the UART controller present on the PS has a maximum size of 64 bytes, meaning 32 bytes per direction.

Since the FIFO allowed the packet size to be a maximum of 32 bytes, only a few bytes could be used for the header, so only the most important information could be included there. Here 4 bytes were used for the header, so that still 28 bytes of the packet can be used for the actual user data.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Figure 1: Packet structure

# **Header**

It is important for the other partner to know to which subsystem the commands are addressed or from which subsystem the data comes, whether the transmission of the packet was without errors and how many valid bytes are in the packet. Various status information about the connection-oriented mode of operation is also necessary.

The following information is transmitted in the header of the packets:

* Subsystem-ID
* CRC-8
* Data Size
* Flags

# **Subsystem-ID**

Two identification numbers are used for each subsystem, one for TCs and one for TMs. This way the respective other side can be informed who is to be addressed or from whom the user data comes.

Currently, the following subsystems each have two IDs:

* Camera
  + TC: 0b00000000 (hex: 00, dec: 0)
  + TM: 0b11110000 (hex: F0, dec: 240)
* BRAM
  + TC: 0b00000101 (hex: 05, dec: 5)
  + TM: 0b11110101 (hex: F5, dec: 245)
* Downlink
  + TC: 0b00001001 (hex: 09, dec: 9)
  + TM: 0b11111001 (hex: F9, dec: 249)
* UART
  + TC: 0b00001010 (hex: 0A, dec: 10)
  + TM: 0b11111010 (hex: FA, dec: 250)
* CPU
  + TC: 0b00001111 (hex: 0F, dec: 15)
  + TM: 0b11111111 (hex: FF, dec: 255)
* DAC
  + TC: 0b00000110 (hex: 06, dec: 6)
  + TM: 0b11110110 (hex: F6, dec: 246)

# **CRC-8**

As error detection for the packets a Cyclic Redundancy Check algorithm was taken.

Since only 1 byte of the header is to be used for the CRC value, a CRC-8 algorithm with the name CRC8\_SAE\_J1850\_ZERO was used in the MSB version. This algorithm is characterized by a generator polynomial 0x1D and an initial and final XOR value of 0x00.

To check the correct course of the packets, the CRC-8 algorithm was slightly modified instead of a sequence number. The last received or last sent value is taken as the new initial value of the CRC calculation in the modified mode of operation, so that the correct sequence can be checked.

The following is a sample calculation of the CRC algorithm:

* Example byte: 0x10 (0b00010000)
* Example *last\_crc\_sent*: 0x01 (0b00000001)

|  |  |  |  |
| --- | --- | --- | --- |
| **Calculation** | | **Explanation** | |
| XOR  = | 0b00010000  0b00000001  **0b00010001** | | START:   * + - Init XOR operation with *last\_crc\_sent* |
| <- | 0b00010001  **0b00100010** | | 1. Step: Left bit == 1?   * No: only shift to left |
| <- | 0b00100010  **0b01000100** | | 2. Step: Left bit == 1?   * No: only shift to left |
| <- | 0b01000100  **0b10001000** | | 3. Step: Left bit == 1?   * No: only shift to left |
| XOR  = | 0b10001000  **0b00010000**  0b00011101  **0b00001101** | | 4. Step: Left bit == 1?   * + - Yes: * shift to left * XOR with generator polynom |
| <- | 0b00001101  **0b00011010** | | 5. Step: Left bit == 1?   * No: only shift to left |
| <- | 0b00011010  **0b00110100** | | 6. Step: Left bit == 1?   * No: only shift to left |
| <- | 0b00110100  **0b01101000** | | 7. Step: Left bit == 1?   * No: only shift to left |
| <- | 0b01101000  **0b11010000** | | 8. Step: Left bit == 1?   * No: only shift to left |
| XOR | 0b11010000  0b00000000  **0b11010000** | | END:   * XOR with *final\_XOR\_value (0x00)* |

* Calculated CRC value: 0xD0

# **Data Size**

For commands or user data that are either smaller than or not an exact multiple of 28 bytes, it is necessary to specify the number of valid bytes in the data field. Therefore, the valid size of the data field is also passed in the header as additional information.

