Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

IC2IC: a Lightweight Serial Interconnect Channel for Multiprocessor Networks

Oliver FAUST, Bernhard H. C. SPUTH, and Alastair R. ALLEN

Department of Engineering, University of Aberdeen, Aberdeen AB24 3UE, UK
{b.sputh, o.faust, a.allen}@abdn.ac.uk

Abstract. IC2IC links introduce blocking functionality to a low latency high performance data link between independent processors. The blocking functionality was achieved with the so-called alternating bit protocol. Furthermore, the protocol hardens the link against message loss and message duplication. The result is a reliable way to transfer bit level information from one IC to another IC. This paper provides a detailed discussion of the link signals and the protocol layer. The practical part shows an example implementation of the IC2IC serial link. This example implementation establishes an IC2IC link between two configurable hardware devices. Each device incorporates a process network which implements the IC2IC transceiver functionality. This example implementation helped us to explore the practical properties of the IC2IC serial interconnect. First, we verified the blocking capability of the link and second we analysed different reset conditions, such as disconnect and bit-error.

Keywords. serial link, hardware channel, embedded systems, system on chip, network on chip, hardware software co-design, multi-core.

Introduction

Parallel processing systems must exchange data in a timely fashion between multiple physically separate processors. According to C. R. Anderson *et al* [1] effective exploitation of multiple processors in a distributed computing environment relies on a low latency, high bandwidth, inter-processor communication network. In the late 1980s the INMOS transputer was a pioneering attempt to built a communication network with these properties for multiprocessor computing [2]. The basic design of the transputer included serial links, that allowed it to communicate with up to four other transputers. The links operated at 5, 10 or 20 Mbit/s — which at the time was faster than existing networks such as Ethernet. The link speed matched the processing speed well, therefore the available resources could be used efficiently. This led to some notable and diverse applications such as neurocomputers [3] and architectures for graphics [4].

While the transputer was simple but powerful compared to many contemporary designs, it never came close to meeting its goal of being used universally in both CPU and microcontroller roles. In the microcontroller realm, the market was dominated by 8-bit machines where cost was the only serious consideration. Here, even the 16-bit transputers were too powerful and expensive for most applications. Furthermore, the concept of communicating processes alienated many users. These are only some of the reasons why the transputer family was not developed further. However, even after the development stopped, the link technology was still used in a limited number of applications. In 1995 the communication links were even standardised as IEEE-1355 [5]. This standard details also “Wormhole Routing” [6], which allows packets of unlimited length to be routed within the network of processors.

The IEEE-1355 link standard continues to generate interest. One reason for this continued interest is the ease of implementation when compared with competing technology such as Ethernet. All twisted pair versions of Ethernet require analogue signal processing of the received signals to extract data – a silicon hungry process [7]. This even led to the claim: "IEEE 1355 data-strobe links: ATM speed at RS232 cost", by Barry M. Cook [8]. On the application side, Greve, O. J. *et al* [9] argue that for heavy mechatronic and robot applications, transputer links can be used to achieve a high-performance and real-time communication between discrete system components. Marcel Boosten *et al* [10] have investigated the construction of a parallel computer using IEEE 1355 high-throughput low-latency DS link networks and high-performance commodity processors running a standard operating system.

Apart from direct applications, the IEEE-1355 technology has influenced a number of other communication standards. The IEEE-1355 encoding scheme has been adopted for the IEEE-1394 standard [11] and the Apple Computer version of IEEE-1394 known as FireWire. Space applications have very high demand on system reliability, because it is difficult, if not impossible, to make changes after the launch of a space craft. These requirements provided the reason why the transputer link technology was adopted as a new communication standard for space applications with relatively minor changes [12]. The European Space Agency plans to use the new SpaceWire standard for most of its future missions. The SpaceWire standard constitutes something like the rebirth of IEEE-1355. Therefore, most of the recent practical projects focus on this standard instead of the original IEEE-1355. In space applications radiation tolerance is very important. B. Fiethe *et al* [13] argue that protection against such effects can be achieved by using parts built of special technology, such as SpaceWire. In a similar argument, Sanjit A. Seshia *et al* [14] state that technology scaling[1] causes reliability problems to become a dominant design challenge. They support their statement with a case study where they analyse the stability of a publicly available implementation of SpaceWire end-nodes. Despite the name, there are attempts to use SpaceWire in earth-bound applications. Sergio Saponara *et al* [15] introduce the SpaceWire standard to the automotive field. This approach is justified, because both space and automotive applications have high demands on reliability of the communication standard.

Apart from these rather high level communication standard considerations, the ideas of transputer links also play a role in the design style for the communication between independent digital circuits. The main reason for this interest is the fact that the communication between independent circuits provides the basis for CSP style hardware design [16]. The two main problems are clock and reset synchronisation between independent circuits. The IEEE-1355 standard provides a solution for both problems. For wire bound communication the clock synchronisation is solved via self-clocking data-strobe signal communication. The reset synchronisation problem is solved with the exchange of silence protocol. However, CSP style communication between independent circuits requires a blocking capability which is not provided by IEEE-1355. The blocking capability ensures that a circuit which is committed to a data transfer can only make progress if the data is exchanged with the communication partner. A minor problem is that many hardware implementations require a higher degree of payload flexibility than the standard can offer. IEEE-1355 defines only data packets with 8 bit payload. This might be too much or too little, depending on the particular application.

This paper discusses the implementation of IC2IC, a lightweight serial interconnect link for multiprocessor networks. This implementation combines IEEE-1355 style synchronisation and encoding with a protocol based blocking functionality. We use an adaptation of the alternating bit protocol (ABP) to ensure the blocking functionality. The protocol guarantees that a particular data packet gets delivered to the receiver. The result is a stiff[2] communica-

[1]Decreasing feature size of digital integrated circuits over the years.

[2]The term 'stiff' describes the characteristic of the implementation with respect to the commitment of the

tion channel which is susceptible to external influences on the physical means of data exchange. We discuss the problems which arise from external influences, such as: bit-error, asynchronous reset and disconnect. To solve these problems was a design challenge, because the logic must react to each problem differently and in some cases 'remember' the system state during which the problem occurred. Both, hardening against message loss and formalised synchronisation, are the core concepts which make the IC2IC serial link reliable. Compared with the problems caused by external influences, the introduction of payload flexibility was relatively easy. Nevertheless, payload flexibility is an important feature of the IC2IC link.

The following section provides an in-depth discussion of the IC2IC serial interconnect channel. This includes a short introduction of the alternating bit protocol, which establishes the blocking functionality. Sections 1.1 and 1.2 introduce link signals and packet level of the protocol. After the definition of both signals and packets we define the IC2IC protocol. This protocol handles initialisation, data transfer and error conditions. Section 2 details an example implementation of the IC2IC serial link between two configurable logic devices. Each logic device executes an IC2IC transceiver circuit. With this setup we show blocking and error recovery in an implementation system.

1. IC2IC

This section introduces IC2IC, a lightweight bidirectional channel between two hardware components. The IC2IC functionality extends IEEE-1355 with blocking functionality. From the protocol standard IC2IC inherits the following functionality:

1. Self clocking — This ensures variable transmission speeds.
2. Asynchronous reset — Implemented through exchange of silence.
3. Disconnect detection — Both communication partners can detect and recover from disconnect.

The blocking communication causes problems after an asynchronous reset. The sender needs to know the state of the receiver when the data transfer resumes. The transfer of the last receiver state is a crucial part of the IC2IC protocol, without it the system is prone to data loss and deadlock.

