

X67BC8780.L12

1 General information

1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX67	X67 system user's manual
MAEMV	Installation / EMC guide

1.2 Order data


Order number	Short description	Figure
	Bus controller modules	
X67BC8780.L12	X67 POWERLINK bus controller, 1x CAN interface, with active wiring help for 8x M12 connections (star wiring), M12 connectors, LED status indicators, high-density module	

Table 1: X67BC8780.L12 - Order data

Required accessories
See " Required cables and connectors " on page 7.
For a general overview, see section "Accessories - General overview" in the X67 system user's manual.

1.3 Module description

This bus controller makes it possible to connect 8 CAN interfaces to the POWERLINK network. The CAN interfaces are operated via an integrated hub, which increases the total cable length defined by classical CAN bus topology.

The interface is equipped with 2 connections. Both connections are connected to an integrated switch. This makes it easy to implement daisy chain cabling.

Functions:

- [POWERLINK](#)
- [CAN interfaces](#)
- [Flatstream](#)

POWERLINK

POWERLINK is a standard protocol for Fast Ethernet equipped with hard real-time characteristics.

CAN interfaces

The module is equipped with 8 CAN interfaces that can be used to transmit and receive CAN objects.

Flatstream communication

"Flatstream" was designed for X2X and POWERLINK networks and allows data transfer to be adapted to individual demands. This allows data to be transferred more efficiently than with standard cyclic polling.

2 Technical description

2.1 Technical data

Order number	X67BC8780.L12
Short description	
Bus controller	POWERLINK (V1/V2) controlled node
General information	
Inputs/Outputs	8 CAN interfaces
Insulation voltage between channel and bus	500 V _{eff}
Nominal voltage	24 VDC
B&R ID code	
Bus controller	0xED84
Status indicators	TX/RX per CAN interface, supply voltage, bus function
Diagnostics	
Outputs	Yes, using LED status indicator and software
Power supply	Yes, using LED status indicator and software
Support	
Dynamic node allocation (DNA)	Yes
Connection type	
Fieldbus	M12, D-coded
Inputs/Outputs	8x M12, A-coded
I/O power supply	M8, 4-pin
Power consumption	
Internal I/O	2.5 W
Certifications	
CE	Yes
UKCA	Yes
EAC	Yes
Fieldbus	
Type	POWERLINK (V1/V2) controlled node
Variant	2x M12 interface (hub), 2x female connector on module
Line length	Max. 100 m between 2 stations (segment length)
Transfer rate	100 Mbit/s
Transfer	
Physical layer	100BASE-TX
Half-duplex	Yes
Full-duplex	No
Autonegotiation	Yes
Auto-MDI/MDIX	Yes
Hub propagation delay	0.96 to 1 µs
Cycle time ¹⁾	200 µs
Interfaces	
Type	CAN
Variant	8x M12 interface (hub)
Max. distance	1000 m
Transfer rate	Bus length ≤25 m: Max. 500 kbps Bus length ≤100 m: Max. 250 kbps Bus length ≤700 m: Max. 50 kbps ²⁾
Hub propagation delay	250 ns
Terminating resistor	Integrated in module
Sensor/Actuator power supply	
Voltage	9 to 32 V
Voltage drop for short-circuit protection at 0.5 A	Max. 2 VDC
Summation current	Max. 1.6 A (200 mA per channel)
Short-circuit proof	Yes
Power supply	
Nominal voltage	24 VDC
Voltage range	9 to 32 VDC
Reverse polarity protection	Yes
Electrical properties	
Electrical isolation	POWERLINK isolated from channel Channel not isolated from channel
Operating conditions	
Mounting orientation	
Any	Yes
Installation elevation above sea level	
0 to 2000 m	No limitation
>2000 m	Reduction of ambient temperature by 0.5°C per 100 m
Degree of protection per EN 60529	IP67

Table 2: X67BC8780.L12 - Technical data

Order number	X67BC8780.L12
Ambient conditions	
Temperature	
Operation	-25 to 60°C
Derating	-
Storage	-40 to 85°C
Transport	-40 to 85°C
Mechanical properties	
Dimensions	
Width	53 mm
Height	155 mm
Depth	42 mm
Torque for connections	
M8	Max. 0.4 Nm
M12	Max. 0.6 Nm

Table 2: X67BC8780.L12 - Technical data

- 1) The minimum cycle time specifies how far the bus cycle can be reduced without communication errors occurring.
- 2) The actual transfer rate depends on the nodes (number and properties of the nodes) and the cable used.

2.2 LED status indicators

Figure	LED	Color	Status	Description
<p>Status indicator 1: Left: L/A IF; right: S/E</p> <p>Status indicator 2: Left: Green, Right: Red</p>	Status indicator 1: Status indicator for POWERLINK bus controller			
	L/A IF	Green	On	The link to the remote station is established.
			Blinking	The link to the remote station is established. The LED blinks if Ethernet activity is taking place on the bus.
	S/E ¹⁾	Green/Red		LED "Status/Error". LED statuses are described in "Status/Error LED "S/E"" on page 4.
	CAN LEDs			
	1-1 to 8-1	Yellow	On	TX - Module transmitting data via the CAN interface
	1-2 to 8-2		On	RX - Module receiving data via the CAN interface
	Status indicator 2: Status indicator for module functionality			
	Left	Green	Off	No power to module
			Single flash	Mode RESET
			Blinking	Mode PREOPERATIONAL
			On	Mode RUN
	Right	Red	Off	Module not supplied with power or everything OK
			On	Error or reset state
			Single flash	I/O error occurred: <ul style="list-style-type: none"> • CAN bus: Warning, passive or off • Buffer overflow
			Double flash	Supply voltage not within the valid range

- 1) LED "Status/Error" is a green/red dual LED.

2.2.1 Status/Error LED "S/E"

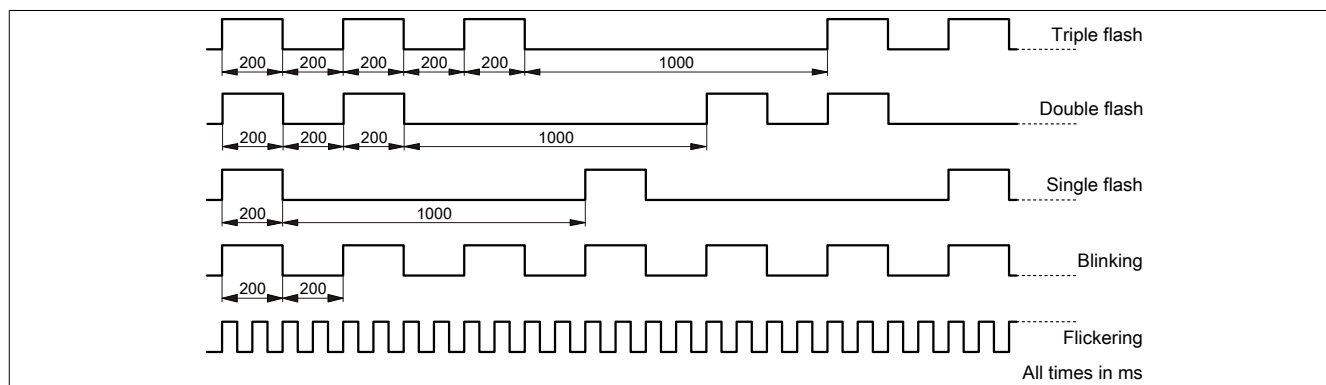
LED "Status/Error" is a green and red dual LED. The color green (status) is superimposed on the color red (error).

Color red - Error	Description
On	<p>The controlled node (CN) is in an error state (failed Ethernet frames, increased number of collisions on the network, etc.). If an error occurs in the following states, the red LED is superimposed by the green flashing LED:</p> <ul style="list-style-type: none"> • PRE_OPERATIONAL_1 • PRE_OPERATIONAL_2 • READY_TO_OPERATE <p>Note:</p> <ul style="list-style-type: none"> • Several red blinking signals are displayed immediately after the device is switched on. This is not an error, however. • The LED lights up red for CNs with set physical node number 0 that have not yet been assigned a node number via dynamic node allocation (DNA).

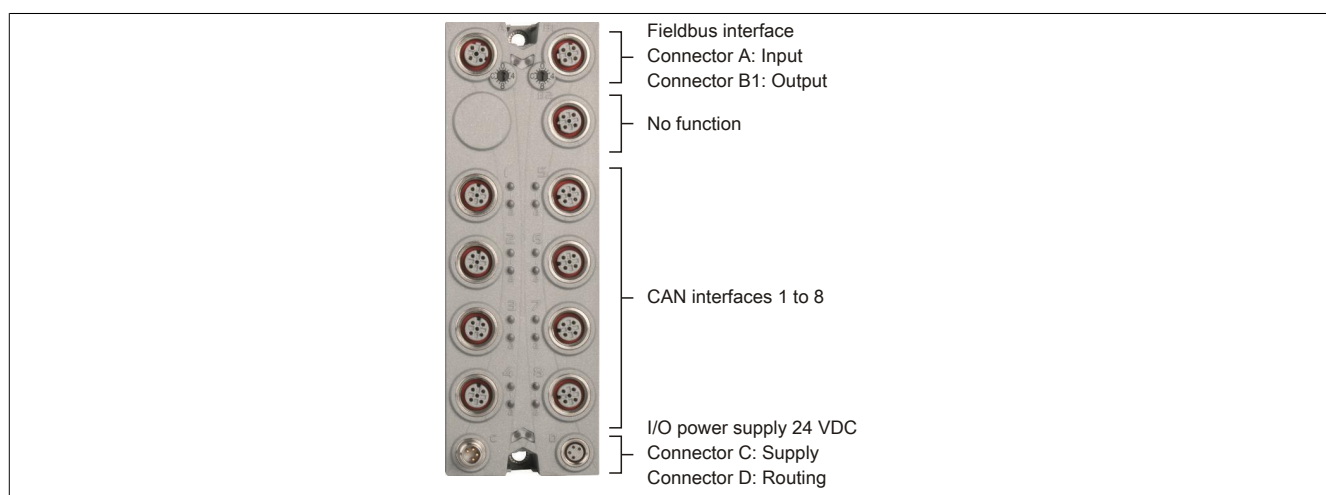
Table 3: Status/Error LED lit red: LED indicating error state

Color green - Status	Description
Off	No power supply or mode NOT_ACTIVE. The controlled node (CN) is either not supplied with power or it is in state NOT_ACTIVE. The CN waits in this state for about 5 s after a restart. Communication is not possible with the CN. If no POWERLINK communication is detected during these 5 s, the CN changes to state BASIC_ETHERNET (flickering). If POWERLINK communication is detected before this time expires, however, the CN immediately changes to state PRE_OPERATIONAL_1.
Green flickering (approx. 10 Hz)	Mode BASIC_ETHERNET. The CN has not detected any POWERLINK communication. In this state, it is possible to communicate directly with the CN (e.g. with UDP, IP). If POWERLINK communication is detected in this state, the CN changes to state PRE_OPERATIONAL_1.
Single flash (approx. 1 Hz)	Mode PRE_OPERATIONAL_1. When operating on a POWERLINK V1 manager, the CN immediately changes to state PRE_OPERATIONAL_2. When operating on a POWERLINK V2 manager, the CN waits until an SoC frame is received and then changes to state PRE_OPERATIONAL_2.
Double flash (approx. 1 Hz)	Mode PRE_OPERATIONAL_2. The CN is normally configured by the manager in this state. It is then switched to state READY_TO_OPERATE by command (POWERLINK V2) or by setting flag "Data valid" in the output data (POWERLINK V1).
Triple flash (approx. 1 Hz)	Mode READY_TO_OPERATE. In a POWERLINK V1 network, the CN switches to state OPERATIONAL automatically as soon as input data is present. In a POWERLINK V2 network, the manager switches to state OPERATIONAL by command.
On	Mode OPERATIONAL. PDO mapping is active and cyclic data is evaluated.
Blinking (approx. 2.5 Hz)	Mode STOPPED. Output data is not being output, and no input data is being provided. It is only possible to switch to or leave this state after the manager has given the appropriate command.

Table 4: Status/Error LED lit green: LED indicating operating state

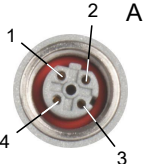
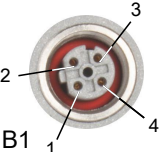


2.3 Operating and connection elements



2.3.1 Fieldbus interfaces

The module is connected to the network using pre-assembled cables. The connection is made using M12 circular connectors.

Connection	Pinout		
	Pin	Name	
	1	TXD	Transmit data
	2	RXD	Receive data
	3	TXD\	Transmit data\
	4	RXD\	Receive data\
	Shield connection made via threaded insert in the module		
	A → D-coded (female), input		
	B1 → D-coded (female), output		

Information:

The color of the wires used in field-assembled cables for connecting to the fieldbus interface may deviate from the standard.

It is very important to ensure that the pinout is correct (see section "Accessories - POWERLINK cables" in the X67 user's manual).

2.3.1.1 Wiring guidelines for bus controllers with Ethernet cable

Some X67 system bus controllers are based on Ethernet technology. POWERLINK cables offered by B&R can be used for wiring.

Order number	Connection type
X67CA0E41.xxxx	Attachment cables - RJ45 to M12
X67CA0E61.xxxx	Connection cables - M12 to M12

The following cabling guidelines must be observed:

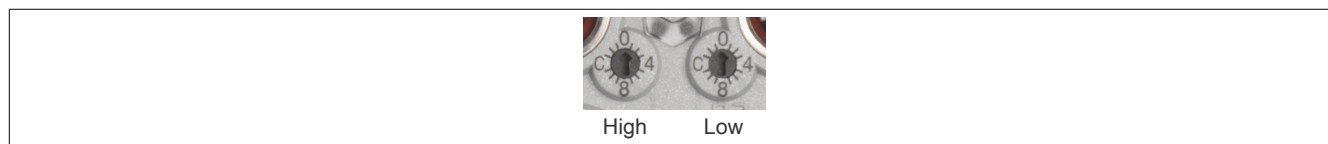
- Use Cat 5 SFTP cables.
- Observe the bend radius of the cable (see the data sheet of the cable)

Information:

Using POWERLINK cables offered by B&R (X67CA0E61.xxxx and X67CA0E41.xxxx) meets product standard EN 61131-2.

The customer must implement additional measures in the event of further requirements.

2.3.1.2 POWERLINK node number

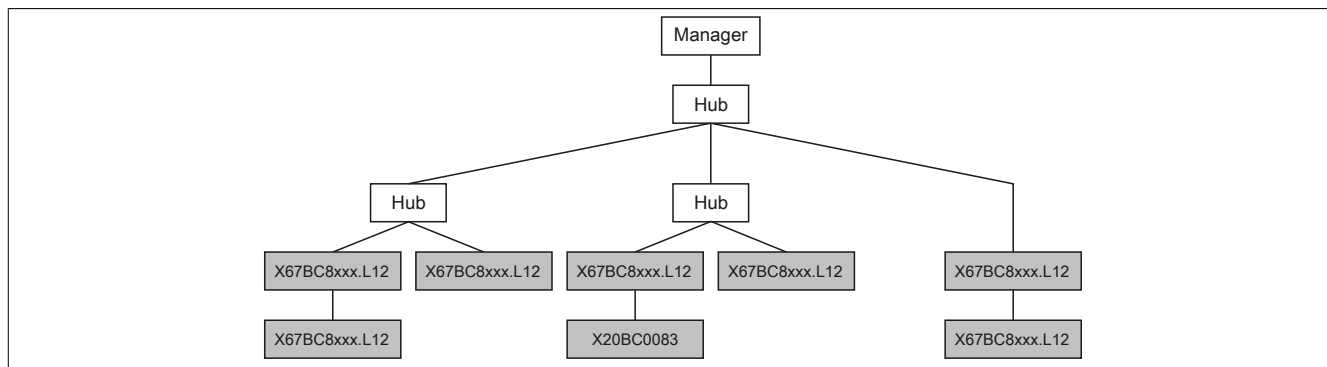


The node number for the POWERLINK node is set using the two number switches.

