Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Abbas Akhundov

Concept and implementation of a Message Handler for an Automotive Communication Controller

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Abbas Akhundov

Concept and implementation of a Message Handler for an Automotive Communication Controller

Bachelorthesisbased on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Aplied Sciences

Supervising examiner : Prof. Dr. -Ing. Lutz Leutelt
Second Examiner : Prof. Dr. -Ing. Franz Schubert

Day of delivery 11. Juli 2016

Abbas Akhundov

Title of the Bachelorthesis

Concept and implementation of a Message Handler for an Automotive Communication Controller

Keywords

Message Handler, FlexRay, Frame Processing Unit, Buffer, RAM, Message

Abstract

Inside this report the concept of the Message Handler and implementation of the Frame Processing Unit, which is part of the Message Handler, is described.

Abbas Akhundov

Titel der Arbeit

Konzept und Realisierung eines Message Handler für einen Automotive Communication Controller

Stichworte

Message Handler, FlexRay, Frame Processing Unit, Buffer, RAM, Message

Kurzzusammenfassung

Diese Arbeit beschreibt die Konzeptionierung des Message Handlers und die Realisierung der Frame Processing Unit, welche Teil des Message Handlers ist.

Contents

List of Tables

List of Figures

1 Introduction

The FlexRay Message Handler is part of the Controller-Host-Interface (CHI) of the FlexRay communication controller. The Message Handler addresses the exchange of message data between the Host and the FlexRay communication protocol. Message transmission pertains to the message data flow from the Host to the FlexRay communication protocol and message reception to the message data flow from the FlexRay communication protocol to the CPU [FlexRay (2005)] . The use of the Message Handler is to arbitrate the access requests to the message buffers by CPU and PE (clients of Message Handler) and hence to avoid any possible conflict between the host and the physical layer attempting to access the Message RAM.

1.1 General Requirements

Req. 1 *Concept is feasible for channel A and B.*

The FlexRay has a dual channel protocol. Several mechanisms in the protocol are replicated on both channels. This means that identical mechanisms are executed for channel A and B. Whenever a specific process is described for channel A, then it is assumed it can be done in identical way for channel B.

Req. 2 *Implementation for channel A only.*

All the processes shall be described and tested for channel A only. Even if some process has to be replicated for channel B it shall not be explicitly described in the thesis. The process of handling messages of channel B or A & B shall not be implemented in the thesis.

Req. 3 *There shall be a mechanism by which the Host informs the Message Handler to which buffer it would like to have access.*

This mechanism shall be able to receive commands from HOST that would tell which operation it would like to make (read, write), then this information shall be transfered to Message Handler. If the operation required is read, then this mechanism shall also transfer the information telling Message Handler which buffer Host would like to read.

1.2 Host - Message Handler Communication Mechanism

Req. 4 *The Communication Mechanism shall only receive header information of a message provided by Host.*

The Communication Mechanism shall not receive payload bits from the Host.

Req. 5 *The Communication Mechanism shall always be able to check if the message received from Host is consistent or not.*

The Communication Mechanism is able to check incoming message for consistency. This shall be done by checking header information(is it a null frame, does the cycle number satisfy the check by cycle mask, is it correct channel) provided by Host.

Req. 6 *The Communication Mechanism shall be able to inform the Message Handler about check result.*

The Communication Mechanism shall be able to inform the Message Handler if the message is consistent or not. In both cases the Communication Mechanism shall receive command from the Message Handler that tells how to proceed depending on the check result and the ability of the Message Handler to currently receive new message (in this case send command to postpone the message).

1.3 Message RAM

Req. 7 *FlexRay messages and associated configuration and status information shall be saved in the Message RAM.*

Configuration information or bits include length of the buffer occupied for current message and header information. The status information represents the current status of the buffer in the Message RAM (i.e. it is locked or not locked). It can be locked if it is currently being used.

Req. 8 *Header and Payload bits shall be saved separately in the Message RAM.*

Messages have Header and Payload sections. These sections shall be saved separately in the RAM in a way that certain message's payload position in the RAM can easily be found by the header of this certain message.

1.4 Host Buffers

Req. 9 *Parameter set from Texas Instruments Drivers shall be a basis for implementation of the Host Buffers.*

Texas Instruments parameters for buffer's length in bits shall be the basis for the implementation. Nevertheless, the Host Buffers module's concept shall be feasible for extending the maximum payload length.

Req. 10 *Host Buffers shall allow 32 bit updates at a given time.*

Host Buffers shall be able to store 32 bits of payload bits at a given time.

Req. 11 *Host Buffers shall only store payload bits.*

Host Buffers shall not store header information of the Message.

1.5 Message Handler Requirements

Req. 12 *The Message Handler shall not violate any FlexRay timing and shall provide fair access for Host and PRT.*

The Message Handler shall always try to be as fast as possible and try to avoid any data locking for both of it's clients.

Req. 13 *The Message Handler shall always store the data that has to be sent in the Message RAM.*

The data that is being sent from one client to another shall be stored in the Message RAM in a way that later it can be identified by the Message Handler and be sent to the other client.

Req. 14 *Message Handler shall not allow client to use a buffer in the Message RAM if it is currently locked.*

The buffer in the Message RAM can be locked if it is currently being used by one of two clients. If the buffer is locked then Null frame is sent to the client as a response to his request.

Req. 15 *The Message Handler allows Host to configure the Message Handler.*

The Message Handler shall allow Host (one of the clients) to configure it's future behavior. Message Handler shall allow it only during the start time of the Message Handler. Any other configuration changes form Host part shall be ignored. The configuration part shall include how long the incoming messages shall be, the check code for incoming messages and the header length. Some of the configuration parts can be missing and if they are, then default values shall be applied for missing bits by the Message Handler. In case if Host does not provide configuration bits then default configuration shall be used.

Req. 16 *The Message Handler shall always ignore the incoming messages from Host or PRT if the message is a null frame.*

The Message Handler shall be always able to reject the message if it does not carry

any useful information, such as a null frame. This means that the payload length of the message is equal to 0.

Req. 17 *The Message Handler is always able to receive, send and store the incoming messages from CPU and PRT.*

The Message Handler shall always be able to receive the message from both of it's clients. It shall store the received messages and send it to Host or PRT when certain message, which is saved, is requested by one of them.

Req. 18 *The Message Handler is always able to read or write data from or to the Message RAM.*

Basic operations of communication to Message RAM are required. This includes read/write operations.

Req. 19 *The Message Handler is always able to identify requested message by its header information (frame ID, cycle ID) sent to it.*

The Message Handler is always able to identify which message is required from the RAM depending on which header information is sent from client. If the message doesn't exist in RAM them null message shall be sent in response.

Req. 20 *The Message Handler is always able to send status information to the HOST.*

The Message Handler shall be able to inform HOST in case some error happens and should be able to receive commands from the HOST in any situation and then react to this commands in a proper manner. The error might happen due to the fact that incoming message's length is not equal to the length booked for the message defined in configuration bits.

Req. 21 *In theory the Message Handler should be able to serve messages from both static and dynamic segments.*

The Message Handler shall only work with the static segment, thus only static segment messages shall be saved. Nevertheless, the Message Handler shall be able to be expanded to handle messages from both dynamic and static segments.

Req. 22 *The Message Handler shall be able to remember the position of the message in the RAM.*

The Message Handler shall be able to memorize or derive in a fast manner the address of the message inside the RAM by its header information.

Req. 23 *The Message Handler shall always be able to safely stop/pause it's operation.*

In case if Message Handler would have to stop/pause it's operation it should be able to safely stop writing or reading data to or from the RAM and block any incoming messages other than new status information from the HOST. This cases can be present when one of the clients tries to save a message in the buffer of the Message RAM and the message length is bigger than expected, according to configuration bits saved in RAM.

2 Concept

2.1 Introduction

This chapter describes the concept of the Message Handler mechanism. In addition modules that are used during the concept and implementation phase are presented. This shall help to understand the concept in general and give a starting point for a continuation of the thesis.

Modules

Input and Output Buffers These Buffers are used to transmit messages from HOST to the Message Handler and vice versa.

Message Handler This controls the data transfer between the input/output buffers and the Message RAM. In addition it controls transfer between the Transient Protocol Buffers and the Message RAM.

Message RAM This is a single ported RAM which stores the configuration and messages [Fujitsu (2007)].

Global Time Unit This is a common base for both channels that provides the microticks and macroticks, cycle counter and the timing control for the static segment of the communication cycle.

Transient Protocol Buffers This is the connection to the physical layer of the network. These buffers are used to transmit messages from FlexRay to the Message Handler and vice versa.

Frame Processing Unit Implements a mechanism that allows HOST to communicate with the Message Handler. In addition this module also acts as a security module that would prevent HOST from corrupting the data on the RAM.

Customer CPU Interface It connects a Host CPU to the Generic CPU Interface.

Generic CPU Interface It is used to control the range of signals to be used to send payload bits from Host to Message Handler and vice versa. Can be set to 8/16/32 bit interface.

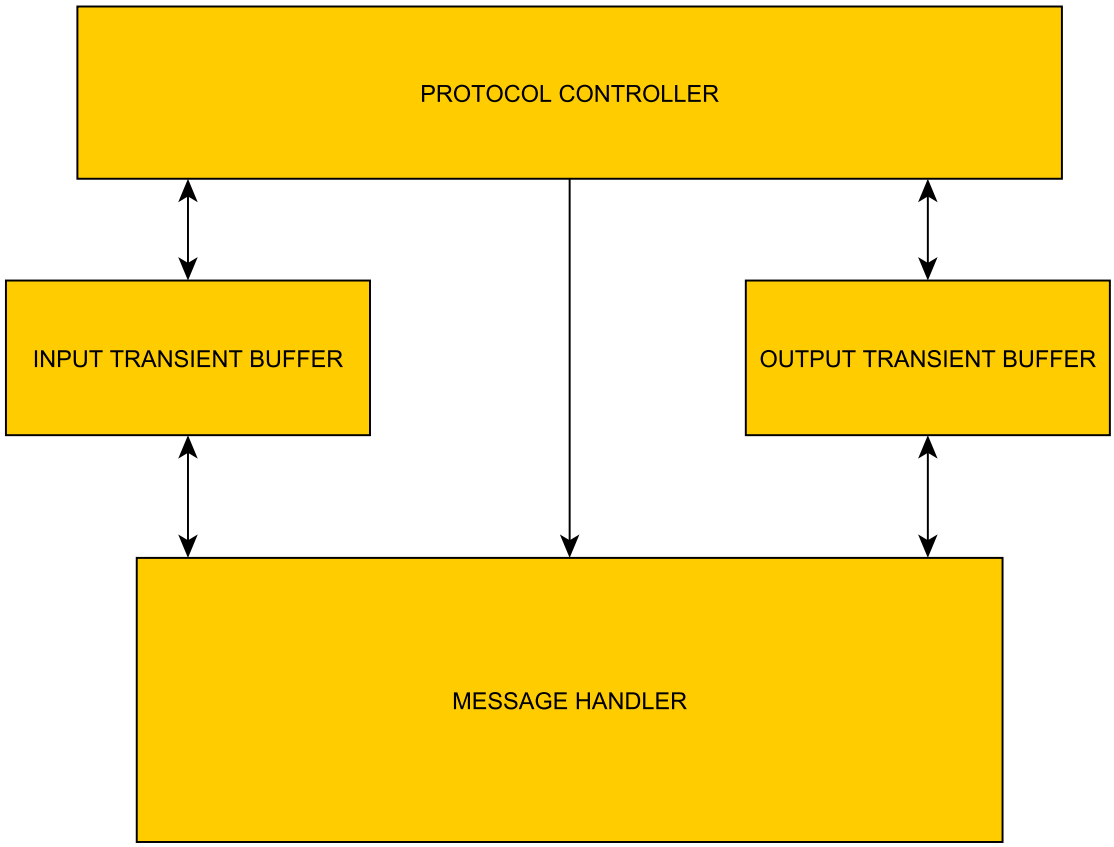

Figure 2.1: Protocol Controller and Message Handler connection concept

2.2 Protocol Controller to Message Handler Concept

The concept for the Protocol Controller connection to the Message Handler did not change throughout the project. The concept can be seen at Figure 2.1. This concept uses two transient buffers (TBF) to send and receive message from Protocol Controller (PRT). The Message Handler is directly connected to the PRT in order to be able to receive header from the PRT and one of the buffers (as requested by the requirement). This concept meets all the requirements for this project.

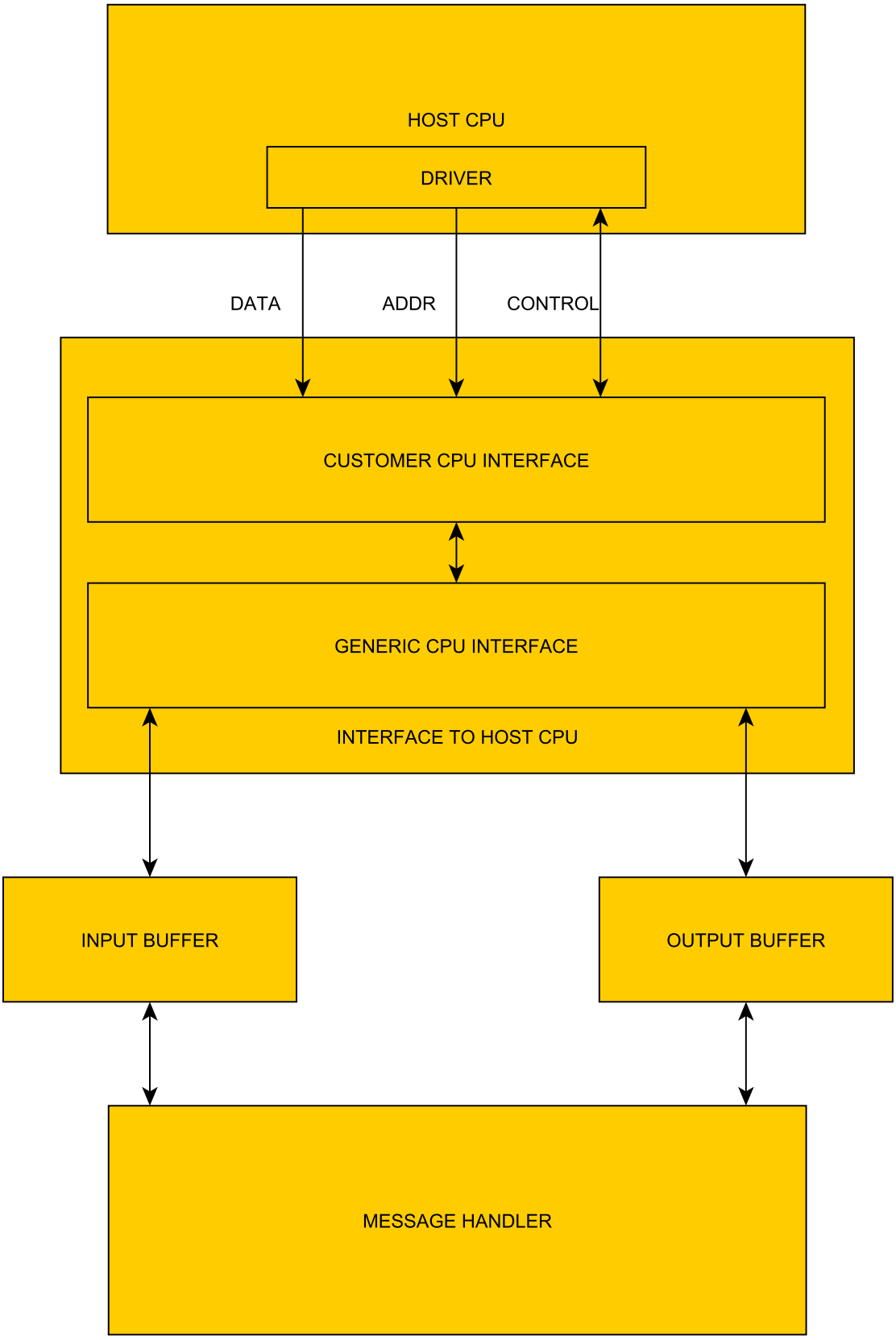

Figure 2.2: First Concept

2.3 First Concept

Figure 2.2 shows first concept that was originally thought to be used. It has an interface to Host CPU which itself can be configured. It also has input and output buffers which have shadow buffer. The shadow buffers is used when the main buffers is currently in use and locked. In this case shadow buffer becomes main buffer.