As example a command of the length 1.3kByte:

Here you can see that in the last packet only 12 bytes are valid instead of 28 bytes.

# **Flags**

For various status information that can be signaled with one bit, one byte is occupied for so-called flags. Currently only 6 of these are in use.

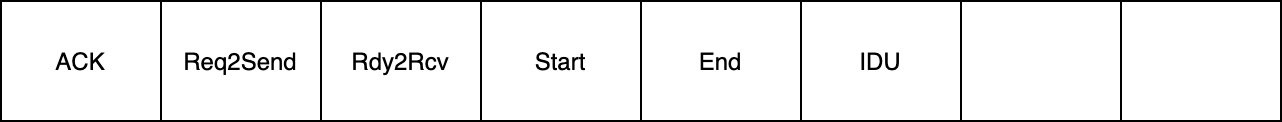


Figure :

Figure 2: Order of the flags

The flags are required for establishing the connection, they also signal the first and the last packet to be sent or the positive or negative acknowledgement of the received packet. In the following the flags are explained:

* **ACK:** This bit confirms the correctness of the last received packet. It is obtained using the CRC-8 algorithm.
* **Request to send (Req2Send):** The bit is set if one of the two devices wants to establish a connection for transmitting payload data.
* **Ready to Receive (Rdy2Rcv):** The bit is set to confirm the connection establishment. It can only be set once a connection request has been made.
* **Start:** The bit is set if it concerns the first packet of payload data to be transferred.
* **End:** The bit is set if it concerns the last packet of payload data to be transferred.
* **ID unknown (IDU):** The bit is set if the received ID does not exist. The ID is checked during the connection setup.

# **Flow Charts**

In order to better understand the flow of the respective processes, both the sending of data and the receiving of data were graphically represented in a flow chart. The reception process is divided into 2 sections: The connection establishment and the reception or transmission of the packets.

# **Receiving data**

# Ein Bild, das Text, Whiteboard enthält. Automatisch generierte Beschreibung

Figure 3: Flow chart for receiving data

# **Transmitting Data**

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Figure 4: Flow chart for transmitting data

# **Functions Overview**

# **Header *UART\_EIVE\_Protocol* for sending protocol**

# **int** UART\_Send\_Data(uint8\_t ID, uint8\_t \*databytes, **int** dataLength);

* **Description**: Main function of the EIVE UART protocol to send data. At first, this function uses the *connect\_()*-method to establish a connection between the transmitter and the receiver. Afterwards, once a connection was established, it uses the method *send\_data()* to send the transferred data to the receiver. It is the main method, which is called by the OBC or the PLOC to send the data
* **Parameters**:
  + **ID**: Identification number of the subsystem to which it refers
  + **databytes**: pointer to the data which is going to be send
  + **dataLength**: length of the data which is going to be send (number of bytes)
* **Return** **value**:
  + **XST\_SUCCESS**: if the data was sent properly
  + **XST\_FAILURE**: if the data was not sent properly
* **Notes**:

**int** connect\_(uint8\_t ID, uint8\_t \*databytes, uint8\_t dataLength, uint8\_t \*lastCRC\_send, uint8\_t \*lastCRC\_rcvd);

* **Description**: This method is used to establish a connection between the transmitter and the receiver
* **Parameters**:
  + **ID**: Identification number of the subsystem to which it refers
  + **databytes**: pointer to the data which is going to be send
  + **dataLength**: length of the data which is going to be send (number of bytes)
  + **lastCRC\_send**: pointer to the last CRC value, which was sent
  + **lastCRC\_rcvd**: pointer to the last CRC value, which was received
* **Return** **value**:
  + **XST\_SUCCESS**: if the connection was established properly
  + **XST\_FAILURE**: if the connection was not established properly
* **Notes**:

**int** send\_request\_to\_send(uint8\_t ID, uint8\_t \*temp32, uint8\_t \*lastCRC\_send, uint8\_t \*flags);

* **Description**: This method sends a connection request to the receiver to establish a connection
* **Parameters**:
  + **ID**: Identification number of the subsystem to which it refers
  + **temp32**: pointer to an array, which is located in the method *connect\_()*, and is used to buffer an entire package
  + **lastCRC\_send**: pointer to the last CRC value, which was sent
  + **flags**: pointer to the flags which are going to be send in the connection request
* **Return** **value**:
  + **XST\_SUCCESS**: if the connection request was sent properly
  + **XST\_FAILURE**: if the connection request was not sent properly
* Notes:

**int** package\_count(**int** dataLength);

* **Description**: Function for counting the number of packets from the transferred data length
* **Parameters**:
  + **dataLength**: length of the data which is going to be send (number of bytes)
* **Return** **value**: It returns the number of packages needed to transfer all the data
* **Notes**:

**void** get\_received\_data(uint8\_t \*header, uint8\_t \*data, uint8\_t \*flags, uint8\_t \*submittedCRC);

* **Description**: Method to save the submitted data and splitting it into header (CRC and flags) and data
* **Parameters**:
  + **header**: pointer to the array in which the submitted header from the receiver is going to be buffered
  + **data**: pointer to the array in which the submitted data from the receiver is going to be buffered
  + **flags**: pointer to the array in which the submitted flags from the receiver are going to be buffered
  + **submittedCRC**: pointer to a variable in which the submitted CRC from the receiver value is going to be buffered
* **Notes**:Since this is a connection-oriented protocol, it is necessary to wait for a response after each transmission. This occurs both when the connection is established and when the packages containing the data are sent. Accordingly, this method is required for these two functions and therefore the arrays are located once in the method *connect()* and another time in the method *send\_data().*

**int** send\_data(uint8\_t ID, uint8\_t \*databytes, **int** dataLength, uint8\_t \*lastCRC\_send, uint8\_t \*lastCRC\_rcvd);

* **Description**: Method which is used to split the data to be sent into packets and to send the packages themselves. For this purpose, the method *UART\_Send()* provided by Xilinx is used (see chapter, functions provided by Xilinx)
* **Parameters**:
  + **ID**: Identification number of the subsystem to which it refers
  + **databytes**: pointer to the data which is going to be send
  + **dataLength**: length of the data which is going to be send (number of bytes)
  + **lastCRC\_send**: pointer to the last CRC value, which was sent
  + **lastCRC\_rcvd**: pointer to the last CRC value, which was received
* **Return** **value**:
  + **XST\_SUCCESS**: if the sending of the data was successful
  + **XST\_FAILURE**: if the sending of the data was not successful
* **Notes**:

**int** wait\_on\_answer(uint8\_t \*send\_array, uint8\_t ID, uint8\_t \*lastCRC\_send);

* **Description**: Method for waiting for the response of the receiver
* **Parameters**:
  + **send\_array**: pointer to the array containing the data to be sent
  + **ID**: Identification number of the subsystem to which it refers
  + **lastCRC\_send**: pointer to the last CRC value, which was sent
* **Return value**:
  + **XST\_SUCCESS**: if an answer was received
  + **XST\_FAILURE**: if an answer was not received
* **Notes**: This method regularly checks the Receive Buffer. To ensure that the protocol runs continuously, a counter is incremented. If the counter reaches its maximum value without filling the receive buffer, the desired package is sent to the receiver again. Another counter is responsible for counting the number of transmissions of the same package to the receiver. If this counter also reaches its maximum value, a failure is sent to the recipient and the method also returns a failure. If the buffer is filled, the method returns a success, even without the timers reaching their maximum values.