1.1. Alternating Bit Protocol

Figure 1 shows a basic setup of a point to point communication system. In this system, port A (sender) sends data packets to port B (receiver) over a lossy communication channel. The channel can swallow a finite number of packets and duplicate the same packet a finite number of times. The first formalised solution for this problem emerged 1969 as a note on reliable full-duplex transmission over half-duplex links [17]. The solution involves an "alternation bit". This idea was picked up by A. W. Roscoe [18, page 130] who proposed a method called alternating bit protocol to establish a secure communication of the lossy channel. This protocol involves a back channel from port B to port A. This back channel has the same characteristic as the data communication channel. The name alternating bit protocol comes from the fact that each packet is extended by one bit which alternates between 0 and 1 before it is sent along the lossy channel. Multiple copies of the same data packet are sent from port A to port B until port A receives a control packet which acknowledges this data packet. The behaviour of port B depends on the environment. If the environment of port B has picked up

communication partners which engage in a message exchange over this link. Once a communication partner has committed there is no way to withdraw.

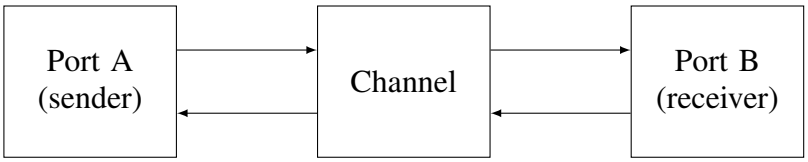

Figure 1. Basic communication system

the data from the previous packet then as soon as port B gets a new packet via the communication channel it sends an acknowledgement packet and offers the data to the environment. Port B acknowledges all resends of this data packet with a copy of the initial acknowledgement packet. In the case that the environment did not pick up the last message port B does not acknowledge the reception of a new data packet. Needless to say that port B does not acknowledge subsequent copies of this data packet until the environment picks up the data of the previous data packet. The two ports can always spot a new message or acknowledgement because of the alternating bit.

Sputh *et al* made relevant adjustments to the alternating bit protocol [19]. They establish safety, stability and functionally of the resettable receiver alternating bit protocol (RRABP). This extended alternating bit protocol creates a reliable and blocking channel between sender and receiver over unreliable non-blocking communication channels. Furthermore, this protocol allows the system to restart the sender at any time: however, not under all conditions without losing a message. The IC2IC serial link protocol is a practical realisation of the formal RRABP.

1.2. Link Signals

Each IC2IC port is connected to four link signals. The two signals tx_data and tx_strobe establish outgoing communication and rx_data and rx_strobe handle incoming communication. Figure 2 shows an IC2IC port as a self-contained entity with the ability to exchange data via the external communication signals. On the signal level the communication happens in two distinct phases, first synchronisation and then data transfer.

Figure 3 shows the timing diagram of the transmission signals in the synchronisation phase of the data exchange. This phase ensures that both communication partners are ready for the data transfer. During this initial phase both data and strobe signals have the same timing. The figure shows two signals, the transmitted data signal tx_data and the received data signal rx_data. At time point 1, the IC2IC transceiver under discussion is ready to transfer data, therefore it assigns low to the tx_data signal. A low timer circuit ensures that the low stays assigned for $100\mu s$ before it changes to high for $10\mu s$. At time point 2, the communication partner starts the transmission, by assigning an active low on the rx_data signal. At time point 3, the transceiver generates a falling edge on the tx_data signal. This falling edge causes the communication partner to reset the low timer. This reset synchronises both communication partners. The timer reset is the reason why the $10\mu s$ high pulse is not present at time point 4. At time point 5, the falling edges of both communication partners are in sync. This is the start signal for the data transfer. After time point 5, the timing diagram shows the bit pattern for the stop packet.

Figure 4 shows a timing diagram of the IC2IC protocol data transfer signals. The first signal shows the bit-sequence to be transmitted. The second row shows the data (tx_data) signal. The third row shows the strobe (tx_strobe) signal. The data signal encodes the bit sequence directly, i.e. the signal is low when a 0 bit is transmitted and the signal is high when a 1 bit is transmitted. The strobe signal generates an edge whenever the data signal stays constant for more then one bit period T_B. In Figure 4, we assume an initial reset condition. Time point 1 indicates the start of a second consecutive 0 data bit. To communicate the start of the second 0 to the receiver, the strobe signal generates a rising edge. Time point 2 indicates

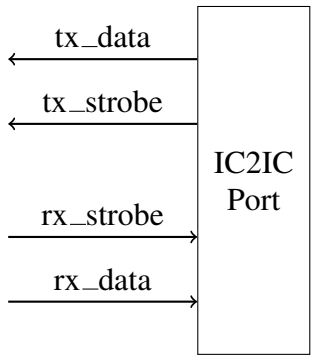

Figure 2. Link signals

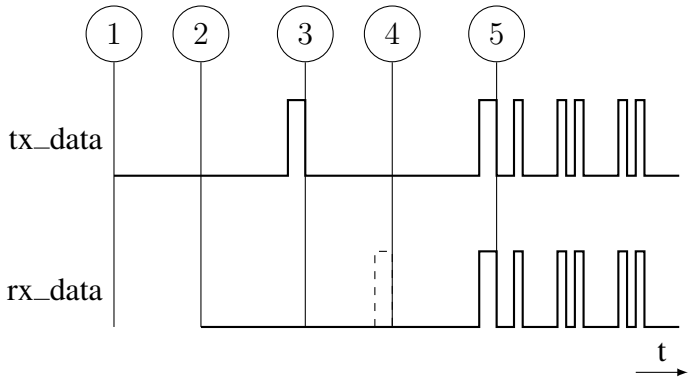

Figure 3. IC2IC synchronisation pattern

the start of a second consecutive 1. At this time point the strobe signal transits from high to low thereby creating a falling edge. Similarly, time point 3 indicates the start of a third consecutive 1. At this time point, the strobe signal generates a rising edge.

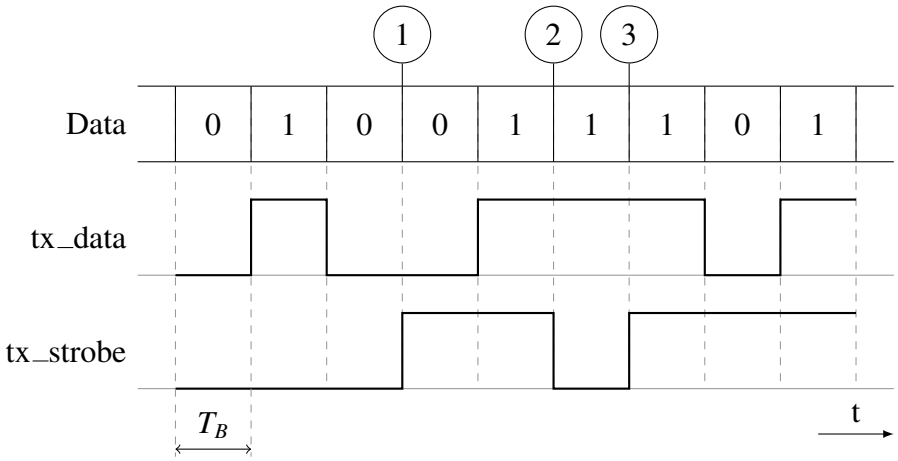

Figure 4. IC2IC data transfer pattern

1.3. Packet Level

The ABP functionality requires data and control packets. Data packets contain header and payload. Control packets are more complex, because they are used for acknowledgement and protocol synchronisation. The following paragraphs detail the composition of data and control packets.

Figure 5 shows the data packet structure. L is the number of payload data bits and $L+3$ is the total number of bits in one data packet. P is the parity bit, it is set for odd parity in the parity region. $\overline{\text{C}}$ indicates not control packet, therefore $\overline{\text{C}} = 0$. ABP is the alternating bit protocol bit.