Advantages

1. Concept is feasible for channel A and B.
2. Host buffers allow 32 bit updates at a given time.

Disadvantages

1. The Host is not able to configure the Message Handler. Only default configuration can be applied.
2. The MH is not able to send status information to the HOST.
3. Incoming message from the Host cannot be checked for integrity (if it is null message or if it does not match the mask filter) before it reaches the Message Handler.
4. Host buffers do not store only payload bits.
5. The Message Handler cannot provide fair access to all its buffers due to the fact that the MH cannot inform the Host that it is not able to receive any additional payload bits at certain time.
6. The Message Handler cannot inform the Host that requested buffer is currently locked, hence it will only send null message in response.
7. The MH is not able to safely stop/pause its operation since it cannot send the status information to the HOST.

Hence this approach violates requirements that need to be fulfilled for this project.

2.4 Second Concept

Figure 2.3 describes the second approach. This approach was designed to get rid of disadvantages presented in the first concept. The Frame Processing Unit (FPU) is added to control the data flow between Host Interface and Message Handler.

This concept holds the advantages from the first approach and all requirements are fulfilled. The Interface module is used to make the system more flexible. The Customer CPU interface allows HOST to set Generic Interface to be used as 8/16/32 bits Generic CPU Interface which hence sets the bit width of the signal that will be sent to Message Handler via Input Buffer. Nevertheless, it requires additional development that is not required for this task. Only 32 bits width signal will be used.

2.5 Final Concept

Figure 2.4 describes the final concept. The interface is removed and the FPU is directly connected to the Host. This allows the Host to communicate with the Message Handler via FPU and does not require additional development of the interface. The FPU itself is a mechanism that allows Host to communicate with Message Handler. This concept meets all the requirements for this project.

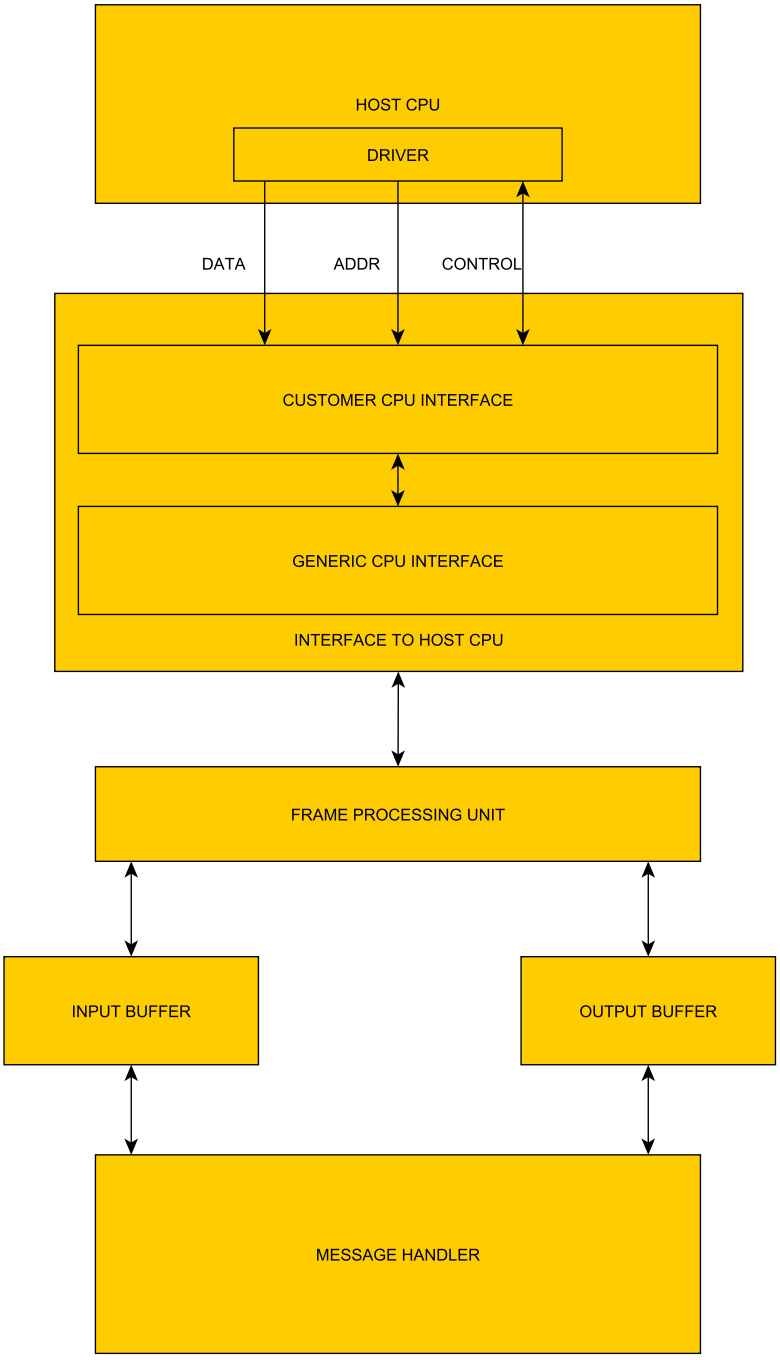

Figure 2.3: Second Concept

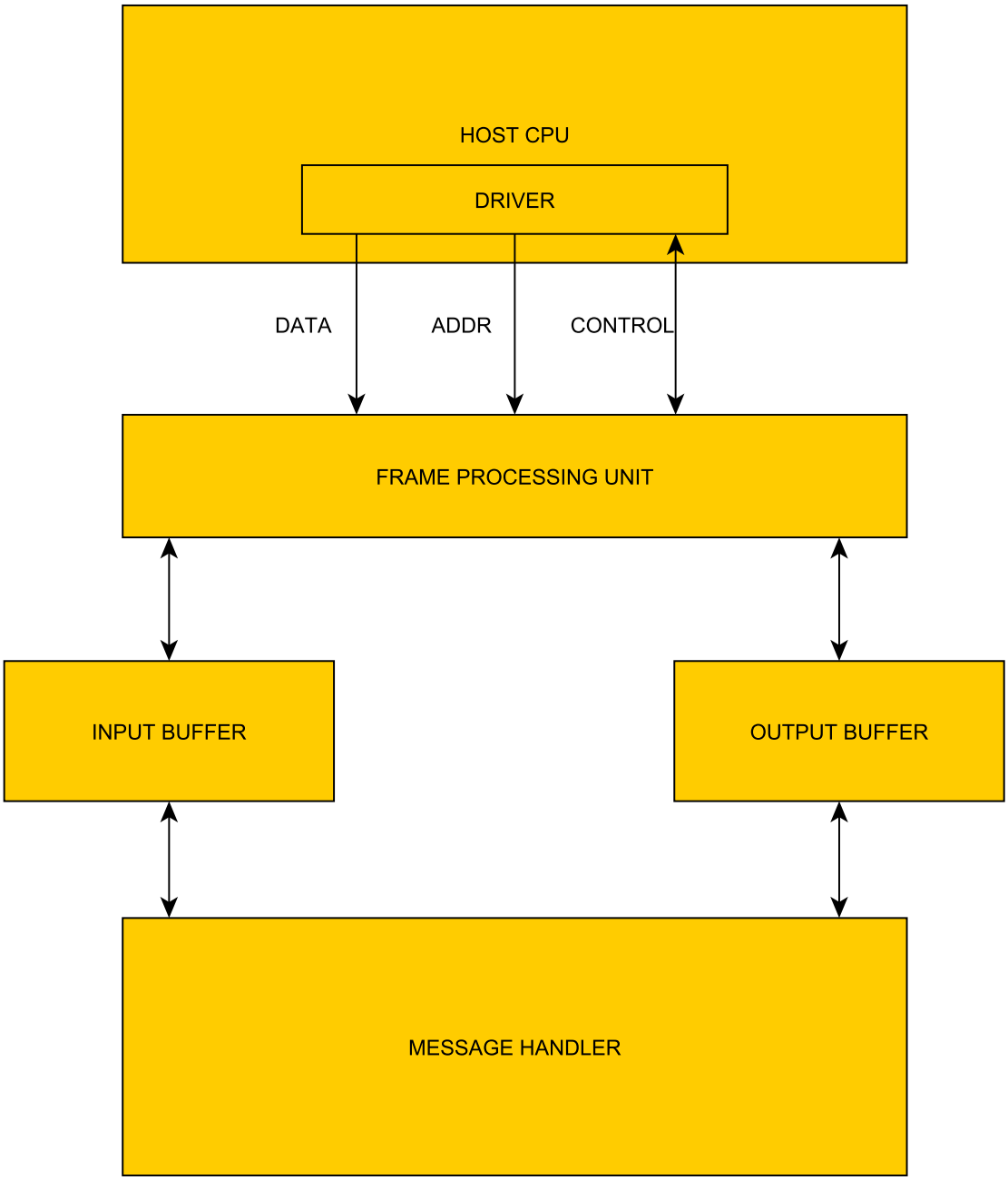

Figure 2.4: Final Concept

3 Design

3.1 Introduction

This chapter describes the design of the Message Handler system. The concept part is extended here to show the details of the system. In addition this chapter gives a broader view of the Message Handler.

3.2 Transitions

This section shall describe the connection between modules and give a clearer understanding of their signal's behavior by providing use cases. Description of use cases shall be separated into different parts to describe only the connection that is stated in certain subsection (i.e. in Subsection 3.2.1 for use case "HOST requests write access to a certain buffer" only the part that includes both HOST and FPU shall be described). This is done to make it easier to understand. The signals characteristics are describe in more details in following sections of this chapter.

3.2.1 Host Access to messages through the FPU

The host CPU is prevented from directly accessing the message buffers in the Message RAM to avoid conflicts between HOST accesses and FlexRay message reception or transmission. The host accesses are relayed via the Frame Processing Unit (FPU). The connection can be seen at Figure 3.1.

From the HOST's perspective several use cases can be described. First, it is when the HOST wants to configure the RAM. To do this HOST would need to send request to the FPU through the CONTROL_HOST_IN signal. Thus, FPU will know what operation HOST would like to make. Secondly, HOST needs to define the amount of buffers that shall be active in the RAM. The last step is to actually configure the RAM. To make the configuration HOST has to send the index of the buffer it wishes to configure (this is done by INDEX_IN signal) and, within the same clock cycle, the payload length (via the DATA_HOST_IN) for specified (by index)

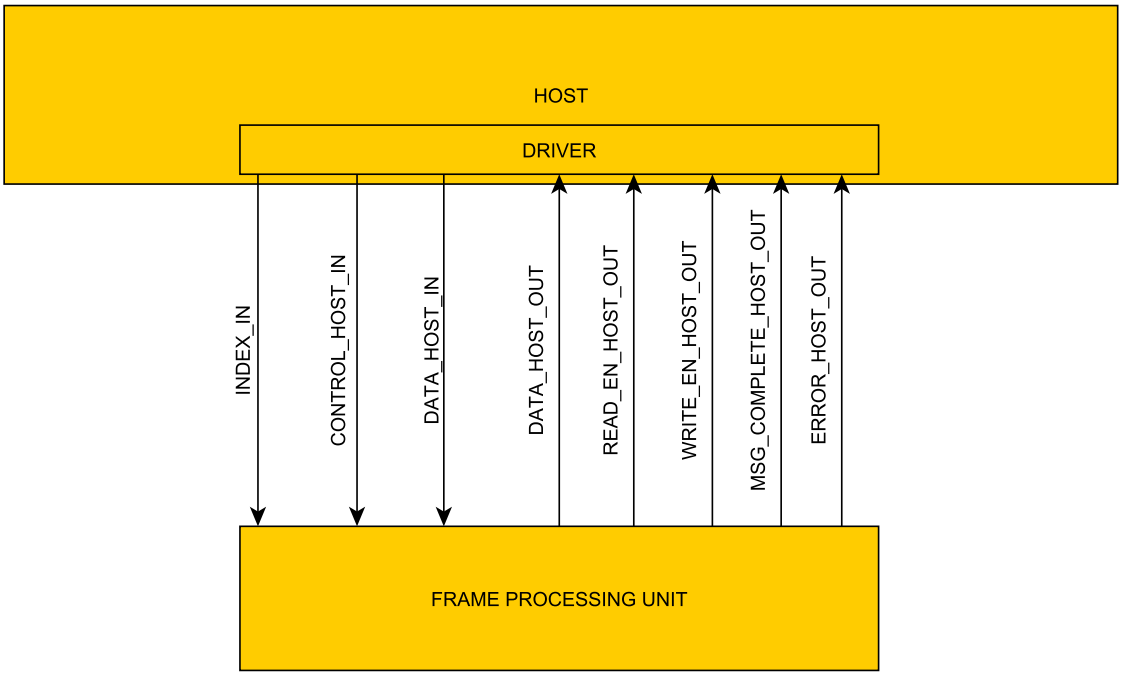

Figure 3.1: HOST CPU to Frame Processing Unit connection

buffer. The last described operation is repeated several times until every buffer is configured. When this point is reached FPU shall notify the HOST that the operation is finished by raising MSG_COMPLETE_HOST_IN flag. If Host does not reach that point, where it configures all the buffers, or tries to configure more buffers than are active, then ERROR_HOST_OUT flag shall be raised.

The second use case is when HOST requires write access to certain buffer. To notify the MH that HOST wants to update certain buffer it has to send appropriate request to the FPU via CONTROL_HOST_IN signal. At the same time HOST sends the index of the buffer that it wants to have access to. This information is processed by the FPU and if Message Handler allows access then FPU raises WRITE_EN_HOST_OUT flag to notify the HOST that it can start sending payload bits. HOST updates DATA_HOST_IN signal by rate of 32 bits per cycle. When FPU notices that amount of updated bits in the buffer equals to the length of the buffer FPU notifies HOST that the message is complete and HOST is not able to update the buffer any further. FPU can also pause the update of the buffer by HOST when MH requests to pause the write operation. To notify the HOST FPU sets the WRITE_EN_HOST_OUT flag to zero and keeps the MSG_COMPLETE_OUT flag zero as well (this way HOST knows that the message is not complete yet). Once MH allows HOST through FPU to continue the write operation write enable flag is raised and the operation is continued.

The last use case is when HOST requires read access to certain buffer. The flow here is similar to the write operation. HOST provides the index of the buffer it wants to read, the FPU processes the request and waits for the payload bits from the Message Handler. When FPU is ready to transmit these bits the READ_EN_HOST_OUT flag is raised. This way HOST knows that it is allowed to read the payload bits from the FPU. Once all payload bits of specified buffer are read the FPU raises MSG_COMPLETE_HOST_OUT flag.

3.2.2 Data transfer from INPUT BUFFER TO MESSAGE HANDLER

Input buffer is a FIFO with a depth of 2. This was considered to be enough for required task. Input buffer is used to transfer payload and configuration bits from Frame Processing Unit to the Message Handler (Figure 3.2). Several use cases can be applied to describe the particular role of the Input Buffer and corresponding interaction between FPU and MH.

During the configuration phase FPU sends command to the Message Handler that configuration phase has started. This is done by sending request via CONTROL_MH_OUT signal. The INDEX_OUT defines which buffer is currently configured. Next step is to transmit the payload length for each buffer. FPU sends these bits through the Input Buffer to the Message Handler. Using DATA_IB_OUT signal FPU sends bits representing the payload length to the Input Buffer and at the same time READ_EN_IB_OUT flag is set for one clock cycle to notify the buffer that it can send received configuration bits to the Message Handler. This is done

to prevent Input Buffer from reading incorrect data. Then, during the next clock cycle, INPUT BUFFER sets the EMPTY_IB_IN flag to zero and transmits payload bits to the Message HANDLER via DATA_IB_IN signal. This process is repeated until all buffers are configured. The FULL_IB_IN signal is used by the FPU to determine whether IB is full or not.

Similar operation is done during buffer update (from HOST) phase. However, this time INDEX_OUT is only set once at the beginning to define the buffer that has to be updated. In addition Input Buffer now receives payload bits from the FPU and transmits them to the MH.

3.2.3 Data transfer from MESSAGE HANDLER to OUTPUT BUFFER

Connection from the Message Handler to the Output Buffer is similar to what was described in Subsection 3.2.2. This can also be seen from the Figure 3.2. The purpose of the Output Buffer is to deliver payload bits from the Message Handler to the FPU for certain buffer that is required by the FPU. The Frame Processing Unit sends read request via the CONTROL_MH_OUT signal and, at the same time transmits the index of the buffer that FPU wants to have access to. Then Message Handler starts sending payload bits through the Output Buffer at a rate of 32 bits per cycle. This process is continued until all payload bits for required buffer are transmitted to the FPU.

