

X20(c)CS1030

Data sheet
3.40 (August 2025)



Publishing information

B&R Industrial Automation GmbH

B&R Strasse 1

5142 Eggelsberg

Austria

Telephone: +43 7748 6586-0

Fax: +43 7748 6586-26

office@br-automation.com

Disclaimer

All information in this document is current as of its creation. The contents of this document are subject to change without notice. B&R Industrial Automation GmbH assumes unlimited liability in particular for technical or editorial errors in this document only (i) in the event of gross negligence or (ii) for culpably inflicted personal injury. Beyond that, liability is excluded to the extent permitted by law. Liability in cases in which the law stipulates mandatory unlimited liability (such as product liability) remains unaffected. Liability for indirect damage, consequential damage, business interruption, loss of profit or loss of information and data is excluded, in particular for damage that is directly or indirectly attributable to the delivery, performance and use of this material.

B&R Industrial Automation GmbH notes that the software and hardware designations and brand names of the respective companies used in this document are subject to general trademark, brand or patent protection.

Hardware and software from third-party suppliers referenced in this document is subject exclusively to the respective terms of use of these third-party providers. B&R Industrial Automation GmbH assumes no liability in this regard. Any recommendations made by B&R Industrial Automation GmbH are not contractual content, but merely non-binding information for which no liability is assumed. When using hardware and software from third-party suppliers, the relevant user documentation of these third-party suppliers must additionally be consulted and, in particular, the safety guidelines and technical specifications contained therein must be observed. The compatibility of the products from B&R Industrial Automation GmbH described in this document with hardware and software from third-party suppliers is not contractual content unless this has been separately agreed in individual cases; in this respect, warranty for such compatibility is excluded in any case, and it is the sole responsibility of the customer to verify this compatibility in advance.

1247782280719-3.40

1 General information

1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX20	X20 System user's manual

1.2 Coated modules

Coated modules are X20 modules with a protective coating for the electronics component. This coating protects X20c modules from condensation and corrosive gases.

The modules' electronics are fully compatible with the corresponding X20 modules.



For simplification purposes, only images and module IDs of uncoated modules are used in this data sheet.

The coating has been certified according to the following standards:

- Condensation: BMW GS 95011-4, 2x 1 cycle
- Corrosive gas: EN 60068-2-60, method 4, exposure 21 days



1.3 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS1030	X20 interface module, 1 RS422/485 interface, max. 115.2 kbit/s	
X20cCS1030	X20 interface module, coated, 1 RS422/485 interface, max. 115.2 kbit/s	
	Required accessories	
	Bus modules	
X20BM11	X20 bus module, 24 VDC keyed, internal I/O power supply connected through	
X20BM15	X20 bus module, with node number switch, 24 VDC keyed, internal I/O power supply connected through	
X20cBM11	X20 bus module, coated, 24 VDC keyed, internal I/O power supply connected through	
	Terminal blocks	
X20TB06	X20 terminal block, 6-pin, 24 VDC keyed	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	

Table 1: X20CS1030, X20cCS1030 - Order data

1.4 Module description

In addition to the standard I/O, complex devices often need to be connected. The X20CS communication modules are intended precisely for cases like this. As normal X20 electronics modules, they can be placed anywhere on the remote backplane.

Functions

- [RS422/485 interface](#)
- [Flatstream](#)

RS485/RS422

The RS485/RS422 interface enables the serial decentralized connection of devices to the X20 System.

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	X20CS1030	X20cCS1030
Short description		
Communication module	1x RS485/RS422	
General information		
B&R ID code	0x1FD0	0xE500
Status indicators	Data transfer, terminating resistor, operating state, module status	
Diagnostics		
Module run/error	Yes, using LED status indicator and software	
Data transfer	Yes, using LED status indicator	
Terminating resistor	Yes, using LED status indicator	
Power consumption		
Bus	0.01 W	
Internal I/O	1.44 W	
Additional power dissipation caused by actuators (resistive) [W]	-	
Certifications		
CE	Yes	
UKCA	Yes	
ATEX	Zone 2, II 3G Ex nA nC IIA T5 Gc IP20, Ta (see X20 user's manual) FTZÚ 09 ATEX 0083X	
UL	cULus E115267 Industrial control equipment	
HazLoc	cCSAus 244665 Process control equipment for hazardous locations Class I, Division 2, Groups ABCD, T5	
DNV	Temperature: B (0 to 55°C) Humidity: B (up to 100%) Vibration: B (4 g) EMC: B (bridge and open deck)	
CCS	Yes	-
LR	ENV1	
KR	Yes	
ABS	Yes	
BV	EC33B Temperature: 5 - 55°C Vibration: 4 g EMC: Bridge and open deck	
KC	Yes	-
Interfaces		
Interface IF1		
Signal	RS485/RS422	
Variant	Connection made using 12-pin terminal block X20TB12	Connection via 12-pin terminal block X20TB12
Number of stations	RS485: Maximum 32 stations RS422: Maximum 10 stations (receivers)	
Max. distance	1200 m	
Transfer rate	Max. 115.2 kbit/s	
FIFO buffer	1 kB	
Terminating resistor	Integrated in module	
Controller	UART type 16C550 compatible	
Electrical properties		
Electrical isolation	RS485/RS422 (IF1) isolated from bus and I/O power supply	
Operating conditions		
Mounting orientation		
Horizontal	Yes	
Vertical	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	IP20	

Table 2: X20CS1030, X20cCS1030 - Technical data

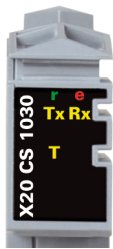
Technical description

Order number	X20CS1030		X20cCS1030
Ambient conditions			
Temperature			
Operation			
Horizontal mounting orientation	-25 to 60°C		
Vertical mounting orientation	-25 to 50°C		
Derating	See section "Derating".		
Storage	-40 to 85°C		
Transport	-40 to 85°C		
Relative humidity			
Operation	5 to 95%, non-condensing	Up to 100%, condensing	
Storage	5 to 95%, non-condensing		
Transport	5 to 95%, non-condensing		
Mechanical properties			
Note	Order 1x terminal block X20T-B06 or X20TB12 separately. Order 1x bus module X20BM11 separately.	Order 1x terminal block X20T-B06 or X20TB12 separately. Order 1x bus module X20cBM11 separately.	
Pitch	12.5 ^{+0.2} mm		

Table 2: X20CS1030, X20cCS1030 - Technical data

2.2 LED status indicators

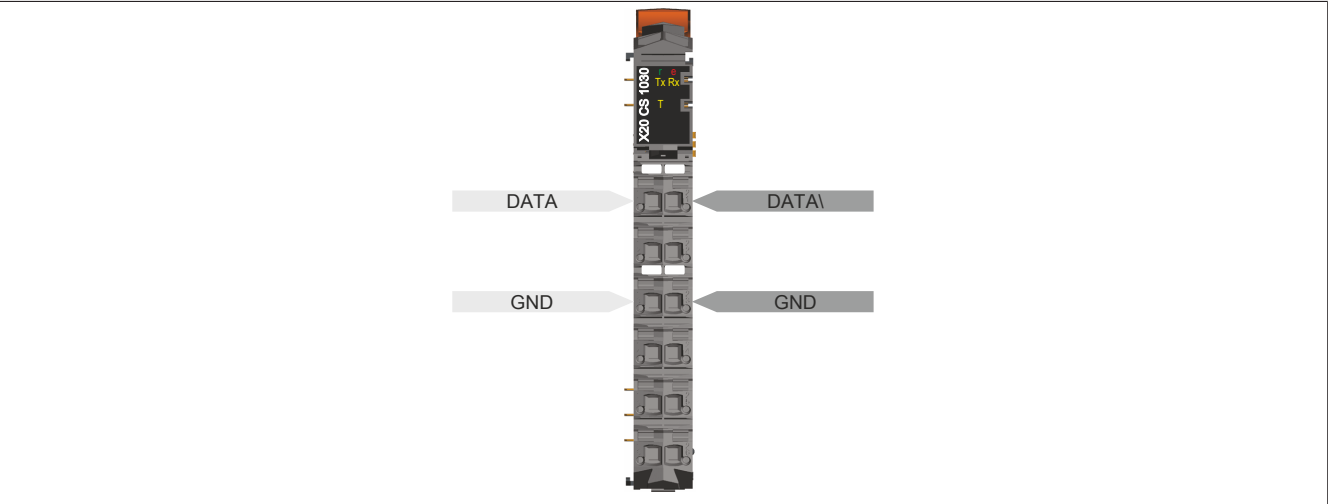
For a description of the various operating modes, see section "Additional information - Diagnostic LEDs" in the X20 system user's manual.

Figure	LED	Color	Status	Description
	r	Green	Off	No power to module
			Single flash	RESET mode
			Double flash	BOOT mode (during firmware update) ¹⁾
			Blinking	PREOPERATIONAL mode
			On	RUN mode
	e	Red	Off	No power to module or everything OK
			Single flash	An I/O error has occurred, see "Error message status bits" on page 46
			On	Error or reset status
	e + r	Red on / Green single flash		Invalid firmware
	Tx	Yellow	On	The module transmits data via the RS485/RS422 interface
	Rx	Yellow	On	The module receives data via the RS485/RS422 interface
	T	Yellow	On	Terminating resistor integrated in the module switched on

1) Depending on the configuration, a firmware update can take up to several minutes.

2.3 Pinout

RS485 mode



3 UL certificate information

To install the module according to the UL standard, the following rules must be observed.



Information:

- **Use copper conductors only. Minimum temperature rating of the cable to be connected to the field wiring terminals: 61°C, 28 - 14 AWG.**
- **All models are intended to be used in a final safety enclosure that must conform with requirements for protection against the spread of fire and have adequate rigidity per UL 61010-1 and UL 61010-2-201.**
- **The external circuits intended to be connected to the device shall be galv. separated from mains supply or hazardous live voltage by reinforced or double insulation and meet the requirements of SELV/PELV circuit.**
- **If the equipment is used in not specified manner, the protection provided by the equipment may be impaired.**
- **Repairs can only be made by B&R.**

4 Function description

4.1 RS422/485 interface

The module is equipped with an RS422/485 interface.