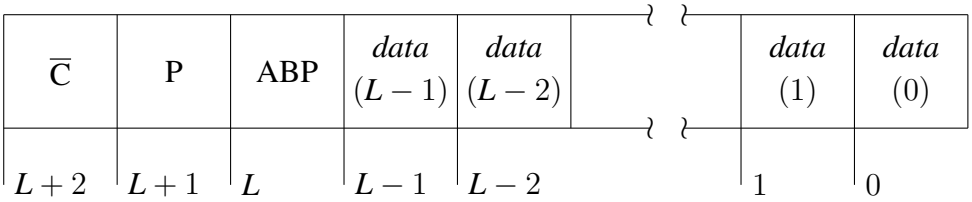

Figure 5. Data packet structure

Figure 6 shows the control packet structure. A control packet always has 6 bits. Setting bit C to 1 identifies this packet as a control packet. P is the parity bit which ensures odd parity in the parity region.

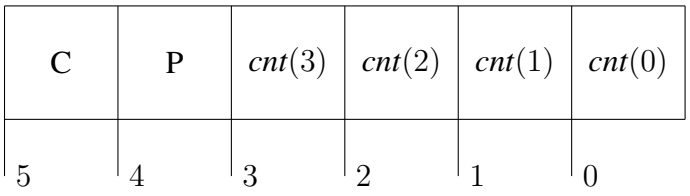

Figure 6. Control packet structure

The four control packet bits *cnt*(3) – *cnt*(0) encode a maximum of $2^4 = 16$ control messages. Table 1 details all messages which are used in the example implementation. The *stop_xy* and *start_xy* messages are used for RRABP protocol synchronisation. *zero_ack* and *one_ack* are the ABP acknowledgement messages sent from port B to port A. The *alive* packet ensures a continuous data stream between the ports.

Table 1. Mapping of the control messages onto the 4 *cnt* bits

Message	*cnt* bits	Message	*cnt* bits
stop_msg	0000	*stop_ack*	1111
start_msg	1110	*start_0_ack*	1101
start_1_ack	1100	*start_rst_ack*	1011
zero_ack	1010	*one_ack*	1000
alive	0111		

The parity bit P of a data packet covers the ABP bit, the *L* data payload bits and the data / control flag in the next packet. Similarly, the parity bit of a control packet covers the 4 control bits *cnt*(3) – *cnt*(0) and the data / control flag in the next packet. Figure 7 shows a parity coverage example. In the example scenario a control packet is followed by a data packet. The parity coverage ensures that an error in any single bit of a packet, including the packet type flag, can be detected even though the packets are not all the same length. The parity bit is set such that the total number of 1s in all the bits covered (including the parity bit) is odd.

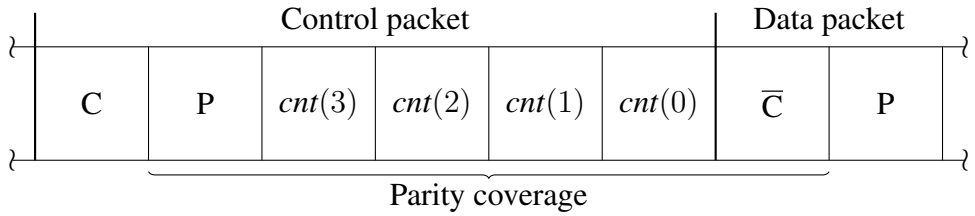

Figure 7. Parity coverage example

1.4. The IC2IC Protocol

The resettable receiver alternating bit protocol, introduced in Section 1.1, handles the protocol synchronisation between port A and port B. However, even the RRABP struggles to handle problems introduced by the physical means of data transmission. These problems include disconnect and bit-errors. In this section we extend the protocol further such that it addresses the practical problems of the IC2IC serial link.

Figure 8 shows the communication diagram for normal operation. Normal operation means there are no errors introduced by the physical means of communication. The figure

shows protocol synchronisation, normal data transfer as well as receiver and sender blocking.

