UART
Connection Orientated Communication Protocol

Documentation

Tobias Festerling
Valentin Stadtlander

26 June 2020

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

Table of Content

List of Figures

1. Introduction

This chapter explains the context in which the UART protocol is to be used and gives a general overview of how it works.

1.1. General Information

For the EIVE satellite project, a payload computer (PLOC) is being developed on an FPGA, which will take over various tasks in space. The individual subsystems of the PLOC are controlled by an on-board computer (OBC) via commands (TCs), which are to be transmitted via UART. A simple high-level protocol is required to enable the exchange of large commands and data.
Initially, it was considered to implement a standardized protocol, such as CCSDS, but this would have been too complex for the required purposes, so it was decided to create a separate and very simple protocol for the required purposes.

1.2. Overview

The protocol is used to exchange 32 Byte packets each, which is limited by the maximum FIFO size. It is based on a connection that must be established between the transmitter and receiver. Therefore, the sending or receiving of data is divided into two parts: connection establishment and the actual transmission. This has the advantage that data and commands longer than 32 bytes can also be exchanged.
During the entire data transmission, each received packet is assigned a CRC value, which is checked and acknowledged with a response either positive or negative, i.e. each packet is followed by an acknowledgement packet.
To ensure that the UART protocol does not have to know anything about commands or data, an ID has been introduced for each subsystem, through which the content of the data does not have to be opened, it only has to be passed on to the respective subsystem with a callback routine.

2. 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.

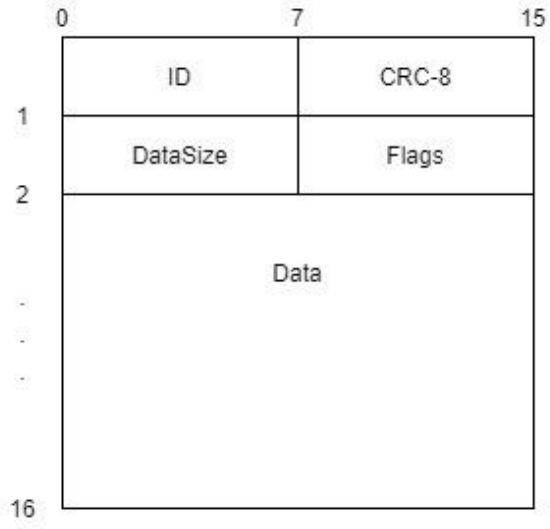

Figure 1: Packet structure

2.1.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

2.1.1. 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)

2.1.2. 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
0b00010000 XOR 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
0b10001000 **0b00010000** XOR 0b00011101 = **0b00001101**	4. Step: Left bit == 1? ➔ Yes: • shift to left • XOR with generator polynomial
<- 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
0b11010000 XOR 0b00000000 **0b11010000**	END: ➔ XOR with *final_XOR_value (0x00)*

➔ Calculated CRC value: 0xD0

2.1.3. 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:

$$\left\lfloor \frac{1300\ Bytes}{28 \frac{Bytes}{Packet}} \right\rfloor = 46\ Packets$$

$$\Leftrightarrow 28\ Byte * 46\ Packets = 1288\ Byte$$

$$\Longrightarrow 1300\ Byte - 1288\ Byte = 12\ Byte\ remaining$$

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

2.1.4. 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.

ACK	Req2Send	Rdy2Rcv	Start	End	IDU		

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 TM/TCs.
- **Ready to Receive (Rdy2Rcv):** The bit is set to confirm the connection establishment. It can only be set once a connection request has been received.
- **Start:** The bit is set if it concerns the first packet of TM/TCs to be transferred.
- **End:** The bit is set if it concerns the last packet of TM/TCs 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.

3. Callback Routine

Because of the subsystem ID in the packet header, the UART interface does not need to know the content of the data or the type of command. A callback routine is required so that the data can then be passed on to the corresponding subsystems. To realize this, a data handler function has to be implemented in each subsystem. This function is called by the UART interface and gets the data and the length of the data as arguments.