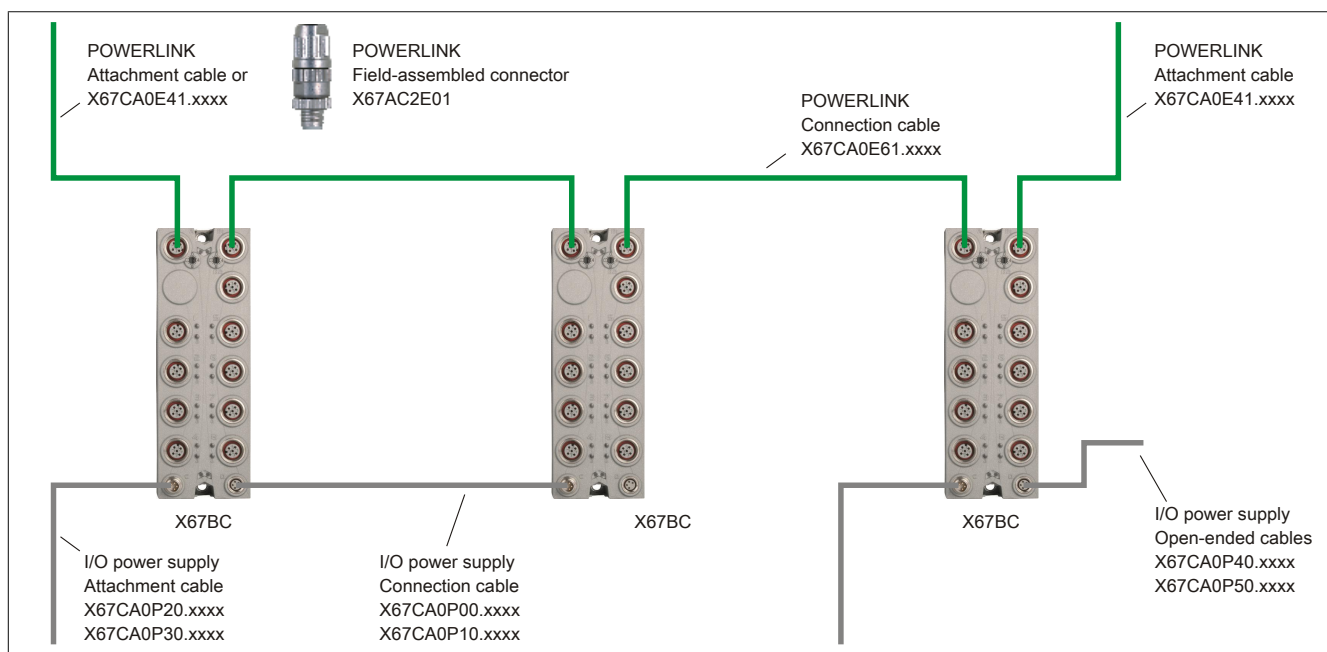
Switch position	Description
0x00	Only permitted when operating the POWERLINK node in DNA mode.
0x01 - 0xEF	Node number of the POWERLINK node. Operation as a controlled node (CN).
0xF0 - 0xFF	Reserved, switch position not permitted.

2.3.1.3 Integrating into a POWERLINK network

The bus controller is used in a tree or line structure as follows:

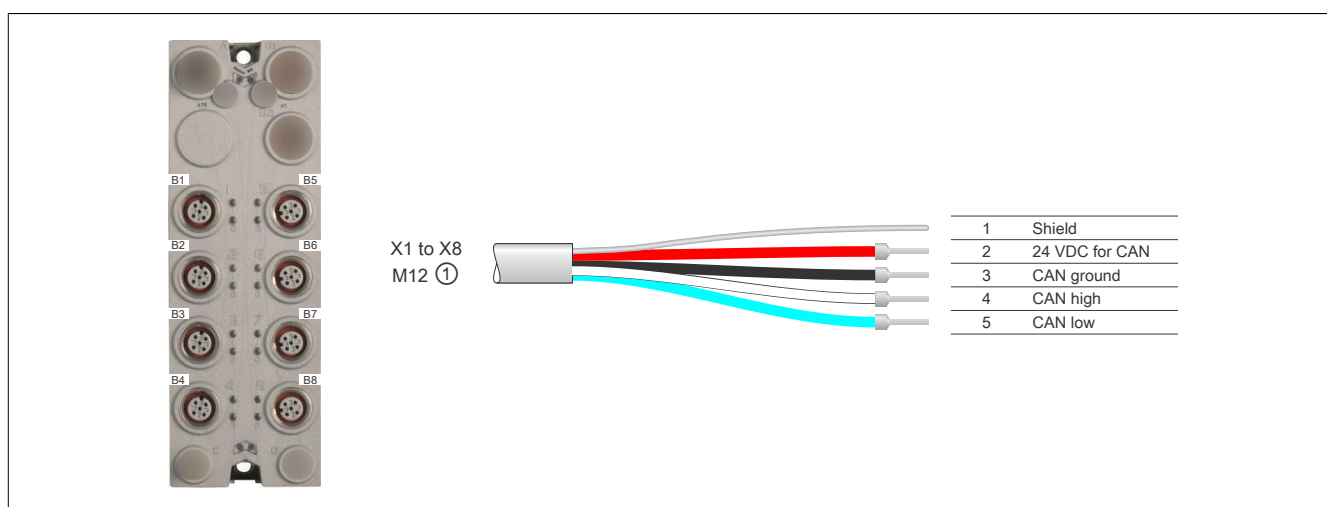


2.3.1.4 Required cables and connectors



2.3.2 CAN interface

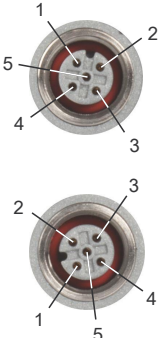
2.3.2.1 Pinout



- ① X67CA0C22.xxxx: M12 CAN bus cable, straight
X67CA0C32.xxxx: M12 CAN bus cable, angled

2.3.2.2 Connector B1 to B8

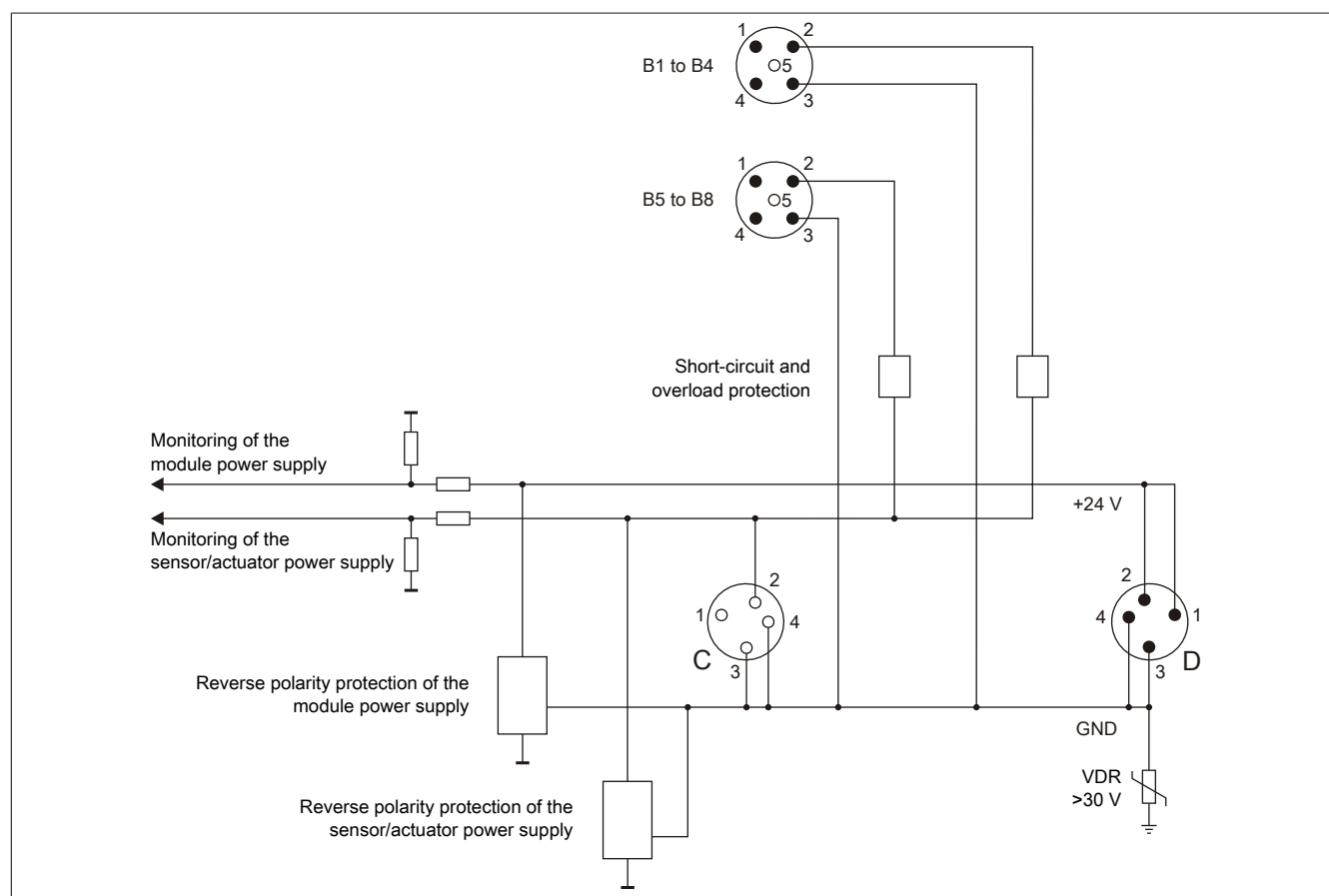
The module is connected to the network using pre-assembled cables. The connection is made using M12 circular connectors.

Connection	Pinout	
	Pin	Name
B1 to B8 	1	Shield
	2	24 VDC for CAN
	3	CAN ground
	4	CAN high
	5	CAN low
	Shield connection made via threaded insert in the module B1 to B8 → A-coded (female), input/output	

2.3.2.3 Terminating resistor

Each CAN interface (B1 to B8) in the module is equipped with a terminating resistor.

2.4 Input/Output circuit diagram

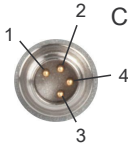
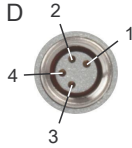


2.5 24 VDC power supply

The power supply is connected via M8 connectors C and D. The power supply is fed via connector C (male). Connector D (female) is used to route the power supply to other modules.

Information:

The maximum permissible current for the power supply is 4 A.

Connection	Pinout		
	Pin	Connector C (male)	Connector D (female)
	1	24 VDC module	24 VDC module
	2	24 VDC actuator/sensor	24 VDC module
	3	GND	GND
	4	GND	GND
	C → Connector (male) in module, feed for power supply D → Connector (female) in module, routing of power supply		

3 Function description

3.1 POWERLINK

POWERLINK is an Ethernet-based, real-time capable fieldbus. POWERLINK extends the IEEE 802.3 Ethernet standard by a deterministic access method and also defines a CANopen-compatible fieldbus interface. POWERLINK distinguishes between process and service data in the same way as CANopen. Process data (PDO) is exchanged cyclically in the cyclic phase, while service data (SDO) is transferred acyclically. Service data objects are transmitted in the acyclic phases of POWERLINK using a connection-oriented protocol. The cyclic transfer of data in PDOs is enabled by "mapping".

For additional information, see [POWERLINK bus controller user's manual](#) and www.br-automation.com/en/technologies/powerlink.

3.2 CAN interfaces

The module is equipped with 8 CAN interfaces that can be used to transmit and receive CAN objects.

3.2.1 Transmitting and receiving

In function model "Flat", CAN information is transferred via cyclic input and output registers.

To transmit a CAN object, the CAN identifier, the CAN data (max. 8 bytes) and the number of bytes to be transmitted must be written to the cyclic I/O data points. "TXCount" is then increased to send the transmission. The data is applied to the module's internal buffer (max. 18 objects) and transmitted over the CAN network at the next opportunity.

The same algorithm is used for receiving information from the CAN network. The module saves the CAN messages in its internal buffer along with the respective identifiers. The CAN identifier, CAN data (max. 8 bytes) and number of bytes to be processed are then written to the cyclic I/O data points. "RXCount" tells the application which data must be applied from these input data points.

3.2.2 The CAN object

A CAN object always consists of 4 bytes identifier and a maximum of 8 following data bytes. This also results in the relationship between CAN object length and the amount of CAN payload data. This is important because the number of CAN user data bytes must always be determined by the frame length when communicating using "Flatstream".

Composition of a CAN object or CAN frame

Byte	Explanation	Information
1	Identifier	ID bits 0 to 7
2		ID bits 8 to 15
3		ID bits 16 to 23
4		ID bits 24 to 31
5 - 12	CAN payload data	0 to 8 CAN payload data bytes

Identifier

The 32 bits (4 bytes) of the CAN identifier are used as follows:

Bit	Description	Value	Information
0	Frame format	0	Standard frame format (SFF) with 11-bit identifier
		1	Extended frame format (EFF) with 29-bit identifier
1	Frame type	0	Data frame
		1	Remote frame (RTR)
2	Reserved	-	
3 - 31	CAN identifier of the telegram to be transmitted	x	Extended frame format (EFF) with 29 bits Standard frame format (SFF) with 11 bits ¹⁾

1) Only bits 21 to 31 are used; bits 3 to 20 = 0.

3.2.3 Data stream of the CAN module

In function model 254, the data packets of a data stream to be transferred are referred to as frames.

Information:

The following applies to the CAN module:

- A frame always contains a CAN object and thus cannot be longer than 12 bytes.
- The CAN object is only applied to the transmit buffer after the frame has been completed.
- The CAN payload data length is firmly related to the frame length or the actual size of the CAN object. The following applies:
 - $\text{CAN payload data length} = \text{Frame length} - 4$
 - $\text{Frame length} = \text{CAN payload data length} + 4$

3.2.4 Consideration of error cases during transmission

Data may be lost on the POWERLINK or X2X Link network due to transfer interference. A one-time failure of cyclic data is tolerated by the I/O systems. This is possible because all I/O data is re-transferred in the following cycle. A transfer error is not visible at the I/O variables; these remain frozen at the value of the last cycle.

This toleration of one-time I/O failures may result in loss or delayed transmission of CAN telegrams. The counter feedback is calculated on the module and used to detect these cases.

Information:

The registers are described under ["TXCountReadBack" on page 44](#) and ["TXCountLatchReadBack" on page 44](#).

3.3 Flatstream communication

3.3.1 Introduction

B&R offers an additional communication method for some modules. "Flatstream" was designed for X2X and POWERLINK networks and allows data transfer to be adapted to individual demands. Although this method is not 100% real-time capable, it still allows data transfer to be handled more efficiently than with standard cyclic polling.

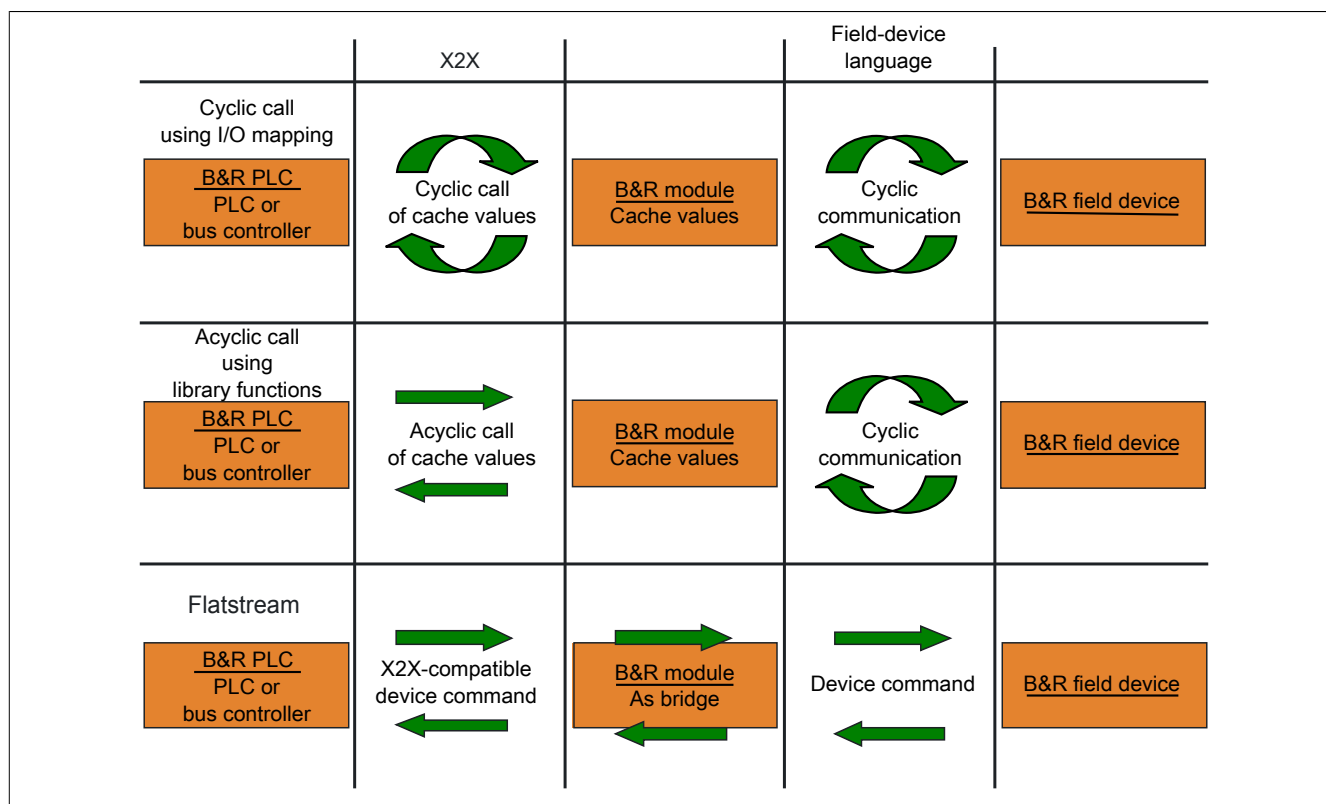


Figure 1: 3 types of communication

Flatstream extends cyclic and acyclic data queries. With Flatstream communication, the module acts as a bridge. The module is used to pass controller requests directly on to the field device.