**void** fill\_packages(uint8\_t ID, **int** dataLength, uint8\_t \*databytes, uint8\_t \*temp, **int** packageCount);

* **Description**: This method is used to fill the packages with the submitted information. The pointer to the submitted value temp is used to buffer the temporary packages
* **Parameters**:
  + **ID**: Identification number of the subsystem to which it refers
  + **dataLength**: length of the data which is going to be send (number of bytes)
  + **databytes**: pointer to the data which is going to be send
  + **temp**: pointer to an array, which is located in the method *send\_data()* and is used to buffer the packets with the corresponding header and data, which are going to be sent
* **Notes**:

uint8\_t fill\_header\_for\_empty\_data(uint8\_t \*header, uint8\_t ID, uint8\_t flags, uint8\_t \*lastCRC\_send);

* **Description**: Method for filling the header of packets which are not to transmit data
* **Parameters**:
  + **header**: pointer to the array in which the submitted information is going to be buffered
  + **ID**: Identification number of the subsystem to which it refers
  + **flags**: array containing the flags which are filed into the header
  + **lastCRC\_send**: pointer to the last CRC value, which was sent
* **Return value**: The method returns the CRC value calculated for this package to be sent
* **Notes**: The method is used to fill the header when empty data packages are to be sent. It is used while waiting for a response if the previously received package was not acknowledged. It is also used while sending a success or a failure, because here only the ACK flag is relevant

# **Header UART\_EIVE\_Protocol for receiving protocol**

**int** UART\_Recv\_Data();

* **Description**: Main method for receiving data if its available. The method *UART\_Recv\_Buffer()* provided by Xilinx is used to fill the receive buffer. Furthermore the method *receive()* is used to analyse the received information and to generate a corresponding response for the transmitter
* **Return value**:
  + **XST\_SUCCESS**: If the receiving process has been completed successfully
  + **XST\_FAILURE**: If the receiving process has not been completed successfully
  + **XST\_NO\_DATA:** if no data is available in the receiving buffer
* **Notes**:

**int** receive();

* **Description**: This method is called in the main method; it is used to receive the data. For this purpose, it implements the algorithm for receiving data
* **Return value**:
  + **XST\_SUCCESS**: If the receiving was correct
  + **XST\_FAILURE**: If the receiving was not correct
* **Notes**: As indicated above, in this method the protocol for receiving data is implemented, which requires a connection to the transmitter. A connection is established to receive packages longer than 28 bytes. First, it is indicated that some data is to be transmitted, to which it responds that data can be received. Then the actual data transmission takes place, where each received package is acknowledged positively or negatively

**int** connection\_establishment(uint8\_t \*last\_crc\_rcv, uint8\_t \*last\_crc\_send, uint8\_t \*new\_flags, uint8\_t \*conn\_id, uint8\_t \*calc\_crc);

* **Description**: Method to connect with the transmitter. It checks the received data from the transmitter whether the request to send flag is set and answers with a package where the ready to receive flag is set
* **Parameters**:
  + **last\_crc\_rcv:** pointer to the last CRC value, which was received
  + **last\_crc\_send:** pointer to the last CRC value, which was sent
  + **new\_flags:** pointer to the flags, which are going to be sent
  + **conn\_id:** pointer to the identification number for the connection establishment
  + **calc\_crc:** pointer to the variable located in the method *recevie()*, to store the calculated CRC
* **Return value:** 
  + **XST\_SUCCESS**: If the connection was established correctly
  + **XST\_FAILURE**: If the connection was not established correctly
* **Notes**:

**int** receive\_data(uint8\_t \*crc\_rcv, uint8\_t \*crc\_send, uint8\_t rcvd\_id, uint8\_t last\_sent\_flags, uint8\_t \*calc\_crc);

* **Description**: this method is used to receive data from the connected transmitter
* **Parameters**:
  + **crc\_rcv**: pointer to the last received CRC value
  + **crc\_send:** pointer to the last send CRC value
  + **last\_sent\_flags:** pointer to the last sent flags
* **Return value**:
  + **XST\_SUCCESS**: If the data was successful received and written in the receiving buffer
  + **XST\_FAILURE**: If the data was not successful received
* **Notes**:

**int** extract\_header(**const** uint8\_t \*rcvBuffer, uint8\_t \*header, uint8\_t \*data);

* **Description**: this method splits the received data into header and payload data
* **Parameters**:
  + **rcvBuffer**: pointer to the buffer with received data
  + **header**: pointer to the header array to store the extracted header
  + **data**: pointer to the data array to store the extracted header
* **Return value**:
  + **XST\_SUCCESS**: If the received header and data were successfully written in the corresponding arrays.
  + **XST\_FAILURE**: If the received header and data were not successfully written in the corresponding arrays.
* **Notes**: Although this method is called extract header, its functionality has been extended and the data is also extracted from the receive buffer and written to the corresponding arrays

**int** check\_ID(uint8\_t ID);

* **Description**:

Function to check if the incoming packages have a known or unknown ID. If the ID is not known, the flag *IDU* is.

* **Parameters**:
  + **ID**: Identification number of the incoming package
* **Return value**:
  + **1:** If the ID is known
  + **0:** If the ID is not known
* **Notes**:

**int** send\_failure(uint8\_t \*last\_crc, uint8\_t old\_id, uint8\_t \*calc\_crc, int id\_unknown);

* **Description**:

This function is used to send an empty packet with the ACK flag not set to signal to the other party that the CRC value does not match and therefore an error has occurred during packet transmission.

* **Parameters**:
  + **last\_crc:** Is the pointer to the last sent CRC value. With this value the new CRC value is calculated.
  + **old\_id:** Is the ID of the currently received packet.
  + **calc\_crc:** Is the pointer to the value in which the newly calculated CRC value is temporarily stored.
  + **id\_unknown:** This value indicates whether the error is due to an unknown ID. Set (1) if the error was caused by an unknown ID, else not set (0).
* **Return value**:
  + **XST\_SUCCESS:** If the sending process was without errors.
  + **XST\_FAILURE:** If an error has occurred during the sending process.
* **Notes**: To fill the header the function *"fill\_header\_for\_empty\_data()"* is called, in which the new CRC value is calculated.

**int** send\_success(uint8\_t \*last\_crc, uint8\_t id, uint8\_t flags, uint8\_t \*calc\_crc);

* **Description**:

This function is used to send an empty packet with the ACK flag set and further flags to signal the remote station that the received CRC value matches and that the packet was received correctly. Other flags to be sent, such as “*Request To Send”*, are passed as parameters and sent with the message.

* **Parameters**:
  + **last\_crc:** the address of the last sent CRC-value.
  + **id:** Is the ID of the currently received packet.
  + **flags:** Is the flag byte to send with this answer.
  + **calc\_crc:** Is the pointer to the value in which the newly calculated CRC value is temporarily stored.
* **Return value**:
  + **XST\_SUCCESS:** If the sending process was without errors.
  + **XST\_FAILURE:** If an error has occurred during the sending process.
* **Notes**: To fill the header the function *"fill\_header\_for\_empty\_data()"* is called, in which the new CRC value is calculated.

**int** UART\_answer(uint8\_t \*header);

* **Description**: This method creates an empty packet with the passed header and calls the lowlevel UART function "UART\_Send()" to send the response packet.
* **Parameters**:
  + **header**: The header for the response to be sent.
* **Return** **value**:
  + **XST\_SUCCESS:** If the sending process was without errors.
  + **XST\_FAILURE:** If an error has occurred during the sending process.
* **Notes**: None.

**int** recv\_TC(uint8\_t \*header, uint8\_t \*databytes, int size\_of\_data);

* **Description**: This method is used for the data callback. Here the data is passed to the callback handler functions of the individual subsystems, so that the UART subsystem does not need to know the content of the data and its further processing.