The handler function should have the following structure:

$$\boldsymbol{void}\ callback\_handler\_ < subsystem\_name > (uint8\_t *\ data, int\ datalength)$$
{

Callback routine for the respective subsystem

}

In this function, the respective subsystem can then evaluate and further process the command or data.
The function "recv_TC()" within the source file UART_Protocol_Recv.c is responsible for the callback routine, where a switch-case statement is used to compare the ID.
The handler function created by the subsystems must then be entered in the case statement of the respective subsystem so that it can be called.

For example: Camera subsystem

UART subsystem:
The UART subsystem receives data.

The ID is checked by the UART subsystem: The data is for the camera subsystem.

The UART subsystem calls the handler function provided by the camera subsystem and passes the data and the data length.

Camera subsystem:
Within the camera handler function the data is evaluated.

The received data is a command. The command is: "Take a picture".

The command is executed.

4. 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 processes are divided into 2 sections: The connection establishment and the reception or transmission of the packets.

4.1.Receiving data

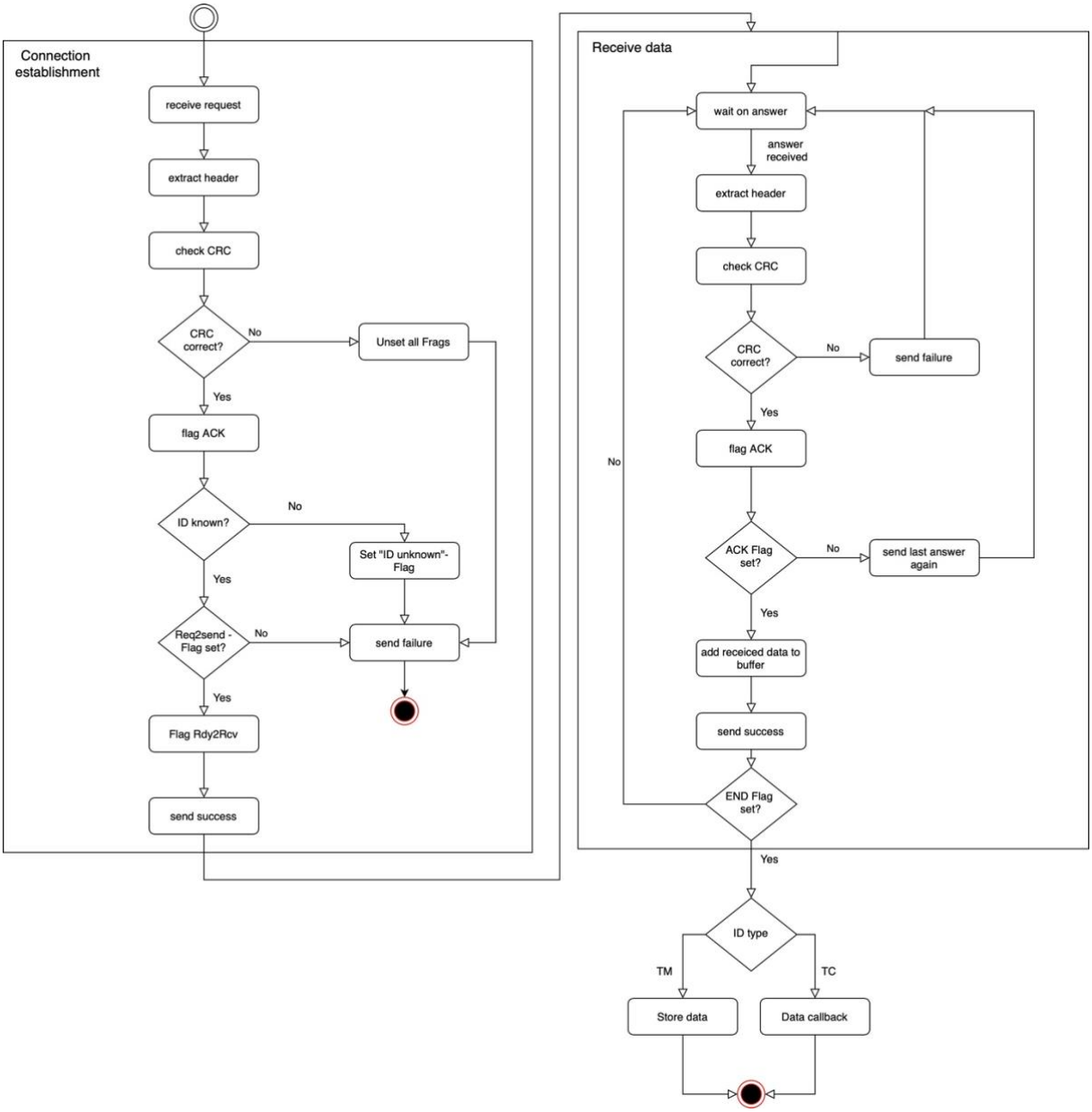

Figure 3: Flow chart for receiving data

4.2. Transmitting Data

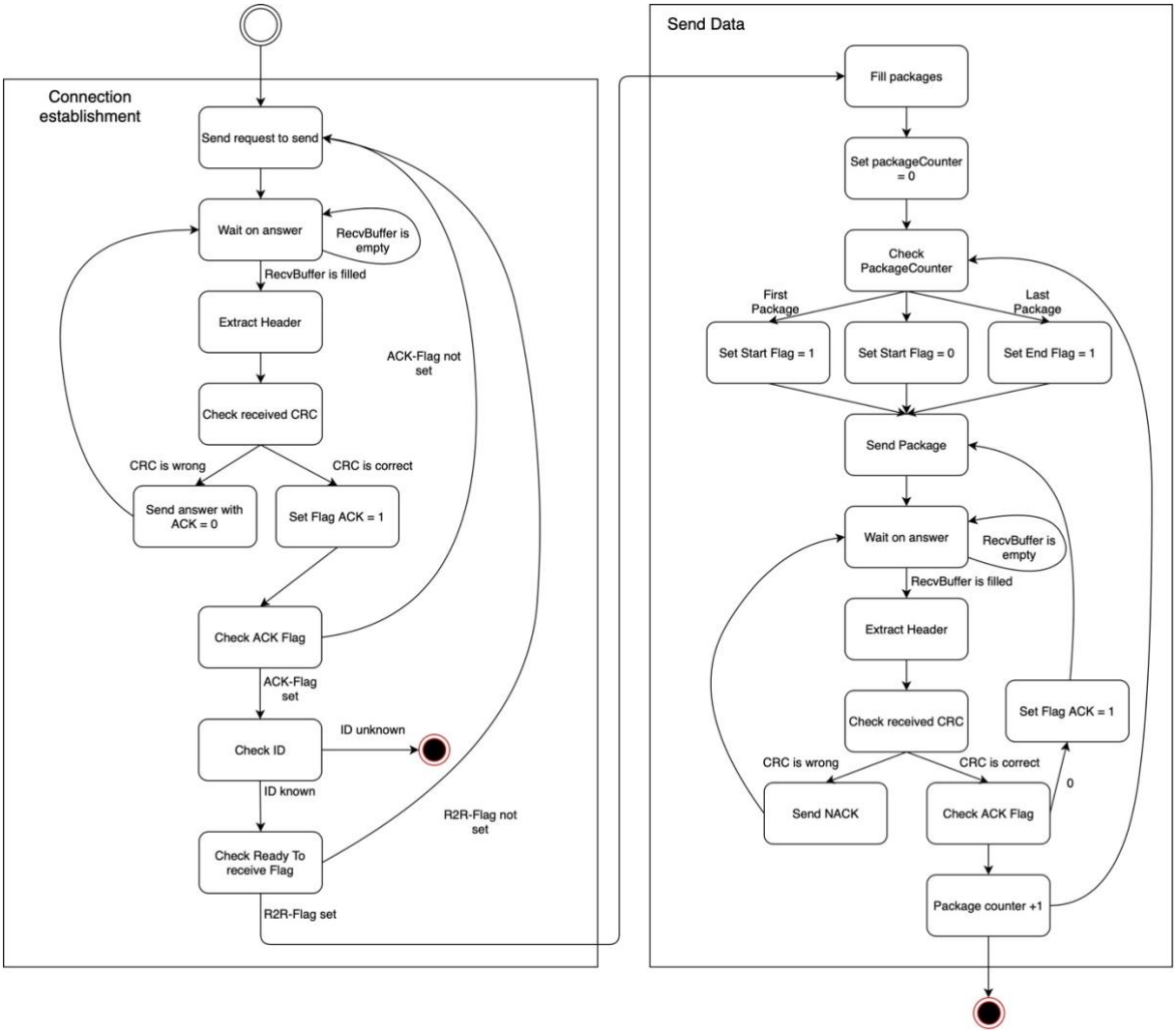

Figure 4: Flow chart for transmitting data

5. 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.

5.1.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.

5.2.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 back a TM.

5.2.1. 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.

5.2.2. UART Protocol Test Receiving TCs

During the reception test only the receiving side of the protocol is tested. Here a Java program, as emulator of the one side, sends a text which is to be received by the FPGA.
The texts used here had different lengths between 100 bytes and 5 kbytes.