3.3.2 Message, segment, sequence, MTU

The physical properties of the bus system limit the amount of data that can be transmitted during one bus cycle. With Flatstream communication, all messages are viewed as part of a continuous data stream. Long data streams must be broken down into several fragments that are sent one after the other. To understand how the receiver puts these fragments back together to get the original information, it is important to understand the difference between a message, a segment, a sequence and an MTU.

Message

A message refers to information exchanged between 2 communicating partner stations. The length of a message is not restricted by the Flatstream communication method. Nevertheless, module-specific limitations must be considered.

Segment (logical division of a message):

A segment has a finite size and can be understood as a section of a message. The number of segments per message is arbitrary. So that the recipient can correctly reassemble the transferred segments, each segment is preceded by a byte with additional information. This control byte contains information such as the length of a segment and whether the approaching segment completes the message. This makes it possible for the receiving station to interpret the incoming data stream correctly.

Sequence (how a segment must be arranged physically):

The maximum size of a sequence corresponds to the number of enabled Rx or Tx bytes (later: "MTU"). The transmitting station splits the transmit array into valid sequences. These sequences are then written successively to the MTU and transferred to the receiving station where they are lined up together again. The receiver stores the incoming sequences in a receive array, obtaining an image of the data stream in the process.

With Flatstream communication, the number of sequences sent are counted. Successfully transferred sequences must be acknowledged by the receiving station to ensure the integrity of the transfer.

MTU (Maximum Transmission Unit) - Physical transport:

MTU refers to the enabled USINT registers used with Flatstream. These registers can accept at least one sequence and transfer it to the receiving station. A separate MTU is defined for each direction of communication. OutputMTU defines the number of Flatstream Tx bytes, and InputMTU specifies the number of Flatstream Rx bytes. The MTUs are transported cyclically via the X2X Link network, increasing the load with each additional enabled USINT register.

Properties

Flatstream messages are not transferred cyclically or in 100% real time. Many bus cycles may be needed to transfer a particular message. Although the Rx and Tx registers are exchanged between the transmitter and the receiver cyclically, they are only processed further if explicitly accepted by register "InputSequence" or "OutputSequence".

Behavior in the event of an error (brief summary)

The protocol for X2X and POWERLINK networks specifies that the last valid values should be retained when disturbances occur. With conventional communication (cyclic/acyclic data queries), this type of error can generally be ignored.

In order for communication to also take place without errors using Flatstream, all of the sequences issued by the receiver must be acknowledged. If Forward functionality is not used, then subsequent communication is delayed for the length of the disturbance.

If Forward functionality is being used, the receiving station receives a transmission counter that is incremented twice. The receiver stops, i.e. it no longer returns any acknowledgments. The transmitting station uses SequenceAck to determine that the transfer was faulty and that all affected sequences must be repeated.

3.3.3 The Flatstream principle

Requirements

Before Flatstream can be used, the respective communication direction must be synchronized, i.e. both communication partners cyclically query the sequence counter on the remote station. This checks to see if there is new data that should be accepted.

Communication

If a communication partner wants to transmit a message to its remote station, it should first create a transmit array that corresponds to Flatstream conventions. This allows the Flatstream data to be organized very efficiently without having to block other important resources.

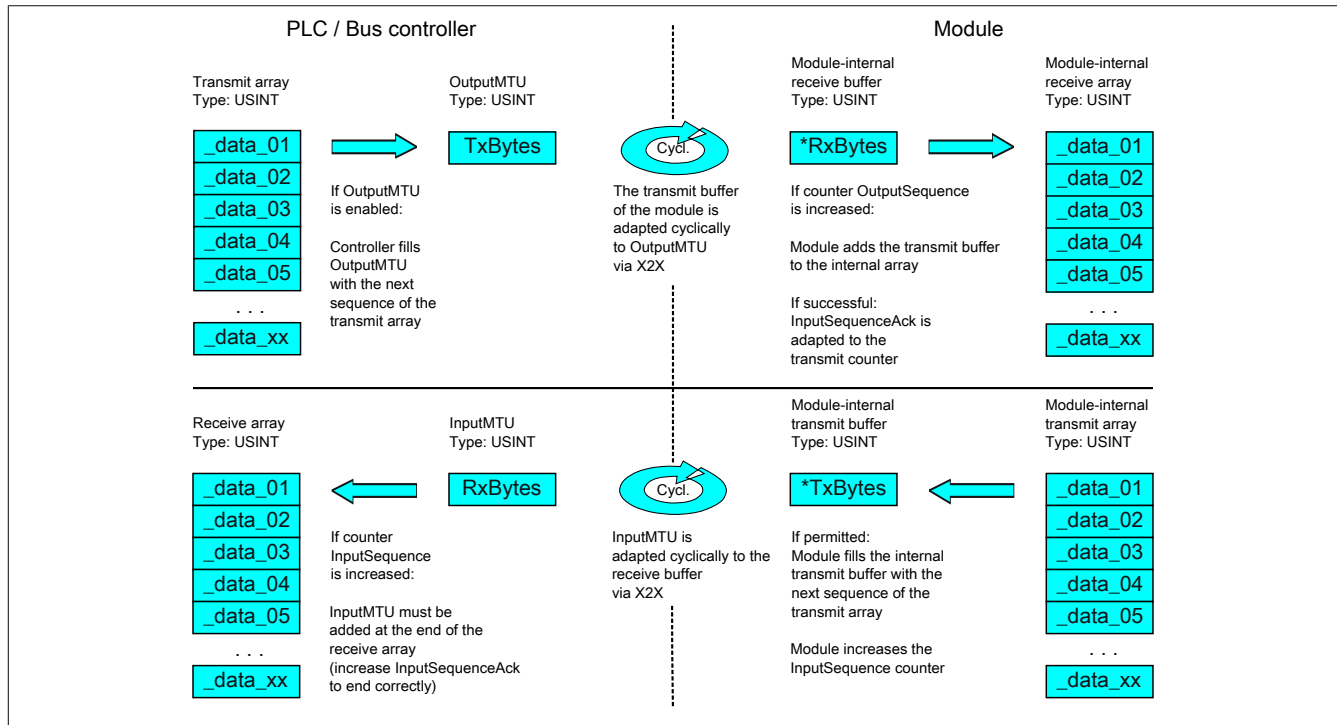


Figure 2: Flatstream communication

Procedure

The first thing that happens is that the message is broken into valid segments of up to 63 bytes, and the corresponding control bytes are created. The data is formed into a data stream made up of one control bytes per associated segment. This data stream can be written to the transmit array. The maximum size of each array element matches that of the enabled MTU so that one element corresponds to one sequence.

If the array has been completely created, the transmitter checks whether the MTU is permitted to be refilled. It then copies the first element of the array or the first sequence to the Tx byte registers. The MTU is transported to the receiver station via X2X Link and stored in the corresponding Rx byte registers. To signal that the data should be accepted by the receiver, the transmitter increases its SequenceCounter.

If the communication direction is synchronized, the remote station detects the incremented SequenceCounter. The current sequence is appended to the receive array and acknowledged by SequenceAck. This acknowledgment signals to the transmitter that the MTU can now be refilled.

If the transfer is successful, the data in the receive array will correspond 100% to the data in the transmit array. During the transfer, the receiving station must detect and evaluate the incoming control bytes. A separate receive array should be created for each message. This allows the receiver to immediately begin further processing of messages that are completely transferred.

3.3.4 Registers for Flatstream mode

5 registers are available for configuring Flatstream. The default configuration can be used to transmit small amounts of data relatively easily.

Information:

The controller communicates directly with the field device via registers "OutputSequence" and "InputSequence" as well as the enabled Tx and Rx bytes. For this reason, the user needs to have sufficient knowledge of the communication protocol being used on the field device.

3.3.4.1 Flatstream configuration

To use Flatstream, the program sequence must first be expanded. The cycle time of the Flatstream routines must be set to a multiple of the bus cycle. Other program routines should be implemented in Cyclic #1 to ensure data consistency.

At the absolute minimum, registers "InputMTU" and "OutputMTU" must be set. All other registers are filled in with default values at the beginning and can be used immediately. These registers are used for additional options, e.g. to transfer data in a more compact way or to increase the efficiency of the general procedure.

The Forward registers extend the functionality of the Flatstream protocol. This functionality is useful for substantially increasing the Flatstream data rate, but it also requires quite a bit of extra work when creating the program sequence.

Information:

In the rest of this description, the names "OutputMTU" and "InputMTU" do not refer to the registers names. Instead, they are used as synonyms for the currently enabled Tx or Rx bytes.

Information:

The registers are described in ["Flatstream registers" on page 48](#).

3.3.4.2 Flatstream operation

When using Flatstream, the communication direction is very important. For transmitting data to a module (output direction), Tx bytes are used. For receiving data from a module (input direction), Rx bytes are used. Registers "OutputSequence" and "InputSequence" are used to control or secure communication, i.e. the transmitter uses them to give instructions to apply data and the receiver confirms a successfully transferred sequence.

Information:

The registers are described in ["Flatstream registers" on page 48](#).

3.3.4.2.1 Format of input and output bytes

Name:

"Format of Flatstream" in Automation Studio

On some modules, this function can be used to set how the Flatstream input and output bytes (Tx or Rx bytes) are transferred.

- **Packed:** Data is transferred as an array.
- **Byte-by-byte:** Data is transferred as individual bytes.

3.3.4.2.2 Transport of payload data and control bytes

The Tx and Rx bytes are cyclic registers used to transport the payload data and the necessary control bytes. The number of active Tx and Rx bytes is taken from the configuration of registers ["OutputMTU"](#) and ["InputMTU"](#), respectively.

In the user program, only the Tx and Rx bytes from the controller can be used. The corresponding counterparts are located in the module and are not accessible to the user. For this reason, the names were chosen from the point of view of the controller.

- "T" - "Transmit" → Controller *transmits* data to the module.
- "R" - "Receive" → Controller *receives* data from the module.

Control bytes

In addition to the payload data, the Tx and Rx bytes also transfer the necessary control bytes. These control bytes contain additional information about the data stream so that the receiver can reconstruct the original message from the transferred segments.

Bit structure of a control byte

Bit	Name	Value	Information
0 - 5	SegmentLength	0 - 63	Size of the subsequent segment in bytes (default: Max. MTU size - 1)
6	nextCBPos	0	Next control byte at the beginning of the next MTU
		1	Next control byte directly after the end of the current segment
7	MessageEndBit	0	Message continues after the subsequent segment
		1	Message ended by the subsequent segment

SegmentLength

The segment length lets the receiver know the length of the coming segment. If the set segment length is insufficient for a message, then the information must be distributed over several segments. In these cases, the actual end of the message is detected using bit 7 (control byte).

Information:

The control byte is not included in the calculation to determine the segment length. The segment length is only derived from the bytes of payload data.

nextCBPos

This bit indicates the position where the next control byte is expected. This information is especially important when using option "MultiSegmentMTU".

When using Flatstream communication with multi-segment MTUs, the next control byte is no longer expected in the first Rx byte of the subsequent MTU, but transferred directly after the current segment.

MessageEndBit

"MessageEndBit" is set if the subsequent segment completes a message. The message has then been completely transferred and is ready for further processing.

Information:

In the output direction, this bit must also be set if one individual segment is enough to hold the entire message. The module will only process a message internally if this identifier is detected.

The size of the message being transferred can be calculated by adding all of the message's segment lengths together.

Flatstream formula for calculating message length:

Message [bytes] = Segment lengths (all CBs without ME) + Segment length (of the first CB with ME)	CB	Control byte
	ME	MessageEndBit

3.3.4.2.3 Communication status

The communication status is determined via registers "OutputSequence" and "InputSequence".

- **OutputSequence** contains information about the communication status of the controller. It is written by the controller and read by the module.
- **InputSequence** contains information about the communication status of the module. It is written by the module and should only be read by the controller.

Relationship between OutputSequence and InputSequence

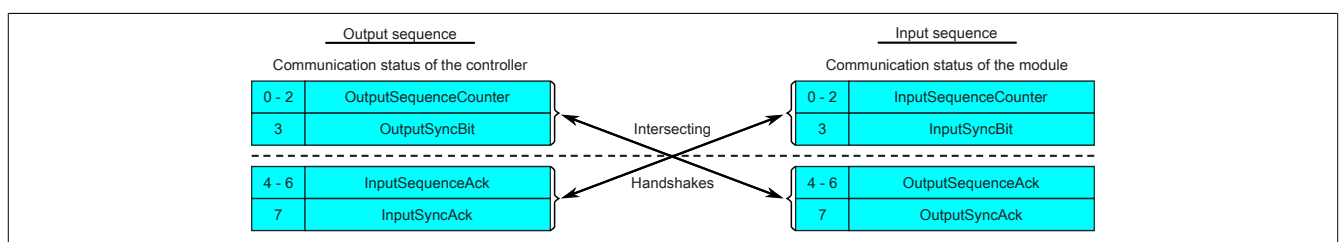


Figure 3: Relationship between OutputSequence and InputSequence

Registers "OutputSequence" and "InputSequence" are logically composed of 2 half-bytes. The low part indicates to the remote station whether a channel should be opened or whether data should be accepted. The high part is to acknowledge that the requested action was carried out.

SyncBit and SyncAck

If SyncBit and SyncAck are set in one communication direction, then the channel is considered "synchronized", i.e. it is possible to send messages in this direction. The status bit of the remote station must be checked cyclically. If SyncAck has been reset, then SyncBit on that station must be adjusted. Before new data can be transferred, the channel must be resynchronized.

SequenceCounter and SequenceAck

The communication partners cyclically check whether the low nibble on the remote station changes. When one of the communication partners finishes writing a new sequence to the MTU, it increments its SequenceCounter. The current sequence is then transmitted to the receiver, which acknowledges its receipt with SequenceAck. In this way, a "handshake" is initiated.

Information:

If communication is interrupted, segments from the unfinished message are discarded. All messages that were transferred completely are processed.

3.3.4.3 Synchronization

During synchronization, a communication channel is opened. It is important to make sure that a module is present and that the current value of SequenceCounter is stored on the station receiving the message.

Flatstream can handle full-duplex communication. This means that both channels / communication directions can be handled separately. They must be synchronized independently so that simplex communication can theoretically be carried out as well.

Synchronization in the output direction (controller as the transmitter):

The corresponding synchronization bits (OutputSyncBit and OutputSyncAck) are reset. Because of this, Flatstream cannot be used at this point in time to transfer messages from the controller to the module.

Algorithm

1) The controller must write 000 to OutputSequenceCounter and reset OutputSyncBit. The controller must cyclically query the high nibble of register "InputSequence" (checks for 000 in OutputSequenceAck and 0 in OutputSyncAck). <i>The module does not accept the current contents of InputMTU since the channel is not yet synchronized.</i> <i>The module matches OutputSequenceAck and OutputSyncAck to the values of OutputSequenceCounter and OutputSyncBit.</i>
2) If the controller registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The controller continues cyclically querying the high nibble of register "OutputSequence" (checks for 001 in OutputSequenceAck and 0 in InputSyncAck). <i>The module does not accept the current contents of InputMTU since the channel is not yet synchronized.</i> <i>The module matches OutputSequenceAck and OutputSyncAck to the values of OutputSequenceCounter and OutputSyncBit.</i>
3) If the controller registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The controller continues cyclically querying the high nibble of register "OutputSequence" (checks for 001 in OutputSequenceAck and 1 in InputSyncAck).
Note: Theoretically, data can be transferred from this point forward. However, it is still recommended to wait until the output direction is completely synchronized before transferring data.
<i>The module sets OutputSyncAck.</i>
The output direction is synchronized, and the controller can transmit data to the module.

Synchronization in the input direction (controller as the receiver):

The corresponding synchronization bits (InputSyncBit and InputSyncAck) are reset. Because of this, Flatstream cannot be used at this point in time to transfer messages from the module to the controller.

Algorithm

<i>The module writes 000 to InputSequenceCounter and resets InputSyncBit.</i> <i>The module monitors the high nibble of register "OutputSequence" and expects 000 in InputSequenceAck and 0 in InputSyncAck.</i>
1) The controller is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The controller has to match InputSequenceAck and InputSyncAck to the values of InputSequenceCounter and InputSyncBit. <i>If the module registers the expected values in InputSequenceAck and InputSyncAck, it increments InputSequenceCounter.</i> <i>The module monitors the high nibble of register "OutputSequence" and expects 001 in InputSequenceAck and 0 in InputSyncAck.</i>
2) The controller is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The controller has to match InputSequenceAck and InputSyncAck to the values of InputSequenceCounter and InputSyncBit. <i>If the module registers the expected values in InputSequenceAck and InputSyncAck, it sets InputSyncBit.</i> <i>The module monitors the high nibble of register "OutputSequence" and expects 1 in InputSyncAck.</i>
3) The controller is permitted to set InputSyncAck.
Note: Theoretically, data could already be transferred in this cycle. If InputSyncBit is set and InputSequenceCounter has been increased by 1, the values in the enabled Rx bytes must be accepted and acknowledged (see also "Communication in the input direction").
The input direction is synchronized, and the module can transmit data to the controller.

3.3.4.4 Transmitting and receiving

If a channel is synchronized, then the remote station is ready to receive messages from the transmitter. Before the transmitter can send data, it needs to first create a transmit array in order to meet Flatstream requirements.