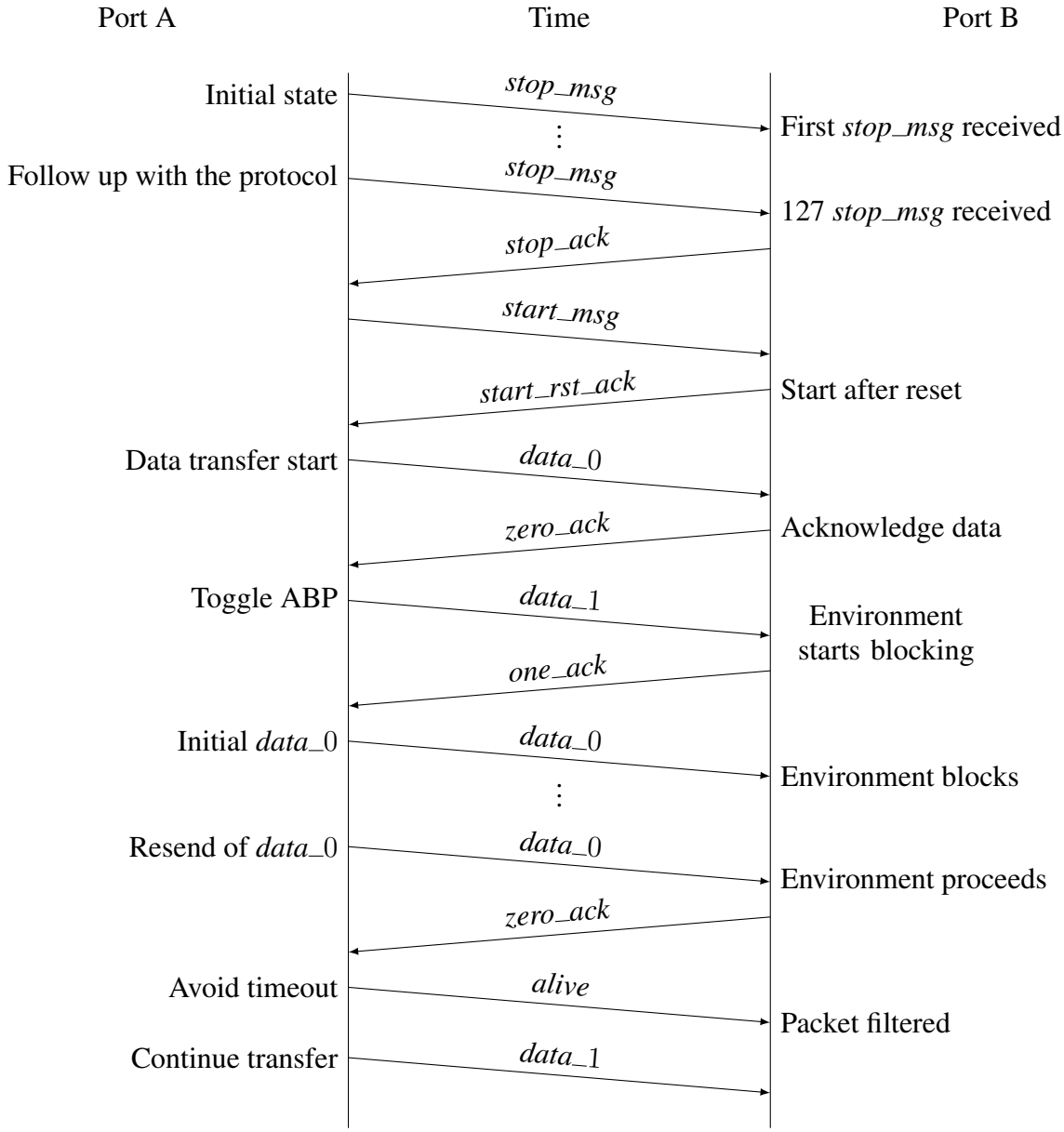

Figure 8. Protocol diagram for normal operation

After synchronisation on the physical layer, shown in Figure 3, the IC2IC protocol takes over. The protocol synchronisation starts with a *stop_msg* control packet sent from port A (sender) to port B (receiver). Port B must receive 127 consecutive *stop_msg* messages before the link is considered fit for data transfer. Port B acknowledges the reception of 127 *stop_msg* packets with a *stop_ack* packet. After port A receives the *stop_ack* packet it sends out *start_msg* packets. Upon reception of such a *start_msg* packet, port B responds with *start_rst_ack* packet. This packet indicates that port B came out of a global reset condition. After having received the *start_rst_ack* packet, port A sends out the first data packet, *data_*0, with ABP bit 0. Port B acknowledges the reception of this packet with the *zero_ack* control packet. Upon reception of the *zero_ack* packet port A toggles the ABP bit and sends out the second data packet. Port B receives and acknowledges this data packet, however it is not picked up by the environment of port B. In other words, the environment on the receiver side blocks the communication and port B acts as a buffer (with one place). Upon reception of the

acknowledgement, port A sends a *data_0* packet. This packet is not acknowledged by port B, because the buffer is full and the environment still blocks. The absence of acknowledgement forces port A to resend *data_0*. After the receiver environment empties the buffer, port B acknowledges *data_0*. After port A receives this acknowledgement (*zero_ack*) the transmitter recognises that there are no further data vectors to be transmitted. In other words, the transmitter environment blocks. To ensure continuous data traffic between port A and B on the physical layer, port A sends an *alive* control packet. This packet is filtered by port B. At any time port A can resume the data transfer by sending out a *data_1* packet.

Figure 9 shows the communication diagram for the case that a bit error in the *stop_msg* control packet occurs. Upon detection of a bit-error in the *stop_msg* port B forces an asynchronous reset by putting a permanent low on the output signals. This behaviour is detected as a disconnect by port A. In reaction to this disconnect, port A also puts its output signals on low. This silence is observed by both ports for 10ms. Port B remembers that the error occurred during the initial protocol synchronisation. After this silence the communication resumes with the initial synchronisation pattern, described in Section 1.2. After the physical synchronisation, the protocol synchronisation starts. Port B concludes the protocol synchronisation by sending *start_rst_ack*. This indicates that port B came out of a global reset condition and the expected ABP bit value for the first data packet is 0.

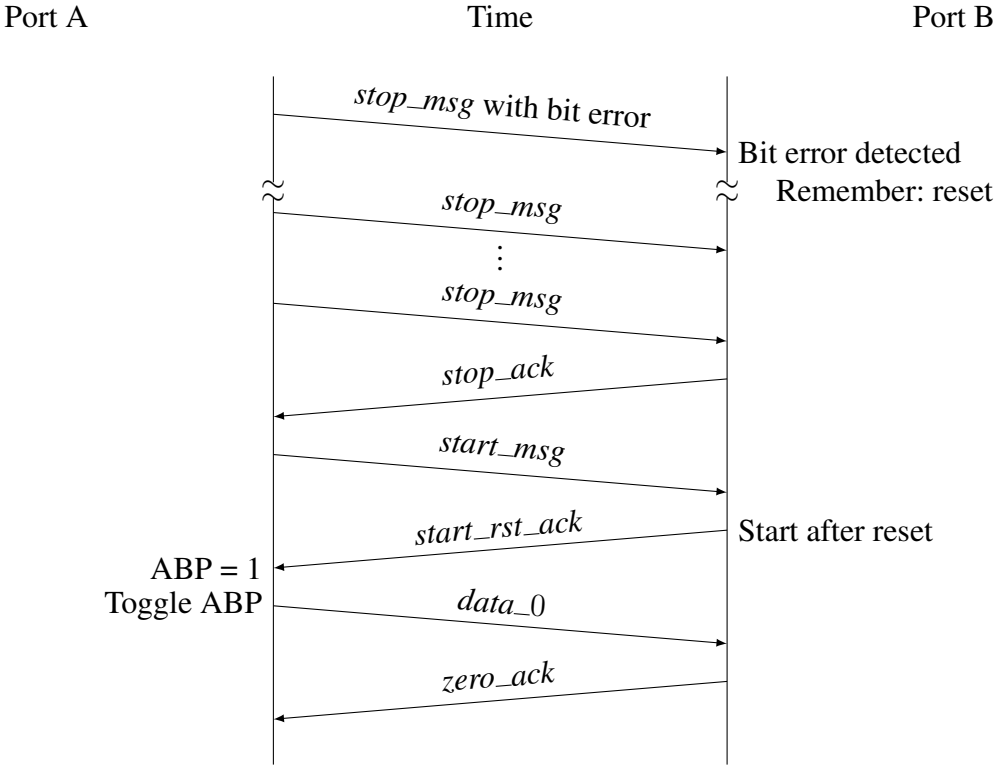

Figure 9. Protocol diagram for *stop_msg* with bit error

Figure 10 shows the communication diagram for the case when a bit error in the *data_1* packet occurs. When port B detects this bit error it sets both output signals to low. This communicates an asynchronous reset to port A. After the silence period has passed, both port A and port B resume the transmission with the physical initialisation sequence. Port B sends out the *start_zero_msg* control packet. This indicates that the value of the last correct received ABP bit was 0. After port A receives this packet it resumes the data transmission with an ABP bit equal to 1, in Figure 10 this is indicated by *data_1*.

Figure 11 shows the communication diagram for the case when a bit error in the control packet *zero_ack* occurs. The error handling is similar to the previous scenario. However, this

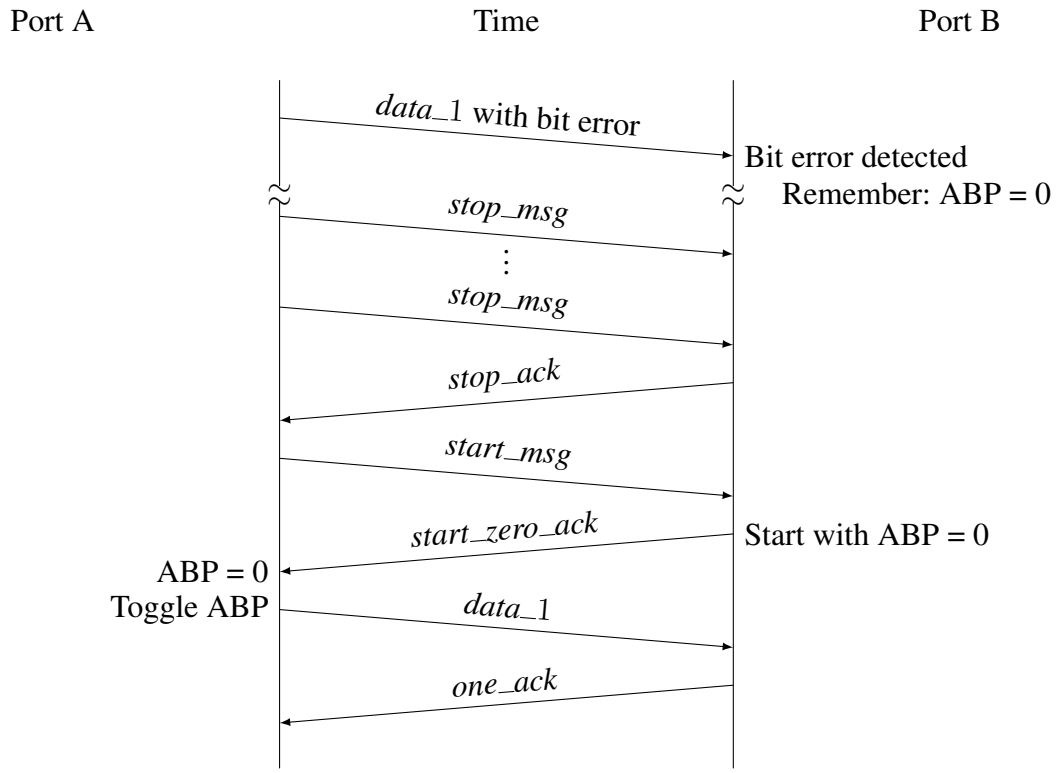

Figure 10. Protocol diagram for *data_1* with bit error

time port A detects the error in the control packet and initiates the exchange of silence. The last correct ABP bit port B has seen was 1, therefore the initialisation protocol is concluded with port B sending *start_zero_ack*.