3.2.4 Data transfer from and to FlexRay Protocol Controllers

The two Transient Buffers (TBF IN/OUT) are used to buffer the data for transfer between the FlexRay Protocol Controller and the Message Handler. Each TBF is build as a double buffer. The structure of the TBF's is the same as in Input and Output Buffers (Figure 3.3). The operation flow is almost the same with several differences.

First, PRT transmits not the index of the buffer it wants to have access to, but the header of required buffer. The index of the buffer is derived by the MH from the header.

Secondly, and more important, is the fact that Protocol Controller is not able to do the same check operations FPU does. Thus, Message Handler has to calculate the amount of incoming/outgoing payload bits in order to check if the message is complete or not.

In addition, Protocol Controller does not have as much control commands as the FPU does (i.e. FPU is not only able to request write and read but also configure and pause the operation cycle of the Message Handler).

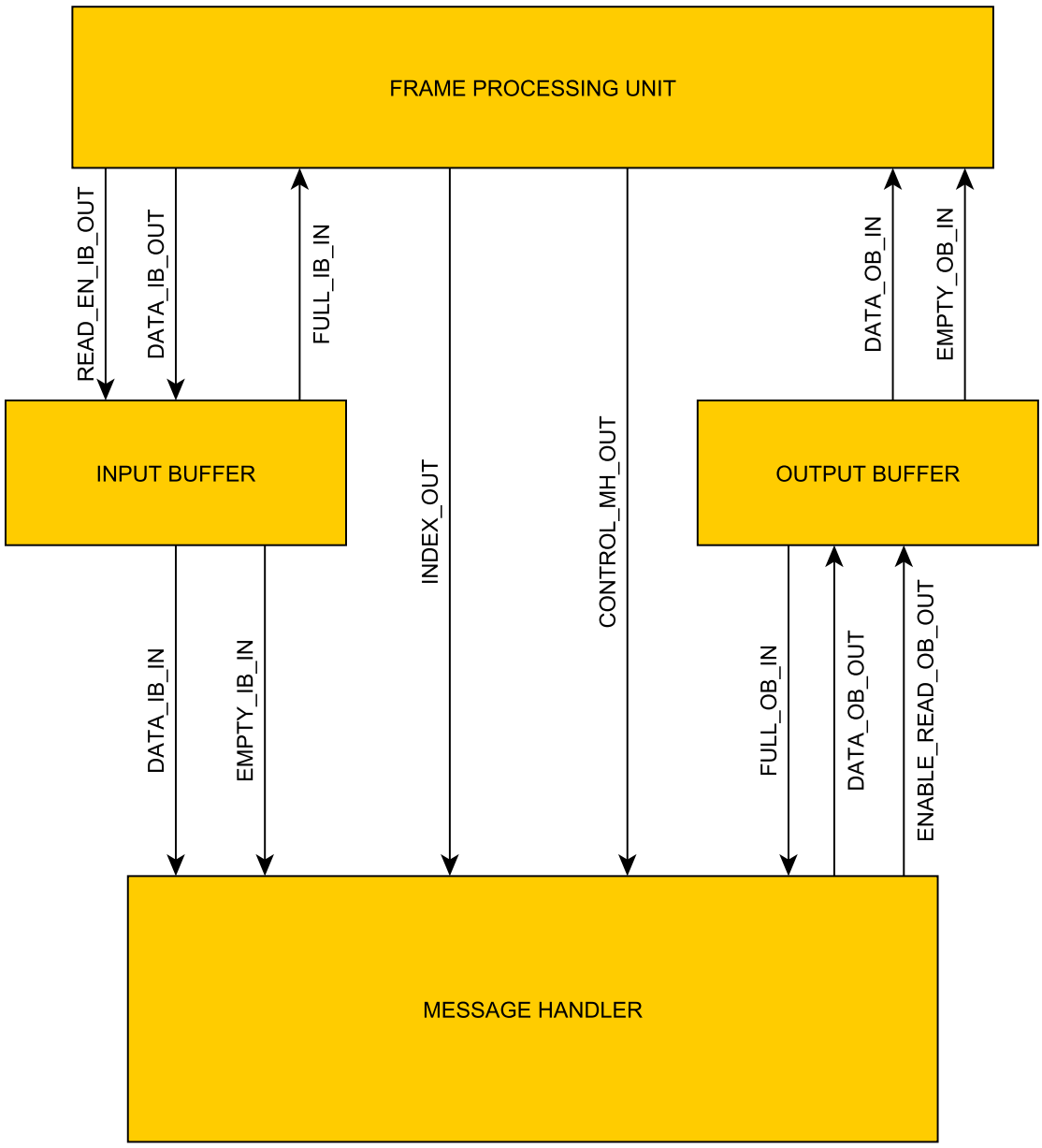

Figure 3.2: Frame Processing Unit to Message Handler connection

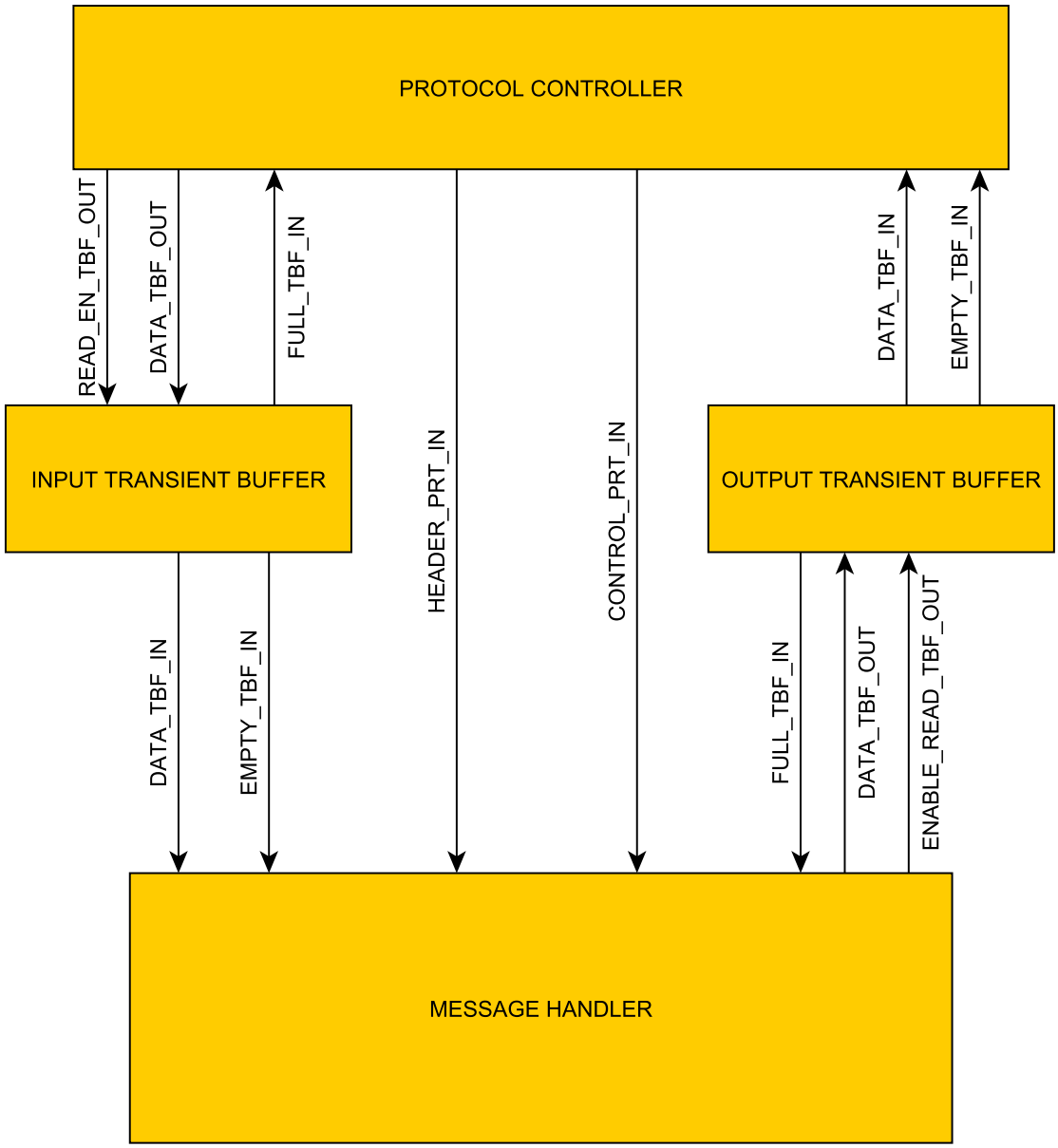

Figure 3.3: Message Handler and Protocol Controller Connection

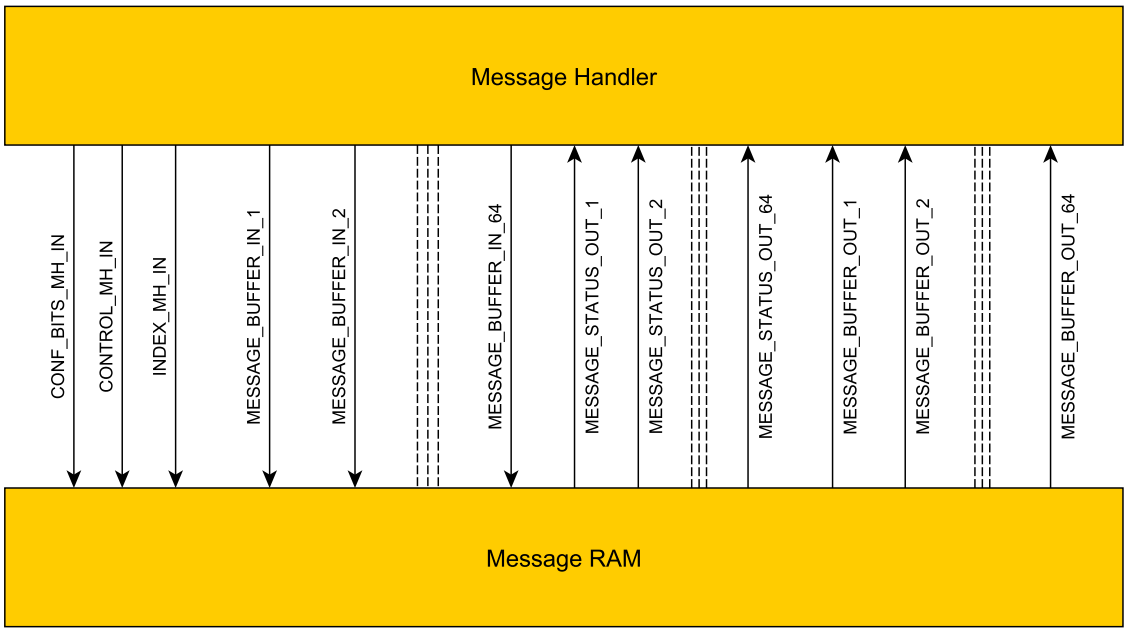

Figure 3.4: Message Handler and Message RAM connection

3.2.5 Data transfer from Message Handler to Message RAM

Message RAM can only be accessed by the Message Handler. This gives MH ability to prevent any half updated message to be read or to update the message that is currently being read. The description of the connection can be found at Figure 3.4.

The Message RAM can store up to 64 messages. The number of active messages is set during the configuration phase. TO start configuration phase MH sets the CONTROL_MH_IN signal to respective value. Then, first the amount of active messages is set via the INDEX_MH_IN signal. The next step is to configure every buffer. The configuration bits (i.e. including payload length) are sent via the CONF_BITS_MH_IN signal and the index of the buffer that is being configured is set via the INDEX_MH_IN signal.

Second use case is when HOST wants to read or write certain buffer. First it requires read/write operation by using CONTROL_MH_IN signal and sets the index of the required buffer via the INDEX_MH_IN. The next step is to read the status of required buffer. MESSAGE_STATUS_OUT_X (where X is the number of required buffer) is updated by the Message RAM and checked by the MH. If the status is ’0’ then buffer can be read/updated. If the status is ’1’ the buffer is busy and MH has to wait until the status will be ’0’. If the buffer was not busy then Message Handler can perform required operation:

Write Operation Send header and payload bits via the MESSAGE_BUFFER_IN_X (where X is the index of required buffer).

Read Operation Read header and payload bits via the MESSAGE_BUFFER_OUT_X (where X is the index of required buffer).

3.3 Frame Processing Unit

This section shall describe the signals that are used for the Frame Processing Unit (FPU). The description shall be brief as it is already well described in the tables. In addition broader discussion on the FPU shall take place in the realization section.

Signals

FPU can be described as a gate that stands between HOST and MH. The reason being is to prevent HOST from direct access to the RAM. This module can be evaluated as request handler and security gateway. This is well reflected on its signals (Table 3.1). Described set of signals show that the FPU is able to handle the communication between HOST and MH. Nevertheless, FPU is also able to make its own decisions based on the current status of the MH and HOST.

Table 3.1: Frame Processing Unit signals characteristics

Signal	Input/Output	From/To	Length(bits)
INDEX_IN	INPUT	HOST	5
CONTROL_HOST_IN	INPUT	HOST	3
EMPTY_OB_IN	INPUT	OB	1
FULL_IB_IN	INPUT	IB	1
DATA_HOST_IN	INPUT	HOST	32
DATA_OB_IN	INPUT	OB	32
INDEX_OUT	OUTPUT	MH	5
CONTROL_MH_OUT	OUTPUT	MH	3
READ_EN_IB_OUT	OUTPUT	IB	1
DATA_HOST_OUT	OUTPUT	HOST	32
DATA_IB_OUT	OUTPUT	IB	32
READ_EN_HOST_OUT	OUTPUT	HOST	1
WRITE_EN_HOST_OUT	OUTPUT	HOST	1
MSG_COMPLETE_HOST_OUT	OUTPUT	HOST	1
ERROR_HOST_OUT	OUTPUT	HOST	1

The description of these signals can be found at Table 3.2. 32 bits width signals are used to transmit bits from HOST to the MH and vice versa. The FPU is able to be controlled by the HOST via the CONTROL_HOST_IN signal. If FPU considers requested operation acceptable then depending on the command FPU shall inform the MH using the signal CONTROL_MH_OUT. It can be write, read, configuration and other requests.

Most of the time the INDEX_OUT signal is reflection of the INDEX_IN excluding some cases that are described in section Realization.

Table 3.2: Frame Processing Unit signals description

Signal	Values
INDEX_IN	Used to define the index of the message in the RAM Host wants to have access to.
CONTROL_HOST_IN	Used to tell the FPU what HOST would like it to do (write, read, configure, pause and etc.)
EMPTY_OB_IN	Used to tell if OUTPUT BUFFER is currently EMPTY and hence no data is there to be read.
FULL_IB_IN	Used to check if INPUT BUFFER is currently FULL and hence no new data can be written to it.
DATA_HOST_IN	Used to receive payload bits from the HOST.
DATA_OB_IN	Used to receive payload bits from the MH.
INDEX_OUT	Used to tell MH to which message FPU would like to have access.
CONTROL_MH_OUT	Used to tell the MH that FPU wants to read/write message. Also is used to pause or reset the MH if Host requests so.
READ_EN_IB_OUT	Used to tell INPUT BUFFER that it can write data to the MESSAGE HANDLER.
DATA_HOST_OUT	Used to send the payload bits received from Output Buffer to the HOST.
DATA_IB_OUT	Used to send the payload bits received from HOST to the Input Buffer.
READ_EN_HOST_OUT	Used to tell host if it can read data or not.
WRITE_EN_HOST_OUT	Used to tell Host if it can write data or not.
MSG_COMPLETE_HOST_OUT	Used to tell HOST if the write or read operation is completed.
ERROR_HOST_OUT	Used to tell Host about the error happened.

FPU provides flags for the HOST that describe the current status. READ_EN_HOST_OUT flag tells HOST that there are valid payload bits that can be read. WRITE_EN_HOST_OUT

allows HOST to update certain buffer. This flag can be set to zero if the Output Buffer is currently full and cannot accept any new payload bits. MSG_COMPLETE_HOST_OUT informs HOST that the required operation cycle has finished. This is applicable for both write, read and configuration operations. ERROR_HOST_OUT is raised when HOST tries to overflow the amount of payload bits at the buffer in the RAM. This can be viewed when HOST has already updated the buffer but still tries to add more payload bits to it.