The received ID is mapped to the respective data callback function of the subsystems by means of a switch-case statement.

* **Parameters**:
  + **header:** Is the pointer to the header array to extract the ID.
  + **databytes:** Are the data bytes received to pass them to the respective subsystem via the data callback.
  + **size\_of\_data:** Is the number of data bytes.
* **Return** **value**:
  + **XST\_SUCCESS**: always.
* **Notes**: The individual subsystems are themselves responsible for providing the TC handler functions. Each handler function is required to have as parameters the data bytes of type uint8\_t\* and the length of the data of type integer.

**int** recv\_TM();

* **Description**: This function is not declared at this time, it is an optional function for storing data into the memory if the received data is not a TC.
* **Parameters**: None.
* **Return** **value**: None.
* **Notes**: This functionality is not used at this time.

**void** default\_operation();

* **Description**: This function can be used for error handling. If some ID is received, which is not a TC and also not a TM.
* **Parameters**: None.
* **Return** **value**: None.
* **Notes**: None.

# **Header UART\_IO**

# **int** Initialize\_UART\_Device(u16 DeviceID);

* **Description**: This method is used to initialize the UART device. Also, the UART do a self-test to check the correctness of the functionalities. After that the mode is set to “normal operation mode” and the receive buffer is filled up with “0x00”.
* **Parameters**:
  + **DeviceID:** the ID of the UART device, given by definitions in the header-file
    - **UART0\_Device\_ID:** Device ID of the UART0
    - **UART1\_Device\_ID:** Device ID of the UART1
* **Return** **value**:
  + **XST\_SUCCESS:** if initialization process has run without errors.
  + **XST\_FAILURE:** if an error has occurred during the initialization process or during the self-test
* **Notes**: Called functions taken from Xilinx uartps polled example (Ref.:xxx)

**int** UART\_Send\_Buffer(u8 SendBuffer[BUFFER\_SIZE]);

* **Description**: This function is called to send the packets. If the UART device works correctly, the Xilinx provided *XUartPs\_Send()* - function is called and the packet to send is passed.

During transmission, the system waits until the process is completed.

* **Parameters**:
  + **SendBuffer[BUFFER\_SIZE]:** is the buffer to send. It may have a maximum length of 32 bytes.
* **Return** **value**:
  + **XST\_SUCCESS:** This is returned when all 32 bytes have been sent correctly.
  + **XST\_FAILURE:** This is returned when the sent count does not match the buffer length or when the UART is not ready.
* **Notes**: Called functions taken from Xilinx uartps polled example (Ref.:xxx)

**int** UART\_Recv\_Buffer(uint8\_t\* RecvBuffer);

* **Description**: This function is called to receive the packets. If the UART device is working correctly, the *XUartPs\_Recv()* function provided by Xilinx is called to write the data into a receive buffer. If no data to be received is available, the function is left before the actual receive process.
* **Parameters**:
  + **RecvBuffer:** is the buffer in which the received data is to be stored.
* **Return** **value**:
  + **XST\_SUCCESS:** This is returned when all 32 Bytes are received correctly.
  + **XST\_NO\_DATA:** This is returned when no data is available to receive.
  + **XST\_FAILURE:** This is returned when the UART is not ready.
* **Notes**: Called functions taken from Xilinx uartps polled example (Ref.: xxx) and Xilinx uartps low echo example (Ref.: xxx)

**int** UART\_Send(u8 \*data);

* **Description**: This function is called by the methods of the higher-level log. Here the data packet to be sent is passed on to the actual transmission method.
* **Parameters**:
  + **Data:** Is the data packet to be sent.
* **Return** **value** (coming from UART\_Send\_Buffer):
  + **XST\_SUCCESS:** This is returned when all 32 bytes have been sent correctly.
  + **XST\_FAILURE:** This is returned when the sent count does not match the buffer length or when the UART is not ready.
* **Notes**: This function only calls the function UART\_Send\_Buffer().