Figure 12 shows the communication diagram for the case that both ports detect a disconnect during a normal data transfer. Therefore, both initiate the exchange of silence. After the silence period has elapsed, port A and port B are ready to continue the data transfer. The physical synchronisation signals ensure that the link resumes after the ports are connected again. In case of a reconnect, port B concludes the initialisation protocol by sending a *start_one_ack* packet. This indicates that the ABP value of the last correctly received data packet was 1.

2. Example Implementation

The process network of the implementation is organised in a decentralised way. That means, there is no central process which controls the IC2IC functionality. Each process within the network has its own functionality and to a certain extent its own agenda. This leads to a hierarchically flat model.

Figure 13 shows the implementation process network. This process network consists of nine processes and CSP style, i.e. blocking, channels for data transfer. All channels are named *from*2*to* where *from* indicates the sender process name and *to* indicates the receiver process name. The process network communicates with the local processing environment via *in* and *out* channel. *tx_data* and *tx_strobe* are output signals, i.e. they leave the local processing environment. Similarly, *rx_data* and *rx_strobe* are input signals. The network functionality can roughly be partitioned into two parts. The first part implements the data handling and the second part implements the ABP functionality.

The data handling starts with messages flowing from the environment to the *P* process via the *in* channel. The *P* process appends the ABP protocol bit and sends the extended

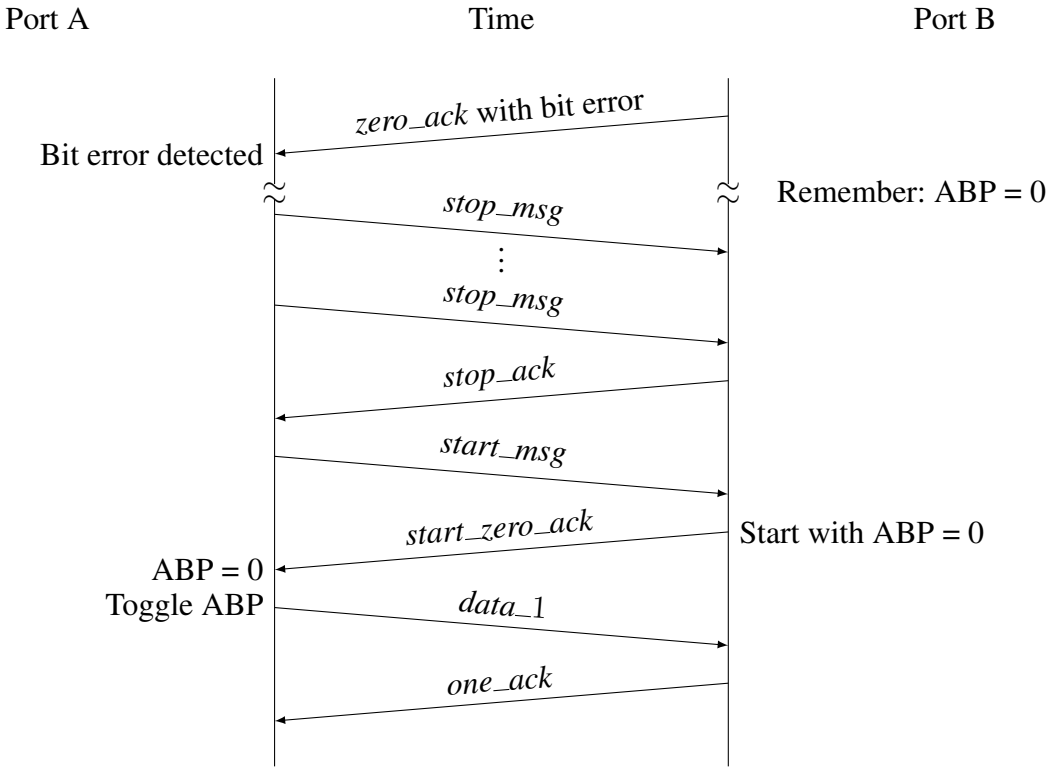

Figure 11. Protocol diagram for *zero_ack* with bit error

message to the *MUXER* process. The *MUXER* extends the message with parity bit and control bit before the message is sent to the *TX* process. The *TX* process translates the message into the self clocking line code for transmission from IC to IC. The message is received with the *RX* process. The *RX* process decodes the transmission signal and sends the received message bits to the *DEMUX* process. The *DEMUX* process assembles and checks the message. The control and parity bit is removed before the message is sent to the *C* process. The *C* process removes the ABP bit before the message is sent to the environment via the *out* channel.

Table 2 shows the packet composition on the TX side and the decomposition on the RX side. The environment provides an L dimensional data vector Data_L. The IC2IC[3] process network adds a three bit header which ensures security and robustness of the data transfer. On the RX side (port B) the IC2IC process network removes the header and offers the received L dimensional data vector to the environment.

Table 2. IC2IC layer structure. Data_L is an L dimensional data vector.

TX		RX	
Entity	Packet	Entity	Packet
Environment	Data_L	Environment	Data_L
P	ABP & Data_L	*C*	ABP & Data_L
MUXER	$\overline{\text{C}}$ & ABP & Data_L	*DEMUX*	$\overline{\text{C}}$ & ABP & Data_L
Physical layer			

The three remaining processes, *TX_ABP*, *RX_ABP* and *CNTMSG* implement the ABP protocol functionality. *TX_ABP* controls the ABP protocol on the TX side. Therefore, it is connected to the *P* process via a bidirectional channel. This bidirectional channel enables the *TX_ABP* process to react to input. Furthermore, *TX_ABP* is connected with the *MUXER*

[3]Capitalised italic fonts indicate a process name. In this case IC2IC is the name of the process which implements the IC2IC functionality.

