Communication in the X20 electronics module

Data sheets

Version: **1.10 (March 2024)**

Order no.: **Communication in the X20 electronics module**

Translation of the original documentation

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

1707054958206-1.10

1 X20CS1012

1.1 General information

The M-Bus master is designed as a single-width module and can be connected anywhere within the X20 I/O system. It can therefore be used decentrally for distributed topologies. The M-Bus master supports transfer rates of 300, 2400 and 9600 bit/s; up to 64 slaves supplied via M-Bus can be connected.

M-Bus (Meter-Bus) is a relatively simple fieldbus for recording consumption data, such as from electricity or heat meters. It is based on a reverse polarity protected two-wire line and works according to the master-slave principle.

- Power supply for up to 64 slaves on the M-Bus
- Decentralized use of the communication interface

1.1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX20	X20 System user's manual
MAEMV	Installation / EMC guide

1.2 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS1012	X20 interface module, 1 M-Bus master interface, integrated slave supply	
	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	
	Terminal blocks	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	

Table 1: X20CS1012 - Order data

1.3 Technical description

1.3.1 Technical data

Order number	X20CS1012
Short description	
Communication module	1 M-Bus master for controlling up to 64 slaves
General information	
B&R ID code	0xCABF
Status indicators	Data transfer, M-Bus power supply, operating state, module status
Diagnostics	
Module run/error	Yes, using LED status indicator and software
Data transfer	Yes, using LED status indicator
M-Bus power supply	Yes, using LED status indicator and software
Power consumption	
Bus	0.2 W
Internal I/O	0.35 W + (Number of slaves * 0.08 W)
Module power dissipation	0.55 W + (Number of slaves * 0.006 W)
Additional power dissipation caused by actuators (resistive) [W]	-
Insulation voltage between M-Bus and X2X Link	500 VDC, 1 min
Certifications	
CE	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
EAC	Yes
Interfaces	
Interface	
Type	M-Bus master
Variant	Connection via 12-pin terminal block X20TB12
Max. distance	See section "M-Bus".
Transfer rate	300, 2400 or 9600 bit/s
Number of slaves	Max. 64
Internal resistance of master	Max. 6 Ω
Bus voltage mark at 0 mA	I/O supply voltage + (11.5 to 13.5) V
Bus voltage drop with space	12 to 13.5 V
Overload shutdown	250 mA ±10%
Bit threshold	6 to 9 mA
Collision threshold	24 to 36 mA
Received readjustment time	Max. 10 s [1)]
Bus cable	Shielded or unshielded
Electrical properties	
Electrical isolation	M-Bus isolated from bus M-Bus not isolated from 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
Ambient conditions	
Temperature	
Operation	
Horizontal mounting orientation	-25 to 60°C
Vertical mounting orientation	-25 to 50°C
Derating	-
Storage	-40 to 85°C
Transport	-40 to 85°C

Table 2: X20CS1012 - Technical data

Order number	X20CS1012
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order 1x terminal block X20TB12 separately. Order 1x bus module X20BM11 separately.
Pitch	12.5$^{+0.2}$ mm

Table 2: X20CS1012 - Technical data

1) After each load change on M-Bus (e.g. switching slaves on or off)