The interface is only permitted to be used with the registers and interface values provided for this purpose. After selecting the interface mode and completely writing to all configuration registers, enabling the interface must be the last write command. If a parameter change is necessary, the interface must first be disabled.



Information:

The interface mode register is described in ["Serial interface - Configuration" on page 40](#).

4.2 Flatstream communication

4.2.1 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.

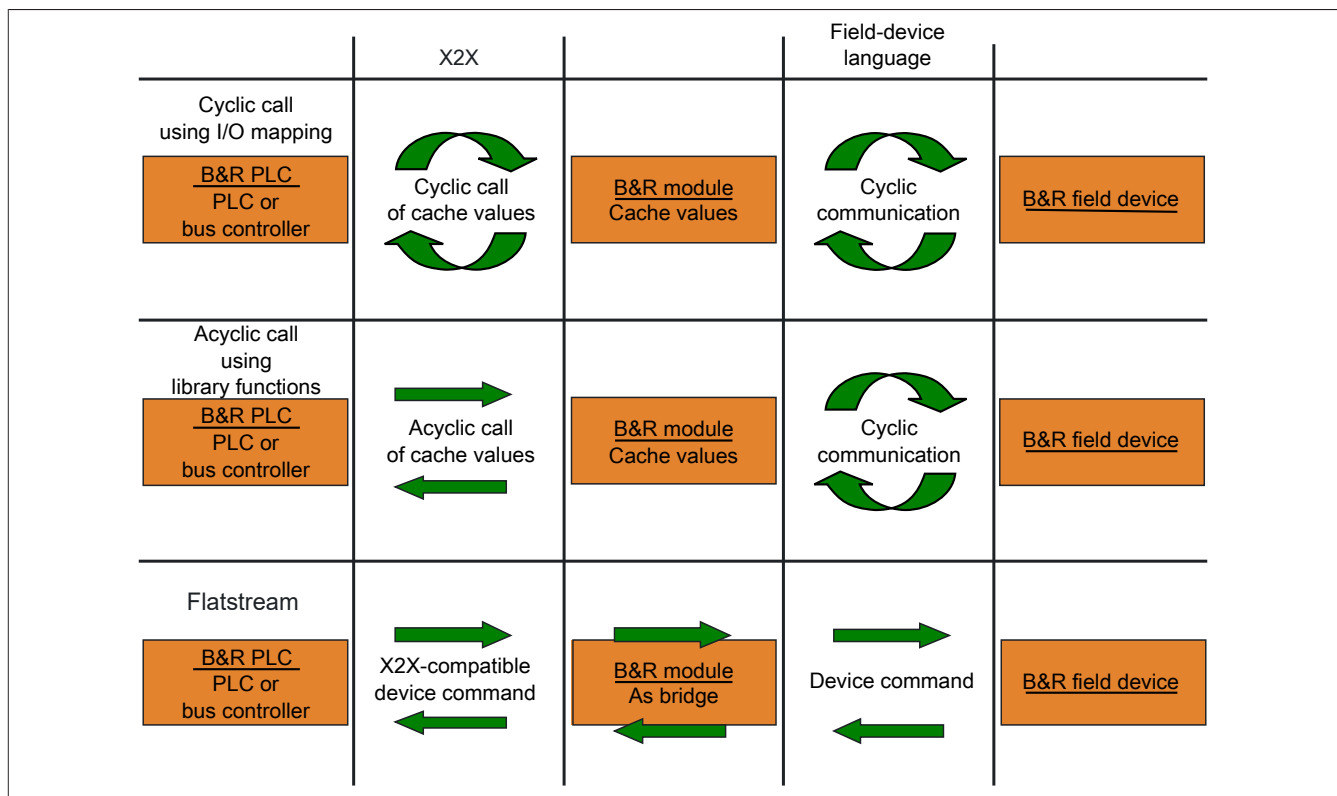


Figure 1: 3 types of communication

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.

4.2.2 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.

4.2.3 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.

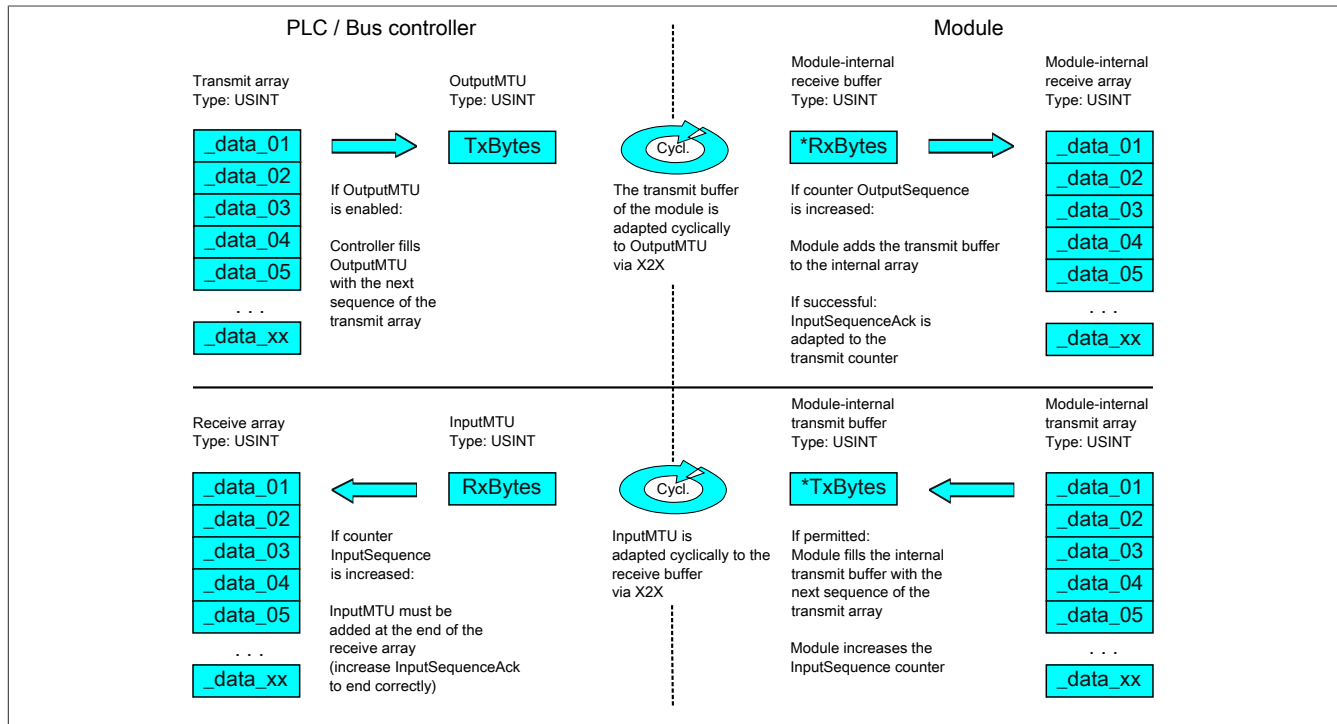


Figure 2: Flatstream communication

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 re-filled. 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.

4.2.4 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 RxBytes bytes. For this reason, the user must have sufficient knowledge of the communication protocol being used on the field device.

4.2.4.1 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 47.

Registers are described in section "Flatstream communication" in the respective data sheets.

4.2.4.2 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 47.

Registers are described in section "Flatstream communication" in the respective data sheets.

4.2.4.2.1 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.

4.2.4.2.2 Transporting 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.

4.2.4.2.2.1 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 MultiSegmentMTUs, 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

Function description

4.2.4.2.3 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.

4.2.4.2.3.1 Relationship between OutputSequence and InputSequence

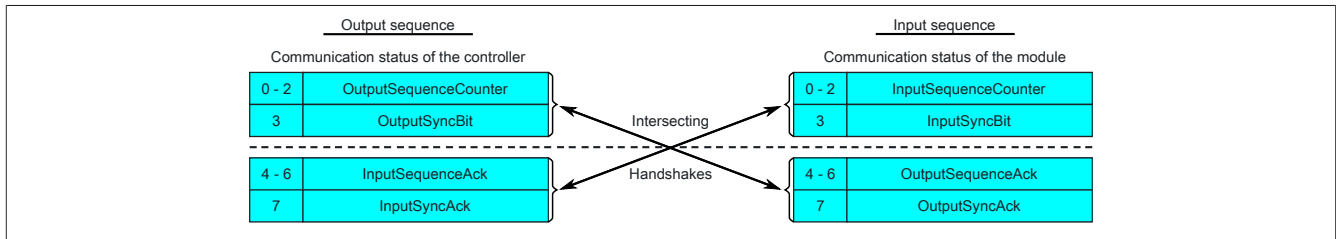


Figure 3: Relationship between OutputSequence and InputSequence

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.

4.2.4.3 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). The module does not accept the current contents of InputMTU since the channel is not yet synchronized. The module matches OutputSequenceAck and OutputSyncAck to the values of OutputSequenceCounter and OutputSyncBit.
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). The module does not accept the current contents of InputMTU since the channel is not yet synchronized. The module matches OutputSequenceAck and OutputSyncAck to the values of OutputSequenceCounter and OutputSyncBit.
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. The module sets OutputSyncAck. 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

The module writes 000 to InputSequenceCounter and resets InputSyncBit. The module monitors the high nibble of register "OutputSequence" and expects 000 in InputSequenceAck and 0 in InputSyncAck.
1) The controller is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The controller must match InputSequenceAck and InputSyncAck to the values of InputSequenceCounter and InputSyncBit. If the module registers the expected values in InputSequenceAck and InputSyncAck, it increments InputSequenceCounter. The module monitors the high nibble of register "OutputSequence" and expects 001 in InputSequenceAck and 0 in InputSyncAck.
2) The controller is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The controller must match InputSequenceAck and InputSyncAck to the values of InputSequenceCounter and InputSyncBit. If the module registers the expected values in InputSequenceAck and InputSyncAck, it sets InputSyncBit. The module monitors the high nibble of register "OutputSequence" and expects 1 in InputSyncAck.
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.