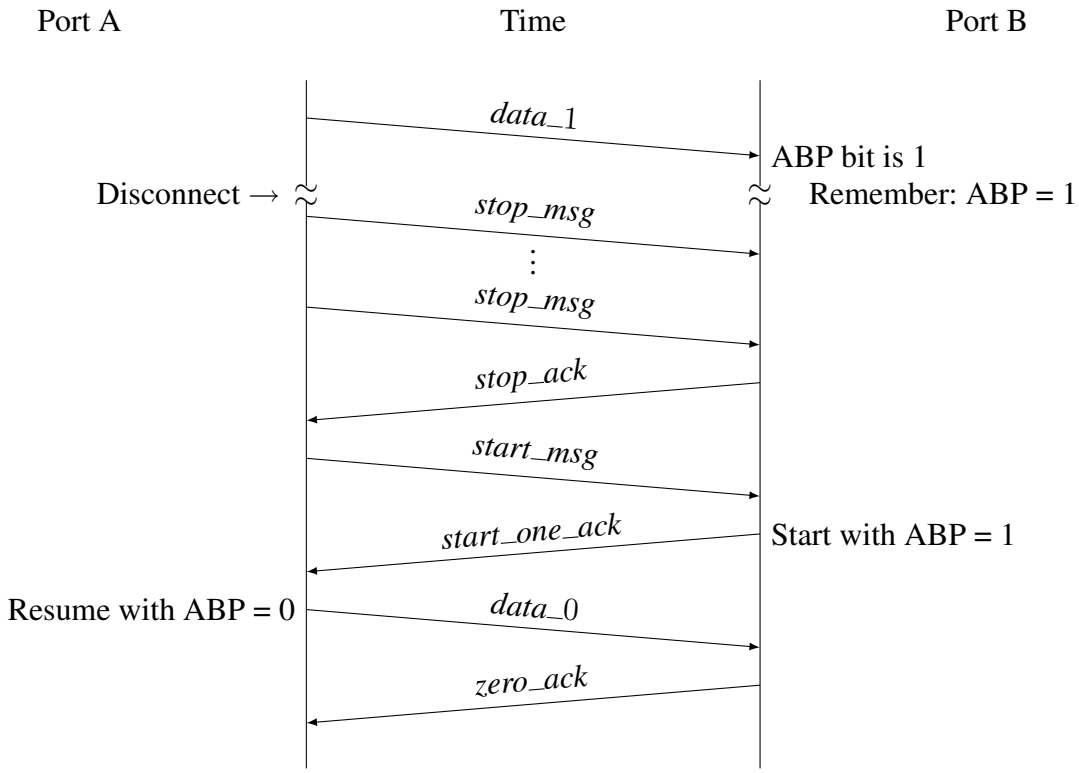

Figure 12. Protocol diagram disconnect error

process. This channel is used to communicate *start* and *stop* messages during the protocol initialisation. The last connection of the *TX_ABP* is with the *CNTMSG* process. Via this channel the *TX_ABP* process receives acknowledgement messages from the communication partner. The *CNTMSG* process routes all other control messages to the *RX_ABP* process. This process is responsible for the protocol on the receiver side. Apart from the control messages it also receives the ABP control bit from the *C* process. Based on this information it generates control messages for the receiver side of the communication partner. In order to send packets to the receiver, the *RX_ABP* process is connected with the *MUXER* process.

Figure 14 shows the reset network which connects the processes of the IC2IC example implementation in parallel with the data and control communication network. The local reset network is used for error handling. A local reset affects the whole IC2IC network implementation, but not the environment. That means after a local reset the IC2IC network is in a predefined state. This state is different from the global reset state, because some processes 'remember' particular information such as the value of the ABP bit. The *DEMUX* process informs the *RX* process that a protocol error, such as a parity error, happened during transmission. The *RX* process is the central point in the reset network, because it controls the *local_rst* signal. This signal is asserted after an *rx_rst* event is received or a disconnect is detected. This *local_rst* enables the process network to react to protocol error or disconnect.

The following sections provide an in-depth discussion of the individual processes. This discussion introduces the functionality of the processes during normal operation and their behaviour during local reset.

2.1. P Process

In its initial or reset state process *P* requests the ABP bit value from the *TX_ABP* process. Once *P* has received this value it is ready to receive a message from the environment via the *in* channel. After having received a message the *P* process appends the ABP token and sends the resulting packets to the *MUXER* process. This is the last step before *P* recurses. A local

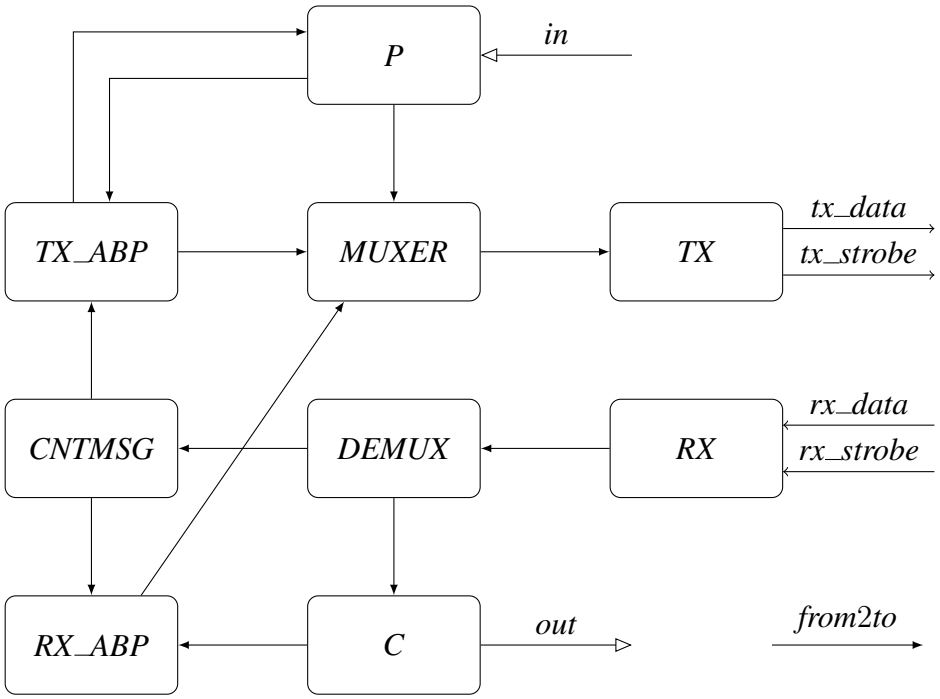

Figure 13. IC2IC process network with all the communication channels between the processes

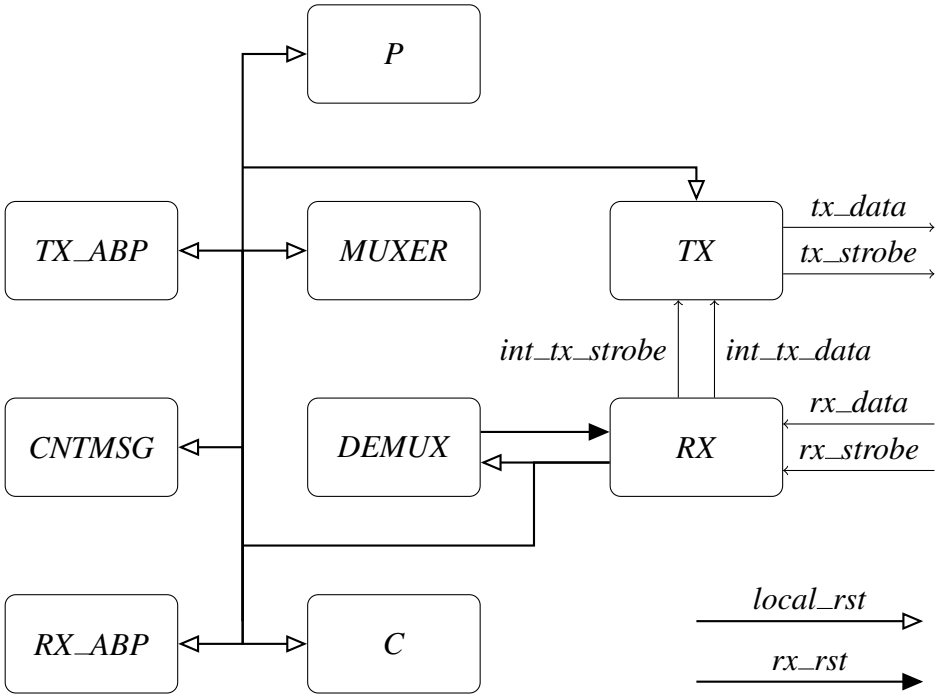

Figure 14. IC2IC reset network

reset forces this process into the initial state. However, it remembers the last data vector from the environment.