The transmitting station must also generate a control byte for each segment created. This control byte contains information about how the subsequent part of the data being transferred should be processed. The position of the next control byte in the data stream can vary. For this reason, it must be clearly defined at all times when a new control byte is being transmitted. The first control byte is always in the first byte of the first sequence. All subsequent positions are determined recursively.

Flatstream formula for calculating the position of the next control byte:

$$\text{Position (of the next control byte)} = \text{Current position} + 1 + \text{Segment length}$$

Example

3 autonomous messages (7 bytes, 2 bytes and 9 bytes) are being transmitted using an MTU with a width of 7 bytes. The rest of the configuration corresponds to the default settings.

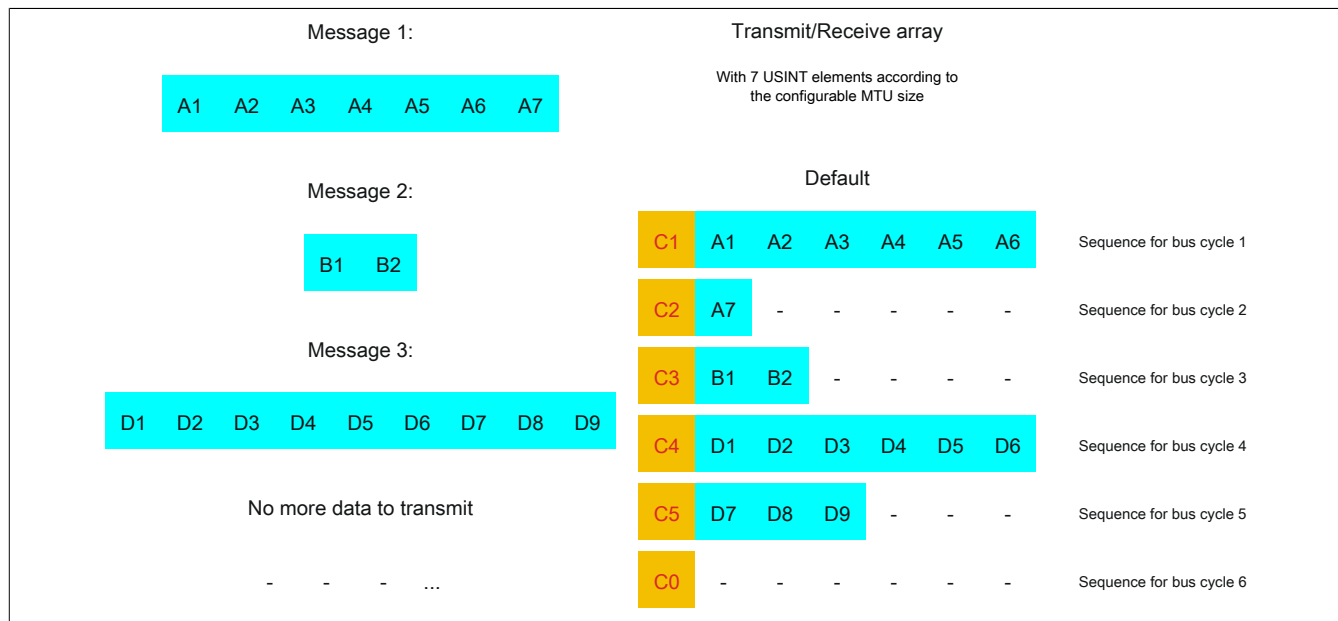


Figure 4: Transmit/Receive array (default)

First, the messages must be split into segments. In the default configuration, it is important to ensure that each sequence can hold an entire segment, including the associated control byte. The sequence is limited to the size of the enable MTU. In other words, a segment must be at least 1 byte smaller than the MTU.

MTU = 7 bytes → Max. segment length = 6 bytes

- Message 1 (7 bytes)
 - ⇒ First segment = Control byte + 6 bytes of data
 - ⇒ Second segment = Control byte + 1 data byte
- Message 2 (2 bytes)
 - ⇒ First segment = Control byte + 2 bytes of data
- Message 3 (9 bytes)
 - ⇒ First segment = Control byte + 6 bytes of data
 - ⇒ Second segment = Control byte + 3 data bytes
- No more messages
 - ⇒ C0 control byte

A unique control byte must be generated for each segment. In addition, the C0 control byte is generated to keep communication on standby.

C0 (control byte 0)			C1 (control byte 1)			C2 (control byte 2)		
- SegmentLength (0)	=	0	- SegmentLength (6)	=	6	- SegmentLength (1)	=	1
- nextCBPos (0)	=	0	- nextCBPos (0)	=	0	- nextCBPos (0)	=	0
- MessageEndBit (0)	=	0	- MessageEndBit (0)	=	0	- MessageEndBit (1)	=	128
Control byte	Σ	0	Control byte	Σ	6	Control byte	Σ	129

Table 5: Flatstream determination of the control bytes for the default configuration example (part 1)

C3 (control byte 3)			C4 (control byte 4)			C5 (control byte 5)		
- SegmentLength (2)	=	2	- SegmentLength (6)	=	6	- SegmentLength (3)	=	3
- nextCBPos (0)	=	0	- nextCBPos (0)	=	0	- nextCBPos (0)	=	0
- MessageEndBit (1)	=	128	- MessageEndBit (0)	=	0	- MessageEndBit (1)	=	128
Control byte	Σ	130	Control byte	Σ	6	Control byte	Σ	131

Table 6: Flatstream determination of the control bytes for the default configuration example (part 2)

3.3.4.4.1 Transmitting data to a module (output)

When transmitting data, the transmit array must be generated in the application program. Sequences are then transferred one by one using Flatstream and received by the module.

Information:

Although all B&R modules with Flatstream communication always support the most compact transfers in the output direction, it is recommended to use the same design for the transfer arrays in both communication directions.

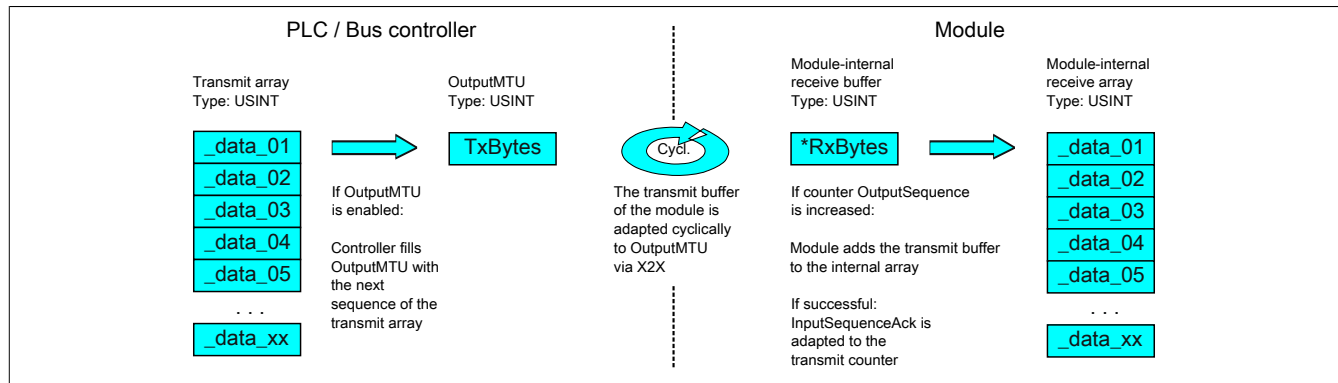


Figure 5: Flatstream communication (output)

Message smaller than OutputMTU

The length of the message is initially smaller than OutputMTU. In this case, one sequence would be sufficient to transfer the entire message and the necessary control byte.

Algorithm

<p><i>Cyclic status query:</i></p> <ul style="list-style-type: none"> - The module monitors OutputSequenceCounter.
<p>0) Cyclic checks:</p> <ul style="list-style-type: none"> - The controller must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The controller must check whether OutputMTU is enabled. → If OutputSequenceCounter > InputSequenceAck: MTU is not enabled because the last sequence has not yet been acknowledged.
<p>1) Preparation (create transmit array):</p> <ul style="list-style-type: none"> - The controller must split up the message into valid segments and create the necessary control bytes. - The controller must add the segments and control bytes to the transmit array.
<p>2) Transmit:</p> <ul style="list-style-type: none"> - The controller transfers the current element of the transmit array to OutputMTU. → OutputMTU is transferred cyclically to the module's transmit buffer but not processed further. - The controller must increase OutputSequenceCounter.
<p><i>Reaction:</i></p> <ul style="list-style-type: none"> - The module accepts the bytes from the internal receive buffer and adds them to the internal receive array. - The module transmits acknowledgment and writes the value of OutputSequenceCounter to OutputSequenceAck.
<p>3) Completion:</p> <ul style="list-style-type: none"> - The controller must monitor OutputSequenceAck. → A sequence is only considered to have been transferred successfully if it has been acknowledged via OutputSequenceAck. In order to detect potential transfer errors in the last sequence as well, it is important to make sure that the length of the <i>Completion</i> phase is run through long enough.
<p>Note:</p> <p>To monitor communication times exactly, the task cycles that have passed since the last increase of OutputSequenceCounter should be counted. In this way, the number of previous bus cycles necessary for the transfer can be measured. If the monitoring counter exceeds a predefined threshold, then the sequence can be considered lost.</p> <p>(The relationship of bus to task cycle can be influenced by the user so that the threshold value must be determined individually.)</p> <ul style="list-style-type: none"> - Subsequent sequences are only permitted to be transmitted in the next bus cycle after the completion check has been carried out successfully.

Message larger than OutputMTU

The transmit array, which must be created in the program sequence, consists of several elements. The user has to arrange the control and data bytes correctly and transfer the array elements one after the other. The transfer algorithm remains the same and is repeated starting at the point *Cyclic checks*.

General flowchart

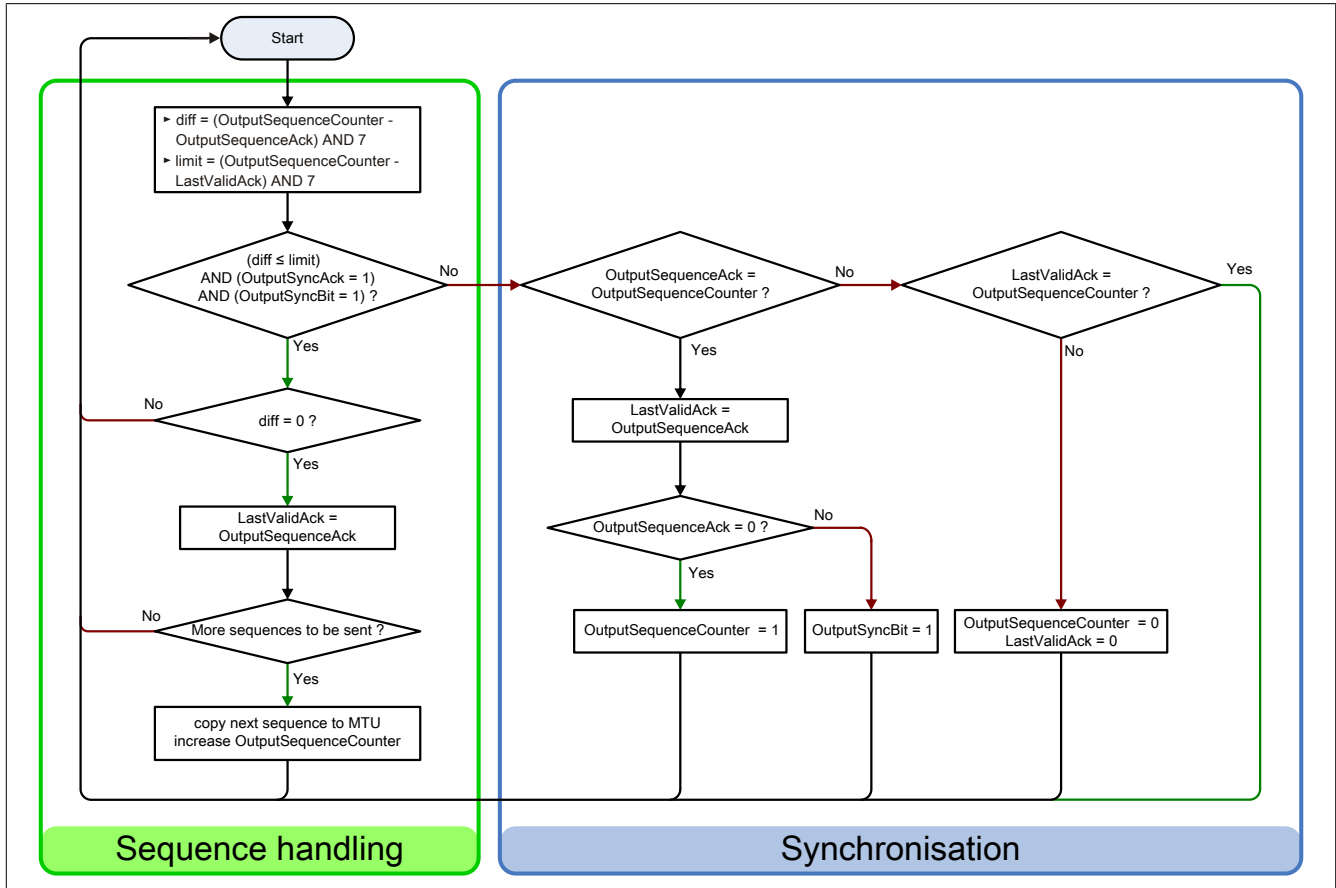


Figure 6: Flowchart for the output direction

3.3.4.4.2 Receiving data from a module (input)

When receiving data, the transmit array is generated by the module, transferred via Flatstream and must then be reproduced in the receive array. The structure of the incoming data stream can be set with the mode register. The algorithm for receiving the data remains unchanged in this regard.

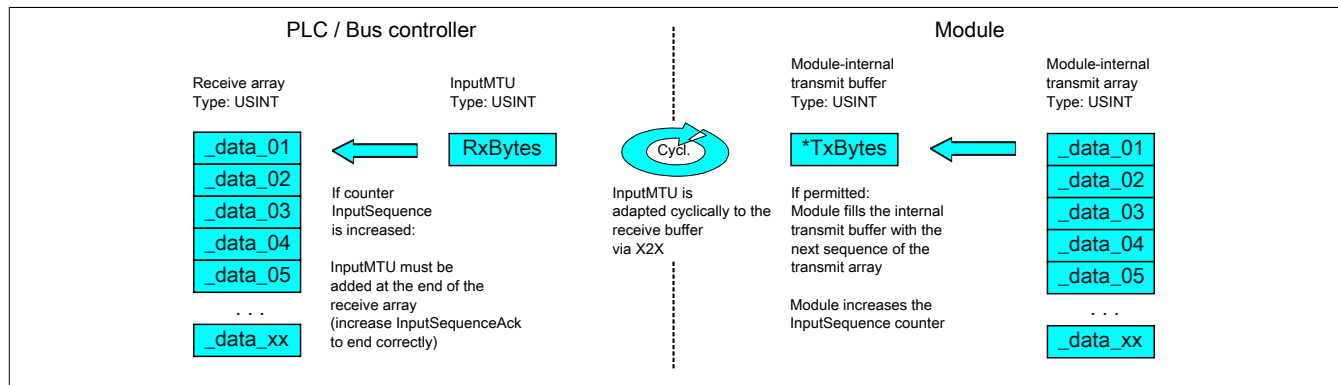


Figure 7: Flatstream communication (input)

Algorithm

0) Cyclic status query: - The controller must monitor InputSequenceCounter.
Cyclic checks: - The module checks InputSyncAck. - The module checks InputSequenceAck.
Preparation: - The module forms the segments and control bytes and creates the transmit array.
Action: - The module transfers the current element of the internal transmit array to the internal transmit buffer. - The module increases InputSequenceCounter.
1) Receiving (as soon as InputSequenceCounter is increased): - The controller must apply data from InputMTU and append it to the end of the receive array. - The controller must match InputSequenceAck to InputSequenceCounter of the sequence currently being processed.
Completion: - The module monitors InputSequenceAck. → A sequence is only considered to have been transferred successfully if it has been acknowledged via InputSequenceAck. - Subsequent sequences are only transmitted in the next bus cycle after the completion check has been carried out successfully.