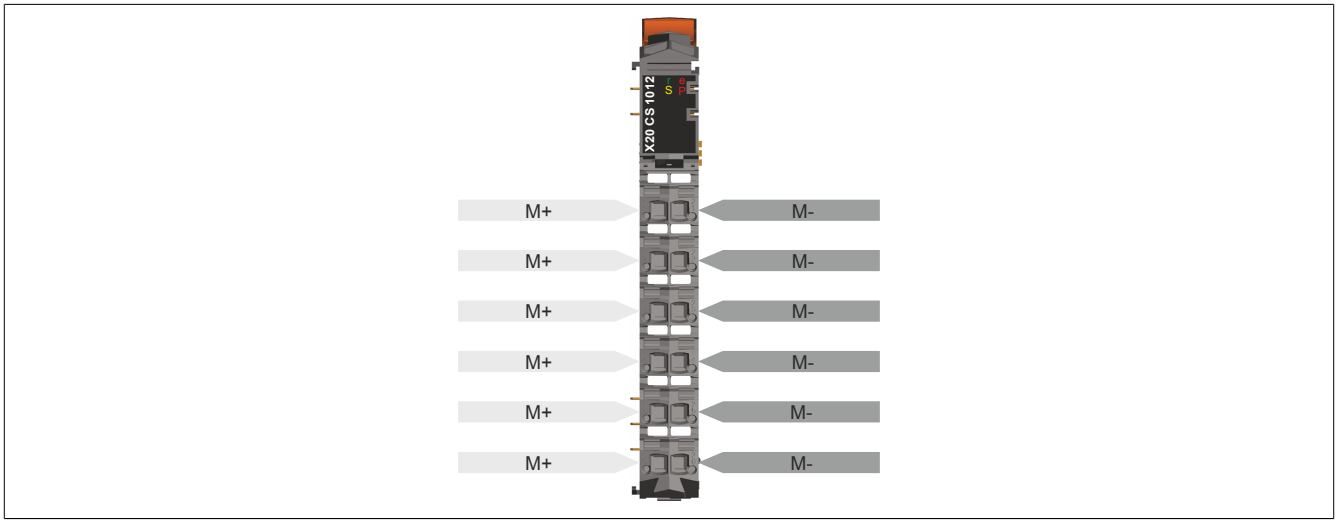
1.3.2 LED status indicators

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

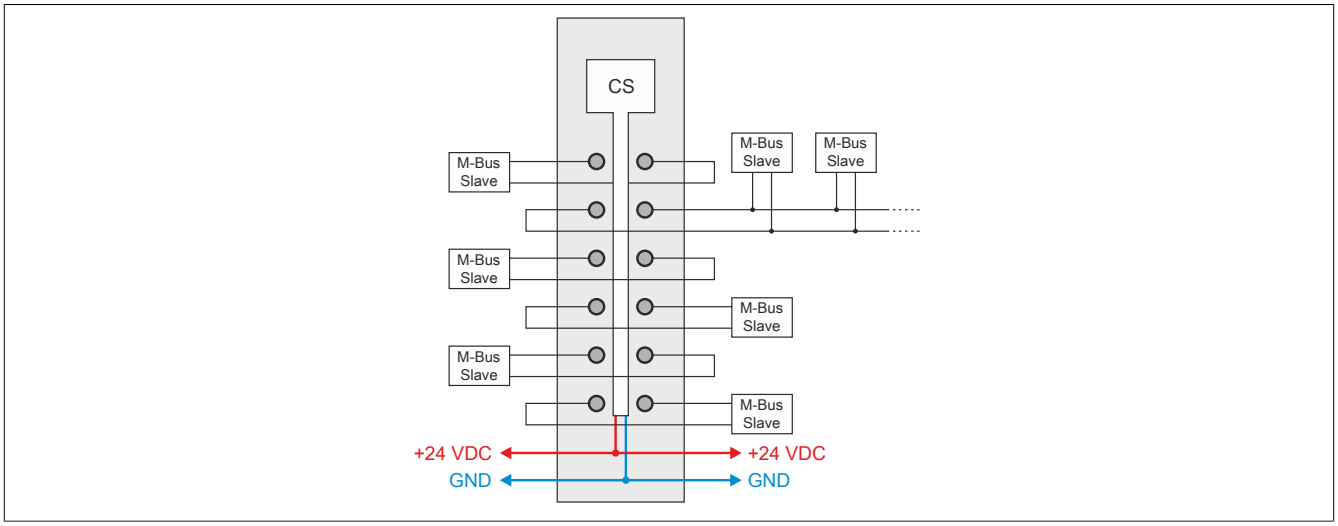
Figure	LED	Color	Status	Description
	r	Green	Off	No power to module
			Single flash	UNLINK mode
			Double flash	Mode BOOT (during firmware update)[1]
			Blinking	PREOPERATIONAL mode
			On	RUN mode
	e	Red	Off	No power to module or everything OK
			On	Error or reset status
	e + r	Red on / Green single flash		Invalid firmware
	S	Yellow	Off	No slaves sending data
			On	At least one slave is sending data via the M-Bus
	P	Red	Off	M-Bus supply ok
			On	Short-circuit or overload on M-Bus