2.2. TX_ABP Process

TX_ABP handles the transmitter part of the ABP protocol. In the initial state, it establishes the protocol synchronisation with the receiver part of the communication partner. This is achieved by sending out *stop_msg* control packet to the *MUXER* process. The *stop_msg* are sent until the first *stop_ack* control packet is received via the channel which connects it to the

CNTMSG process. After having received the *stop_ack* message the *TX_ABP* process sends out the *start_msg*. The receiver can respond to the start message with one of three different control packets:

1. *start_rst_ack* – This control packet is used to indicate that port B (receiver) came out of a global reset.
2. *start_zero_ack* – This control packet indicates that the ABP bit of the last accepted data packet had ABP = 0.
3. *start_one_ack* – This control packet indicates that the ABP bit of the last accepted data packet had ABP = 1.

After initialisation, the *TX_ABP* process establishes the ABP functionality on the transmitter side. It provides the *P* process with the correct value of the ABP bit. The value of the ABP bit is negated when the acknowledgement packet (*one_ack* and *zero_ack*) indicates the same value as the current ABP bit. For *TX_ABP* local and global resets have the same effect. During the initialisation the communication partner provides the ABP bit information via one of the three possible responses to *start_msg*.

2.3. C Process

The *C* process consumes the messages from the *DEMUX* process and transfers them to the environment. The process acts as a message buffer, which means that initially it is able to receive a packet before the environment can block the acknowledgement of subsequent packets. The ABP bit is stripped from this packet and internally stored as well as sent to the *RX_ABP* process. After the ABP bit is stripped from the packet, the process is ready to communicate the message to the environment. In case the environment blocks this communication, the *C* process consumes all subsequent packets from the *MUXER* process but it does not send any *ABP* bits to the *RX_ABP* process. This effectively implements the blocking functionality, because the sender will continuously resend the same packet. In effect, this prevents the sender side from making progress, i.e. the sender is blocked. This block is released when the receiver environment picks up the packet from the *C* process. A local reset forces this process into the initial state, but it retains all information local to the process. Furthermore, the ability to deliver buffered data to the environment is not affected.

2.4. RX_ABP Process

The *RX_ABP* process handles the receiver side of the protocol. In the initial state the process is poised to receive 127 *stop_msg* control packets from the *CNTMSG*. The 128th and all subsequent *stop_msg* are acknowledged by sending *stop_ack* packets to the *MUXER*. Subsequently, a *start_msg* is acknowledged with *start_rst_ack*. This concludes the protocol synchronisation and *RX_ABP* handles the receiver (port B) functionality of the ABP protocol. In this state the *RX_ABP* process waits for input from *C* or *CNTMSG*. The *C* process can request the acknowledgement of new data packets. In case a *stop_msg* is received from the *CNTMSG* process or a local reset occurs the *RX_ABP* process returns to the initial protocol handling state. However, this time, the initial protocol is concluded with either *start_zero_ack* or *start_one_ack* depending on the last data packet which was acknowledged.

2.5. MUXER Process

The *MUXER* process relays or multiplexes the packets from *P*, *TX_ABP* and *RX_ABP* to the *TX* process. Furthermore, the *MUXER* process inserts an *alive* message if none of the connected processes supplies a message. This ensures a constant data transmission which allows the *RX* process in the receiver (port B) to detect a disconnect. The message transfer

to the *TX* process is a serial bit-stream. For the *MUXER* process local and global reset are identical.

2.6. DEMUX Process

The *DEMUX* process receives a serial bit-stream from the *RX* process. This bit-stream is decoded and the individual messages are assembled. During this assembly the parity is checked. If the parity check is successful, control packets are passed to the *CNTMSG* process and data packets are passed to the *C* process. All packets are passed on without the parity bit. In case the parity check fails the *DEMUX* process sends out a reset signal to the *RX* process via the *rx_rst* channel. For the *DEMUX* process local and global reset are identical.

2.7. RX Process

The *RX* process performs either control or data extraction tasks. In normal operation, *RX* extracts the data bits from the received *rx_data* and *rx_strobe* signals and sends them on to the *DEMUX* process. During control tasks the *RX* process asserts the *local_rst* signal and manages the output signals: *tx_data* and *tx_strobe*. *RX* has two distinct control tasks, exchange of silence and physical synchronisation. There are two conditions which cause *RX* to start an exchange of silence:

1. No signal transition (edge) was observed during a 1ms time window on either *rx_data* or *rx_strobe*. The absence of signal transitions indicates a disconnect from the communication partner.
2. An event is received via the *rx_rst* line. This indicates the *DEMUX* process has detected a bit error.

During exchange of silence *tx_data* and *tx_strobe* are set to low. After the silence period has elapsed, *RX* performs initial synchronisation. Initial synchronisation is also triggered by a global reset. During initial synchronisation *RX* behaves according to the specification given in Section 1.2.

2.8. TX Process

The *TX* process abstracts the layer-0 functionality of the transmitter. This abstraction involves synchronisation and data transfer functionality. Section 1.2 describes the physical synchronisation and data transfer which is carried out by the *TX* process. When a local reset signal is received the *TX* process hands over the control of the output signals to the *RX* process.

2.9. IC2IC test setup

Figure 15 shows the block diagram of the IC2IC test setup. Each FPGA executes the IC2IC process network. The test setup provides the means to introduce bit errors and disconnect the link at arbitrary time points. To have more testing flexibility, in addition to the IC2IC process network the ML403 hosts also a soft processor. This soft-processor was used to generate test sequences and monitor the received and transmitted messages. The SRSv02 hosts an additional process which mirrors all the messages. In other words, the test setup constitutes a loop, where data source and sink are located in the ML403.

The following list details the tests conducted with this setup:

- The system starts up and transfers data according to the protocol diagram for normal operation, shown in Figure 8. The data transfer starts regardless of the sequence in which the communication partners come online.

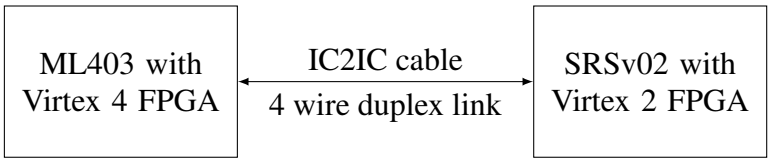

Figure 15. Block diagram of the IC2IC test setup

- The data stream was interrupted by disconnect at various time points. The stream recovered according to the protocol diagram for disconnect error shown in Figure 12.
- The data stream was interrupted by bit errors at various time points. The stream recovered according to the protocol diagrams for bit errors in various packets, see Figures 9, 10 and 11.

All these tests were conducted with a payload of $\text{DATA}_L = 8, 16$ and 32 bits. For each payload setup the test duration was at least 10 hours, i.e. the stream was running for this time. According to the test software, executed by the soft-processor, during all these tests no messages were lost or duplicated. This establishes confidence in the implementation stability.

The tests validate all IC2IC protocol features which were discussed in Section 1.4. Furthermore, the IC2IC process network was implemented in two different flexible logic devices[4]. This shows that the implementation is portable to a wide range of flexible logic devices. Finally, the link was tested with the help of a soft-processor. The results show that the blocking functionality is pervasive. That means, if the link is interrupted the software, executed by the soft-processor, is not able to send or receive a message, i.e. it is blocked. This shows blocking communication over two borders, software / hardware and IC (Virtex 4) / IC (Virtex 2).