General flowchart

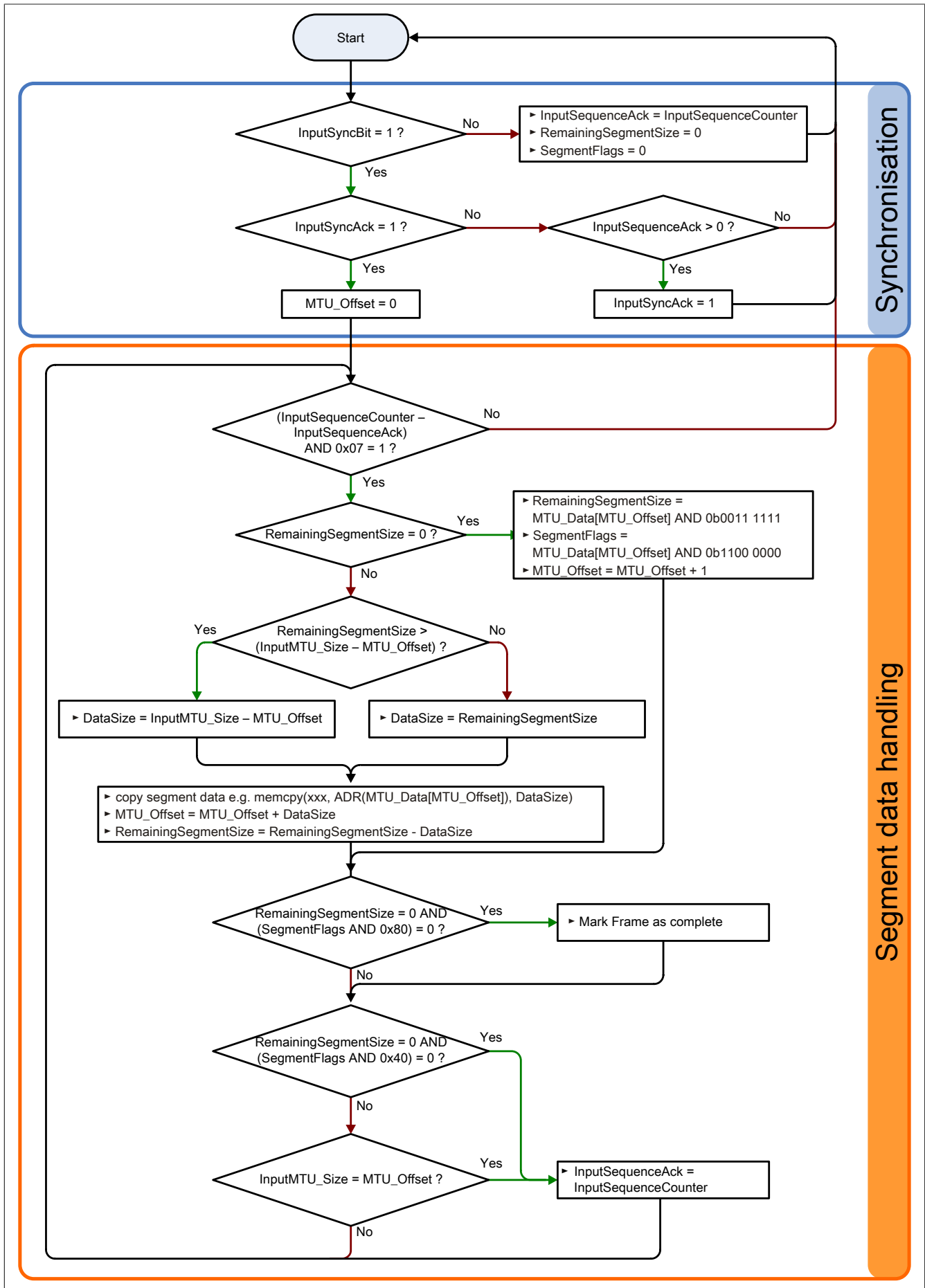


Figure 8: Flowchart for the input direction

3.3.4.4.3 Details

It is recommended to store transferred messages in separate receive arrays.

After a set MessageEndBit is transmitted, the subsequent segment should be added to the receive array. The message is then complete and can be passed on internally for further processing. A new/separate array should be created for the next message.

Information:

When transferring with MultiSegmentMTUs, it is possible for several small messages to be part of one sequence. In the program, it is important to make sure that a sufficient number of receive arrays can be managed. The acknowledge register is only permitted to be adjusted after the entire sequence has been applied.

If SequenceCounter is incremented by more than one counter, an error is present.

In this case, the receiver stops. All additional incoming sequences are ignored until the transmission with the correct SequenceCounter is retried. This response prevents the transmitter from receiving any more acknowledgments for transmitted sequences. The transmitter can identify the last successfully transferred sequence from the remote station's SequenceAck and continue the transfer from this point.

Information:

This situation is very unlikely when operating without "Forward" functionality.

Acknowledgments must be checked for validity.

If the receiver has successfully accepted a sequence, it must be acknowledged. The receiver takes on the value of SequenceCounter sent along with the transmission and matches SequenceAck to it. The transmitter reads SequenceAck and registers the successful transmission. If the transmitter acknowledges a sequence that has not yet been dispatched, then the transfer must be interrupted and the channel resynchronized. The synchronization bits are reset and the current/incomplete message is discarded. It must be sent again after the channel has been resynchronized.

3.3.4.5 Flatstream mode

In the input direction, the transmit array is generated automatically. Flatstream mode offers several options to the user that allow an incoming data stream to have a more compact arrangement. These include:

- Standard
- MultiSegmentMTUs allowed
- Large segments allowed:

Once enabled, the program code for evaluation must be adapted accordingly.

Information:

All B&R modules that offer Flatstream mode support options "Large segments" and "MultiSegmentMTUs" in the output direction. Compact transfer must be explicitly allowed only in the input direction.

Standard

By default, both options relating to compact transfer in the input direction are disabled.

1. The module only forms segments that are at least one byte smaller than the enabled MTU. Each sequence begins with a control byte so that the data stream is clearly structured and relatively easy to evaluate.
2. Since a Flatstream message is permitted to be any length, the last segment of the message frequently does not fill up all of the MTU's space. By default, the remaining bytes during this type of transfer cycle are not used.

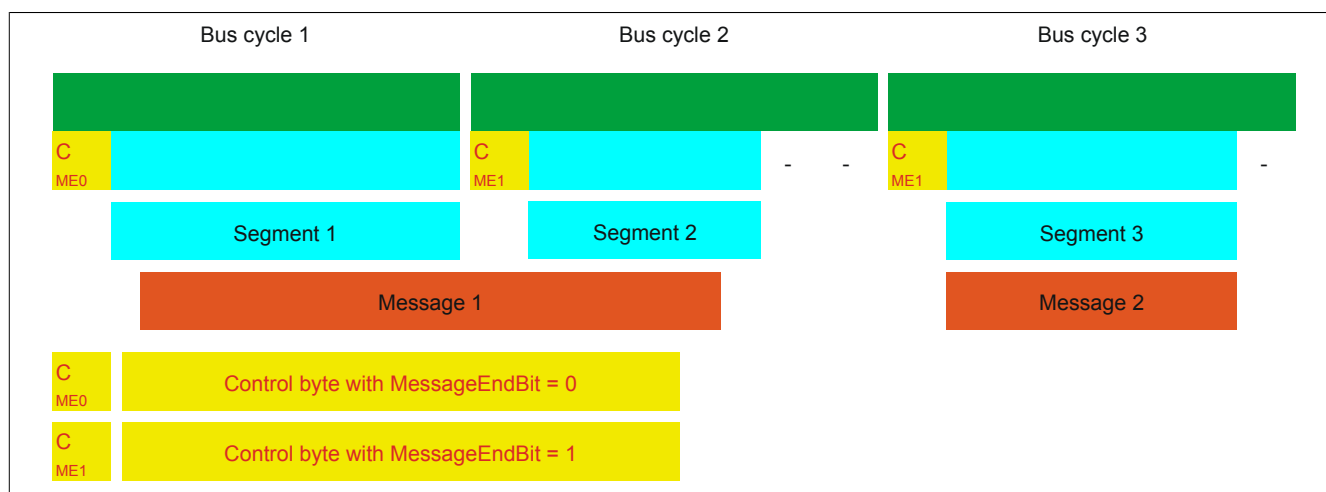


Figure 9: Message arrangement in the MTU (default)

MultiSegmentMTUs allowed

With this option, InputMTU is completely filled (if enough data is pending). The previously unfilled Rx bytes transfer the next control bytes and their segments. This allows the enabled Rx bytes to be used more efficiently.

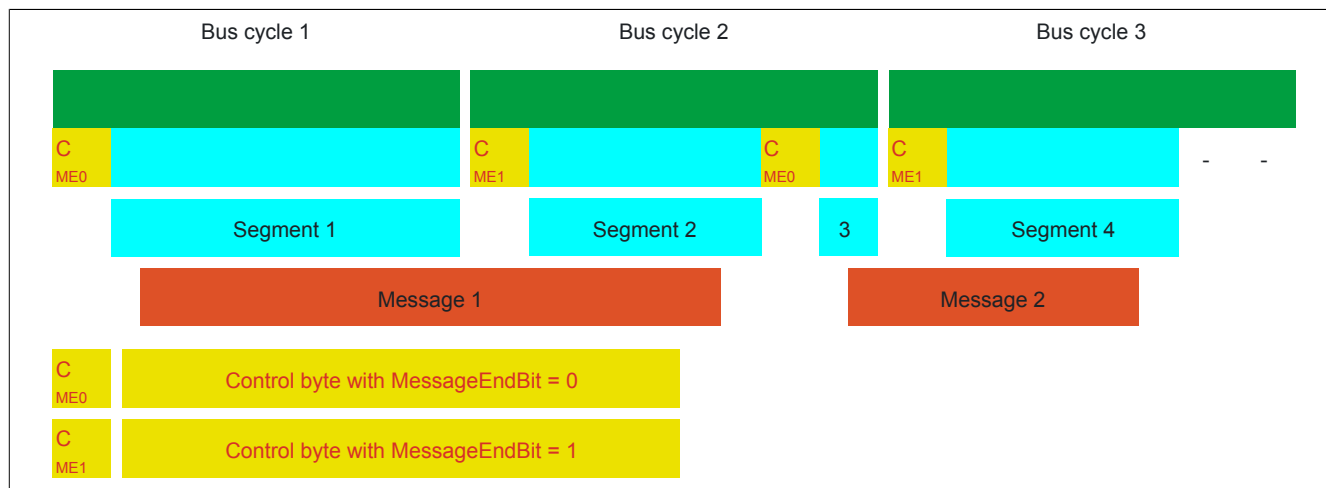


Figure 10: Arrangement of messages in the MTU (MultiSegmentMTUs)

Large segments allowed:

When transferring very long messages or when enabling only very few Rx bytes, then a great many segments must be created by default. The bus system is more stressed than necessary since an additional control byte must be created and transferred for each segment. With option "Large segments", the segment length is limited to 63 bytes independently of InputMTU. One segment is permitted to stretch across several sequences, i.e. it is possible for "pure" sequences to occur without a control byte.

Information:

It is still possible to split up a message into several segments, however. If this option is used and messages with more than 63 bytes occur, for example, then messages can still be split up among several segments.

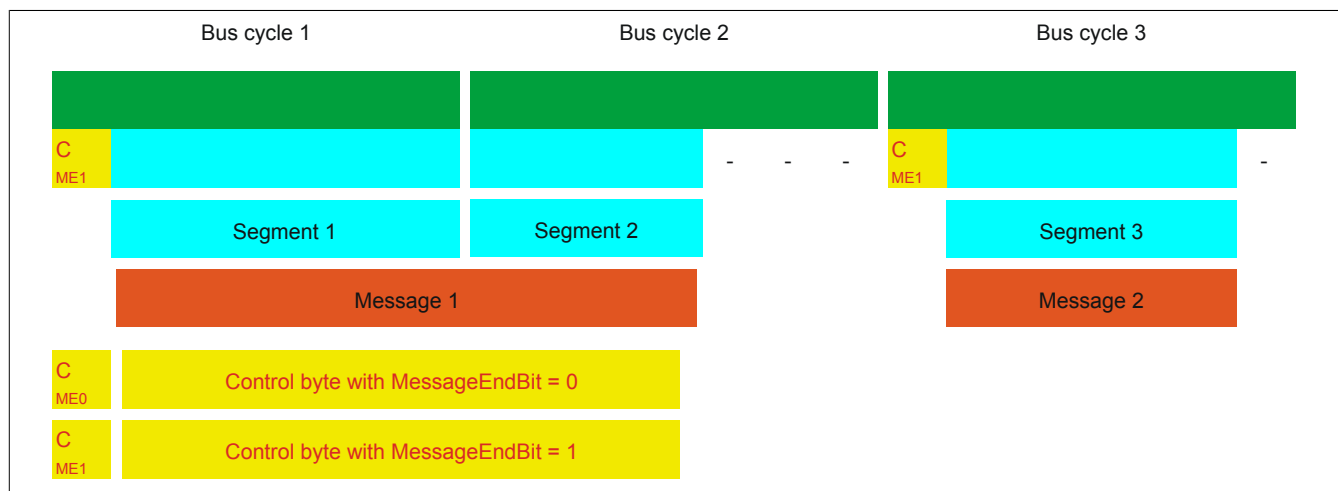


Figure 11: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

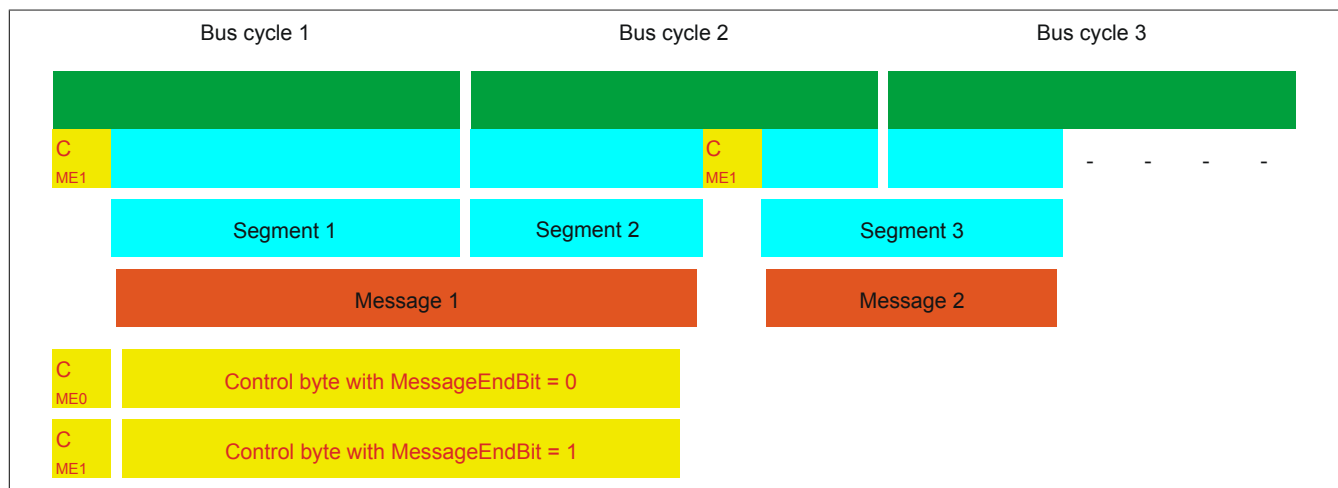


Figure 12: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

3.3.4.6 Adjusting the Flatstream

If the way messages are structured is changed, then the way data in the transmit/receive array is arranged is also different. The following changes apply to the example given earlier.

MultiSegmentMTU

If MultiSegmentMTUs are allowed, then "open positions" in an MTU can be used. These "open positions" occur if the last segment in a message does not fully use the entire MTU. MultiSegmentMTUs allow these bits to be used to transfer the subsequent control bytes and segments. In the program sequence, the "nextCBPos" bit in the control byte is set so that the receiver can correctly identify the next control byte.

Example

3 autonomous messages (7 bytes, 2 bytes and 9 bytes) are being transmitted using an MTU with a width of 7 bytes. The configuration allows the transfer of MultiSegmentMTUs.

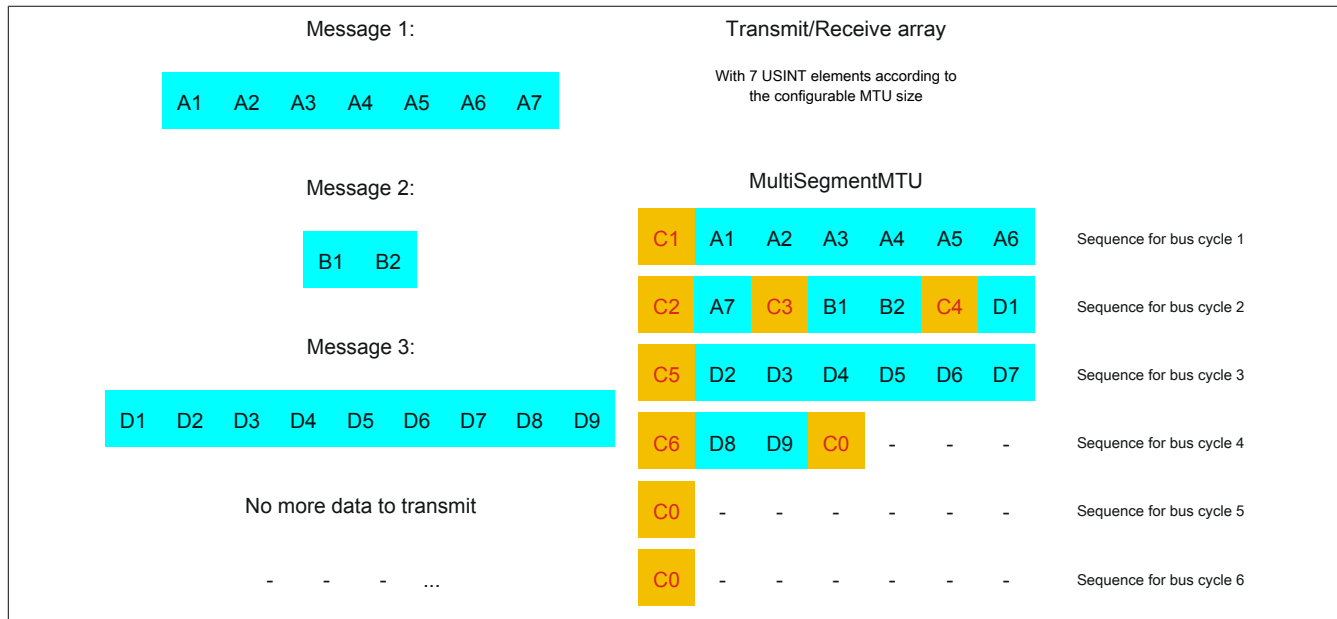


Figure 13: Transmit/receive array (MultiSegmentMTUs)

First, the messages must be split into segments. As in the default configuration, it is important for each sequence to begin with a control byte. The free bits in the MTU at the end of a message are filled with data from the following message, however. With this option, the "nextCBPos" bit is always set if payload data is transferred after the control byte.