Function description

4.2.4.4 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 must 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:

Position (of the next control byte) = Current position + 1 + 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.

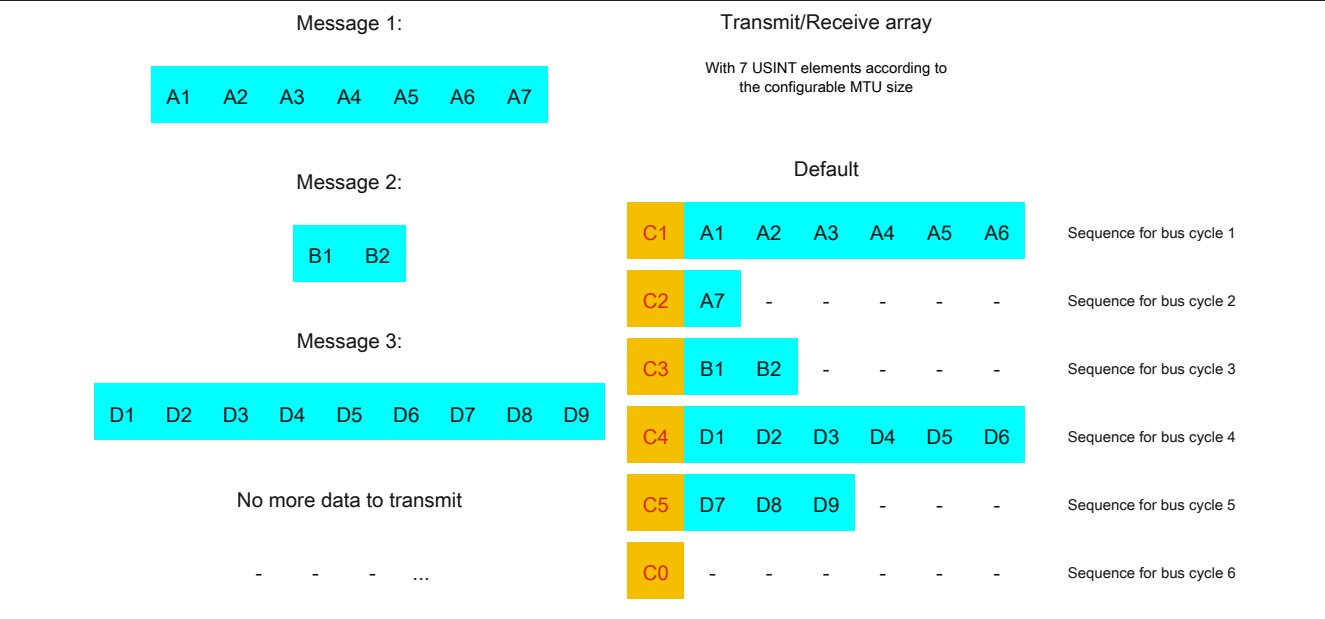


Figure 4: Transmit/Receive array (default)

The messages must first 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 3: 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 4: Flatstream determination of the control bytes for the default configuration example (part 2)

4.2.4.4.1 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.

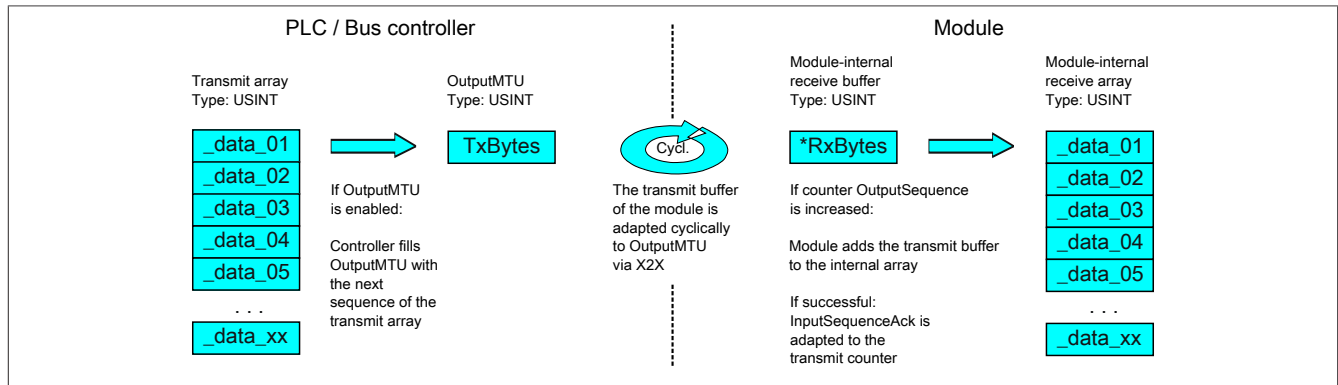


Figure 5: Flatstream communication (output)

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

Cyclic status query: - The module monitors OutputSequenceCounter.
0) Cyclic checks: - 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.
1) Preparation (create transmit array): - 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.
2) Transmit: - 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.
Reaction: - 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.
3) Completion: - 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 Completion phase is run through long enough.
Note: 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. (The relationship of bus to task cycle can be influenced by the user so that the threshold value must be determined individually.) - 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 must 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

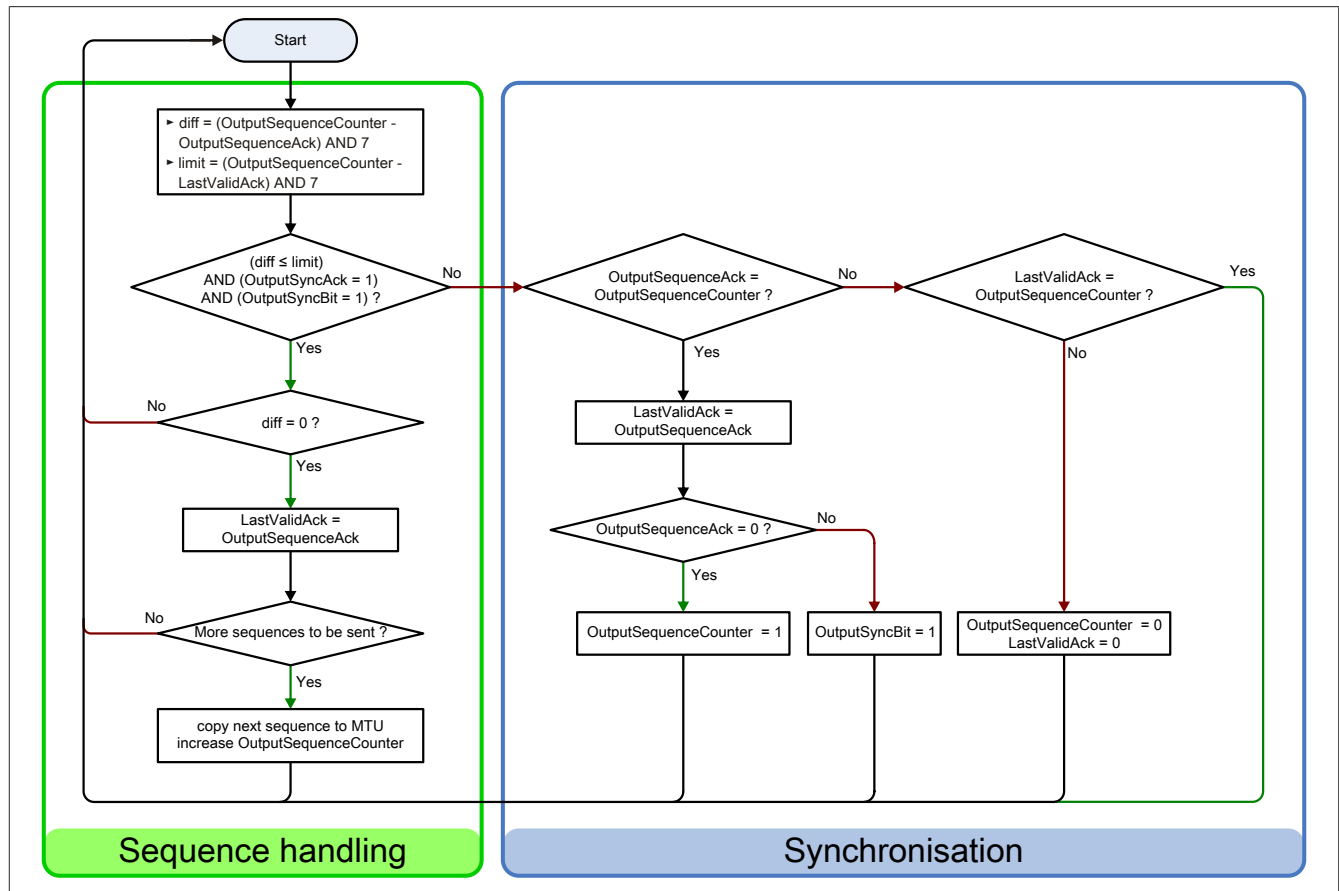


Figure 6: Flowchart for the output direction

Function description

4.2.4.4.2 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.