# **Used functions provided by Xilinx (Ref: xxx)**

**XUartPs\_Config \*** XUartPs\_LookupConfig (u16 DeviceId);

* **Description**: Looks up the device configuration based on the unique device ID. The table contains the configuration info for each device in the system.
* **Parameters**:
  + **DeviceId:** contains the ID of the device
* **Return** **value**: A pointer to the configuration structure or NULL if the specified device is not in the system.
* **Notes**: None.

**s32** XUartPs\_CfgInitialize (XUartPs \*InstancePtr, XUartPs\_Config \*Config, u32 EffectiveAddr);

* **Description**: Initializes a specific XUartPs instance such that it is ready to be used.The data format of the device is setup for 8 data bits, 1 stop bit, and no parity by default. The baud rate is set to a default value specified by Config->DefaultBaudRate if set, otherwise it is set to 19.2K baud. The receive FIFO threshold is set for 8 bytes. The default operating mode of the driver is polled mode.
* **Parameters**:
  + **InstancePtr:** is a pointer to the XUartPs instance.
  + **Config:** is a reference to a structure containing information about a specific XUartPs driver.
  + **EffectiveAddr:** is the device base address in the virtual memory address space. The caller is responsible for keeping the address mapping from EffectiveAddr to the device physical base address unchanged once this function is invoked. Unexpected errors may occur if the address mapping changes after this function is called. If address translation is not used, pass in the physical address instead.
* **Return** **value**:
  + **XST\_SUCCESS:** if initialization was successful
  + **XST\_UART\_BAUD\_ERROR:** if the baud rate is not possible because the inputclock frequency is not divisible with an acceptable amount of error
* **Notes**: The default configuration for the UART after initialization is:
  + 19,200 bps or XPAR\_DFT\_BAUDRATE if defined
  + 8 data bits
  + 1 stop bit
  + no parity
  + FIFO's are enabled with a receive threshold of 8 bytes
  + The RX timeout is enabled with a timeout of 1 (4 char times)
  + All interrupts are disabled.

**XUartPs\_Send** (XUartPs \*InstancePtr, u8 \*BufferPtr, u32 NumBytes);

* **Description**: This function sends the specified buffer using the device in either polled or interrupt driven mode.

This function is non-blocking, if the device is busy sending data, it will return and indicate zero bytes were sent. Otherwise, it fills the TX FIFO as much as it can, and return the number of bytes sent.

In a polled mode, this function will only send as much data as TX FIFO can buffer. The application may need to call it repeatedly to send the entire buffer.

In interrupt mode, this function will start sending the specified buffer, then the interrupt handler will continue sending data until the entire buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending.

* **Parameters**:
  + **InstancePtr:** is a pointer to the XUartPs instance.
  + **BufferPtr:** is pointer to a buffer of data to be sent.
  + **NumBytes:** contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.
* **Return** **value**: The number of bytes actually sent.
* **Notes**: The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

**XUartPs\_Recv** (XUartPs \*InstancePtr, u8 \*BufferPtr, u32 NumBytes);

* **Description**: This function attempts to receive a specified number of bytes of data from the device and store it into the specified buffer.

This function works for both polled or interrupt driven modes. It is non-blocking.

In a polled mode, this function will only receive the data already in the RX FIFO. The application may need to call it repeatedly to receive the entire buffer. Polled mode is the default mode of operation for the device.

In interrupt mode, this function will start the receiving, if not the entire buffer has been received, the interrupt handler will continue receiving data until the entire buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of the receiving or error conditions.

* **Parameters**:
  + **InstancePtr:** is a pointer to the XUartPs instance
  + **BufferPtr:** is pointer to buffer for data to be received into
  + **NumBytes:** is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.
* **Return** **value**: The number of bytes received.
* **Notes**: The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

**void** XUartPs\_SetOperMode (XUartPs \*InstancePtr, u8 OperationMode);

* **Description**: This function sets the operational mode of the UART. The UART can operate in one of four modes: Normal, Local Loopback, Remote Loopback, or automatic echo.
* **Parameters**:
  + **InstancePtr:** is a pointer to the XUartPs instance.
  + **OperationMode:** is the mode of the UART.
* **Return** **value**: None.
* **Notes**: None.