Table 3.3: Frame Processing Unit signals values

Signal	Values
INDEX_IN	00000 to 11111
CONTROL_HOST_IN	{000 : RESET , 001: PAUSE , 010 : CONTINUE , 011 : WR, 100 : RD , 101 : CONF, 110 : DEFAULT_CONF, 111 : IDLE}
EMPTY_OB_IN	{ 0 : NOT EMPTY, 1 : EMPTY}
FULL_IB_IN	{ 0 : NOT FULL, 1 : FULL}
DATA_HOST_IN	Payload bits
DATA_OB_IN	Payload bits
INDEX_OUT	00000 to 11111
CONTROL_MH_OUT	{ 000: PAUSE , 001 : CONTINUE , 010 : WR, 011 : RD, 100 : CONF, 111 : IDLE }
READ_EN_IB_OUT	{ 0 : DISABLE, 1 : ENABLE}
DATA_HOST_OUT	Payload bits
DATA_IB_OUT	Payload bits
READ_EN_HOST_OUT	{ 0 : DISABLE, 1 : ENABLE}
WRITE_EN_HOST_OUT	{ 0 : DISABLE, 1 : ENABLE }
MSG_COMPLETE_HOST_OUT	{ 0 : Message is not complete, 1 : Message is complete}
ERROR_HOST_OUT	{ 0 : NO ERROR, 1 : ERROR}

3.4 Message Handler

The main purpose of the Message Handler is to give fair access to its clients (FlexRay and HOST) and provide them with requested buffers. The MH is the only module that has access to the RAM. This is done to prevent HOST and FlexRay from directly accessing the buffers in the RAM.

Signals

Message Handler has 4 signals responsible for sending and receiving payload bits (Table 3.4). They can be divided into two parts. First part includes DATA_IB_IN and DATA_OB_OUT. These two signals are used to transmit and receive payload values from HOST. Second part consists of DATA_TBF_IN and DATA_TBF_OUT which are used to exchange payload bits with the FlexRay.

Table 3.4: Message Handler signals characteristics

Signal	Input/Output	From/To	Length(bits)
DATA_IB_IN	INPUT	IB	32
DATA_TBF_IN	INPUT	TBFI	32
INDEX_FPU_IN	INPUT	FPU	5
HEADER_PRT_IN	INPUT	PRT	17
CONTROL_FPU_IN	INPUT	FPU	2
CONTROL_PRT_IN	INPUT	PRT	2
EMPTY_IB_IN	INPUT	IB	1
FULL_OB_IN	INPUT	OB	1
DATA_OB_OUT	OUTPUT	OB	32
DATA_TBF_OUT	OUTPUT	TBFO	32
ENABLE_READ_OB_OUT	OUTPUT	OB	1
ENABLE_READ_TBF_OUT	OUTPUT	TBFO	1

Nevertheless, Message Handler recognizes required buffer differently for HOST and FlexRay. For the HOST it is very straight forward. Index is fed to the MH and the buffer with corresponding number is taken from the RAM. However, for the FlexRay it is not the same. FlexRay provides header information instead of an index. The buffer that is required is derived from the header bits. Several algorithms can be applied here to solve this issue. First is that the MH remembers which header refers to which buffer during the configuration stage. Second option is to provide function in VHDL that will derive the index from the header. And the final approach is to save this information in the RAM. The last approach is more appropriate as it will help to reduce the duplication of the configuration files (by giving direct access for the FPU to the configuration part of the RAM).

MH gives opportunity to be controlled by both FlexRay and HOST (through the FPU). However, FlexRay is only able to request read or write operation using this CONTROL_PRT_IN signal. On the other hand, HOST is able to pause the work flow (as well as continue if paused), configure and reset the MH in addition to the write and read request.

Nevertheless, HOST is restricted as well. The HOST is not allowed to reconfigure the RAM once the configuration phase has passed. Thus, it is only possible for HOST to request configuration when Message Handler is in the state IDLE.

3.5 Input Buffer

Input Buffer is typical FIFO (First In, First Out) with depth being equal to 2 [Daniel (2013)]. As the name suggests the first 32 bits written into a FIFO will be the first one to appear on the output. The FIFO here is used as it is expected that HOST will have higher frequency than the Message Handler.

Signals

In order to write the data to the Input Buffer first the data has to be pushed to the DATA_FPU_IN and then the READ_EN has to be set t high for one clock cycle. At the same time the input is saved in internal signal and can be read by the Message Handler. The EMPTY signal is as well triggered to the value high.

When the FULL flag goes high, this means that the Input Buffer's memory is full and will not accept any more writes until data is read. If data is written while the FULL flag is high it will be ignored.

3.6 Output Buffer

Output Buffer, as Input Buffer, is a typical FIFO with the depth of 2 [Daniel (2013)].

Signals

The same technique is applied here with only opposite direction. Now the MH writes to this buffer and FPU reads and transmits the bits to the HOST.

3.7 Message RAM

Direct access to the message buffer in the message RAM is not available in order to avoid collisions between the HOST access to the message RAM and transmission and reception of FlexRay messages. Access is protected via the Message Handler. The message RAM can store up to 64 message buffers. The message buffers are currently only used for static messages. However, this can be extended to the RAM separated by static and dynamic messages. The message RAM module is ported in the Message Handler module.

Signals

Message RAM has 64 inputs for updating the buffer and 64 outputs for reading it. This was made to allow MH write or read different buffers at the same time. At the same time status of every buffer is implemented as well. The reason being is that the MH can be informed if one of its clients (HOST or FlexRay) is trying to read certain buffer that is currently being updated. This shall prevent non-integral buffers to be read. The same is true for the opposite case, when one of the clients is reading certain buffer, and during this time another client is trying to update it. This might have driven to wrong data to be read. However this is as well prevented by checking the current status of the buffer (is it busy or not) before accessing it.

The RAM should be able to be configured during respective phase. CONTROL_MH_IN signal is used to tell the RAM that Message Handler would like to configure it.The next step is to actually send the configuration bits to the message RAM. This is done through the CONF_BITS_MH_IN signal.

CONTROL_MH_IN is also used when MH would like to update or read certain message buffer. The index is written to this signal and the next step is that the message RAM prepares buffer for the update and raises corresponding busy flag.

Table 3.5: Message Handler signals description

Signal	Input/Output
DATA_IB_IN	Payload bits sent by HOST via Input Buffer.
DATA_TBF_IN	Payload bits sent by FLEXRAY via Transient Buffer.
INDEX_FPU_IN	Index of the message required by HOST.
HEADER_PRT_IN	Header of the message required by FLEXRAY.
CONTROL_FPU_IN	Control bits from FPU telling what action HOST wants to do. Including write, read, pause (can be caused by error in FPU or HOST) and continue (which is also IDLE) which is set when there is no action required or when MH waits in PAUSE state and HOST allows to continue. STOP is done by RESET input signal.
CONTROL_PRT_IN	Control bits from Protocol Controller telling what action FLEXRAY would like to make: no action, write, read. It can require write, and after providing HEADER proceed with requiring read without pausing the MH from doing both actions at the same time (unless FLEXRAY requires)to read and write the same message in the MESSAGE RAM.
EMPTY_IB_IN	Is used when the FPU wants to transfer payload bits from the HOST and sends the index to the MH telling to which message it would like to have access. The Message Handler receives the index and then reacts when the Input Buffer is not Empty anymore. Only then MH starts to read values from the IB.
FULL_OB_IN	Is used when the HOST wants to read some message with specified index. If the FULL_OB_IN signal is low then the MH sends the payload bits. If the FULL_OB_IN is high then MH waits until message is read by FPU and the EMPTY signal is low again. Then the MH starts transferring payload bits to the OB.
DATA_OB_OUT	Payload bits sent to the HOST via OB.
DATA_TBF_OUT	Payload bits sent to the FLEXRAY via TBFO.
ENABLE_READ_OB_OUT	Is set to 1 for one clock cycle when MH has sent payload bits to OB and FPU can start reading them.
ENABLE_READ_TBF_OUT	Is set to 1 for one clock cycle when MH has sent payload bits to TBFO and PRT can start reading them.

Table 3.6: Message Handler signals values

Signal	Value
DATA_IB_IN	Payload bits
DATA_TBF_IN	Payload bits
INDEX_FPU_IN	00000 to 11111
HEADER_PRT_IN	FRAME_ID (11 bits) + CYCLE_COUNT (6 bits)
CONTROL_FPU_IN	{ PAUSE : 00, WR : 01, RD : 10, CONTINUE = IDLE : 11 }
CONTROL_PRT_IN	{ IDLE : 00, WR : 01, RD: 10 }
EMPTY_IB_IN	{ 1 : EMPTY , 0 : NOT EMPTY }
FULL_OB_IN	{ 1: FULL , 0 : NOT FULL }
DATA_OB_OUT	Payload bits
DATA_TBF_OUT	Payload bits
ENABLE_READ_OB_OUT	{ 1 : ENABLE, 0 : DISABLE }
ENABLE_READ_TBF_OUT	{ 1 : ENABLE, 0 : DISABLE }

Table 3.7: Input Buffer signals characteristics

Signal	Input/Output	From/To	Length (bits)
DATA_FPU_IN	INPUT	FPU	32
READ_EN	INPUT	FPU	1
DATA_MH_OUT	OUTPUT	MH	32
EMPTY	OUTPUT	MH	1
FULL	OUTPUT	FPU	1

Table 3.8: Input Buffer signals description

Signal	Description
DATA_FPU_IN	Payload bits from FPU.
READ_EN	Enables read operation for the MH.
DATA_MH_OUT	Payload bits to MH.
EMPTY	Is set to sow MH if there is no data to be read.
FULL	Is set to show if FPU can or cannot write new data.

Table 3.9: Input Buffer signals value

Signal	Values
DATA_FPU_IN	Payload bits
READ_EN	{1 : ENABLE, 0 : DISABLE}
DATA_MH_OUT	Payload bits
EMPTY	{1 : FIFO IS EMPTY, 0 : FIFO IS NOT EMPTY}
FULL	{1 : FIFO IS FULL, 0 : FIFO IS NOT FULL}

Table 3.10: Output Buffer signals characteristics

Signal	Input/Output	From/To	Length (bits)
DATA_MH_IN	INPUT	MH	32
READ_EN	INPUT	MH	1
DATA_FPU_OUT	OUTPUT	MH	32
EMPTY	OUTPUT	FPU	1
FULL	OUTPUT	MH	1

Table 3.11: Output Buffer signals description

Signal	Description
DATA_MH_IN	Payload bits from FPU.
READ_EN	Enables read operation for the FPU.
DATA_FPU_OUT	Payload bits to FPU.
EMPTY	Is set to sow FPU if there is no data to be read.
FULL	Is set to show if MH can or cannot write new data.

Table 3.12: Output Buffer signals value

Signal	Values
DATA_MH_IN	Payload bits
READ_EN	{1 : ENABLE, 0 : DISABLE}
DATA_FPU_OUT	Payload bits
EMPTY	{1 : FIFO IS EMPTY, 0 : NOT EMPTY}
FULL	{1 : FIFO IS FULL, 0 : NOT FULL}

Table 3.13: Message RAM signals characteristics

Signal	Input/Output	From/To	Length (bits)
INDEX_MH_IN	INPUT	MH	6
CONF_BITS_MH_IN	INPUT	MH	32
MESSAGE_BUFFER_IN_1-64	INPUT	MH	32
CONTROL_MH_IN	INPUT	MH	2
MESSAGE_BUFFER_OUT_1-64	OUTPUT	MH	32
MESSAGE_STATUS_OUT_1-64	OUTPUT	MH	1

Table 3.14: Message RAM signals description

Signal	Description
INDEX_MH_IN	Is used to define to which buffer MH would like to have access.
CONF_BITS_MH_IN	Configuration bits to setup the amount of buffers that will be used and their payload length. Only possible during configuration stage.
MESSAGE_BUFFER_IN_1-64	Is used to obtain and further save payload and header information for certain buffer.
CONTROL_MH_IN	Is used to tell Message Ram to which buffer Message Handler would like to have access to.
MESSAGE_BUFFER_OUT_1-64	Is used to deliver payload and header information for certain buffer.
MESSAGE_STATUS_OUT_1-64	Is used to deliver current status of certain buffer.

Table 3.15: Message RAM signals value

Signal	Values
INDEX_MH_IN	000000 to 111111
CONF_BITS_MH_IN	Configuration bits
MESSAGE_BUFFER_IN_1-64	Header bits + Payload bits
CONTROL_MH_IN	00 : WR, 01 : RD, 10 : CONF
MESSAGE_BUFFER_OUT_1-64	Header bits + Payload bits
MESSAGE_STATUS_OUT_1-64	{0 : BUFFER CAN BE USED, 1 : BUFFER IS BUSY}

4 Realisation

4.1 FPU

Frame Processing Unit (FPU) shall be the main concern for the realization part as it has never been developed to be used as a part of the Message Handler system. The main purpose of the FPU is to control incoming bits from the HOST and handle them respectively.

The FPU can be described as a communicator and controller between HOST and MH. FPU is able to receive commands from the HOST and send respective request, if made so, to the MH. It is able to get the index of requested message from HOST and send respective command and index to the MH.

Overall, FPU can be divided in several states : IDLE, CONFIG, DEFAULT CONFIG, PASSIVE, ACTIVE (write or read) and PAUSE. This section shall describe all states that are used in FPU and provide with the respective algorithm for each state.

4.1.1 IDLE state

IDLE state is used to reset and/or start the FPU. Only during this state HOST can require configuration. As a Moore output all signals retain there default values (which are there reset values).

This state can only redirect to the default or HOST configuration state. This can be viewed on 4.2.

FPU can jump to IDLE from any other state including PAUSE. During the PAUSE state HOST can require to stop the operation and hence FPU will be reseted.

4.1.2 CONFIG state

CONFIG state is accessed whenever HOST requires to configure the MH and Message RAM, thus no default configuration is applied. When entering this state MH is notified that the HOST would like to start the configuration of the Message RAM.

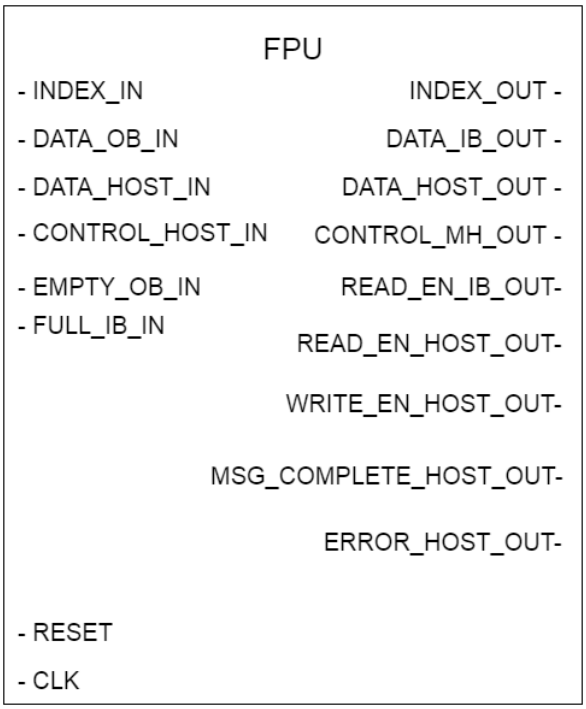

Figure 4.1: Frame Processing Unit design

During the first stage of the configuration HOST sets the amount of message buffers it would like to see in ram. This value should not exceed maximum or minimum value it can take (this is described in details in section 4.1.2). When the amount of buffers is set FPU sends this information to the MH, which now knows how many buffers shall be used.

The second stage of state CONFIG is required to set the payload length for each buffer. Due to the fact that the implementation should be feasible for dynamic segments as well; HOST can configure different payload sizes for various buffers. One clock cycle is required to send all bits that MH requires in order to configure the RAM. During this clock cycle FPU sends index of the buffer that is currently to be configured and its respective payload size.

Length check

This is the first sub-state of CONFIG. This state is responsible for checking the amount of buffers HOST would like to configure for the RAM. The algorithm of this sub-state can be seen on 4.3.

The minimum RAM length is defined as a constant in VHDL file as is used to check whether HOST has provided value bigger than minimum or not. If the value provided by HOST is less than minimum, then the constant minimum value is used as a length of the RAM.