MTU = 7 bytes → Max. segment length = 6 bytes

- Message 1 (7 bytes)
 - ⇒ First segment = Control byte + 6 bytes of data (MTU full)
 - ⇒ Second segment = Control byte + 1 byte of data (MTU still has 5 open bytes)
- Message 2 (2 bytes)
 - ⇒ First segment = Control byte + 2 bytes of data (MTU still has 2 open bytes)
- Message 3 (9 bytes)
 - ⇒ First segment = Control byte + 1 byte of data (MTU full)
 - ⇒ Second segment = Control byte + 6 bytes of data (MTU full)
 - ⇒ Third segment = Control byte + 2 bytes of data (MTU still has 4 open bytes)
- No more messages
 - ⇒ C0 control byte

A unique control byte must be generated for each segment. In addition, the C0 control byte is generated to keep communication on standby.

C1 (control byte 1)			C2 (control byte 2)			C3 (control byte 3)		
- SegmentLength (6)	=	6	- SegmentLength (1)	=	1	- SegmentLength (2)	=	2
- nextCBPos (1)	=	64	- nextCBPos (1)	=	64	- nextCBPos (1)	=	64
- MessageEndBit (0)	=	0	- MessageEndBit (1)	=	128	- MessageEndBit (1)	=	128
Control byte	Σ	70	Control byte	Σ	193	Control byte	Σ	194

Table 7: Flatstream determination of the control bytes for the MultiSegmentMTU example (part 1)

Warning!

The second sequence is only permitted to be acknowledged via SequenceAck if it has been completely processed. In this example, there are 3 different segments within the second sequence, i.e. the program must include enough receive arrays to handle this situation.

C4 (control byte 4)			C5 (control byte 5)			C6 (control byte 6)		
- SegmentLength (1)	=	1	- SegmentLength (6)	=	6	- SegmentLength (2)	=	2
- nextCBPos (6)	=	6	- nextCBPos (1)	=	64	- nextCBPos (1)	=	64
- MessageEndBit (0)	=	0	- MessageEndBit (1)	=	0	- MessageEndBit (1)	=	128
Control byte	Σ	7	Control byte	Σ	70	Control byte	Σ	194

Table 8: Flatstream determination of the control bytes for the MultiSegmentMTU example (part 2)

Large segments

Segments are limited to a maximum of 63 bytes. This means they can be larger than the active MTU. These large segments are divided among several sequences when transferred. It is possible for sequences to be completely filled with payload data and not have a control byte.

Information:

It is still possible to subdivide a message into several segments so that the size of a data packet does not also have to be limited to 63 bytes.

Example

3 autonomous messages (7 bytes, 2 bytes and 9 bytes) are being transmitted using an MTU with a width of 7 bytes. The configuration allows the transfer of large segments.

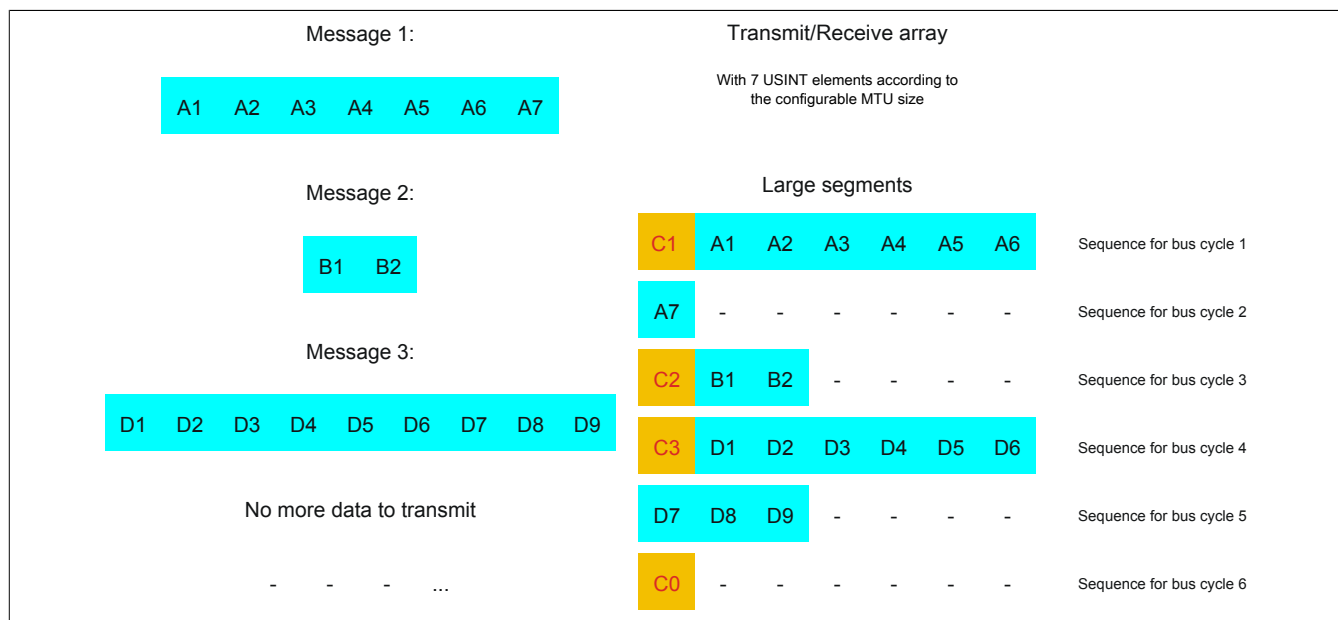


Figure 14: Transmit/receive array (large segments)

First, the messages must be split into segments. The ability to form large segments means that messages are split up less frequently, which results in fewer control bytes generated.

Large segments allowed → Max. segment length = 63 bytes

- Message 1 (7 bytes)
 - ⇒ First segment = Control byte + 7 bytes of data
- Message 2 (2 bytes)
 - ⇒ First segment = Control byte + 2 bytes of data
- Message 3 (9 bytes)
 - ⇒ First segment = Control byte + 9 bytes of data
- No more messages
 - ⇒ C0 control byte

A unique control byte must be generated for each segment. In addition, the C0 control byte is generated to keep communication on standby.

C1 (control byte 1)			C2 (control byte 2)			C3 (control byte 3)		
- SegmentLength (7)	=	7	- SegmentLength (2)	=	2	- SegmentLength (9)	=	9
- nextCBPos (0)	=	0	- nextCBPos (0)	=	0	- nextCBPos (0)	=	0
- MessageEndBit (1)	=	128	- MessageEndBit (1)	=	128	- MessageEndBit (1)	=	128
Control byte	Σ	135	Control byte	Σ	130	Control byte	Σ	137

Table 9: Flatstream determination of the control bytes for the large segment example

Large segments and MultiSegmentMTU

Example

3 autonomous messages (7 bytes, 2 bytes and 9 bytes) are being transmitted using an MTU with a width of 7 bytes. The configuration allows transfer of large segments as well as MultiSegmentMTUs.

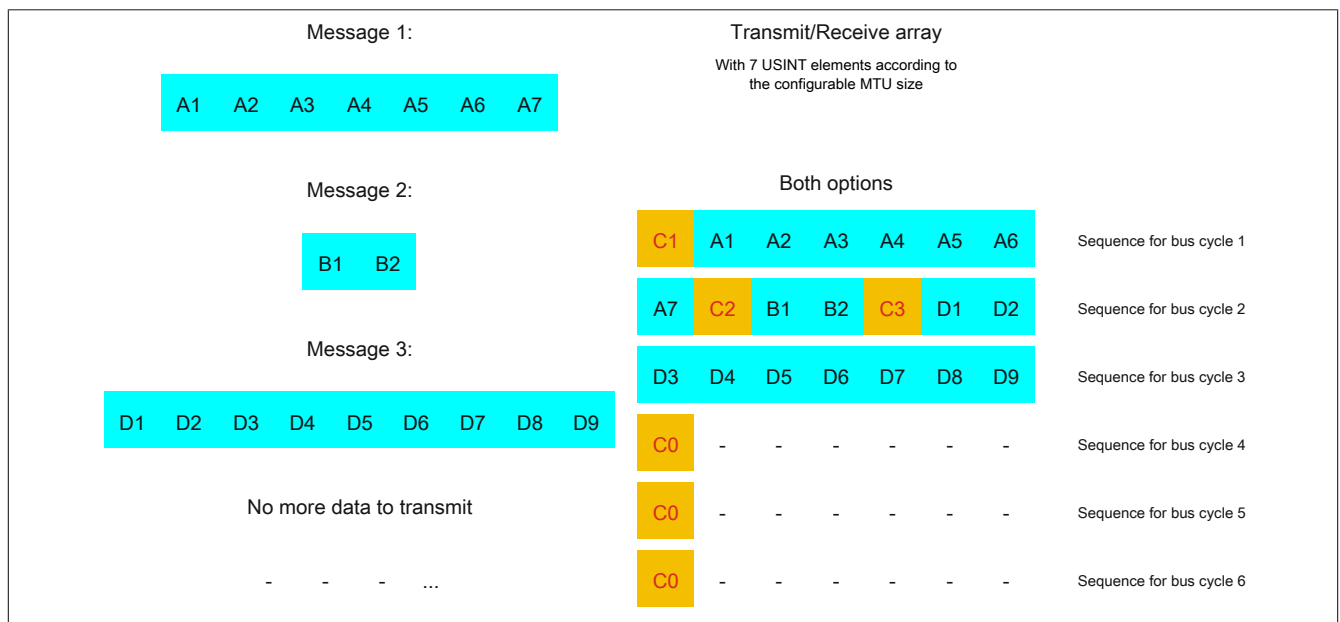


Figure 15: Transmit/receive array (large segments and MultiSegmentMTUs)

First, the messages must be split into segments. If the last segment of a message does not completely fill the MTU, it is permitted to be used for other data in the data stream. Bit "nextCBPos" must always be set if the control byte belongs to a segment with payload data.

The ability to form large segments means that messages are split up less frequently, which results in fewer control bytes generated. Control bytes are generated in the same way as with option "Large segments".

Large segments allowed → Max. segment length = 63 bytes

- Message 1 (7 bytes)
 - ⇒ First segment = Control byte + 7 bytes of data
- Message 2 (2 bytes)
 - ⇒ First segment = Control byte + 2 bytes of data
- Message 3 (9 bytes)
 - ⇒ First segment = Control byte + 9 bytes of data
- No more messages
 - ⇒ C0 control byte

A unique control byte must be generated for each segment. In addition, the C0 control byte is generated to keep communication on standby.

C1 (control byte 1)			C2 (control byte 2)			C3 (control byte 3)		
- SegmentLength (7)	=	7	- SegmentLength (2)	=	2	- SegmentLength (9)	=	9
- nextCBPos (0)	=	0	- nextCBPos (0)	=	0	- nextCBPos (0)	=	0
- MessageEndBit (1)	=	128	- MessageEndBit (1)	=	128	- MessageEndBit (1)	=	128
Control byte	Σ	135	Control byte	Σ	130	Control byte	Σ	137

Table 10: Flatstream determination of the control bytes for the large segment and MultiSegmentMTU example

3.3.5 Example of function "Forward" with X2X Link

Function "Forward" is a method that can be used to substantially increase the Flatstream data rate. The basic principle is also used in other technical areas such as "pipelining" for microprocessors.

3.3.5.1 Function principle

X2X Link communication cycles through 5 different steps to transfer a Flatstream sequence. At least 5 bus cycles are therefore required to successfully transfer the sequence.

	Step I	Step II	Step III	Step IV	Step V
Actions	Transfer sequence from transmit array, increase SequenceCounter	Cyclic synchronization of MTU and module buffer	Append sequence to receive array, adjust SequenceAck	Cyclic synchronization MTU and module buffer	Check SequenceAck
Resource	Transmitter (task to transmit)	Bus system (direction 1)	Recipients (task to receive)	Bus system (direction 2)	Transmitter (task for Ack checking)

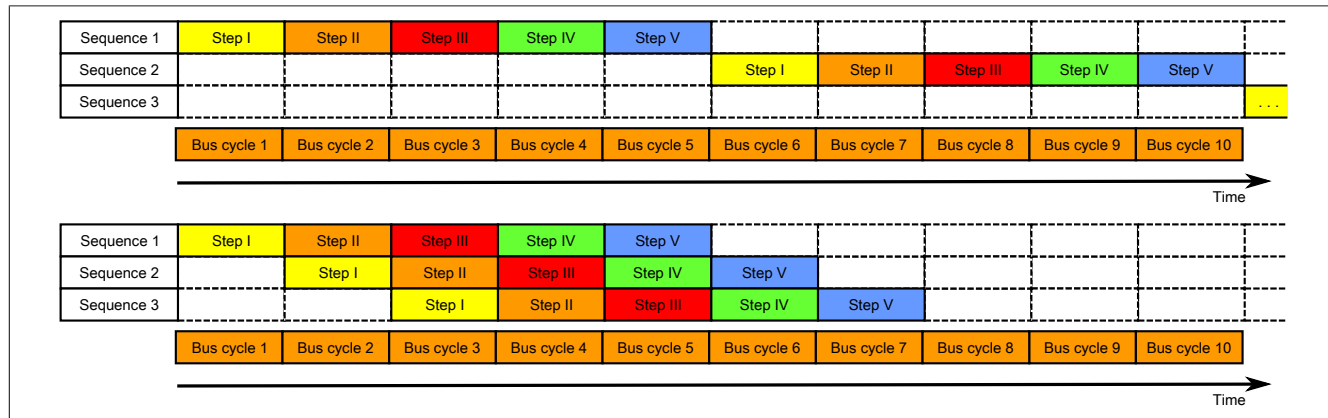


Figure 16: Comparison of transfer without/with Forward

Each of the 5 steps (tasks) requires different resources. If Forward functionality is not used, the sequences are executed one after the other. Each resource is then only active if it is needed for the current sub-action.

With Forward, a resource that has executed its task can already be used for the next message. The condition for enabling the MTU is changed to allow for this. Sequences are then passed to the MTU according to the timing. The transmitting station no longer waits for an acknowledgment from SequenceAck, which means that the available bandwidth can be used much more efficiently.

In the most ideal situation, all resources are working during each bus cycle. The receiver still has to acknowledge every sequence received. Only when SequenceAck has been changed and checked by the transmitter is the sequence considered as having been transferred successfully.

3.3.5.2 Configuration

The Forward function must only be enabled for the input direction. Flatstream modules have been optimized in such a way that they support this function. In the output direction, the Forward function can be used as soon as the size of OutputMTU is specified.

Information:

The registers are described in ["Flatstream registers" on page 48](#).

3.3.5.2.1 Delay time

The delay time is specified in microseconds. This is the amount of time the module has to wait after sending a sequence until it is permitted to write new data to the MTU in the following bus cycle. The program routine for receiving sequences from a module can therefore be run in a task class whose cycle time is slower than the bus cycle.

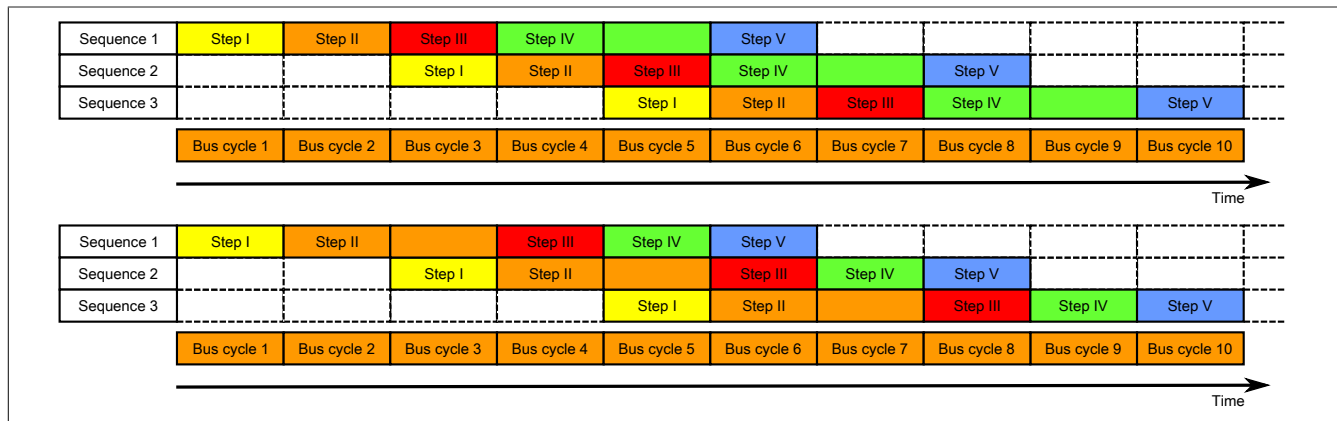


Figure 17: Effect of ForwardDelay when using Flatstream communication with the Forward function

In the program, it is important to make sure that the controller is processing all of the incoming InputSequences and InputMTUs. The ForwardDelay value causes delayed acknowledgment in the output direction and delayed reception in the input direction. In this way, the controller has more time to process the incoming InputSequence or InputMTU.