**s32** XUartPs\_SelfTest (XUartPs \*InstancePtr);

* **Description**: This function runs a self-test on the driver and hardware device.

This self test performs a local loopback and verifies data can be sent and received. The time for this test is proportional to the baud rate that has been set prior to calling this function. The mode and control registers are restored before return.

* **Parameters**:
  + **InstancePtr:** is a pointer to the XUartPs instance
* **Return** **value**:
  + **XST\_SUCCESS:** if the test was successful
  + **XST\_UART\_TEST\_FAIL:** if the test failed looping back the data
* **Notes**: This function can hang if the hardware is not functioning properly.

**u32** XUartPs\_IsSending (XUartPs \*InstancePtr);

* **Description**: This function determines if the specified UART is sending data.
* **Parameters**:
  + **InstancePtr:** is a pointer to the XUartPs instance.
* **Return** **value**:
  + **TRUE:** if the UART is sending data
  + **FALSE:** if UART is not sending data
* **Notes**: None.

#define XUartPs\_IsReceiveData(BaseAddress)

* **Description**: **(MACRO)** Determine if there is receive data in the receiver and/or FIFO.
  + Value:

!((Xil\_In32((BaseAddress) + XUARTPS\_SR\_OFFSET) & \ (u32)XUARTPS\_SR\_RXEMPTY) == (u32)XUARTPS\_SR\_RXEMPTY)

* **Parameters**:
  + **BaseAddress:** contains the base address of the device.
* **Return** **value**:
  + **TRUE:** if there is receive data, FALSE otherwise.
* **Notes**: C-Style signature: u32 XUartPs\_IsReceiveData(u32 BaseAddress)

1. **Testing**

During the development phase there were tests in software as well as at the end directly on the hardware. The software test was carried out both via LAN using a client-server setup and with a version ported to the Java programming language before the protocol was tested on the hardware.

* 1. **Testing in Software**

The software test using a client-server connection was performed because no UART low-level functionality was available between two laptops. The low-level functionality of the UART was replaced by the low-level functionality of the LAN connection, where the server represented the receiver side and the client the transmitter side.

The *UART\_Send()* function was replaced by the *send()* function of the network connection and the *UART\_Recv\_Buffer()* function by the corresponding *recv()* function.

In order to be able to test the protocol later on the hardware, the protocol was translated into the programming language Java. Java was chosen because it offers the option of a simple graphical test interface. Serial ports were opened to allow access to the UART functionalities.

The software test in Java was carried out via emulated serial ports, which provide the UART functionality virtually.

Both tests worked with small amounts of data (< 100 bytes) as well as with large amounts of data (> 2 kBytes), whereby the software tests achieved a positive result.

* 1. **Testing in Hardware**

A total of three hardware tests were performed: one test for the low-level functionality of the UART interface, one test for receiving data with the protocol and one test for receiving a TC and then sending a TM.

* + 1. **UART Low-level Test**

The low-level test included a character byte sequence sent to the FPGA which was received by the low-level function *UART\_Recv\_Buffer().* Since this functionality was provided by Xilinx in a tested example, this test also achieved a positive result.

* + 1. **UART Protocol Test Receiving TCs**
    2. **UART Complete Protocol Test**

1. **To be adjusted**
2. **References**
   * + - Uartps Documentation from Xilinx <https://xilinx.github.io/embeddedsw.github.io/uartps/doc/html/api/index.html>
       - XUARTPS\_Polled\_Example

<https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/uartps/examples/xuartps_polled_example.c>

* + - * XUARTPS\_Low\_Echo\_Example

<https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/uartps/examples/xuartps_low_echo_example.c>

1. **(placeholder)**