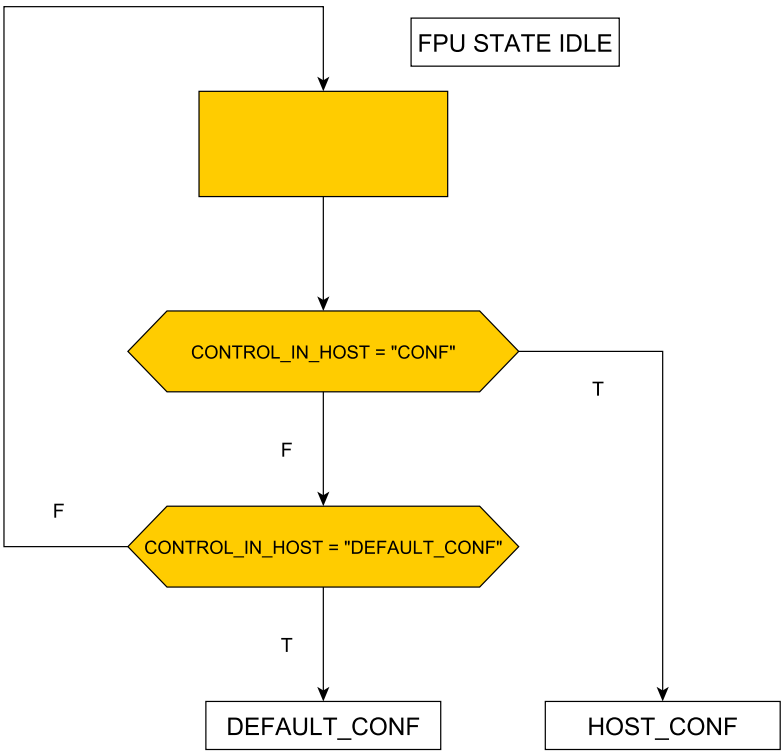

Figure 4.2: Frame Processing Unit state IDLE

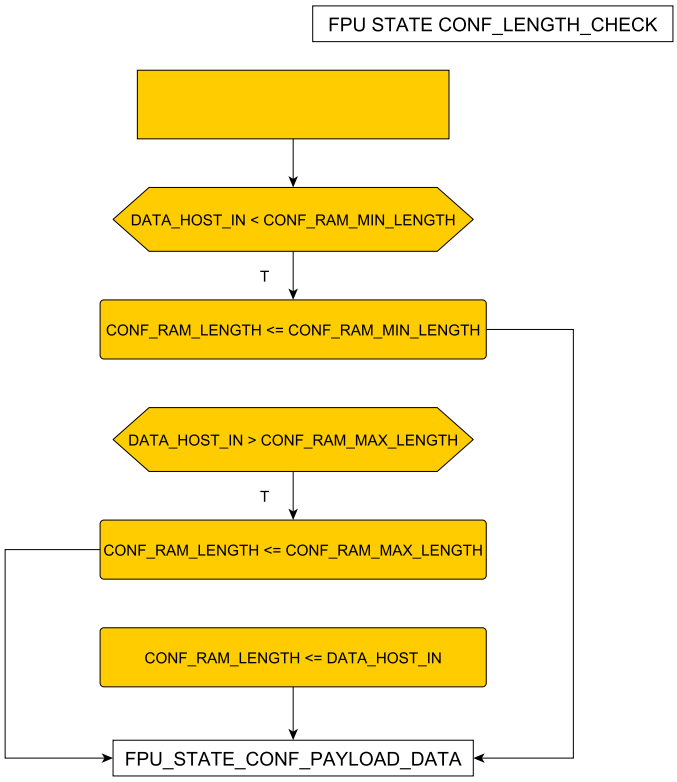

Figure 4.3: Frame Processing Unit state CONF_LENGTH_CHECK

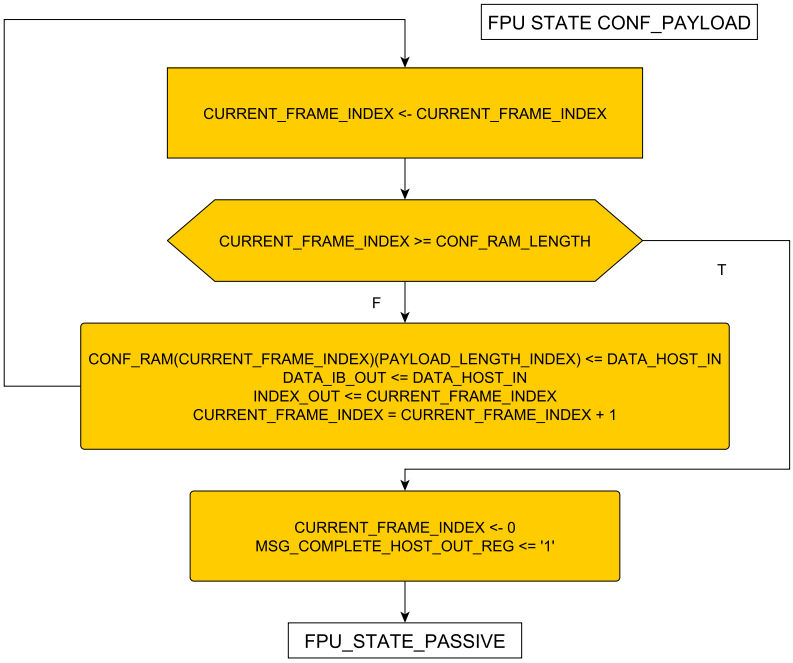

Figure 4.4: Frame Processing Unit state CONF_PAYLOAD_DATA

```
CONSTANT CONF_RAM_MIN_LENGTH : INTEGER := 32;
```

The same applies for the maximum value.

```
CONSTANT CONF_RAM_MAX_LENGTH : INTEGER := 128;
```

Payload data

This is the second sub-state which is accessed after the length check. This stage runs for n clock cycles. Where n is the length of the RAM defined in previous sub-state. The algorithm can be found at 4.4.

The main concern here is the fact that all configuration data are duplicated in the FPU and in the MH. This is done because both of these modules require these data to be saved internally. FPU requires these values to check incoming requests from the HOST. On the other hand MH uses it to configure the RAM according to the configuration bits. However, in future updates this can be changed, in order to remove duplications.

This can be done in a way that all configuration data are stored in the RAM and FPU has direct access to it. Another option would be to divide RAM into two parts. One of them would

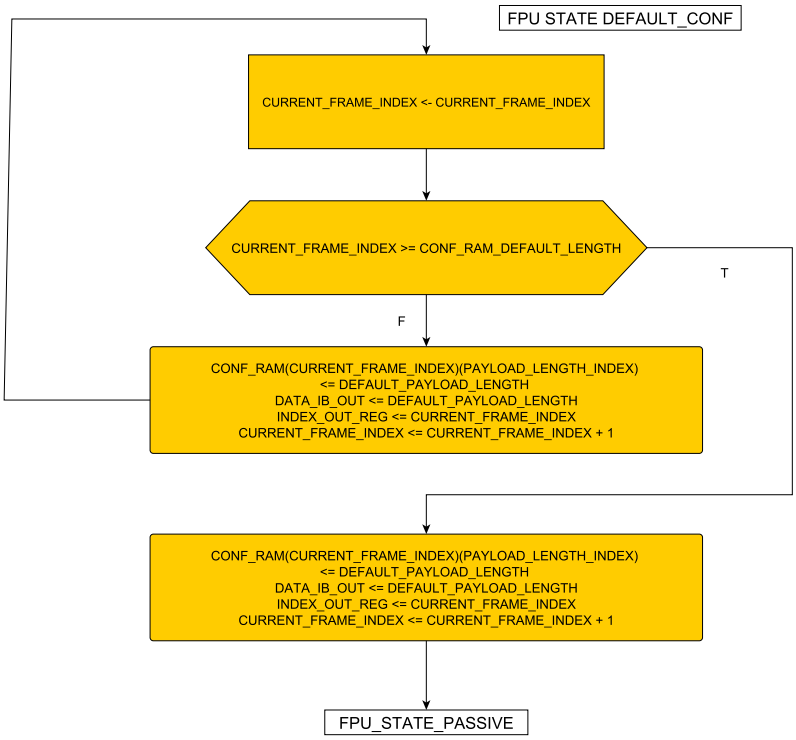

Figure 4.5: Frame Processing Unit state DEFAULT_CONF

be for storing the payload data and another for storing configuration data. Thus FPU will only have access to the configuration bits and hence no security issues can be caused by it.

4.1.3 DEFAULT CONF state

Default configuration is applied if HOST does not want to configure the RAM itself and hence requires FPU to configure the RAM using default values. The logic behind this can be seen at 4.5.

As it can be seen from the name of current state this configuration applies default values to the MH and hence to the RAM.

The default values for the RAM are as follows:

```
CONSTANT CONF_RAM_DEFAULT_LENGTH : INTEGER := 56;
CONSTANT DEFAULT_PAYLOAD_LENGTH : INTEGER := 72;
```

The configuration default length is derived from the payload length. The RAM's maximum size is 4096 bits and hence: $4096/56 \sim 72$.

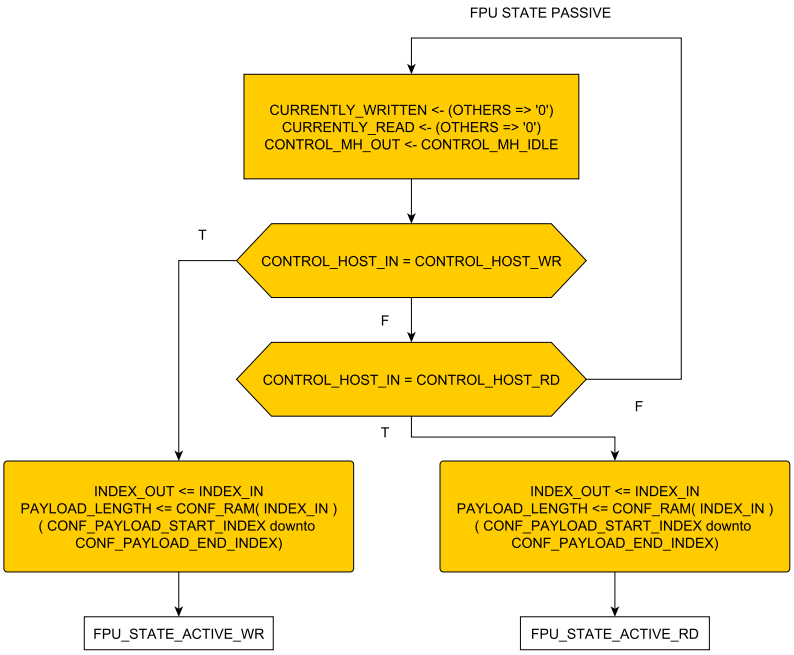

Figure 4.6: Frame Processing Unit state PASSIVE

Despite the fact that it currently uses static payload length for the whole configuration cycle, it is still preferable to keep the algorithm as it is right now, i.e. it takes one cycle for each buffer to be configured. This is done due to the fact that the default configuration might be changed in future updates as it was stated that the concept shall be feasible for dynamic segments as well.

4.1.4 PASSIVE state

This stage is used when RAM is already configured and HOST can request any new operation at any time. This stage is entered no matter what configuration type was selected by HOST. The algorithm is very straight forward and can be seen at Figure 4.6.

While FPU stays in this state it sends default control command to the MH that currently nothing happens. It is only during this state that HOST can require write or read operation. Once required HOST has to provide index of the buffer it wants to have access to. When index is received FPU grabs payload length for required buffer. This values will be used during required operation (write or read) in order to have an idea on how long the payload length is for current buffer and only accept as many bits as is needed.

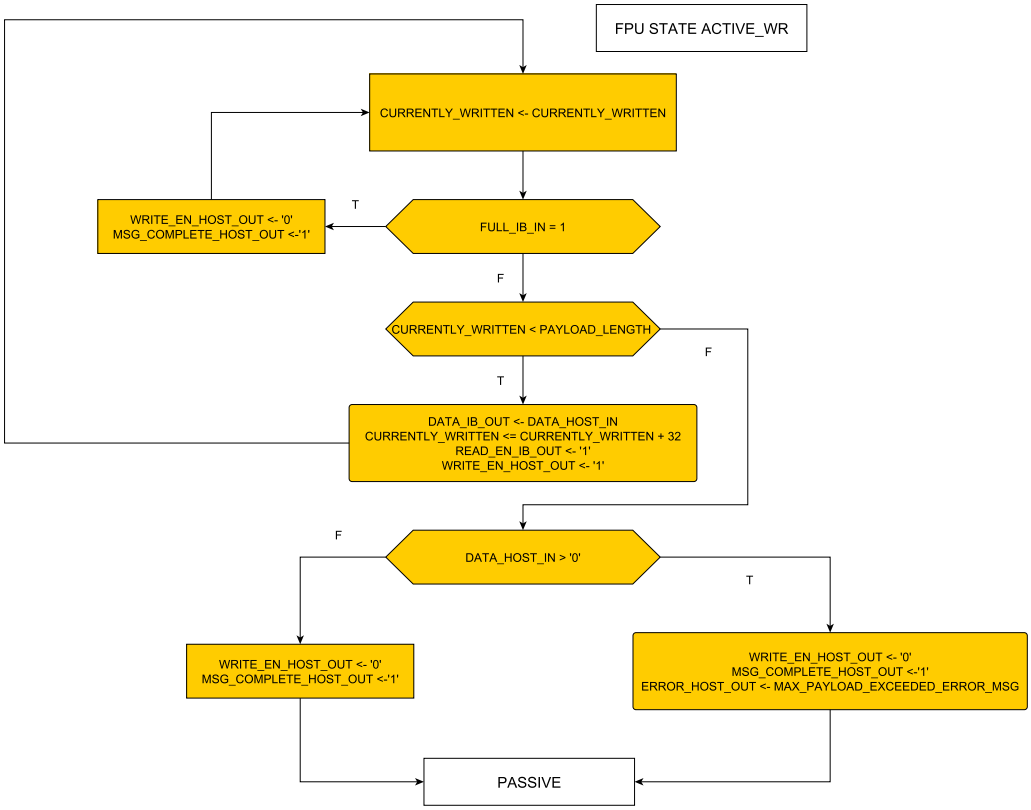

Figure 4.7: Frame Processing Unit state ACTIVE_WR

When FPU receives new operation required by HOST it follows to the ACTIVE state that has two sub-states.

4.1.5 ACTIVE state

Active state is where read and write operations are done. This state cannot be considered as completely valid state. The reason being is that there is no order or connection between sub-states. This is done to only make the concept easier to understand.

WRITE ACCESS state

Write access state is used when HOST requires to update certain buffer (index is provided in state passive). The logic of this state can be seen at Figure 4.7. During this state FPU accepts payload bits from HOST and transmits them via Input Buffer. The payload bit update rate is 32 bits per cycle. If Input Buffer is full then it rises a full flag and FPU tells HOST to

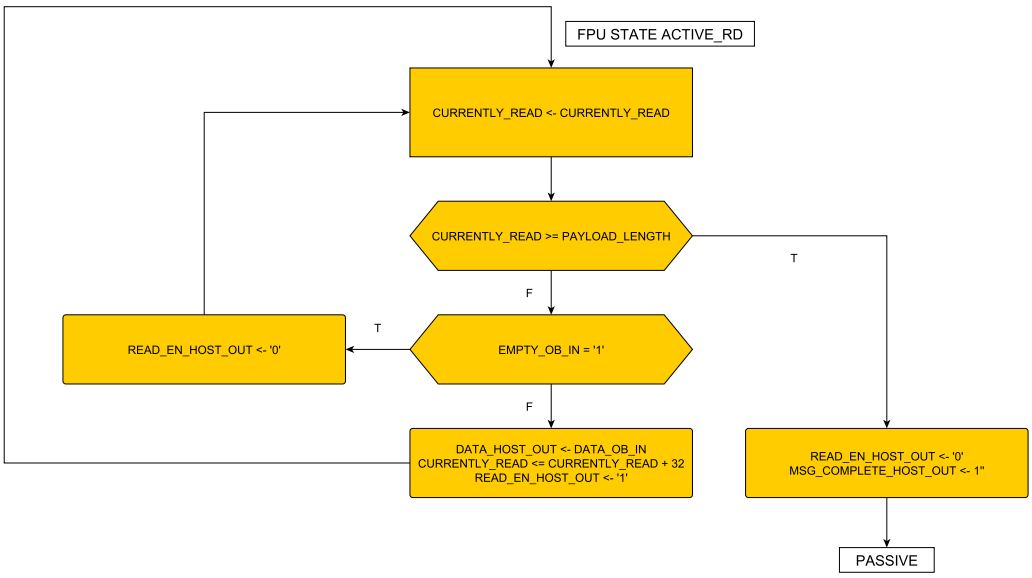

Figure 4.8: Frame Processing Unit state ACTIVE_RD

pause sending payload bits. Once the flag is cleared HOST can continue sending payload bits.

To keep track of how many bits were already transfered this state uses an internal counter that increments by 4 bytes (or 32 bits) with every clock cycle when HOST is allowed to send payload bits. With every clock cycle this counter is compared to the payload length retrieved in state PASSIVE. Once this values is reached FPU tells HOST to stop sending any additional payload values. IF HOST neglects this command the error flag is raised and no more additional bits are accepted.