3.3.5.3 Transmitting and receiving with Forward

The basic algorithm for transmitting and receiving data remains the same. With the Forward function, up to 7 unacknowledged sequences can be transmitted. Sequences can be transmitted without having to wait for the previous message to be acknowledged. Since the delay between writing and response is eliminated, a considerable amount of additional data can be transferred in the same time window.

Algorithm for transmitting

<p><i>Cyclic status query:</i></p> <ul style="list-style-type: none"> - The module monitors <i>OutputSequenceCounter</i>.
<p>0) Cyclic checks:</p> <ul style="list-style-type: none"> - The controller must check <i>OutputSyncAck</i>. → If <i>OutputSyncAck</i> = 0: Reset <i>OutputSyncBit</i> and resynchronize the channel. - The controller must check whether <i>OutputMTU</i> is enabled. → If <i>OutputSequenceCounter</i> > <i>OutputSequenceAck</i> + 7, then it is not enabled because the last sequence has not yet been acknowledged.
<p>1) Preparation (create transmit array):</p> <ul style="list-style-type: none"> - The controller must split up the message into valid segments and create the necessary control bytes. - The controller must add the segments and control bytes to the transmit array.
<p>2) Transmit:</p> <ul style="list-style-type: none"> - The controller must transfer the current part of the transmit array to <i>OutputMTU</i>. - The controller must increase <i>OutputSequenceCounter</i> for the sequence to be accepted by the module. - The controller is then permitted to <i>transmit</i> in the next bus cycle if the MTU has been enabled.
<p><i>The module responds since $OutputSequenceCounter > OutputSequenceAck$:</i></p> <ul style="list-style-type: none"> - The module accepts data from the internal receive buffer and appends it to the end of the internal receive array. - The module is acknowledged and the currently received value of <i>OutputSequenceCounter</i> is transferred to <i>OutputSequenceAck</i>. - The module queries the status cyclically again.
<p>3) Completion (acknowledgment):</p> <ul style="list-style-type: none"> - The controller must check <i>OutputSequenceAck</i> cyclically. → A sequence is only considered to have been transferred successfully if it has been acknowledged via <i>OutputSequenceAck</i>. In order to detect potential transfer errors in the last sequence as well, it is important to make sure that the algorithm is run through long enough. <p>Note:</p> <p>To monitor communication times exactly, the task cycles that have passed since the last increase of <i>OutputSequenceCounter</i> should be counted. In this way, the number of previous bus cycles necessary for the transfer can be measured. If the monitoring counter exceeds a predefined threshold, then the sequence can be considered lost (the relationship of bus to task cycle can be influenced by the user so that the threshold value must be determined individually).</p>

Algorithm for receiving

<p>0) Cyclic status query:</p> <ul style="list-style-type: none"> - The controller must monitor <i>InputSequenceCounter</i>.
<p><i>Cyclic checks:</i></p> <ul style="list-style-type: none"> - The module checks <i>InputSyncAck</i>. - The module checks if <i>InputMTU</i> for enabling. → Enabling criteria: <i>InputSequenceCounter</i> > <i>InputSequenceAck</i> + Forward
<p><i>Preparation:</i></p> <ul style="list-style-type: none"> - The module forms the control bytes / segments and creates the transmit array.
<p><i>Action:</i></p> <ul style="list-style-type: none"> - The module transfers the current part of the transmit array to the receive buffer. - The module increases <i>InputSequenceCounter</i>. - The module waits for a new bus cycle after time from <i>ForwardDelay</i> has expired. - The module repeats the action if <i>InputMTU</i> is enabled.
<p>1) Receiving (<i>InputSequenceCounter</i> > <i>InputSequenceAck</i>):</p> <ul style="list-style-type: none"> - The controller must apply data from <i>InputMTU</i> and append it to the end of the receive array. - The controller must match <i>InputSequenceAck</i> to <i>InputSequenceCounter</i> of the sequence currently being processed.
<p><i>Completion:</i></p> <ul style="list-style-type: none"> - The module monitors <i>InputSequenceAck</i>. → A sequence is only considered to have been transferred successfully if it has been acknowledged via <i>InputSequenceAck</i>.

Details/Background

1. Illegal SequenceCounter size (counter offset)

Error situation: MTU not enabled

If the difference between SequenceCounter and SequenceAck during transmission is larger than permitted, a transfer error occurs. In this case, all unacknowledged sequences must be repeated with the old SequenceCounter value.

2. Checking an acknowledgment

After an acknowledgment has been received, a check must verify whether the acknowledged sequence has been transmitted and had not yet been unacknowledged. If a sequence is acknowledged multiple times, a severe error occurs. The channel must be closed and resynchronized (same behavior as when not using Forward).

Information:

In exceptional cases, the module can increment OutputSequenceAck by more than 1 when using Forward.

An error does not occur in this case. The controller is permitted to consider all sequences up to the one being acknowledged as having been transferred successfully.

3. Transmit and receive arrays

The Forward function has no effect on the structure of the transmit and receive arrays. They are created and must be evaluated in the same way.

3.3.5.4 Errors when using Forward

In industrial environments, it is often the case that many different devices from various manufacturers are being used side by side. The electrical and/or electromagnetic properties of these technical devices can sometimes cause them to interfere with one another. These kinds of situations can be reproduced and protected against in laboratory conditions only to a certain point.

Precautions have been taken for transfer via X2X Link in case such interference should occur. For example, if an invalid checksum occurs, the I/O system will ignore the data from this bus cycle and the receiver receives the last valid data once more. With conventional (cyclic) data points, this error can often be ignored. In the following cycle, the same data point is again retrieved, adjusted and transferred.

Using Forward functionality with Flatstream communication makes this situation more complex. The receiver receives the old data again in this situation as well, i.e. the previous values for SequenceAck/SequenceCounter and the old MTU.

Loss of acknowledgment (SequenceAck)

If a SequenceAck value is lost, then the MTU was already transferred properly. For this reason, the receiver is permitted to continue processing with the next sequence. The SequenceAck is aligned with the associated SequenceCounter and sent back to the transmitter. Checking the incoming acknowledgments shows that all sequences up to the last one acknowledged have been transferred successfully (see sequences 1 and 2 in the image).

Loss of transmission (SequenceCounter, MTU):

If a bus cycle drops out and causes the value of SequenceCounter and/or the filled MTU to be lost, then no data reaches the receiver. At this point, the transmission routine is not yet affected by the error. The time-controlled MTU is released again and can be rewritten to.

The receiver receives SequenceCounter values that have been incremented several times. For the receive array to be put together correctly, the receiver is only permitted to process transmissions whose SequenceCounter has been increased by one. The incoming sequences must be ignored, i.e. the receiver stops and no longer transmits back any acknowledgments.

If the maximum number of unacknowledged sequences has been sent and no acknowledgments are returned, the transmitter must repeat the affected SequenceCounter and associated MTUs (see sequence 3 and 4 in the image).

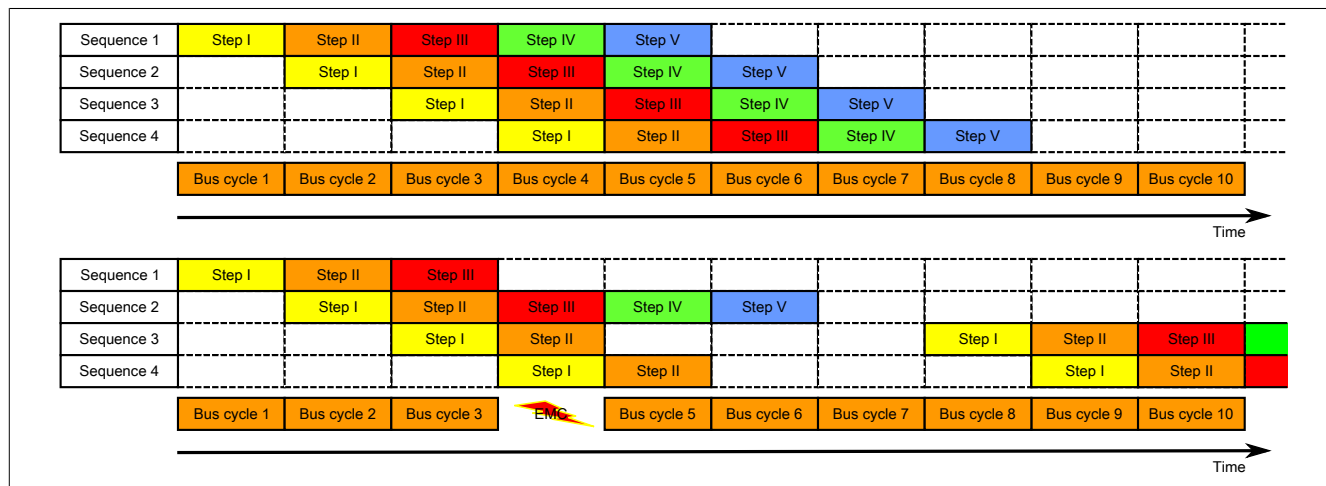


Figure 18: Effect of a lost bus cycle

Loss of acknowledgment

In sequence 1, the acknowledgment is lost due to disturbance. Sequences 1 and 2 are therefore acknowledged in Step V of sequence 2.

Loss of transmission

In sequence 3, the entire transmission is lost due to disturbance. The receiver stops and no longer sends back any acknowledgments.

The transmitting station continues transmitting until it has issued the maximum permissible number of unacknowledged transmissions.

5 bus cycles later at the earliest (depending on the configuration), it begins resending the unsuccessfully sent transmissions.

4 Commissioning

4.1 System requirements

The following minimum versions are recommended to generally be able to use all functions:

- Automation Studio 4.3.4.95
- Automation Runtime A4.26

4.2 Line length and sample rate

Determining the line length

The possible line length depends partly on the CAN nodes used and sampling point. In addition, the processing time of the signals through the CAN hub must also be taken into account. This is a maximum of 250 ns. To calculate the maximum possible transfer rate, the length of the longest CAN branch must be used for each CAN interface. Shortening one CAN branch has no influence on the maximum length of another CAN branch.

Setting the sampling point

The set sampling point corresponds to the sampling point of the CAN hub as well as the connected sensors. Therefore, only sensors whose sampling point is after the sampling point of the CAN hub are permitted to be connected.

The sampling point itself is determined by the synchronization jump width (ConfigSJW) and the sampling instant offset (ConfigSPO).

ConfigSJW	ConfigSPO	Sampling point	Information
1	0	93.75%	
2	0	87.50%	
3	0	81.25%	Bus controller default setting
4	0	75.00%	
1	1	87.50%	
2	1	81.25%	
3	1	75.00%	
4	1	68.75%	

Information:

The registers are described in "[Synchronization jump width](#)" on page 41 and "[Offset for the sampling point](#)" on page 41.

5 Register description

5.1 General data points

In addition to the registers described in the register description, the module has additional general data points. These are not module-specific but contain general information such as serial number and hardware variant.

General data points are described in section "Additional information - General data points" in the X67 system user's manual.

5.2 Function model 0 - Flat

In function model "Flat", CAN information is transferred via cyclic input and output registers. All data for a CAN object (8 CAN data bytes, identifier, status, etc.) is accessible as individual data points (see also ["The CAN object" on page 10](#)).

Information:

- Libraries "ArCAN" and "CAN_Lib" cannot be used.

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Interface - Configuration						
257	ConfigBaudrate	USINT				•
259	ConfigSJW	USINT				•
261	ConfigSPO	USINT				•
266	ConfigTXtrigger	UINT				•
673	Cfo_FIFOTXlimit	USINT				•
677	Cfo_TXRXinfoFlags	USINT				•
Interface - Communication						
641	TXCount	USINT			•	
513	TXCountReadBack	USINT	•			
545	TXCountLatchReadBack	USINT	•			
515	RXCount	USINT	•			
547	RXCountLatch	USINT	•			
Transmit buffer						
645	TXDataSize	USINT			•	
652	TXIdent	UDINT			•	
Index * 2 + 657	TXDataByte0 to TXDataByte7	USINT			•	
Index * 4 + 658	TXDataWord0 to TXDataWord3	UINT			•	
Index * 8 + 660	TXDataLong0 to TXDataLong1	UDINT			•	
Receive buffer 0						
517	RXDataSize0	USINT	•			
524	RXIdent0	UDINT	•			
Index * 2 + 529	RXData0Byte0 to RXData0Byte7	USINT	•			
Index * 4 + 530	RXData0Word0 to RXData0Word3	UINT	•			
Index * 8 + 532	RXData0Long0 to RXData0Long1	UDINT	•			
Receive buffer 1						
549	RXDataSize1	USINT	•			
556	RXIdent1	UDINT	•			
Index * 2 + 561	RXData1Byte0 to RXData1Byte7	USINT	•			
Index * 4 + 562	RXData1Word0 to RXData1Word3	UINT	•			
Index * 8 + 564	RXData1Long0 to RXData1Long1	UDINT	•			

5.3 Function model 2 - Stream and Function model 254 - Cyclic stream

Function models "Stream" and "Cyclic stream" use a module-specific driver of the controller's operating system. The interface can be controlled using libraries "ArCAN" and "CAN_Lib" and reconfigured at runtime.

Function model "Stream"

In function model "Stream", the controller communicates with the module acyclically. The interface is relatively convenient to operate, but the timing is very imprecise.

Function model "Cyclic stream"

Function model "Cyclic stream" was implemented later. From the application's point of view, there is no difference between function models "Stream" and "Cyclic stream". Internally, however, the cyclic I/O registers are used to ensure that communication follows deterministic timing.

Information:

- B&R controllers of type "SG4" must be used in order to use function models "Stream" and "Cyclic stream".
- These function models can only be used in X2X Link and POWERLINK networks.
- When using library "CAN_Lib", "ST1.IF1" must be added to the PLC address of CAN interface IF1.

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Module - Configuration						
-	AsynSize	-				
Interface - Configuration						
6273	CfO_ErrorID0007	USINT				•
Interface - Communication						
6145	CAN error state	USINT	•			
	CANwarning	Bit 0				
	CANpassive	Bit 1				
	CANbusoff	Bit 2				
	CANRXoverrun	Bit 3				
6209	CAN error acknowledgment	USINT			•	
	QuitCANwarning	Bit 0				
	QuitCANpassive	Bit 1				
	QuitCANbussoff	Bit 2				
	QuitCANRXoverrun	Bit 3				

5.4 Function model 254 - Flatstream

Flatstream provides independent communication between an X2X Link master and the module. This interface was implemented as a separate function model for the CAN module. CAN information (identifier, status, etc.) is transferred via cyclic input and output registers. The sequence and control bytes are used to control this data stream (see "[Flatstream communication](#)" on page 12).

When using function model Flatstream, the user can choose whether to use library "AsFltGen" in Automation Studio for implementation or to adapt Flatstream handling directly to the individual requirements of the application.

Information:

- Libraries "ArCAN" and "CAN_Lib" cannot be used.
- Higher data rates can be achieved between X2X master and module compared to function model "Flat".

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Interface - Configuration						
257	ConfigBaudrate	USINT				•
259	ConfigSJW	USINT				•
261	ConfigSPO	USINT				•
266	ConfigTXtrigger	UINT				•
6273	CfO_ErrorID0007	USINT				•
Interface - Communication						
6145	CAN error state	USINT	•			
	CANwarning	Bit 0				
	CANpassive	Bit 1				
	CANbusoff	Bit 2				
	CANRXoverrun	Bit 3				
6209	CAN error acknowledgment	USINT			•	
	QuitCANwarning	Bit 0				
	QuitCANpassive	Bit 1				
	QuitCANbusoff	Bit 2				
	QuitCANRXoverrun	Bit 3				
Flatstream - Configuration						
193	outputMTU	USINT				•
195	inputMTU	USINT				•
197	mode	USINT				•
199	forward	USINT				•
206	forwardDelay	UINT				•
Flatstream - Communication						
0	InputSequence	USINT	•			
Index * 1 + 0	RxByte1 to RxByte27	USINT	•			
32	OutputSequence	USINT			•	
Index * 1 + 32	TxByte1 to TxByte27	USINT			•	

5.5 Interface - Configuration

5.5.1 Transfer rate

Name:

ConfigBaudrate

"Baud rate" in the Automation Studio I/O configuration.

Configuration of the CAN transfer rate for the interface.