Here there were errors in the receiving process on the side of the FPGA. Normally every received packet is answered with an acknowledgement, but this did not arrive correctly on the opposite side. Here the error was caused by the not adapted lowlevel transmission method. After the adjustment the process worked without errors.

5.2.3. UART Complete Protocol Test

In the last test, the protocol is tested for the interaction of receiving and sending data. A Java program, which simulates the opposite UART interface, sends a command, whereupon data is sent back from the FPGA.
The test was done with 3 text-based commands, which are sent to the FPGA. For each command a text of a certain length is sent back as response.

The three commands are:

- "Get_First_Text_For_TC_Command_Longer_than_32_Bytes!!!!"
  ➔ Length of the response text: 72 Bytes

- "GetSecondText"
  ➔ Length of the response text: 3125 Bytes

- "GetThirdText"
  ➔ Length of the response text: 19597 Bytes

An error occurred during the connection setup for sending back the requested data. After the FPGA has received the command, the corresponding text should be sent back directly. The Java program did not receive any data and the FPGA was in a deadlock while waiting for the response of the connection setup.
On the FPGA side, the protocol sent the connection request faster than the Java program was able to respond, causing both sides to wait for a response, which resulted in a deadlock.
This problem was solved by a "*sleep(1)*" statement, which causes the protocol on the FPGA side to wait one second before sending the actual response to the command.
The deadlock was resolved from a blocking query to a non-blocking query by means of a modified receive process.

DEBUG PROCESS:

In order to initialize the UART interface, the initialize process (as shown in Figure 5 and Figure 6) must first be stepped over. After that you can either continue stepping or let the program run alone.