READ ACCESS state

Read access state is used when HOST requires to read certain buffer, the index of which has been defined in state Passive. The algorithm of this state can be seen at Figure 4.8.

During this state FPU transmits payload bits from MH via Output Buffer. The update rate is 32 bits per cycle. If Output Buffer is empty (i.e. did not receive any payload bits from the MH yet) then respective flag is raised in Output Buffer and FPU tells HOST(by clearing read enable flag) that currently there is nothing to transmit. Once the flag is cleared and FPU is able to read new data from the Output Buffer; FPU continues to send payload bits that it reads from the Ouput Buffer.

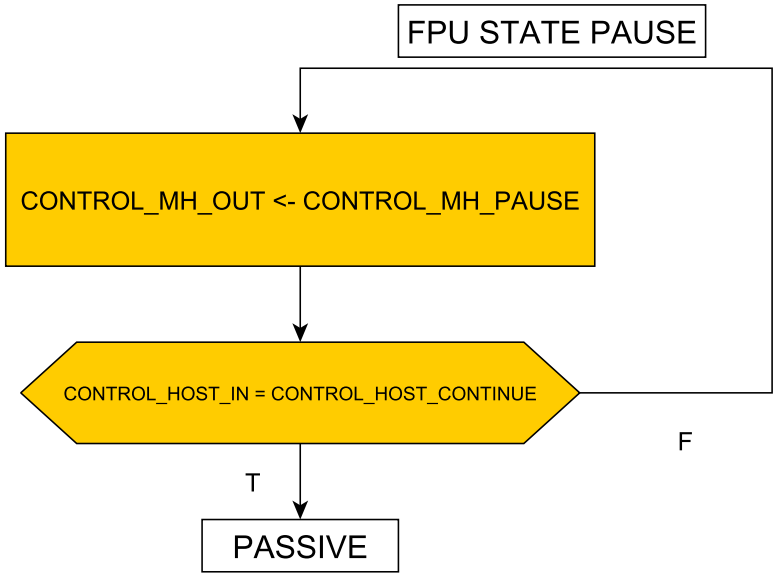

Figure 4.9: Frame Processing Unit state PAUSE

In order to keep track of how many payload bits are currently read FPU uses internal counter that is incremented every clock cycle by 4 bytes (32 bits) when it is able to transmit new payload bits to the HOST. With every clock cycle this values is compared to the payload length defined in state passive. When this value is reached then FPU informs HOST, by raising message complete flag, that there will be no additional payload bits coming. Once this is done FPU changes its state back to the Passive and is ready to accept new commands.

4.1.6 PAUSE state

This state is accessed only when HOST requires to pause the operation. This can be used when there is an internal error in HOST that would require the FPU and MH to currently pause its operation. The logic is very straight forward and can be seen at Figure 4.9. During this state FPU sends pause command to the MH. This state can be leaved either by resetting the FPU module or if HOST sends command to continue the operation. If HOST requires to continue then FPU will change its state to PASSIVE and all previous operations (write or read) would be reseted as well, and data will be considered as non integral.

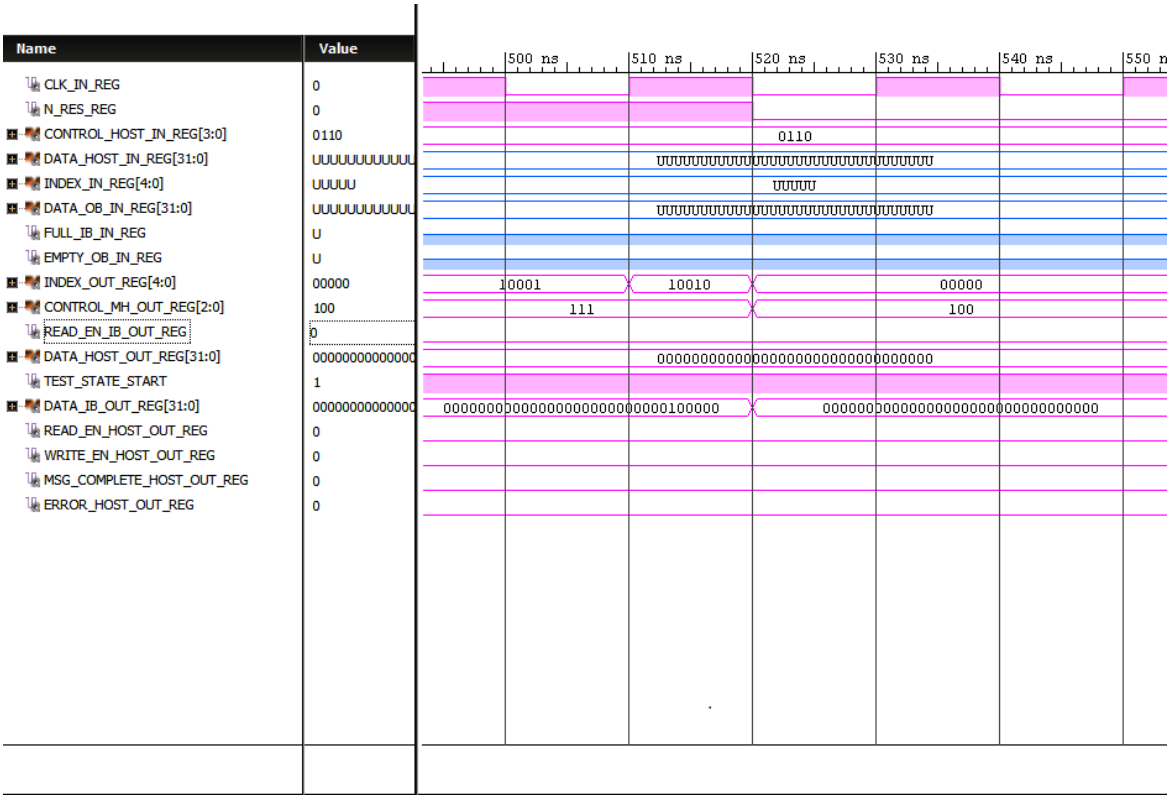

Figure 4.10: Frame Processing Unit Reset Test

4.2 Frame Processing Unit Tests

This section's goal is to introduce test results. The code that has been used for this aim can be found in Appendix A.1.

4.2.1 Reset Test

This test will show the results of reseting the FPU module. The output can be seen at Figure 4.10. The result is what has been anticipated it to be. All output signals are reseted to their default values. The state of the FPU after reset operation has changed to IDLE. Reset can be performed from any state and it includes state Pause as well.

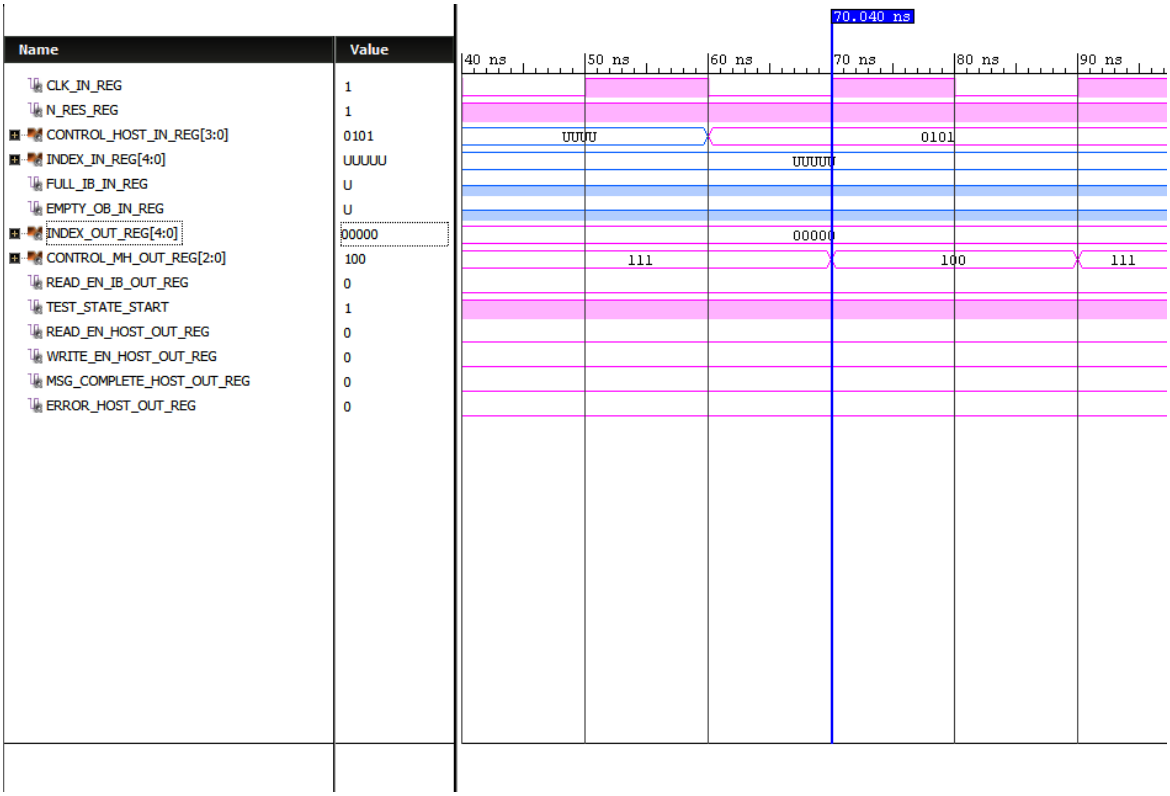

Figure 4.11: Frame Processing Unit CONF state start command transmission

4.2.2 Configuration Test

This test was executed to check the behavior of the FPU when HOST wants to configure the RAM and hence no default values shall be applied. When HOST requests FPU to start the configuration phase then FPU sends corresponding command to the MH. This can be seen by analyzing the CONTROL_MH_OUT_REG signal on Figure 4.11. First HOST required to start the configuration phase by setting CONTROL_HOST_IN_REG signal to "0101" value. This value stands for the configuration start command (not default). When this signal is received FPU changes its state from IDLE to CONF and updates CONTROL_MH_OUT_REG to be "100" for one clock cycle. This signal tells MH that HOST is starting the configuration phase.

By next step HOST is required to provide amount of buffers it would like to use in RAM. This is done during the first configuration stage. Once the value is received via DATA_HOST_IN_REG it is checked and transmitted to the Input Buffer (Figure 4.12).

FPU tests whether the amount of buffers required by HOST is within acceptable range. The

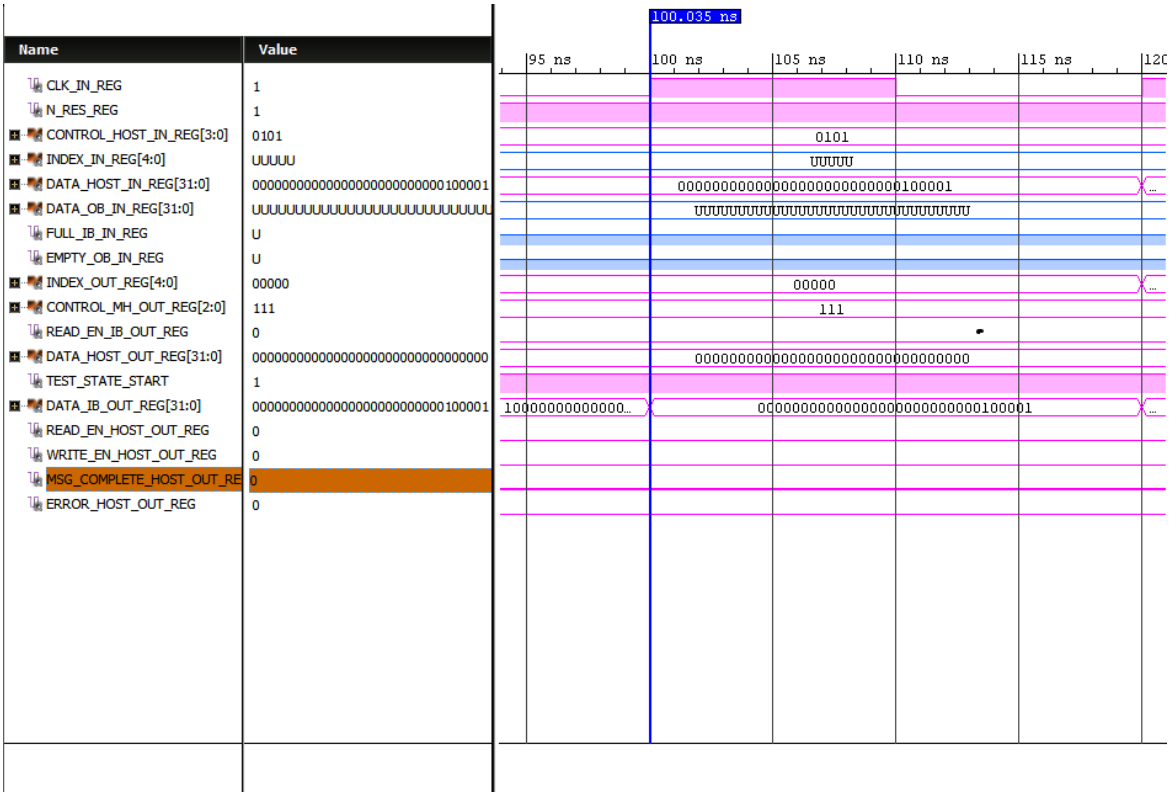

Figure 4.12: Frame Processing Unit amount of Message Buffers configuration bits

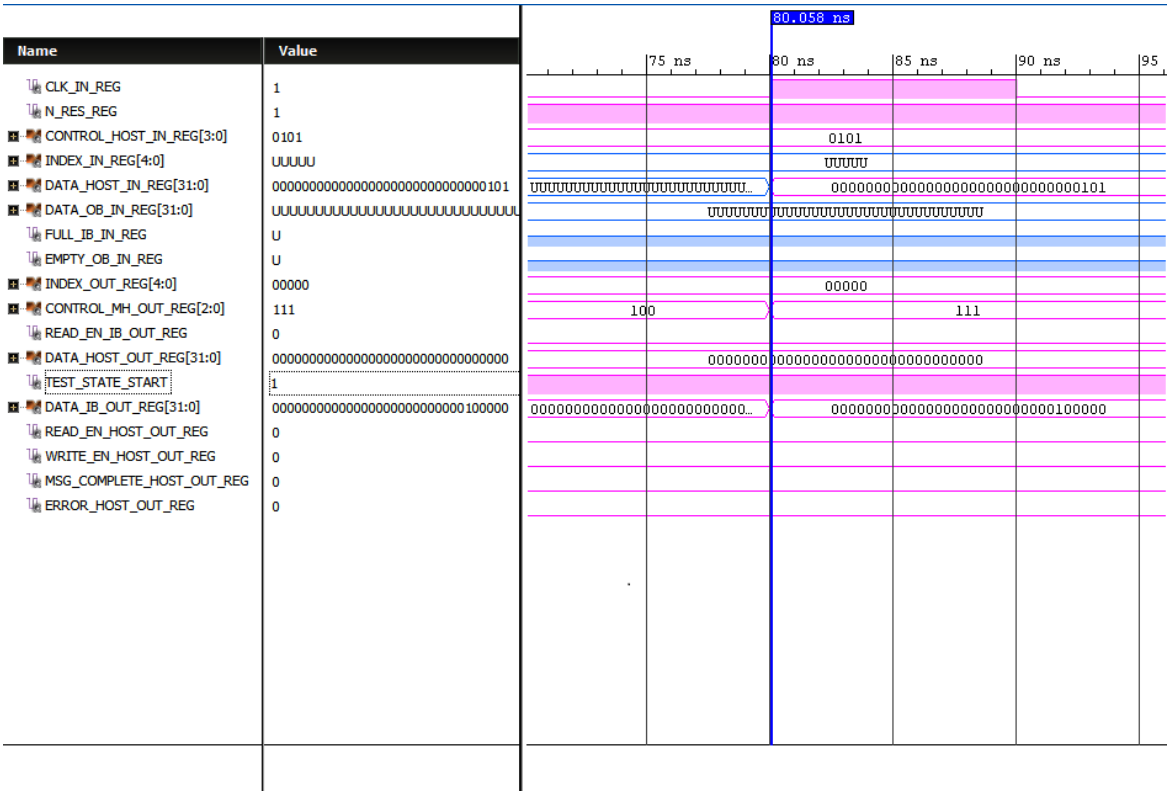

Figure 4.13: Frame Processing Unit Test Message Buffer amount configuration bits lower than minimum

acceptable range stands between minimum and maximum value it can take. On Figure 4.12 this condition is met. However additional tests were made to check the behavior of the FPU.