Data type	Values	Bus controller default setting
USINT	See the bit structure.	0

Bit structure:

Bit	Description	Value	Information
0 - 3	Transfer rate	0	Interface disabled (bus controller default setting)
		1	10 kbit/s
		2	20 kbit/s
		3	50 kbit/s
		4	100 kbit/s
		5	125 kbit/s
		6	250 kbit/s
		7	500 kbit/s
4 - 7	Reserved	-	

5.5.2 Synchronization jump width

Name:

ConfigSJW

"Synchronization jump width" in the Automation Studio I/O configuration.

The synchronization jump width (SJW) is used to resynchronize the sample point within a CAN telegram.

For a more detailed description of the synchronization jump width, see the CAN specification.

Data type	Values	Explanation
USINT	1 to 4	Synchronization jump width. Bus controller default setting: 3

5.5.3 Offset for the sampling point

Name:

ConfigSPO

"Sampling point offset" in the Automation Studio I/O configuration.

Offset for the sample point of the individual bits on the CAN bus.

For a more detailed description of the sampling point offset, see the CAN specification.

Data type	Values	Explanation
USINT	0 to 1	Bus controller default setting: 0

5.5.4 Starting the transmission procedure

Name:

ConfigTXtrigger

"TX objects / TX triggers" in the Automation Studio I/O configuration.

Defines the number of CAN objects that must be transferred to the transmit buffer before the transmission procedure is started.

Data type	Values	Explanation
UINT	0 to 8	Number of CAN objects in the transmit buffer before transmission is started. Bus controller default setting: 1

5.5.5 Configuring error messages

Name:

CfO_ErrorID0007

The error messages to be transferred must first be configured with this register. If the corresponding enable bit is not set, no error state will be reported to the higher-level system when the error occurs.

Data type	Values	Bus controller default setting
USINT	See the bit structure.	0

Bit structure:

Bit	Description	Value	Information
0	CANwarning	0	Disabled (bus controller default setting)
		1	Enabled
1	CANpassive	0	Disabled (bus controller default setting)
		1	Enabled
2	CANbussoff	0	Disabled (bus controller default setting)
		1	Enabled
3	CANRXoverrun	0	Disabled (bus controller default setting)
		1	Enabled
4 - 7	Reserved	-	

5.5.6 Size of the transmit buffer

Name:

Cfo_FIFOTXlimit

"TX FIFO size" in the Automation Studio I/O configuration.

Determines the size of the transmit buffer for the respective interface.

Data type	Values	Explanation
USINT	0 to 18	Size of the transmit buffer

5.5.7 Displaying unprocessed elements remaining in the transmit/receive buffer

Name:

Cfo_TXRXinfoFlags

These registers can be used to configure for the interface that the number of unprocessed elements in the transmit or receive buffer is indicated in the upper 4 bits of registers "TXCountReadBack" and "RXCount".

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	TxFifoInfo "Mode of channel TXCountReadBack" in the Automation Studio I/O configuration	0	Registers "TXCountReadBack" on page 44 and "TXCount-LatchReadBack" on page 44 are used to read back "TXCount".
		1	The lower 4 bits of registers "TXCountReadBack" on page 44 and "TXCountLatchReadBack" on page 44 are used to read back "TXCount". The upper 4 Bits are used to return the number of frames in the transmit buffer that have not been transmitted.
1	RxFifoInfo "Mode of channel RXCount" in the Automation Studio I/O configuration	0	Registers "RXCount" on page 44 and "RXCountLatch" on page 45 are used to indicate the number of telegrams that have been received.
		1	The lower 4 bits of registers "RXCount" on page 44 and "RX-CountLatch" on page 45 are used to indicate the number of telegrams received. The upper 4 bits are used to indicate the number of received but not acknowledged telegrams in the receive buffer.
2 - 7	Reserved	-	

5.6 Interface - Communication

5.6.1 CAN error state

Name:

CAN error state

The bits in this register indicate the error states defined in the CAN protocol. If an error occurs, the corresponding bit is set. For an error bit to be reset, the corresponding bit must be acknowledged (see ["CAN error acknowledgment" on page 43](#)).

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	CANwarning	0	No error
		1	CANwarning error on IF1
1	CANpassive	0	No error
		1	CANpassive error on IF1
2	CANbusoff	0	No error
		1	CANbusoff error on IF1
3	CANRXoverrun	0	No error
		1	CANRXoverrun error on IF1
4 - 7	Reserved	-	

CANwarning

A faulty frame was detected on the CAN bus. This can include bit errors, bit stuffing errors, CRC errors, format errors in the telegram and acknowledgment errors, for example.

CANpassive

The internal transmit and/or receive error counter is greater than 127. CAN communication continues to run, but the interface can only issue a "passive error frame". Likewise, "error passive stations" have less ability to send new telegrams altogether.

CANbusoff

The internal transmit error counter is greater than 255. The bus is switched off, and CAN communication with the module no longer takes place.

CANRXoverrun

An overflow occurred in the module's receive buffer.

5.6.2 CAN error acknowledgment

Name:

CAN error acknowledgment

By setting the respective bit in this register, the error assigned to the bit is acknowledged and the corresponding bit in register "CAN error state" is cleared. The application thus informs the module that it has detected the error state.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	QuitCANwarning	0	No acknowledgment
		1	Acknowledge CANwarning error on IF1
1	QuitCANpassive	0	No acknowledgment
		1	Acknowledge CANpassive error on IF1
2	QuitCANbusoff	0	No acknowledgment
		1	Acknowledge CANbusoff error on IF1
3	QuitCANRXoverrun	0	No acknowledgment
		1	Acknowledge CANRXoverrun error on IF1
4 - 7	Reserved	-	

5.6.3 New CAN telegram for transmit buffer

Name:
TXCount

By increasing this value, the application informs the module that a new CAN telegram should be transferred to the transmit buffer.

Data type	Values
USINT	0 to 255

5.6.4 Reading back "TXCount"

Name:
TXCountReadBack

The value of "TXCount" is copied from the module to this register. This allows the application task to verify that the data for the CAN telegram has been correctly applied by the module.

The meaning of the value depends on bit "TxFifoInfo". This is located in register "[Cfo_TXRXinfoFlags](#)" on page 42.

Data type	Values	Bit "TxFifoInfo"	Explanation
USINT	0 to 255	0	Read back "TXCount"
		1	See the bit structure.

Bit structure:

Bit	Explanation	Values	Information
0 - 3	Read back "TXCount"	0 to 15	Only the lower 4 bits
4 - 7	Number of still untransmitted frames in the transmit buffer	0 to 15	If this number exceeds value 15 (maximum 18 possible), value 15 is returned.

5.6.5 Reading back "TXCount" from the previous cycle

Name:
TXCountLatchReadBack

The module copies the value of "TXCount" from the previous cycle into this register. In the event of a transfer error on the X2X Link or POWERLINK network, this can be used to verify whether the error occurred on the path from the controller to the module or on the path from the module to the controller (see "[Consideration of error cases during transmission](#)" on page 11).

The meaning of the value depends on bit "TxFifoInfo" in register "[Cfo_TXRXinfoFlags](#)" on page 42.

Data type	Values	Bit "TxFifoInfo"	Explanation
USINT	0 to 255	0	Read back "TXCount" from the previous cycle
		1	See the bit structure.

Bit structure:

Bit	Explanation	Values	Information
0 - 3	Read back "TXCount" from the previous cycle	0 to 15	Only the lower 4 bits
4 - 7	Number of still untransmitted frames in the transmit buffer	0 to 15	From the previous cycle

5.6.6 Counter for received CAN telegrams

Name:
RXCount

This counter is incremented by 1 with each CAN telegram received. The application task can thus detect the receipt of new data and retrieve it accordingly from the "RXData" registers.

The meaning of the value depends on bit "RxFifoInfo" in register "[Cfo_TXRXinfoFlags](#)" on page 42.

Data type	Values	Bit "RxFifoInfo"	Explanation
USINT	0 to 255	0	Counter for received telegrams
		1	See the bit structure.

Bit structure:

Bit	Explanation	Values	Information
0 - 3	Counter for received telegrams	0 to 15	Only the lower 4 bits
4 - 7	Number of still unacknowledged telegrams in the receive buffer	0 to 15	

5.6.7 Reading back "RXCount" from the previous cycle

Name:

RXCountLatch

This register always contains the value of "RXCount" from the previous cycle. This can be used to detect transfer errors from the module to the controller (see ["Consideration of error cases during transmission" on page 11](#)).

The meaning of the value depends on bit "RxFifoInfo" in register "Cfo_TXRXinfoFlags" on page 42.

Data type	Values	Bit "RxFifoInfo"	Explanation
USINT	0 to 255	0	Counter for received telegrams from the previous cycle
		1	See the bit structure.

Bit structure:

Bit	Explanation	Values	Information
0 - 3	Counter for received telegrams from the previous cycle	0 to 15	Only the lower 4 bits
4 - 7	Number of telegrams in the receive buffer from the previous cycle	0 to 15	

5.7 Transmit buffer

5.7.1 Number of CAN payload data bytes

Name:

TXDataSize

Amount of CAN payload data bytes to be transmitted. If the value is less than 0, this CAN telegram is marked as invalid and thus not accepted into the transmit buffer. This is useful in connection with transfer error detection between the module and controller (see "[Consideration of error cases during transmission](#)" on page 11).

Data type	Values	Explanation
USINT	-128 to 8	Amount of CAN payload data to be transmitted

5.7.2 Identifier of the CAN telegram

Name:

TXIdent

Identifier of the CAN telegram to be transmitted. The frame format and identifier format are also defined in this register.

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Frame format	0	Standard frame format (SFF) with 11-bit identifier
		1	Extended frame format (EFF) with 29-bit identifier
1	Frame type	0	Data frame
		1	Remote frame (RTR)
2	Reserved	-	
3 - 31	CAN identifier of the telegram to be transmitted	x	Extended frame format (EFF) with 29 bits Standard frame format (SFF) with 11 bits ¹⁾

1) Only bits 21 to 31 are used; bits 3 to 20 = 0.

5.7.3 Configuration of the CAN payload data to be transmitted

Name:

TXDataByte0 to TXDataByte7

TXDataWord0 to TXDataWord3

TXDataLong0 to TXDataLong1

CAN payload data in the transmit direction. Depending on requirements, the 8 payload data bytes of a telegram can be used as 8 individual bytes, 4 word or 2 long data points.

Data type	Values	Description
USINT	0 to 255	Transmitted CAN payload data as bytes
UINT	0 to 65535	Transmitted CAN payload data as word
UDINT	0 to 4,294,967,295	Transmitted CAN payload data as long

5.8 Receive buffers 0 and 1

5.8.1 Number of valid CAN payload data bytes

Name:

RXDataSize0

RXDataSize1

Number of valid CAN payload data bytes.

This register also indicates a general error or a gap in the input data stream by the value -1 (0xFF). Details about the error that has occurred are indicated in register ["CAN error state" on page 43](#).

Data type	Values	Explanation
USINT	1 to 8	Amount of CAN payload data
	-1	Error

5.8.2 Identifier of the received data

Name:

RXIdent0

RXIdent1

Identifier to which the received data is assigned. The frame format and identifier format can also be read from this register.

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Frame format	0	Standard frame format (SFF) with 11-bit identifier
		1	Extended frame format (EFF) with 29-bit identifier
1	Frame type	0	Data frame
		1	Remote frame (RTR)
2	Reserved	-	
3 - 31	CAN identifier of the telegram to be transmitted	x	Extended frame format (EFF) with 29 bits Standard frame format (SFF) with 11 bits ¹⁾

1) Only bits 21 to 31 are used; bits 3 to 20 = 0.

5.8.3 Configuration of the CAN payload data to be received

Name:

RXData0Byte0 to RXData0Byte7

RXData0Word0 to RXData0Word3

RXData0Long0 to RXData0Long1

RXData1Byte0 to RXData1Byte7

RXData1Word0 to RXData1Word3

RXData1Long0 to RXData1Long1

The CAN object's payload data that should be transferred from the receive buffer to the controller in the current cycle are stored in these registers. If new data is received or if there are still additional CAN objects in the receive buffer, these registers are overwritten with the new data in the next cycle.

To ensure as far as possible that no CAN objects are lost, it is necessary that the application responds immediately to a change of "RXCount" and recopies the data from these registers.

The maximum 8 bytes of a CAN telegram can optionally be used as 8 individual bytes, 4 words or 2 long data points.

Data type	Values	Description
USINT	0 to 255	Received CAN payload data as bytes
UINT	0 to 65535	Received CAN payload data as word
UDINT	0 to 4,294,967,295	Received CAN payload data as long

5.9 Flatstream registers

At the absolute minimum, registers "InputMTU" and "OutputMTU" must be set. All other registers are filled in with default values at the beginning and can be used immediately. These registers are used for additional options, e.g. to transfer data in a more compact way or to increase the efficiency of the general procedure.

Information:

For detailed information about Flatstream, see ["Flatstream communication" on page 12](#).

5.9.1 Number of enabled Tx and Rx bytes

Name:

OutputMTU

InputMTU

These registers define the number of enabled Tx or Rx bytes and thus also the maximum size of a sequence. The user must consider that the more bytes made available also means a higher load on the bus system.

Data type	Values
USINT	See the register overview.

5.9.2 Transporting payload data and control bytes

Name:

TxByte1 to TxByteN

RxByte1 to RxByteN

(The value the number N is different depending on the bus controller model used.)

The Tx and Rx bytes are cyclic registers used to transport the payload data and the necessary control bytes. The number of active Tx and Rx bytes is taken from the configuration of registers ["OutputMTU"](#) and ["InputMTU"](#), respectively.

- "T" - "Transmit" → Controller *transmits* data to the module.
- "R" - "Receive" → Controller *receives* data from the module.

Data type	Values
USINT	0 to 255

5.9.3 Communication status of the controller

Name:

OutputSequence

This register contains information about the communication status of the controller. It is written by the controller and read by the module.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0 - 2	OutputSequenceCounter	0 - 7	Counter for the sequences issued in the output direction
3	OutputSyncBit	0	Output direction (disable)
		1	Output direction (enable)
4 - 6	InputSequenceAck	0 - 7	Mirrors InputSequenceCounter
7	InputSyncAck	0	Input direction not ready (disabled)
		1	Input direction ready (enabled)

OutputSequenceCounter

The OutputSequenceCounter is a continuous counter of sequences that have been issued by the controller. The controller uses OutputSequenceCounter to direct the module to accept a sequence (the output direction must be synchronized when this happens).

OutputSyncBit

The controller uses OutputSyncBit to attempt to synchronize the output channel.

InputSequenceAck

InputSequenceAck is used for acknowledgment. The value of InputSequenceCounter is mirrored if the controller has received a sequence successfully.

InputSyncAck

The InputSyncAck bit acknowledges the synchronization of the input channel for the module. This indicates that the controller is ready to receive data.

5.9.4 Communication status of the module

Name:

InputSequence

This register contains information about the communication status of the module. It is written by the module and should only be read by the controller.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0 - 2	InputSequenceCounter	0 - 7	Counter for sequences issued in the input direction
3	InputSyncBit	0	Not ready (disabled)
		1	Ready (enabled)
4 - 6	OutputSequenceAck	0 - 7	Mirrors OutputSequenceCounter
7	OutputSyncAck	0	Not ready (disabled)
		1	Ready (enabled)

InputSequenceCounter

The InputSequenceCounter is a continuous counter of sequences that have been issued by the module. The module uses InputSequenceCounter to direct the controller to accept a sequence (the input direction must be synchronized when this happens).

InputSyncBit

The module uses InputSyncBit to attempt to synchronize the input channel.

OutputSequenceAck

OutputSequenceAck is used for acknowledgment. The value of OutputSequenceCounter is mirrored if the module has received a sequence successfully.

OutputSyncAck

The OutputSyncAck bit acknowledges the synchronization of the output channel for the controller. This indicates that the module is ready to receive data.

5.9.5 Flatstream mode

Name:

FlatstreamMode

A more compact arrangement can be achieved with the incoming data stream using this register.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	MultiSegmentMTU	0	Not allowed (default)
		1	Permitted
1	Large segments	0	Not allowed (default)
		1	Permitted
2 - 7	Reserved		

5.9.6 Number of unacknowledged sequences

Name:
Forward

With register "Forward", the user specifies how many unacknowledged sequences the module is permitted to transmit.

Recommendation:

X2X Link: Max. 5

POWERLINK: Max. 7

Data type	Values
USINT	1 to 7 Default: 1

5.9.7 Delay time

Name:
ForwardDelay

This register is used to specify the delay time in microseconds.

Data type	Values
UINT	0 to 65535 [μ s] Default: 0

5.10 Acyclic frame size

Name:
AsynSize

When using the stream, the data is exchanged internally between the module and controller. A defined number of acyclic bytes is reserved for this slot for this purpose.

Increasing the acyclic frame size results in increased data throughput on this slot.

Information:

This configuration involves a driver setting that cannot be changed during runtime!

Data type	Values	Information
-	8 to 28	Acyclic frame size in bytes. Default = 24

5.11 Minimum cycle time

The minimum cycle time specifies how far the bus cycle can be reduced without communication errors occurring. It is important to note that very fast cycles reduce the idle time available for handling monitoring, diagnostics and acyclic commands.

Minimum cycle time
200 μ s

5.12 Minimum I/O update time

The minimum I/O update time specifies how far the bus cycle can be reduced so that an I/O update is performed in each cycle.

Minimum I/O update time
200 μ s