```
27 /**
28  * Main Method for UART
29  */
30 int main(void)
31 {
32     //xil_printf("UART Test!\n");
33     Initialize_UART_Device(UART0_DEVICE_ID);
34     while(1)
35     {
36         //UART_Recv_Hello_World(&Uart_Ps);
37         UART_Protocol_Test(&Uart_Ps);
38     }
39 }
40
```

Figure 5: Initialize UART

```
30 int main(void)
31 {
32     //xil_printf("UART Test!\n");
33     Initialize_UART_Device(UART0_DEVICE_ID);
34     while(1)
35     {
36         //UART_Recv_Hello_World(&Uart_Ps);
37         UART_Protocol_Test(&Uart_Ps);
38     }
39 }
```

Figure 6: Run Protocol Test

- In the next step the Java program must be started. There you can see the output from Figure 7 in the console. Here you have to choose between
  - 1 (short text),
  - 2 (medium text) and
  - 3 (long text),

  which should be sent back by the other side.

➔ 4 (extra long text) is no longer supported here.

```
Which command do you want to send?
 1: send back text1 (short)
 2: send back text2 (medium)
 3: send back text3 (long)
 4: send back text4 (extra long)
>4: exit
```

Figure 7: Output Java Program (Selection)

- After the selection the transmission process starts: The Java program sends a command (TC) to the opposite side and gets back the desired text (TM). If the exchange is successful, the output appears in Figure 8.

```
Datasize: 19597 Byte(s)
Data received!
Status Receive() 0
Status UART_Recv_Data() 0
Time for Receiving: 11.206 Seconds

-> TC done!
```

Figure 8: Output Java Program (Successfully Completed Text 3)

6. To be adjusted

The first major milestone was the functioning hardware test of the complete protocol (see section 5.2.3), which was successfully completed. In the next step, the protocol has to be adjusted to guarantee better performance and a safe and correct operation. The adjustments listed here are, after completion of the protocol and a first working hardware test, mainly feasible with the OBC and therefore not yet completed.