First it was checked how FPU will react if the desired amount, provided by HOST, is lower than acceptable minimum. The results can be seen at Figure 4.13.

Despite the fact the required amount was set to be "101" via DATA_HOST_IN_REG signal the output was "100000" (DATA_IB_OUT_REG). This has happened FPU internally compared incoming value to the minimum:

```
CONSTANT CONF_RAM_MIN_LENGTH : INTEGER := 32;
```

So it can be seen that the incoming amount is less than declared minimum and hence the default minimum value was used.

Equal test was executed for testing the behavior of the FPU when value bigger than the default maximum was required. The result of this test can be seen at Figure 4.14. As

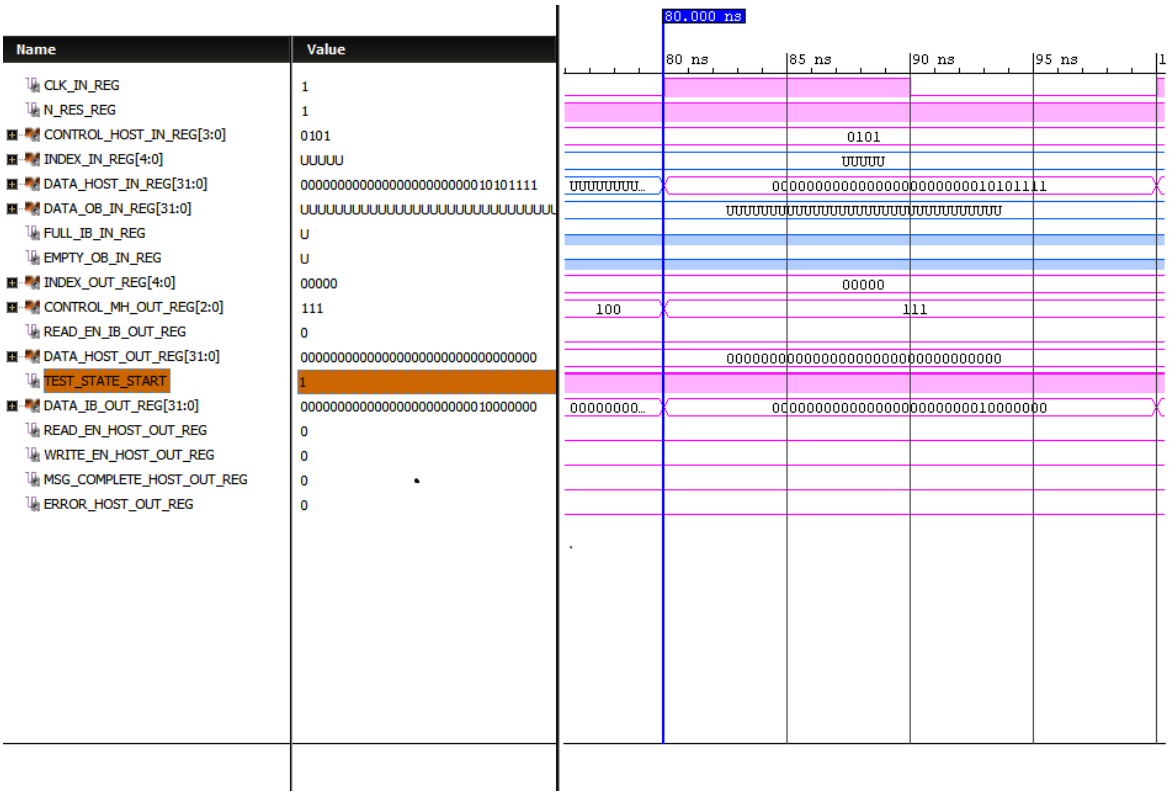

Figure 4.14: Frame Processing Unit Test Message Buffer amount configuration higher than maximum

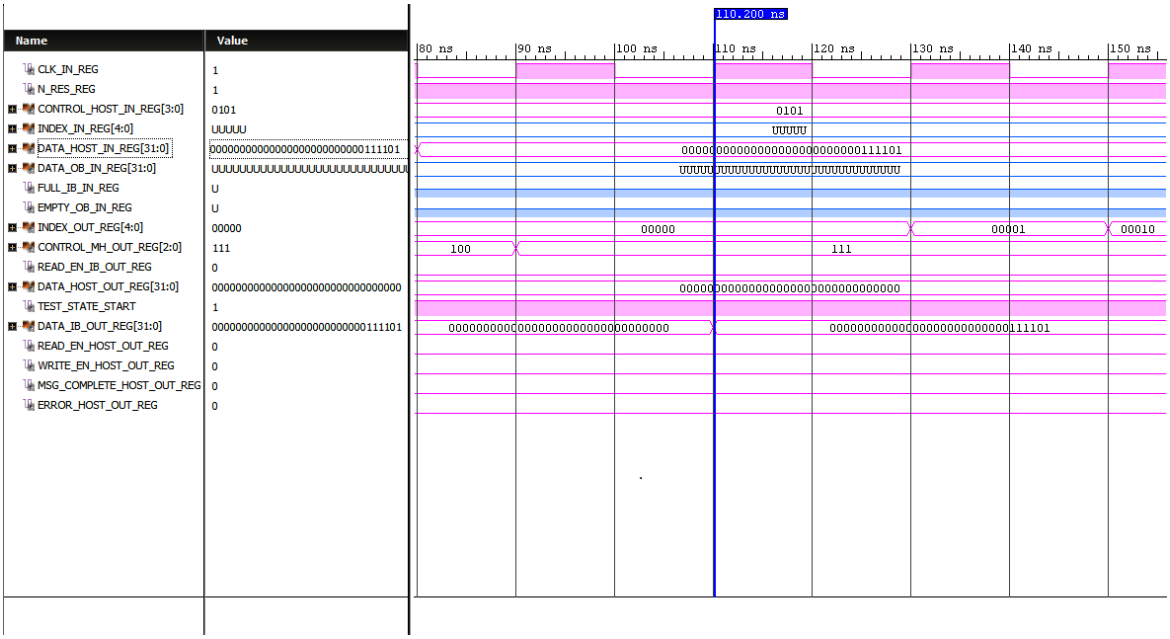

Figure 4.15: Frame Processing Unit configuration bits transmission from HOST to the Message Handler

it can be seen from the Figure 4.14 value "10101111" was fed to the FPU via signal DATA_HOST_IN_REG. This value is bigger than declared maximum:

```
CONSTANT CONF_RAM_MAX_LENGTH : INTEGER := 128;
```

Hence the maximum value shall be applied. This can be seen on Figure 4.14 on output signal DATA_IB_OUT_REG. This is the behavior that was expected.

After FPU has transmitted the bits for the amount buffers that shall be used the next stage of the configuration state takes place. During this phase payload length bits are sent by the HOST through FPU to the MH.

Test was executed to evaluate the behavior of the FPU during this stage. The result can be seen at Figure 4.11.

It can be seen that one clock cycle after the amount of buffers was set FPU starts to transmit both payload size and current buffer's index which is incremented by one each clock cycle in order to be able to properly distinguish the payload size bits for different buffers. The index increments before it reaches the RAM length value. When it is done then the configuration phase is finished and FPU changes its state to Passive.

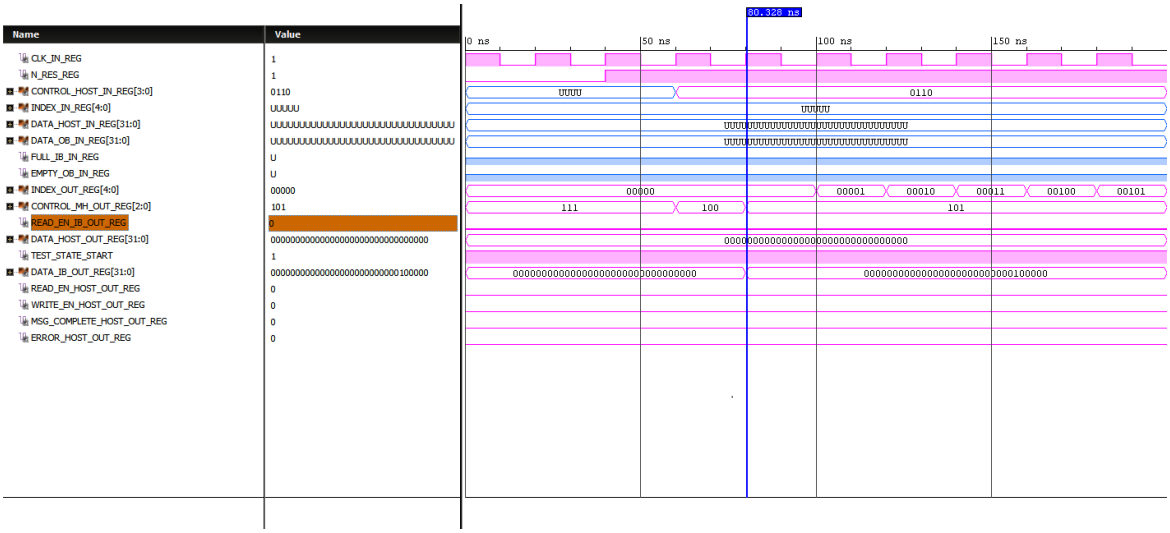

Figure 4.16: Frame Processing Unit Test Default Configuration

4.2.3 Default Configuration Test

The default configuration state is very much the same as the non-default configuration with a small difference: HOST does not provide any values and only default ones are used. The result of th default configuration state testing can be seen at Figure 4.16.

Here CONTROL_HOST_IN_REG is set to "0110" which means that HOST wants the FPU to configure using the default values. After this value is set the same operations on the output can be seen that took place in HOST configuration state. Nevertheless, now the output values are the default ones.

4.2.4 Transition to Passive State Test

No matter what configuration was used they both are followed by the Passive state. The test was executed to check if FPU behaves as expected. The result can be seen at Figure 4.17.

It can be seen that when FPU has left the configuration phase MSG_COMPLETE_HOST_OUT_REG flag was raised to tell HOST that the configuration phase has finished. During the same clock cycle FPU changes its state to the Passive and forces CONTROL_MH_OUT_REG to "111" which stands for IDLE. Now FPU is ready to execute write or read operation for the HOST if it is required so.

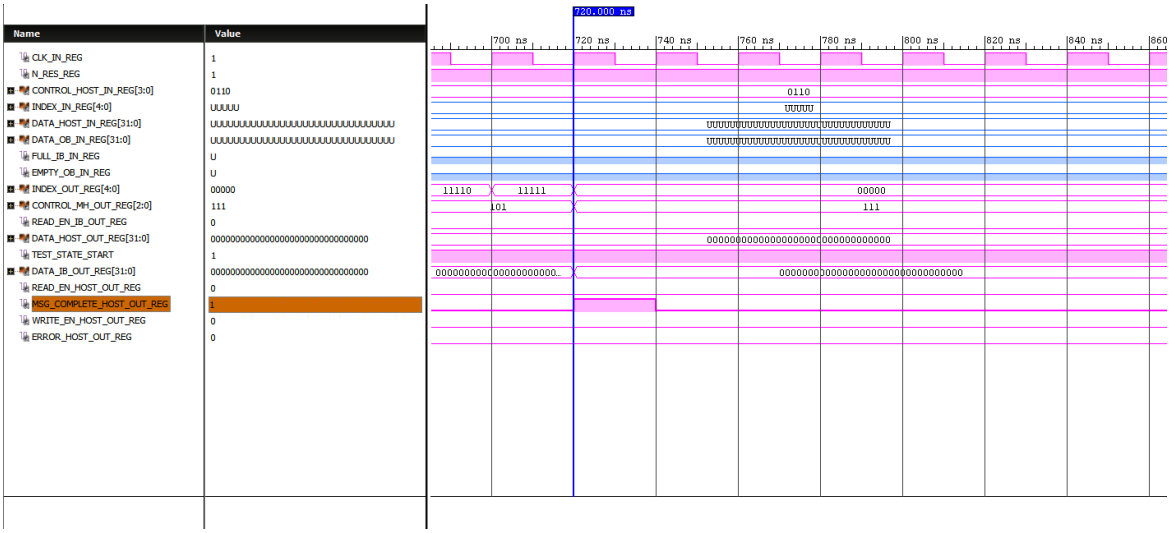

Figure 4.17: Frame Processing Unit Transition to state Passive after configuration

4.2.5 Write Test

When in passive state HOST can require write operation. If done so then the state is changed from Passive to Active Write. The behavior of this phase was tested. The result can be seen at Figure 4.18. It can be seen that HOST required the write operation by forcing CONTROL_HOST_IN_REG signal to "0011" value and as follow th FPU sets CONTROL_MH_OUT_REG to "010" for one clock cycle. When done so READ_EN_IB_OUT_REG is set to allow Input Buffer to read incoming payload bits from the FPU. The same is done for the WRITE_EN_HOST_OUT_REG flag to allow HOST to write payload bits for the buffer that it wants to update.

Next, it was tested what happens when the HOST should finish writing new payload bits to the required buffer but it still continues sending payload bits. The result of the test can be seen at Figure 4.19.

As it can be seen the WRITE_EN_HOST_OUT_REG flag was set to zero, however since HOST did continue to write the error flag was raised as well as the MSG_COMPLETE_HOST_OUT_REG. This allows FPU to be sure that HOST is informed about the fact that it is not allowed to write. However if HOST ignores this fact HOST is prevented to further update the buffer.