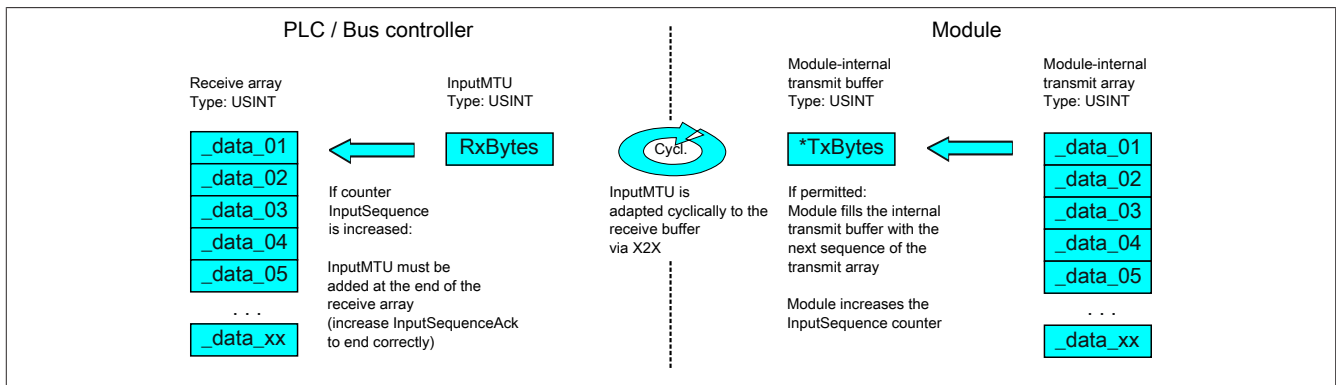


Figure 7: Flatstream communication (input)

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

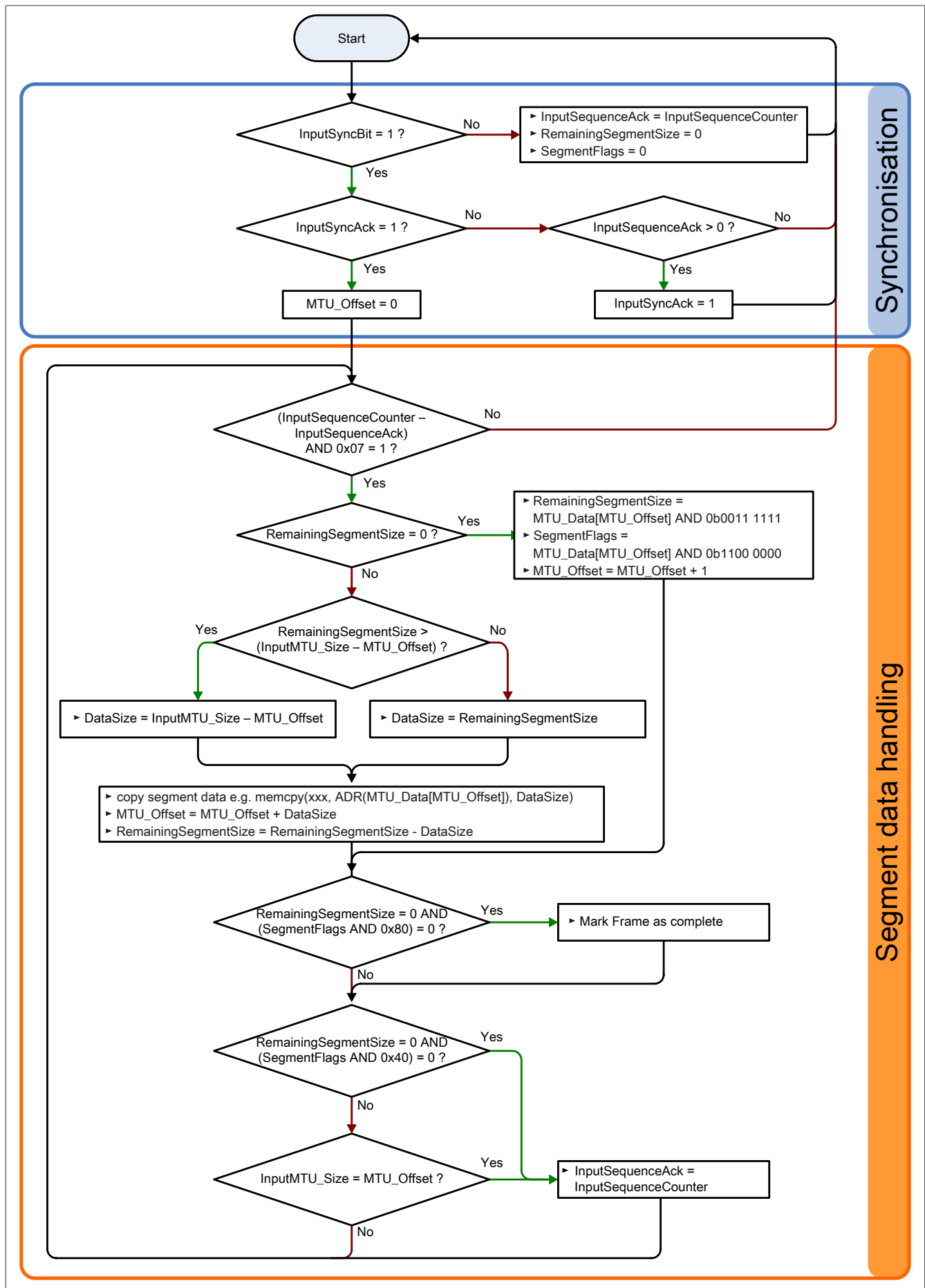


Figure 8: Flowchart for the input direction

4.2.4.4.3 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.

4.2.4.5 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](#)
- [MultiSegmentMTU 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 "MultiSegmentMTU" 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.

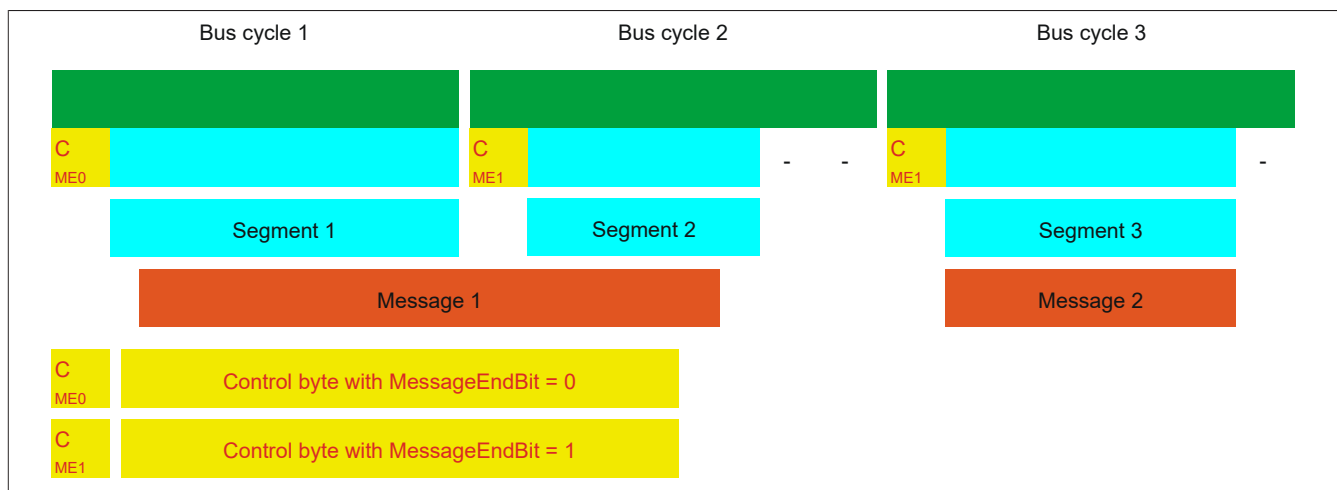


Figure 9: Message arrangement in the MTU (default)

Function description

MultiSegmentMTU 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.

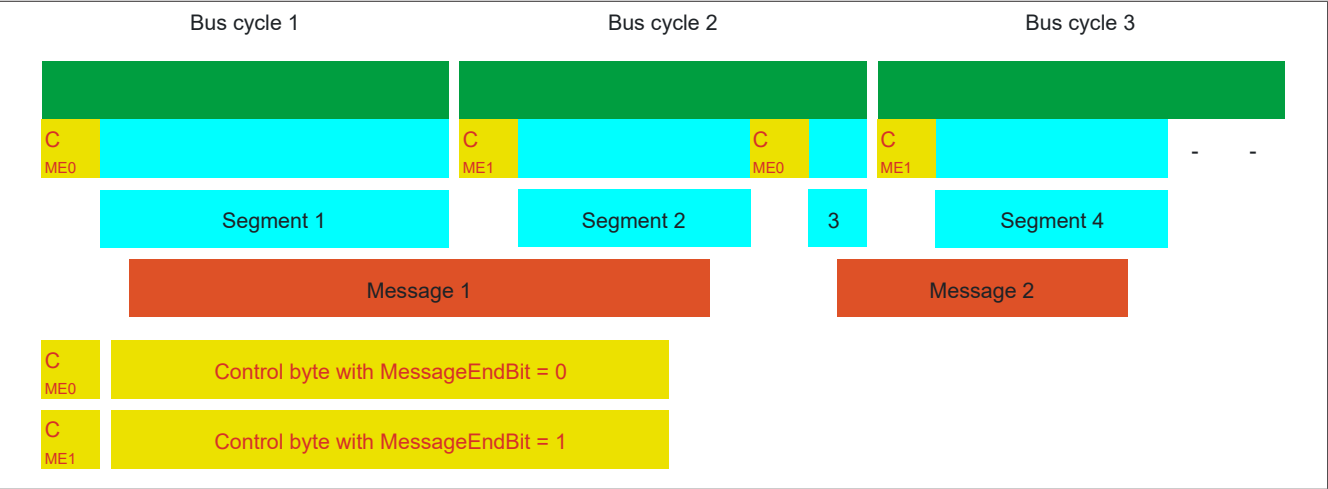


Figure 10: Arrangement of messages in the MTU (MultiSegmentMTU)

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.