To avoid blocking processes within the protocol, counters have been implemented that interrupt the reception of acknowledgements after a certain time.
In addition, there are retransmission timers, whereby the last packet sent can be retransmitted if there is no response. If a value is exceeded, the entire transmission is stopped, and an error value is returned (*XST_FAILURE*).

The following end values are currently set:

- MAX_TIMER:
  - Value = 10000
  - Used in
    - "*UART_EIVE_Protocol_Send.c*" while waiting on acknowledgement
    - "*UART_EIVE_Protocol_Recv.c*" while waiting on acknowledgement
- RETRANSMISION_TIMER:
  - Value = 50
  - Used in
    - "*UART_Protocol_Send.c*" for maximum number of retransmissions of request to send
    - "*UART_Protocol_Send.c*" for maximum number of retransmissions of the data packets

The end values of the timers could not be tested because the OBC is needed to find the correct values.
A sleep function is currently required to give the program time to adjust to receiving data after transmission. This was necessary for the Java program but can be disabled or adjusted if it is not needed or does not fit correctly.
To verify the robustness of the protocol, a long-term test should be performed. During this test, the correct operation of the implemented error detection can be tested, so that error-free operation can be guaranteed even with faulty incoming packets.

This test should be divided into 2 parts:

- a test with a program emulating the other side of the protocol (e.g. the Java program with adaptations for long-term test) to prove the correct operation and to test the error detection,
- a test with the OBC in order to match the functionalities

The long-term test should last several hours or days to ensure that the protocol can be used in long-term operation. Various commands should be sent to the protocol side on the FPGA and, if necessary, replies should be received. If the sending of TMs starting from the PLOC is desired (optional, has to be implemented, see function "*recv_TM()*"), this can be included in the test.

7. Prospect

There are many possibilities to improve and optimize the protocol. A few suggestions are addressed and explained below.

Currently no data is sent with the connection setup, the data field is filled with 0. It is possible to reduce the number of packets by one by using the data field already there.

Currently a response packet of 32 bytes length is sent for each received packet. Since no data is sent, 28 bytes are filled with only 0 and are actually unnecessary.
One possibility to increase the data rate would be to send only the header without data field as acknowledgement instead of the whole packet. This would require 28 bytes less to be transmitted, which would improve the data rate considerably.
In addition, instead of acknowledging each individual packet, it is possible to send 8 packets directly and acknowledge these received packets by means of 8 "headers" in one packet. This could achieve a higher throughput and still maintain error protection.
With this adaptation it is necessary to store the last 8 CRC values.

An error analysis function would also be feasible. Here, after a certain number of failed attempts, the transmitter could send a test packet known from both sides in order to determine whether the error lies with the receiver and a restart of the transmission process is necessary, or whether there is a line error that cannot be corrected by software.
This would require a packet with a test ID and a known data field. It is possible to use the ID for TCs of the UART subsystem as a test ID, since no TCs are currently necessary for the UART subsystem.
If the test packet is received correctly, there is an error at the receiver if the test packet is also not received correctly

Currently it is assumed that the OBC only sends commands (TCs) to the PLOC. It is also possible that data (TMs) are sent from the OBC to the PLOC, for this purpose there is a function "*recv_TM()*" which is currently not used. It could be implemented according to the "*recv_TC()*" function and could also pass the data to the subsystems using callback functions.

8. Functions Overview

8.1. Headerfile *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**: None.

```
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**: None.

```
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: None.

```
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**: None.

```
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**: None.

```
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**: None.

```
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.

8.2.Headerfile 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.
- **Parameters**: None.
- **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**: None.

```
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.
- **Parameters**: None.
- **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**: None.