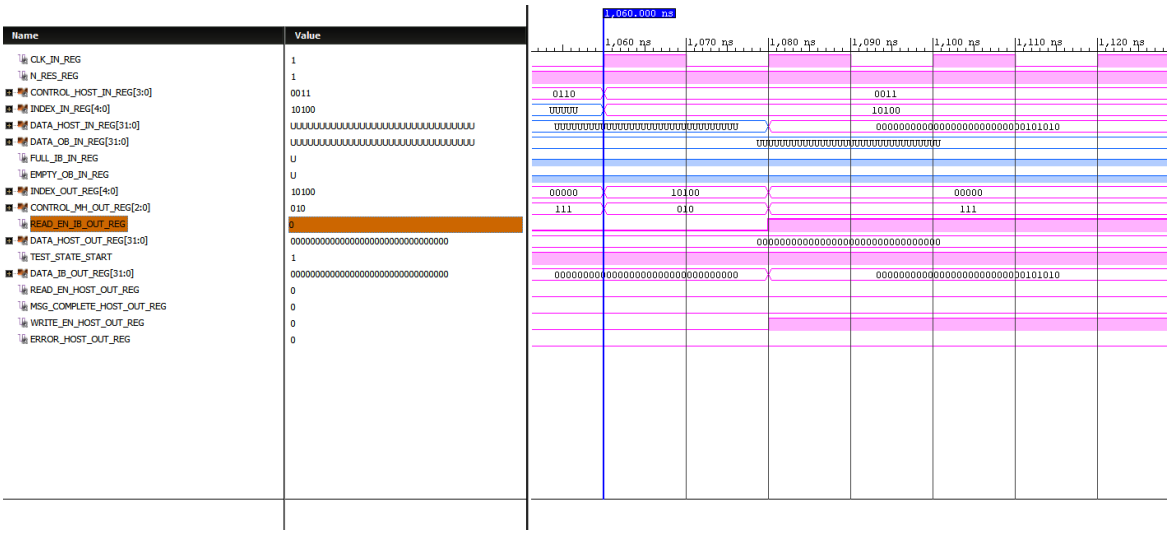

Figure 4.18: Frame Processing Unit Test start of the write operation

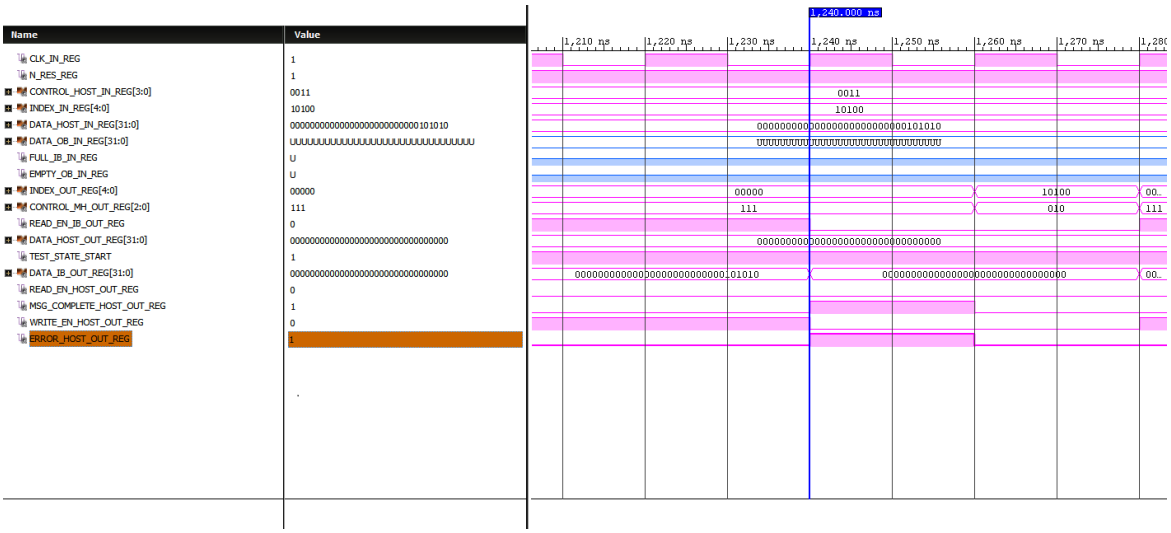

Figure 4.19: Frame Processing Unit Test overflow error

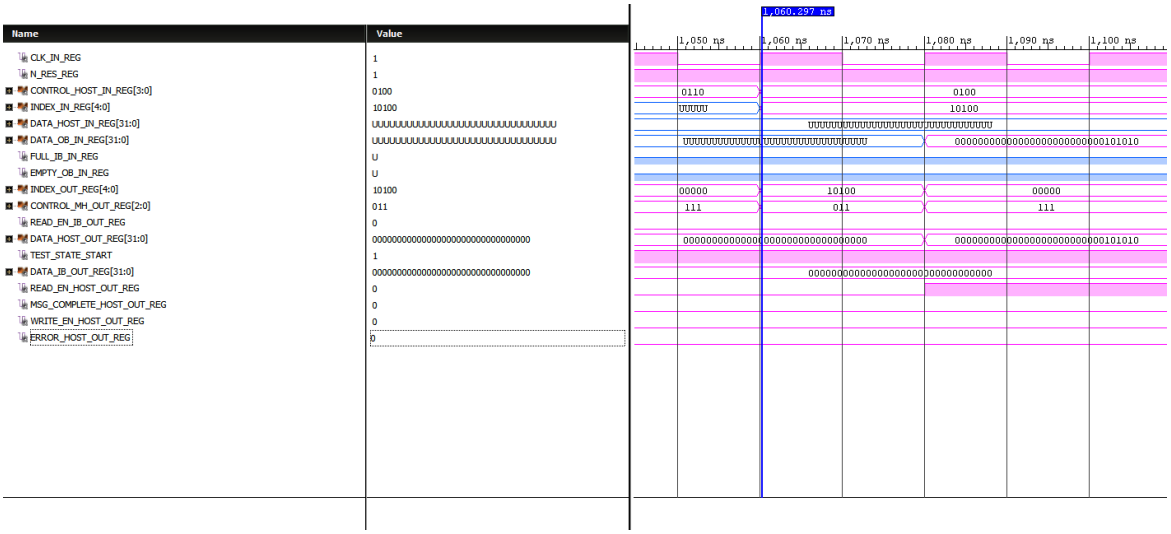

Figure 4.20: Frame Processing Unit Test start of the read operation

4.2.6 Read Test

FPU can be at state Active Read when HOST requires read access to certain buffer. This requirement can only be made during Passive state. In order to check if this transaction is made correctly test was executed (Figure 4.20). HOST requires read operation using input signal CONTROL_HOST_IN_REG. This signal is set to "0100" which stands for read operation requirement. After this signal is received from the HOST FPU sends respective request to the MH using CONTROL_MH_OUT_REG signal. This signal is set for only one clock cycle. At the same time it can be seen that HOST did provide index of the message it wants to have access to. This data, taken from INDEX_IN_REG is reflected to the INDEX_OUT_REG. During the next clock cycle MH starts to transmit required buffer's payload bits and the READ_EN_HOST_OUT_REG flag is also set to one to inform the HOST that it now can start reading payload bits.

The end behavior of this state is as well tested. The results can be seen at Figure 4.21. As expected when FPU knows that the last payload bit for the current buffer was read it informs the HOST that there is nothing more to read by setting the MSG_COMPLETE_HOST_OUT_REG flag to one. At the same time FPU sends the signal to the HOST that it does not allow any more read operations at this certain time. Meaning that there is pause or, as in this case, that there is no more payload bits to be read.

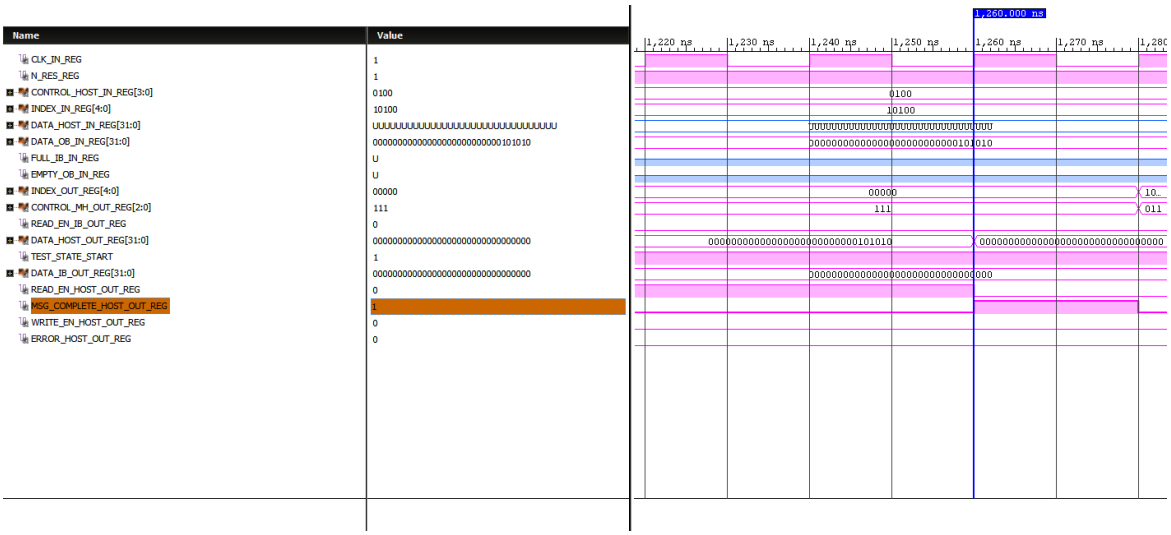

Figure 4.21: Frame Processing Unit Test Host over-requires payload bits

5 Conclusion

The message handling concept described in this paper is a new development. It balances the strictly time-scheduled message transfers between the message memory and the serial communication controllers with the accesses to the message memory requested by the host application program. Concurrent accesses to the same message buffer do not destroy the buffer's data integrity, without the need of buffer locking and without interrupting the communication schedule. No unpredictable latency time is introduced when the host needs to access message data.

The implementation of the Frame Processing Unit declares a new way to control the communication between the HOST and the Message Handler. This way Message Handler is assured that all messages coming from the HOST are integral and valid. This allows to decrease the computational logic of the Message Handler.

In future works related to the Message Handler a new module can be added, which is similar to the Frame Processing Unit, between the Message Handler and the Protocol Controller in order to reduce FlexRay message request related checks in the Message Handler. In addition, the Frame Processing Unit, Message Handler and Message RAM could share the same module that will held the configuration bits for the RAM. This will decrease the overall cost of the Message Handler production.

Appendices

A.1 VHDL test code for the Message Handler

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY FPU_TEST IS


END FPU_TEST;

ARCHITECTURE Behavioral OF FPU_TEST IS

COMPONENT FPU
PORT (
CLK_IN : IN STD_LOGIC;
N_RES : IN STD_LOGIC;
INDEX_IN : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
CONTROL_HOST_IN : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
EMPTY_OB_IN : IN STD_LOGIC;
FULL_IB_IN : IN STD_LOGIC;
DATA_HOST_IN : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
DATA_OB_IN : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
INDEX_OUT : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
CONTROL_MH_OUT : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
READ_EN_IB_OUT : OUT STD_LOGIC;
DATA_HOST_OUT : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
DATA_IB_OUT : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
READ_EN_HOST_OUT : OUT STD_LOGIC;
WRITE_EN_HOST_OUT : OUT STD_LOGIC;
MSG_COMPLETE_HOST_OUT : OUT STD_LOGIC;
ERROR_HOST_OUT : OUT STD_LOGIC
```

```
);
END COMPONENT;

SIGNAL CLK_IN_REG : std_logic;
SIGNAL N_RES_REG : std_logic;
SIGNAL CONTROL_HOST_IN_REG : std_logic_vector (3 DOWNTO 0);
SIGNAL DATA_HOST_IN_REG : std_logic_vector(31 DOWNTO 0);
SIGNAL INDEX_IN_REG : std_logic_vector(4 DOWNTO 0);
SIGNAL DATA_OB_IN_REG : std_logic_vector(31 DOWNTO 0);
SIGNAL FULL_IB_IN_REG : std_logic;
SIGNAL EMPTY_OB_IN_REG : std_logic;

SIGNAL INDEX_OUT_REG : std_logic_vector(4 DOWNTO 0);
SIGNAL CONTROL_MH_OUT_REG : std_logic_vector(2 DOWNTO 0);
SIGNAL READ_EN_IB_OUT_REG : std_logic;
SIGNAL DATA_HOST_OUT_REG : std_logic_vector(31 DOWNTO 0);
SIGNAL DATA_IB_OUT_REG : std_logic_vector(31 DOWNTO 0);
SIGNAL READ_EN_HOST_OUT_REG : std_logic;
SIGNAL WRITE_EN_HOST_OUT_REG : std_logic;
SIGNAL MSG_COMPLETE_HOST_OUT_REG : std_logic;
SIGNAL ERROR_HOST_OUT_REG : std_logic;


SIGNAL TEST_STATE_START : std_logic := '1';
TYPE TEST_STATES IS (TEST_RESET, TEST_DEFAULT_CONF, TEST_CONF_MIN,
 TEST_CONF_MAX, TEST_CONF, TEST_PASSIVE, TEST_WR, TEST_RD);
SIGNAL TEST_STATE : TEST_STATES := TEST_RD;
BEGIN
FPU_CONN : FPU
PORT MAP(
CLK_IN => CLK_IN_REG, N_RES => N_RES_REG, INDEX_IN => INDEX_IN_REG,
 CONTROL_HOST_IN => CONTROL_HOST_IN_REG, EMPTY_OB_IN =>
 EMPTY_OB_IN_REG, FULL_IB_IN => FULL_IB_IN_REG, DATA_HOST_IN =>
 DATA_HOST_IN_REG, DATA_OB_IN => DATA_OB_IN_REG, INDEX_OUT =>
 INDEX_OUT_REG, CONTROL_MH_OUT => CONTROL_MH_OUT_REG,
 READ_EN_IB_OUT => READ_EN_IB_OUT_REG, DATA_HOST_OUT =>
 DATA_HOST_OUT_REG, DATA_IB_OUT => DATA_IB_OUT_REG,
 READ_EN_HOST_OUT => READ_EN_HOST_OUT_REG,
 WRITE_EN_HOST_OUT => WRITE_EN_HOST_OUT_REG,
 MSG_COMPLETE_HOST_OUT => MSG_COMPLETE_HOST_OUT_REG,
```

```
 ERROR_HOST_OUT => ERROR_HOST_OUT_REG
);

PROCESS
BEGIN
CASE TEST_STATE IS
WHEN TEST_RESET =>
N_RES_REG <= '0';
WAIT FOR 100 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0110";
WAIT FOR 400 ns;
N_RES_REG <= '0';
WAIT FOR 20 ns;
WHEN TEST_DEFAULT_CONF =>

N_RES_REG <= '0';
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0110";
WAIT FOR 2000 ns;
WHEN TEST_CONF_MIN =>

N_RES_REG <= '0';
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0101";
WAIT FOR 20 ns;
DATA_HOST_IN_REG(5 DOWNTO 0) <= "000101";
DATA_HOST_IN_REG(31 DOWNTO 6) <= (OTHERS => '0');
WAIT FOR 20 ns;
DATA_HOST_IN_REG(31 DOWNTO 6) <= (OTHERS => '0');
DATA_HOST_IN_REG(5 DOWNTO 0) <= "111101";
WAIT FOR 2000 ns;
WHEN TEST_CONF_MAX =>

N_RES_REG <= '0';
```

```
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0101";
WAIT FOR 20 ns;
DATA_HOST_IN_REG(7 DOWNTO 0) <= "10101111";
DATA_HOST_IN_REG(31 DOWNTO 8) <= (OTHERS => '0');
WAIT FOR 20 ns;
DATA_HOST_IN_REG(31 DOWNTO 6) <= (OTHERS => '0');
DATA_HOST_IN_REG(5 DOWNTO 0) <= "111101";
WAIT FOR 2000 ns;
WHEN TEST_CONF =>

N_RES_REG <= '0';
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0101";
WAIT FOR 20 ns;
DATA_HOST_IN_REG(5 DOWNTO 0) <= "100001";
DATA_HOST_IN_REG(31 DOWNTO 6) <= (OTHERS => '0');
WAIT FOR 20 ns;
DATA_HOST_IN_REG(5 DOWNTO 0) <= "111101";
WAIT FOR 2000 ns;
WHEN TEST_PASSIVE =>

N_RES_REG <= '0';
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0110";
WAIT FOR 5000 ns;
WHEN TEST_WR =>

N_RES_REG <= '0';
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0110";
WAIT FOR 1000 ns;
```

```
CONTROL_HOST_IN_REG <= "0011";
INDEX_IN_REG <= "10100";
WAIT FOR 20ns;
DATA_HOST_IN_REG(5 DOWNTO 0) <= "101010";
DATA_HOST_IN_REG(31 DOWNTO 6) <= (OTHERS => '0');
WAIT FOR 1000ns;

WHEN TEST_RD =>

N_RES_REG <= '0';
WAIT FOR 40 ns;
N_RES_REG <= '1';
WAIT FOR 20 ns;
CONTROL_HOST_IN_REG <= "0110";
WAIT FOR 1000 ns;
CONTROL_HOST_IN_REG <= "0100";
INDEX_IN_REG <= "10100";
WAIT FOR 20ns;
DATA_OB_IN_REG(5 DOWNTO 0) <= "101010";
DATA_OB_IN_REG(31 DOWNTO 6) <= (OTHERS => '0');
WAIT FOR 1000ns;
WHEN OTHERS =>

END CASE;
END PROCESS;

PROCESS
BEGIN
CLK_IN_REG <= '1';
WAIT FOR 10 ns;
CLK_IN_REG <= '0';
WAIT FOR 10 ns;
END PROCESS;

END Behavioral;
```

Bibliography

[Arndt 2009] ARNDT, Michael: Implementation of a FlexRay Communication Interface for Linux. (2009). – URL http://www.scriptkiller.de/pub/flexray4linux/arndt09_flexray_comm_interface.pdf

[Chu. 2006] CHU., Pong P.: *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability and Scalability.* Wiley-IEEE Press, 2006

[Daniel 2013] DANIEL: VHDL: Standart FIFO. (2013). – URL http://www.deathbylogic.com/2013/07/vhdl-standard-fifo/

[FlexRay 2005] FLEXRAY: *FlexRay Communications System Protocol Specification Version 2.1*. FlexRay Consortium, 2005

[Fujitsu 2007] FUJITSU: *FlexRay ASSP MB88121B User's Manual*. 2007. – URL https://www.fujitsu.com/downloads/MICRO/fma/pdfmcu/um-mb88121-am15-11201-1e.pdf

[Rogers und Schmechtig 2006] ROGERS, Bill ; SCHMECHTIG, Stefan: *FlexRay Message Buffers*. IPextreme, Inc., 2006. – URL http://www.ip-extreme.com/downloads/flexray_mb_wp.pdf

[Shaw 2009] SHAW, Robert: *Improving the Reliability and Performance of FlexRay Vehicle Network Applications Using Simulation Techniques*, Waterford Institute of Technology, Master Thesis, 2009. – URL http://repository.wit.ie/1373/1/Improving_the_Reliability_and_Performance_of_FlexRay_Vehicle_Applications_Using_Simulation_Techniques.pdf

[William J. Dally 2016] WILLIAM J. DALLY, Tor M. A.: *Digital Design Using VHDL*. Cambridge University Press, 2016

Acronyms

MH Message Handler

FPU Frame Processing Unit

IB Input Buffer

OB Output Buffer

TBF Transient Buffer

PRT Protocol Controller

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, July 11, 2016

City, Date

sign