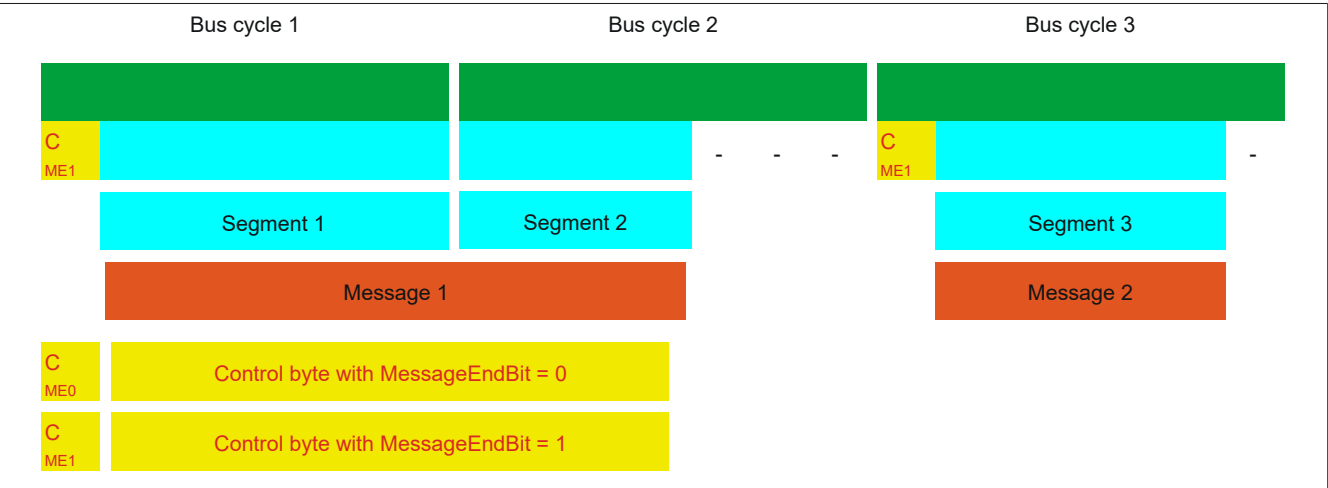


Figure 11: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

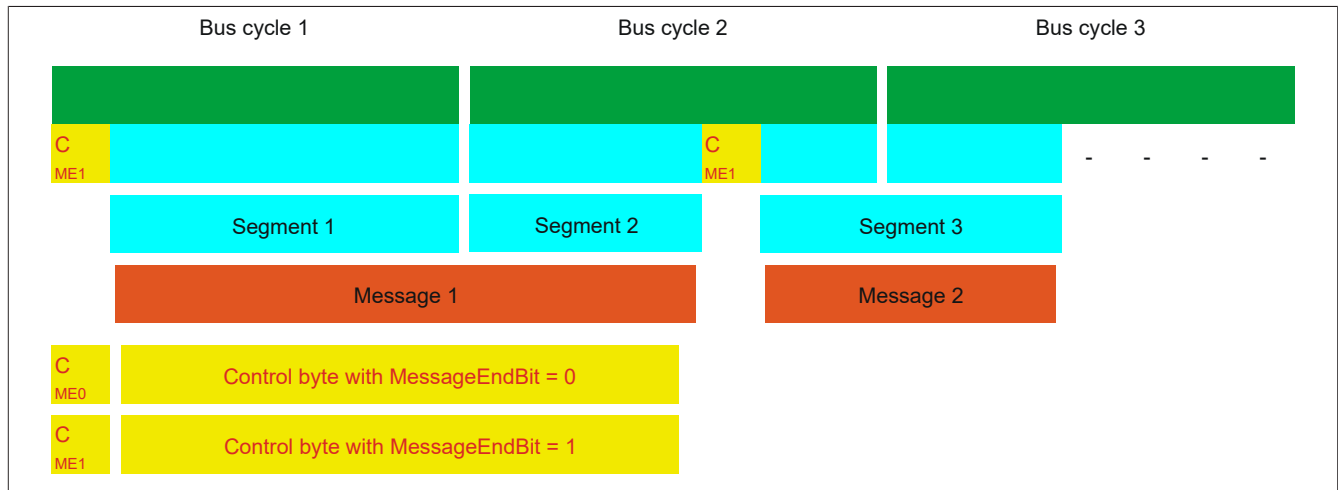


Figure 12: Arrangement of messages in the MTU (large segments and MultiSegmentMTU)

4.2.4.6 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 "nextCB-Pos" 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.

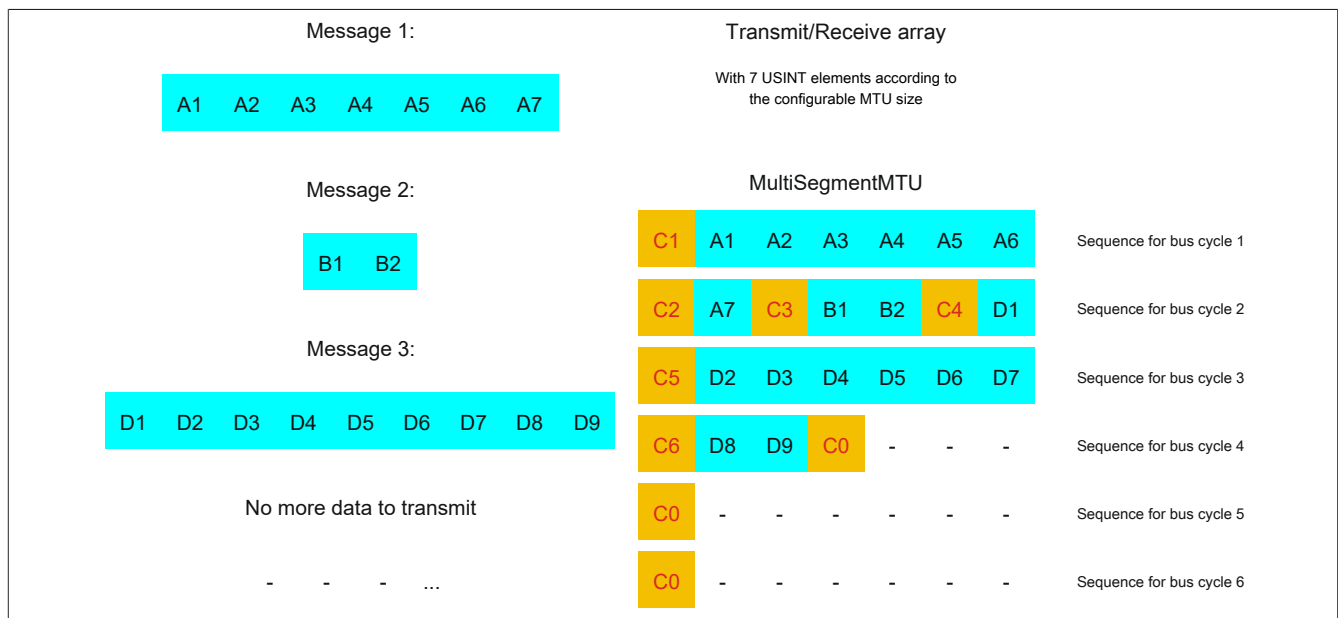


Figure 13: Transmit/Receive array (MultiSegmentMTU)

Function description

The messages must first 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 5: 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.



Mise en garde !

La deuxième séquence ne peut être acquittée via SequenceAck que si elle a été entièrement traitée. Dans cet exemple, il y a 3 segments différents dans la deuxième séquence, c'est-à-dire que le programme doit inclure suffisamment de tableaux de réception pour gérer cette 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 6: 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.

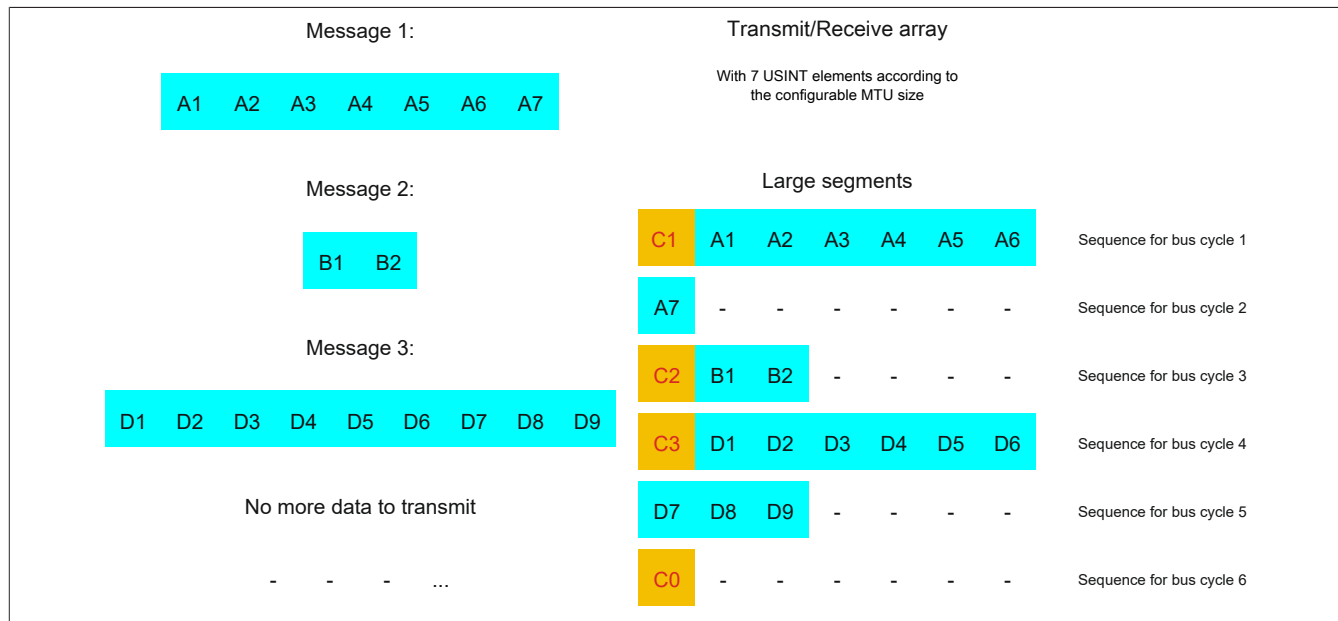


Figure 14: Transmit/receive array (large segments)

The messages must first 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 7: Flatstream determination of the control bytes for the large segment example

Function description

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.