```
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**: None.

```
int extract_header(const uint8_t *rcvBuffer, uint8_t *header,
uint8_t *data);
```

- **Description**: this method splits the received data into header and 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**: None.

```
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**: This functionality is not used at this time.

8.3. Header *UART_IO*

```
int Initialize_UART_Device(u16 DeviceID);
```

- **Description**: This method is used to initialize the UART device. Also, the UART does 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.: [1])

```
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.: [1])

```
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.: [1]) and Xilinx uartps low echo example (Ref.: [2])

```
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().

8.4.Headerfile *CRC*

```
uint8_t calc_crc8_for_one_byte(uint8_t start_crc, uint8_t byte);
```

- **Description**: Method for determining the CRC value of a single byte. This method is passed the start CRC value as well as the corresponding byte. These are used to determine the desired value.
- **Parameters**:
  - **start_crc**: Initial CRC value to determine the new CRC value.
  - **byte:** Byte from which the CRC should be determined.
- **Return value**: It returns the 8-bit calculated CRC value.
- **Notes**: This method is used to calculate the CRC value of the entire 32-byte buffer. So this method gets the single bytes and calculates the desired CRC value from them

```
uint8_t calc_crc8_for_data(uint8_t bytes[], int length,  uint8_t
crc_initval);
```

- **Description**: This method is used to calculate the CRC value for data longer than one byte.
- **Parameters**:

  - **bytes:** array of the passed data which are longer than one byte and from which the CRC value should be calculated.
  - **length:** length of the array containing the past data.
  - **crc_initval:** Intial value to determine the new CRC value.
- **Return value**: It returns the 8-bit calculated CRC value.
- **Notes**: This method is the one that is called when a new CRC value is to be calculated. It passes the received bytes one by one to the method *calc_crc8_for_one_byte* with the corresponding initial value and thus determines the CRC value of the array.

```
int check_crc(uint8_t crc_val, uint8_t *rcv_buffer, uint8_t
crc_initval);
```

- **Description**: This method receives a CRC value, which is compared with the CRC value that is determined from the receive buffer.
- **Parameters**:
  - **crc_val**: CRC value with which the CRC of the receive buffer is to be compared.
  - **rcv_buffer**: Pointer to the Buffer which is filled with the received data.
  - **crc_initval**: Initial value with which the CRC of the receiving buffer is to be determined.
- **Return value**:
  - **XST_SUCCESS:** If the CRC values are equal.
  - **XST_FAILURE:** If the CRC values are not equal.
- **Notes**: This method is used in the protocol when the CRC values of the received data must be compared with the previously stored CRC.

8.5.Headerfile *Flags*

```
void set_<FlagName>_Flag(uint8_t *flags, uint8_t SET);
```

- **Description**: Methods for setting the multiple flags.
- **Parameters**:
  - **flags**: Pointer to store the flags.
  - **SET**: This parameter should be filled with a 1 if the corresponding flag should be set, or with a 0 if the corresponding flag should not be set.
- **Return value**: None.
- **Notes**: This method exists for each of the individual flags, where <*FlagName*> is replaced with the corresponding flag.

```
int get_<FlagName>_flag(uint8_t flags);
```

- **Description**: Methods for getting the multiple flags.
- **Parameters**:
  - **flags**: Byte with the flags from which it should be determined whether the corresponding flag is set
- **Return value**:
  - **1:** If the corresponding flag is set
  - **0:** If the corresponding flag is not set
- **Notes**: This method exists for each of the individual flags, where <*FlagName*> is replaced with the corresponding flag.

8.6. Used functions provided by Xilinx (Ref.: [3])

`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).

9. References

[1] **XUARTPS_Polled_Example:**
https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/uartps/examples/xuartps_polled_example.c
Last call: 26.06.2019

[2] **XUARTPS_Low_Echo_Example:**
https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/uartps/examples/xuartps_low_echo_example.c
Last call: 26.06.2019

[3] **Uartps Documentation from Xilinx:**
https://xilinx.github.io/embeddedsw.github.io/uartps/doc/html/api/index.html
Last call: 26.06.2019