3. Conclusion

In this paper we introduced IC2IC a lightweight serial interconnect channel for multiprocessor networks. This serial link establishes a low latency high performance data link between independent processors. The result is a reliable way to transfer bit level information from one IC to another IC. The blocking functionality and ability to recover from various error conditions distinguishes the IC2IC link from other serial interconnect links.

The practical part of the paper introduces the alternating bit protocol. This protocol constitutes the cornerstone of the IC2IC implementation, because it provides blocking functionality and hardens the link against message loss and message duplication. It was not possible to implement the alternating bit protocol directly, because in the basic form the protocol does not have the flexibility to cope with various practical reset conditions. Therefore, we extended the alternating bit protocol in order to cope with error conditions such as disconnect, bit error and receiver reset. The extended version of the protocol is called IC2IC. On the physical layer the IC2IC link uses a data strobe signal setup and the initial synchronisation is done with a specific synchronisation pattern.

The example implementation constitutes an IC2IC transceiver system. We designed the system as a network of independent processes which communicate via blocking channels. With this approach we carried on the idea of blocking channels. Apart from the communication channels the processes are also connected to a local reset signal. A local reset is asserted in case of an error on the physical link. The local reset signal interrupts all processes and forces them into a predefined state. The processes have the ability to 'remember' certain information after local reset. We use this memory functionality to avoid deadlock states and possible data loss after resynchronisation.

[4]Two different generations of the Xilinx Virtex family.

Process networks for hardware logic are a very elegant way to utilise the inherent parallelism of these devices. Key to these process networks are the blocking communication channels between them. It is relatively easy to establish blocking channels between processes within one chip. However, due to degrading effects on the physical means of communication and the unpredictability of the communication partner, it is difficult to establish blocking communication between logic circuits located in different ICs. The properties of the IC2IC serial link were validated with various tests on a hardware setup. These tests built up confidence in the claim that IC2IC serial links release process networks from the prison of only one IC. The IC2IC link enables blocking communication between two ICs. This blocking point to point communication is the corner stone for CSP style process networks which span over multiple ICs.

With the IC2IC protocol each individual payload message is acknowledged. Therefore, the IC2IC serial link is best described as buffered channel. The buffer has room for two massages, one in the producer (P) and one in the consumer (C). These buffers are necessary to expose a uniform interface to the environment. So, to their environment the IC2IC input and output looks like an ordinary hardware channel – even though the messages are transferred or received from another chip. If this interface is not enforced, back to back synchronisation on individual messages is possible. The possibility of back to back synchronisation is one of the strongest points in favour of the ABP protocol and indeed the IC2IC serial link. With a buffered channel there is always the uncertainty about whether or not a message has reached the destination. This uncertainty is potentially dangerous especially when important control messages are transferred. This predictability of the IC2IC serial link leads to simple abstract models. In CSP the IC2IC serial link with uniform interfaces is modelled as a buffered channel with two places. In case these interfaces are not enforced, the IC2IC serial link can be abstracted as a CSP style channel.

3.1. Future Work

The development of the IC2IC protocol is the first step towards process networks implemented in multiple ICs. The next development step is concerned with particular examples. These examples help to establish the merits of the protocol. The translation step from the theoretical model to the implementation is not proven, therefore examples are necessary to built up trust in the protocol. Based on particular examples, speed and robustness properties can be tested. The speed property is concerned with the data rate. There is a theoretical maximum data rate and specific data rates for individual applications. To compare the individual with the maximum data rates reveals some insights into the IC2IC protocol. Robustness considerations are concerned with particulars of the channel. The communication channel between two ICs might introduce bit errors. Future work will be concerned with analysing the effects of bit errors on the protocol. Furthermore, the communication channel might not be stable. That means, there are random disconnects due to bouncing connections. Some work will focus on simulating such bouncing conditions and analysing the performance of the IC2IC protocol.

After having established trust in the protocol implementation we can start looking towards more involved systems. Process networks are the most widely used mechanism to model complex systems with non-linear behaviour. The size of the process networks, i.e. the number of processes involved, depends on the complexity of the task. For many practical problems the task is ill defined, therefore the complexity of the system is not defined. However, more complex systems can react to a wider range of environments. That means, systems which model artificial intelligence tend to work better with larger process networks. The IC2IC protocol might be one way to extend such networks beyond the borders of a single IC without functional side effects. The only negative side effect might be insufficient communication speed. But, this is a field of further research.

Acknowledgements

This work was supported by the European FP6 project "WARMER" (contract no.: FP6-034472).

References

[1] C. R. Anderson, Marcel Boosten, R. W. Dobinson, S. Haas, R. Heeley, N. A. H. Madsen, B. Martin, J. Pech, D. A. Thornley, and C. L. Ullod. IEEE 1355 DS-Links: Present Status and Future Prospects. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 69–80, mar 1998.

[2] Colin Whitby-Strevens. The transputer. *SIGARCH Comput. Archit. News*, 13(3):292–300, 1985.

[3] A. Johannet, G. Loheac, L. Personnaz, I. Guyon, and G. Dreyfus. A transputer based neurocomputer. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 120–127, sep 1987.

[4] M. Meriaux, A. Atamenia, and E. Lepretre. A transputer-based architecture for graphics. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 297–306, sep 1987.

[5] Standard for Heterogeneous Inter-Connect (HIC). Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction, 1995. 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[6] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36(5):547–553, 1987.

[7] Barry M. Cook and C. P. H. Walker. SpaceWire - DS-Links Reborn. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages –, sep 2006.

[8] Barry M. Cook. IEEE 1355 data-strobe links: ATM speed at RS232 cost. *Microprocessors and Microsystems*, 21(7-8):421–428, March 1998.

[9] O. J. Greve, M. H. Schwirtz, Gerald H. Hilderink, Jan F. Broenink, and Andrè W. P. Bakkers. An A/D D/A board using IEEE-1355 DS-Links for Heterogeneous Multiprocessor Environment. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 27–38, mar 1998.

[10] Marcel Boosten, R. W. Dobinson, B. Martin, and P. D. V. van der Stok. A PCI-based Network Interface Controller for IEEE 1355 DS-Links. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 49–68, mar 1998.

[11] IEEE-1394-1996 High Speed Serial Bus, 1996. 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[12] ECSS-E-50-12A SpaceWire – Linkes, nodes, routers and networks. (24 January 2003).

[13] B. Fiethe, H. Michalik, C. Dierker, B. Osterloh, and G. Zhou. Reconfigurable system-on-chip data processing units for space imaging instruments. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 977–982, San Jose, CA, USA, 2007. EDA Consortium.

[14] Sanjit A. Seshia, Wenchao Li, and Subhasish Mitra. Verification-guided soft error resilience. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1442–1447, San Jose, CA, USA, 2007. EDA Consortium.

[15] Sergio Saponara, Esa Petri, Marco Tonarelli, Iacopo Del Corona, and Luca Fanucci. FPGA-based networking systems for high data-rate and reliable in-vehicle communications. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 480–485, San Jose, CA, USA, 2007. EDA Consortium.

[16] Ad M. G. Peeters. Implementation of Handshake Components. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 98–132. Springer, 2004.

[17] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–265, 1969.

[18] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, New Jersey 07485 United States of America, first edition, 1997.

[19] Bernhard Sputh, Oliver Faust, and Alastair R. Allen. RRABP: Point-to-Point Communication over Unreliable Components. In P.H. Welch et al., editor, *Communicating Process Architectures 2008*, pages 203–217, 2008.