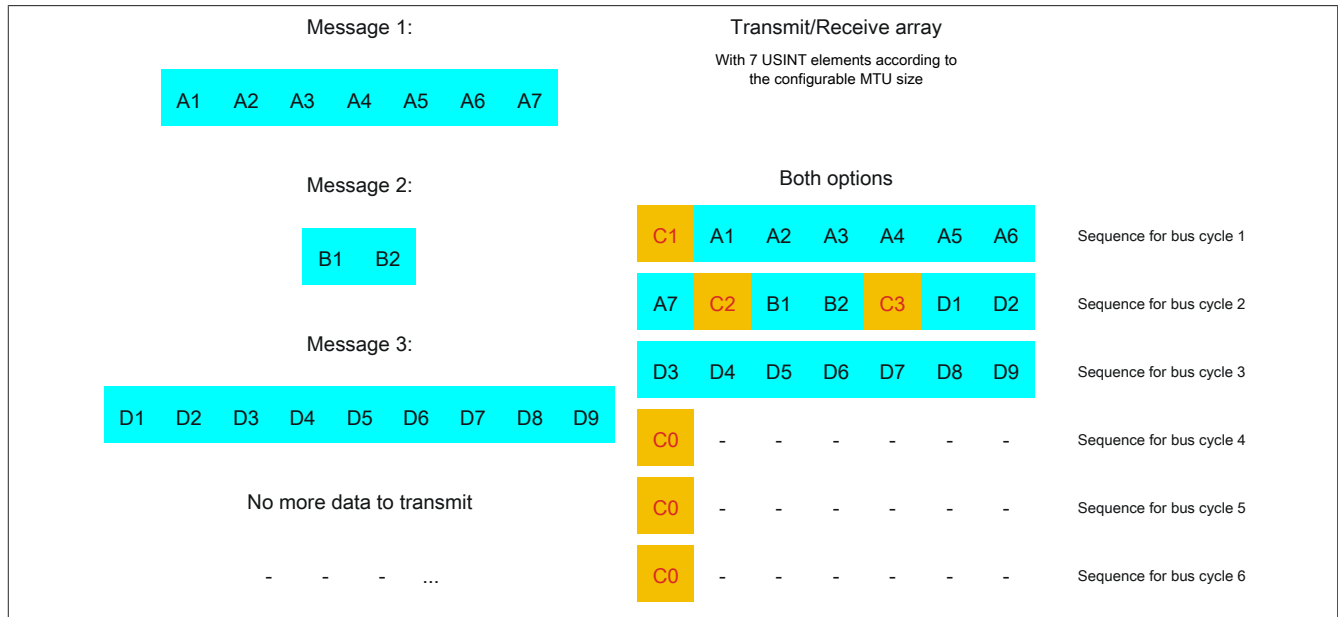


Figure 15: Transmit/Receive array (large segments and MultiSegmentMTU)

The messages must first 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 8: Flatstream determination of the control bytes for the large segment and MultiSegmentMTU example

4.2.5 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.

4.2.5.1 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 Sequence-Counter	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)

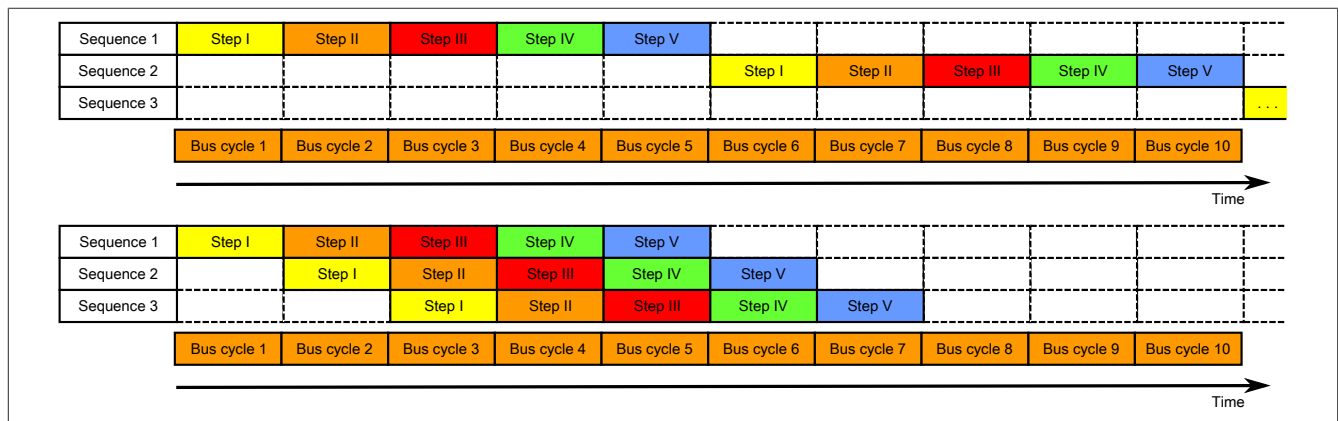


Figure 16: Comparison of transfer without/with Forward

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 must still acknowledge every sequence received. Only when SequenceAck has been changed and checked by the transmitter is the sequence considered as having been transferred successfully.

4.2.5.2 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 47](#).

Registers are described in section ["Flatstream communication"](#) in the respective data sheets.

4.2.5.2.1 Delay time

The delay time is specified in microseconds. This is the amount of time the module must 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.

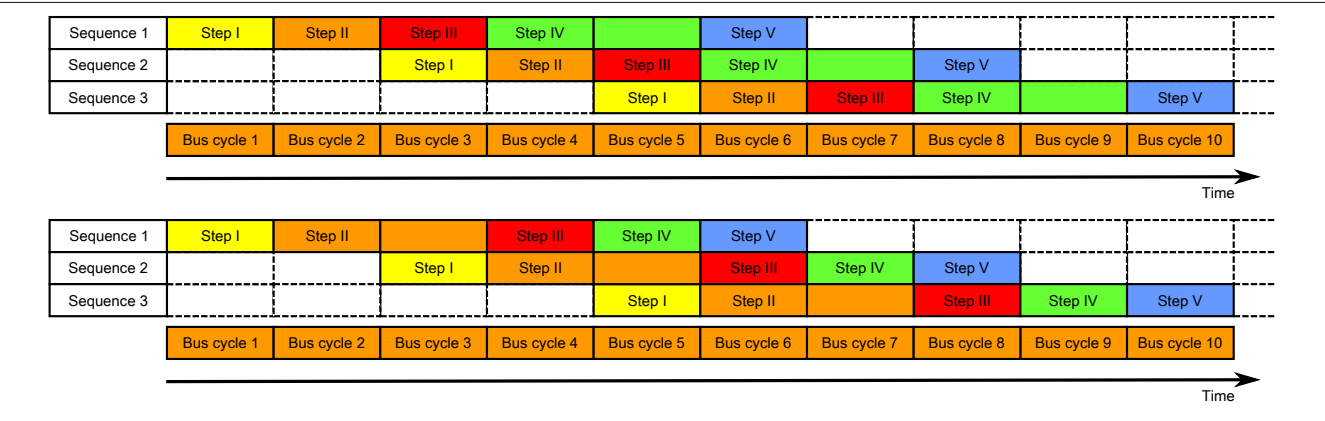


Figure 17: Effect of ForwardDelay when using Flatstream communication with the Forward function

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.

4.2.5.3 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

Cyclic status query: - The module monitors OutputSequenceCounter.
0) Cyclic checks: - The controller must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The controller must check whether OutputMTU is enabled. → If OutputSequenceCounter > OutputSequenceAck + 7, then it is not enabled because the last sequence has not yet been acknowledged.
1) Preparation (create transmit array): - 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.
2) Transmit: - The controller must transfer the current part of the transmit array to OutputMTU. - The controller must increase OutputSequenceCounter for the sequence to be accepted by the module. - The controller is then permitted to transmit in the next bus cycle if the MTU has been enabled.
The module responds since OutputSequenceCounter > OutputSequenceAck: - 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 OutputSequenceCounter is transferred to OutputSequenceAck. - The module queries the status cyclically again.
3) Completion (acknowledgment): - The controller must check OutputSequenceAck cyclically. → 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 algorithm is run through long enough.
Note: 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 (the relationship of bus to task cycle can be influenced by the user so that the threshold value must be determined individually).

Algorithm for receiving

0) Cyclic status query: - The controller must monitor InputSequenceCounter.
Cyclic checks: - The module checks InputSyncAck. - The module checks if InputMTU for enabling. → Enabling criteria: InputSequenceCounter > InputSequenceAck + Forward
Preparation: - The module forms the control bytes / segments and creates the transmit array.
Action: - The module transfers the current part of the transmit array to the receive buffer. - The module increases InputSequenceCounter. - The module waits for a new bus cycle after time from ForwardDelay has expired. - The module repeats the action if InputMTU is enabled.
1) Receiving (InputSequenceCounter > InputSequenceAck): - 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.

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.

4.2.5.4 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).

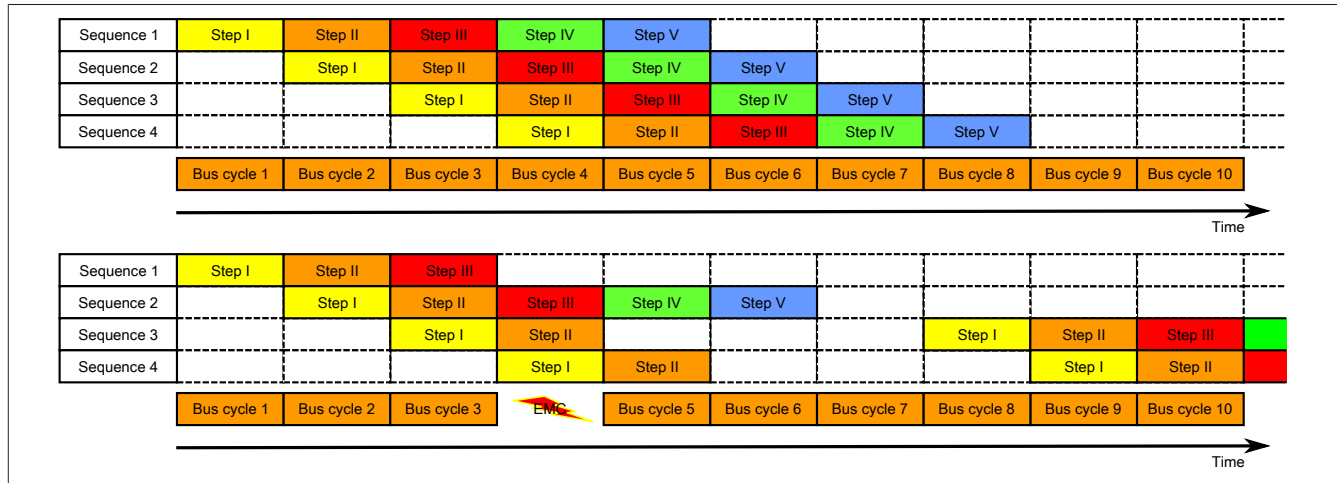


Figure 18: Effect of a lost bus cycle

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.

5 Commissioning

5.1 Usage after the X20IF1091-1

If this module is operated after X2X Link module X20IF1091-1, delays may occur during the Flatstream transfer. For detailed information, see section "Data transfer on the Flatstream" in X20IF1091-1.

5.2 Serial with FlatStream

When using FlatStream communication, the module acts as a bridge between the X2X Link master and an intelligent field device connected to the module. FlatStream mode can be used for either point-to-point connections as well as for multidrop systems. Specific algorithms such as timeout and checksum monitoring are usually managed automatically. During normal operation, the user does not have access to these details.

In a serial network, the module is always the master (DTE). Various adjustments can be made to ensure that signals are transmitted without errors.

The user can, for example, define a handshake algorithm or set the baud rate in order to adapt the transmission quality to the requirements of the application.

Operation

When using FlatStream, the general structure of the FlatStream frame must be maintained.

Input/Output sequence	Tx/Rx bytes	
(unchanged)	Control byte (unchanged)	Serial frame (without handshake or similar measures)

5.3 Using the module on the bus controller

Function model 254 "Bus controller" is used by default only by non-configurable bus controllers. All other bus controllers can use other registers and functions depending on the fieldbus used.

For detailed information, see section "Additional information - Using I/O modules on the bus controller" in the X20 user's manual (version 3.50 or later).

5.3.1 CAN I/O bus controller

The module occupies 1 analog logical slot on CAN I/O.

5.4 Interface configuration

5.4.1 Handshake - Configuration

In order to ensure serial communication runs smoothly, the size of the receive buffer being used in the module must be made known. In addition, the user can specify a software- or hardware-side handshake algorithm.



Information:

The registers are described in ["Handshake - Configuration" on page 41](#).

5.4.1.1 Locking/Unlocking the receive buffer

The two registers "Lock" and "Unlock" can be used for "flow control" monitoring of the communication. If the amount of data from the module input exceeds the value of register "Lock", flow control switches to state "Passive". To return to state "Active" or "Ready", the amount of data in the receive buffer must fall below the default value of register "Unlock".



Information:

These registers simulate the behavior of a Schmitt trigger, so the value of register "Lock" must be greater than the value of register "Unlock".

5.4.1.2 RTS/CTS evaluation

The two registers "TxF" and "RxF" can be used to enable flow control for the input or output direction. Communication takes place here via a ring buffer.



Information:

Only one hsh register can be configured for controlling the RTS line.

5.4.1.3 Software handshake

The two registers "Xon" and "Xoff" can be used to initiate a software handshake for flow control. A valid ASCII character must be configured in both registers for this.

Values	Explanation
0 to 255	XOn/Off ASCII characters
65535	No software handshake

Some applications require periodic repetition of the current state for software-side handshakes. The repetition time in milliseconds can be specified for this purpose.

5.4.1.4 Disabling/Enabling the handshake

The two registers "Set" and "Clr" can be used to manually manage the handshake via the application.

By using these registers, the output level of the hardware handshake line RTS can be forcibly held to active or passive.



Information:

Only one hsh register can be configured for controlling the RTS line.

Values	Explanation
0	RTS line freely available for other flow control methods
16	The RTS line is enabled.

5.4.2 Frame - Configuration

Different message end identifiers can be defined in order to correctly form the transmitted Tx frames and correctly interpret the received Rx frames.



Information:

The registers are described in "[Frame - Configuration](#)" on page 43.

5.4.2.1 Hardware-based frame detection

Frame detection on the hardware side can generally be enabled. The RTS line is active as long as data is being transmitted. This Tx framing mode can be used to control external interface converters.

5.4.2.2 Timeout

The message is considered to be terminated when nothing is transferred for the specified duration. The time is specified here in characters to ensure that it is independent of the transfer rate. The number of characters is then multiplied by the time needed to transfer a character.

5.4.2.3 Number of transmit/receive bytes

The message is considered ended as soon as a frame of the defined size in bytes has been transferred. The largest possible frame length corresponds to the receive buffer of 4096 bytes. Larger frames when receiving result in error Receive Overrun.

After transmitting the frame, the configured transmit timeout is observed.

5.4.2.4 Delimiters

Transmit and receive delimiters can be configured. The message is considered ended as soon as one of the defined characters is transferred.

6 Register description

6.1 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 X20 System user's manual.

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

Function models "Stream" and "Cyclic stream" use a module-specific driver for the operating system. The interface can be controlled using library "DvFrame" and reconfigured at runtime.

Function model - Stream

In function model "Stream", the CPU communicates with the module acyclically. The interface is convenient, 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:

- In order to use function models "Stream" and "Cyclic stream", you must be using B&R controllers of type "SG4".
- These function models can only be used in X2X Link and POWERLINK networks.

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Module – Configuration						
-	AsynSize	-				
Status messages – Configuration						
50	CfO_RxStatelgnoreMask	UINT				•
6273	CfO_ErrorID0007	USINT				•
Status messages – Communication						
6145	ErrorByte	USINT	•			
	StartBitError	Bit 0				
	StopBitError	Bit 1				
	ParityError	Bit 2				
	RXoverrun	Bit 3				
6209	ErrorQuitByte	USINT			•	
	QuitStartBitError	Bit 0				
	QuitStopBitError	Bit 1				
	QuitParityError	Bit 2				
	QuitRXoverrun	Bit 3				

6.3 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 module. Serial information 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 9](#)).

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.

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Serial interface - Configuration						
1	phyMode	USINT				•
12	phyBaud	UDINT				•
3	phyData	USINT				•
5	phyStop	USINT				•
7	phyParity	USINT				•
Handshake – Configuration						
66	rxLock	UINT				•
70	rxUnlock	UINT				•
34	hssXOn	UINT				•
38	hssXOff	UINT				•
42	hssPeriod	UINT				•
Frame – Configuration						
74	rxCto	UINT				•
106	txCto	UINT				•
78	rxEomSize	UINT				•
110	txEomSize	UINT				•
N * 4 + 82	rxEomCharN (index N = 0 to 3)	UINT				•
N * 4 + 114	txEomCharN (index N = 0 to 3)	UINT				•
Status messages – Configuration						
50	CfO_RxStatelgnoreMask	UINT				•
6273	CfO_ErrorID0007	USINT				•
Status messages – Communication						
6145	ErrorByte	USINT	•			
	StartBitError	Bit 0				
	StopBitError	Bit 1				
	ParityError	Bit 2				
	RXoverrun	Bit 3				
6209	ErrorQuitByte	USINT			•	
	QuitStartBitError	Bit 0				
	QuitStopBitError	Bit 1				
	QuitParityError	Bit 2				
	QuitRXoverrun	Bit 3				
Flatstream						
225	OutputMTU	USINT				•
227	InputMTU	USINT				•
229	Mode	USINT				•
231	Forward	USINT				•
238	ForwardDelay	UINT				•
128	InputSequence	USINT	•			
N + 128	RxByteN (index N = 1 to 27)	USINT	•			
160	OutputSequence	USINT			•	
N + 160	TxByteN (index N = 1 to 27)	USINT			•	

6.4 Function model 254 - Bus controller

Function model "Bus controller" is a reduced form of function model "Flatstream". Instead of up to 27 Tx / Rx bytes, a maximum of 7 Tx / Rx bytes can be used.

Register	Offset ¹⁾	Name	Data type	Read		Write	
				Cyclic	Non-cyclic	Cyclic	Non-cyclic
Serial interface - Configuration							
257	-	phyMode_CANIO	USINT				•
268	-	phyBaud_CANIO	UDINT				•
259	-	phyData_CANIO	USINT				•
261	-	phyStop_CANIO	USINT				•
263	-	phyParity_CANIO	USINT				•
Handshake – Configuration							
322	-	rxILock_CANIO	UINT				•
326	-	rxIUnlock_CANIO	UINT				•
290	-	hssXOn_CANIO	UINT				•
294	-	hssXOff_CANIO	UINT				•
298	-	hssPeriod_CANIO	UINT				•
Frame – Configuration							
330	-	rxCto_CANIO	UINT				•
362	-	txCto_CANIO	UINT				•
334	-	rxEomSize_CANIO	UINT				•
366	-	txEomSize_CANIO	UINT				•
N*4 + 338	-	rxEomCharN (index N = 0 to 3)	UINT				•
N*4 + 370	-	txEomCharN (index N = 0 to 3)	UINT				•
Status messages – Configuration							
306	-	CfO_RxStateIgnoreMask_CANIO	UINT				•
6273	-	CfO_ErrorID0007	USINT				•
Status messages – Communication							
6145	-	ErrorByte	USINT		•		
		StartBitError	Bit 0				
		StopBitError	Bit 1				
		ParityError	Bit 2				
		RXoverrun	Bit 3				
6209	-	ErrorQuitByte	USINT				•
		QuitStartBitError	Bit 0				
		QuitStopBitError	Bit 1				
		QuitParityError	Bit 2				
		QuitRXoverrun	Bit 3				
FlatStream							
225	-	OutputMTU	USINT				•
227	-	InputMTU	USINT				•
229	-	Mode	USINT				•
231	-	Forward	USINT				•
238	-	ForwardDelay	UINT				•
128	0	InputSequence	USINT	•			
N + 128	N	RxByteN (index N = 1 to 7)	USINT	•			
160	0	OutputSequence	USINT			•	
N + 160	N	TxByteN (index N = 1 to 7)	USINT			•	

1) The offset specifies the position of the register within the CAN object.

6.5 Serial interface - Configuration

The user has to configure 5 registers to operate the serial interface.

6.5.1 Mode

Name:

phyMode

phyMode_CANIO

This register is used to determine the current operating mode of the interface.

Enabling the interface is only permitted after complete configuration of the other registers. If parameters need to be changed, the interface must first be disabled.

Data type	Value	Description
USINT	0	Interface disabled (bus controller default setting)
	4	RS422 interface enabled ¹⁾
	5	RS422 interface enables as a bus ²⁾
	6	RS485 interface enabled with echo
	7	RS485 interface enabled without echo

1) Connection between 2 stations

2) Connections between multiple stations possible. Transmit lines connected as with RS485 TriState.

6.5.2 Baud rate

Name:

phyBaud

phyBaud_CANIO

This register sets the baud rate of the interface in bit/s.

Data type	Value	Function
UDINT	1200	1.2 kbaud
	2400	2.4 kbaud
	4800	4.8 kbaud
	9600	9.6 kbaud
	19200	19.2 kbaud
	38400	38.4 kbaud
	57600	57.6 kbaud (bus controller default setting)
	115200	115.2 kbaud

6.5.3 Number of data bits

Name:

phyData

phyData_CANIO

This register is used to specify the number of bits to be transferred for each character.

Data type	Value	Description
USINT	7	7 data bits
	8	8 data bits (bus controller default setting)

6.5.4 Number of stop bits

Name:

phyStop

phyStop_CANIO

This register is used to define the number of stop bits.

Data type	Values	Explanation
USINT	2	1 stop bit (bus controller default setting)
	4	2 stop bits

6.5.5 Type of parity check

Name:

phyParity

phyParity_CANIO

This register is used to define the parity check type. Possible values are ASCII coded.

Data type	Value	Description
USINT	48	"0" - (low) bit is always 0
	49	"1" - (high) bit is always 1
	69	"E" - (even) even parity (bus controller default setting)
	78	"N" - (no) no bit
	79	"O" - (odd) odd parity

6.6 Handshake - Configuration

6.6.1 Locking the receive buffer

Name:

rxILock

rxILock_CANIO

This register is used to configure the upper threshold of the receive buffer.

Data type	Value	Description
UINT	0 to 4095	Upper limit of receive buffer. Bus controller default setting: 1024

6.6.2 Unlocking the receive buffer

Name:

rxIUnlock

rxIUnlock_CANIO

This register is used to configure the lower threshold of the receive buffer.

Data type	Value	Description
UINT	0 to 4095	Lower limit of receive buffer. Bus controller default setting: 512

6.6.3 Turn on software handshake

Name:

hssXOn

hssXOn_CANIO

This register can be used to configure the XOn character. The value 17 is the default, but any other value can also be configured.

Data type	Value	Description
UINT	0 to 255	XOn ASCII character
	65535	No software handshake (bus controller default setting)

6.6.4 Turn off software handshake

Name:

hssXOff

hssXOff_CANIO

This register can be used to configure the XOff character. The value 19 is the default, but any other value can also be configured.

Data type	Value	Description
UINT	0 to 255	XOff ASCII character
	65535	No software handshake (bus controller default setting)

6.6.5 Handshake repetition

Name:

hssPeriod

hssPeriod_CANIO

Some applications require periodic repetition of the current state for software-side handshakes. The repetition time in milliseconds can be specified in this register for this purpose.

Data type	Value	Description
UINT	0	Automatic status repeat disabled
	500 to 10000	Retry interval in ms. Bus controller default setting: 5000

6.7 Frame - Configuration

6.7.1 Terminating when a receive timeout occurs

Name:

rxCto

rxCto_CANIO

This register is used to set the duration until a receive timeout is triggered.

Data type	Value	Description
UINT	0	Function disabled
	1 to 65535	Receive timeout in characters. Bus controller default setting: 4

6.7.2 Terminating when a transmit timeout occurs

Name:

txCto

txCto_CANIO

This register is used to set the duration until a transmit timeout is triggered.

Data type	Value	Description
UINT	0	Function disabled
	1 to 65535	Transmit timeout in characters. Bus controller default setting: 5

6.7.3 Maximum number of bytes received

Name:

rxEomSize

rxEomSize_CANIO

These registers configure the maximum number of bytes in the receive frame.

Data type	Value	Description
UINT	0	Function disabled
	1 to 4096	Configurable receive frame length in characters. Bus controller default setting: 256

6.7.4 Maximum number of bytes transmitted

Name:

txEomSize

txEomSize_CANIO

These registers configure the maximum number of bytes in the transmit frame.

Data type	Value	Description
UINT	0	Function disabled
	1 to 4096	Configurable transmit frame length in characters. Bus controller default setting: 4096

6.7.5 Define receive terminator

Name:

rxEomChar0 to rxEomChar3

rxEomChar0_CANIO to rxEomChar3_CANIO

It is possible to configure a receive terminator for all registers.

The message is considered to be terminated as soon as one of the defined characters is transferred.

Data type	Value	Description
UINT	0 to 255	Frame terminator (ASCII code)
	65535	Function disabled (bus controller default setting)

6.7.6 Define transmit terminator

Name:

txEomChar0 to txEomChar3

txEomChar0_CANIO to txEomChar3_CANIO

It is possible to configure a transmit terminator for all registers.

The message is considered to be terminated as soon as one of the defined characters is transferred.

Data type	Value	Description
UINT	0 to 255	Frame terminator (ASCII code)
	65535	Function disabled (bus controller default setting)

6.8 Status messages - Configuration

The status messages provide the user with information about the current situation in the downstream serial network.

6.8.1 Error detection setting

Name:

CfO_RxStatelgnoreMask

CfO_RxStatelgnoreMask_CANIO

This register has a direct effect on UART operation. Error detection in general can be disabled using the low byte. If error detection is not disabled, the high byte can be used to specify that a detected error should be interpreted as the end of the message.

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

Bit structure:

Bit	Name	Value	Information
0 - 3	Reserved	0	
4	StartBitError	0	Detect invalid start bit (bus controller default setting)
		1	Ignore
5	StopBitError	0	Detect invalid stop bit (bus controller default setting)
		1	Ignore
6	ParityError	0	Detect invalid parity bit (bus controller default setting)
		1	Ignore
7	RXoverrun	0	Detect overflow in receive direction (bus controller default setting)
		1	Ignore
8 - 11	Reserved	0	
12	StartBitError corresponds to the end of the frame (if bit 4 = 0)	0	Indicate error in module only (bus controller default setting)
		1	Also signal end of frame
13	StopBitError corresponds to the end of the frame (if bit 5 = 0)	0	Indicate error in module only (bus controller default setting)
		1	Also signal end of frame
14	ParityError corresponds to the end of the frame (if bit 6 = 0)	0	Indicate error in module only (bus controller default setting)
		1	Also signal end of frame
15	RXoverrun corresponds to the end of the frame (if bit 7 = 0)	0	Indicate error in module only (bus controller default setting)
		1	Also signal end of frame

6.8.2 Forward error to the application

Name:

CfO_ErrorID0007

This register sets which error messages are forwarded to the application.

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

Bit structure:

Bit	Name	Value	Information
0	StartBitError	0	Ignore (bus controller default setting)
		1	Indicating a faulty start bit
1	StopBitError	0	Ignore (bus controller default setting)
		1	Indicating a faulty stop bit
2	ParityError	0	Ignore (bus controller default setting)
		1	Indicating a faulty parity bit
3	RXoverrun	0	Ignore (bus controller default setting)
		1	Indicating an overflow in the receive direction
4 - 7	Reserved	0	

6.9 Status messages - Communication

After configuration is completed, up to four status messages can be evaluated in the application.

6.9.1 Error message status bits

Name:

StartBitError

StopBitError

ParityError

RXoverrun

This register transfers the individual bits that indicate an error. If a error occurs, the corresponding bit is set and maintained until it is acknowledged.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	StartBitError	0	No error
		1	Start bit error occurred ¹⁾
1	StopBitError	0	No error
		1	Stop bit error occurred ¹⁾
2	ParityError	0	No error
		1	Parity bit error occurred ¹⁾
3	RXoverrun	0	No error
		1	Receive buffer overflow occurred ²⁾
4 - 7	Reserved	0	

1) This error can result from things such as mismatched interface configurations or problems with the wiring.

2) This data point reports a receive buffer overrun. The buffer capacity on the module is exhausted and all subsequent data arriving at the interface is lost. An overrun always means that the data received on the module is not read fast enough by the higher-level system. The solution here is to optimize the cycle times of all transfer routes and task classes involved and utilize the available handshake options.

6.9.2 Acknowledging the status bits

Name:

QuitStartBitError

QuitStopBitError

QuitParityError

QuitRXoverrun

This register is used to transfer the individual bits that acknowledge an indicated error state. After one of the bits has been set, it can be reset using the corresponding acknowledgment bit.

If the error is still actively pending, the error status bit is not deleted. The acknowledgment bit can only be reset if the error status bit is no longer set.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	QuitStartBitError	0	No acknowledgment
		1	Acknowledge start bit error
1	QuitStopBitError	0	No acknowledgment
		1	Acknowledge stop bit error
2	QuitParityError	0	No acknowledgment
		1	Acknowledge parity bit error
3	QuitRXoverrun	0	No acknowledgment
		1	Acknowledge receive buffer overflow error
4 - 7	Reserved	0	

6.10 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](#).

6.10.1 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.

6.10.2 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

6.10.3 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.

6.10.4 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.

6.10.5 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		

6.10.6 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

6.10.7 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

6.11 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

6.12 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

6.13 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