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

1.3.3 Pinout



1.3.4 Connection example



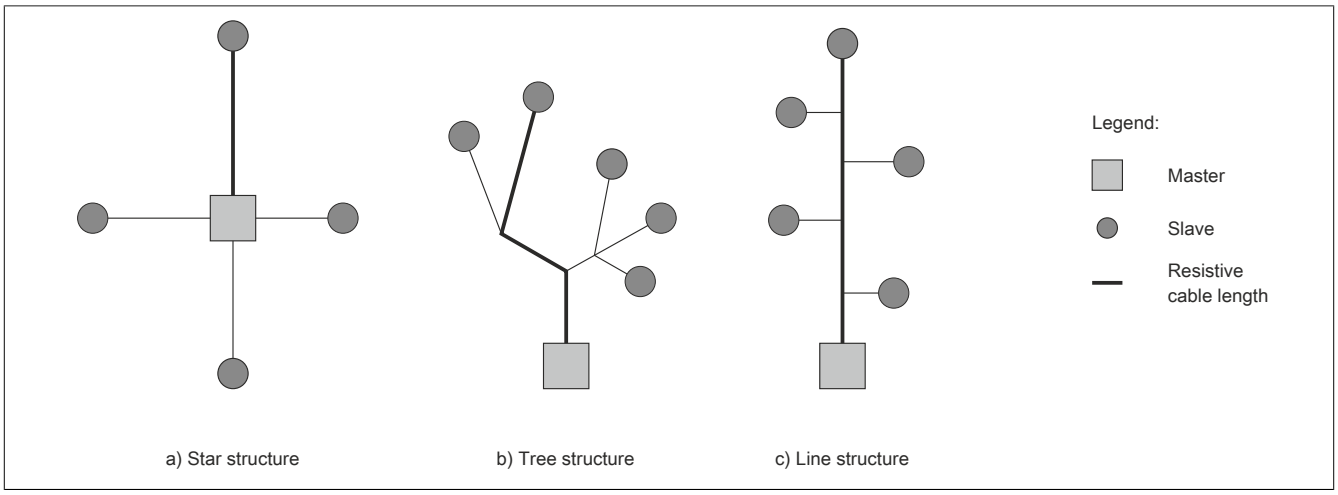
1.3.5 Usage after the X20IF1091-1

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.

1.4 M-Bus

1.4.1 Bus topology

The bus topology has a significant influence on the maximum load of an M-Bus network. In general a star structure is preferred over a tree structure and in turn a tree structure is preferred over a line structure. Furthermore, connecting the slaves to the bus the same way provides better values than connecting them all at the end of the branches after all other parameters have been determined.



1.4.2 Cable cross section

The cable being used has a specific capacity and resistance, which in turn has an effect on the operation of the bus. The resistive influence of the cable means a loss of voltage on the line, which is subsequently not available for supplying the bus. In order to guarantee sufficient power, the voltage on the slaves must never be less than 12 V neither when sending from the master to the slave, nor in the opposite direction. The deciding factor in this case is the longest branch of the network whose length is referred to as the resistive cable length.

The cable's capacity causes signal distortion during data transfer because the slew rates of the rising and falling edges are slowed down. For example, replacing a 3 km branch in a network with two 1.5 km branches will improve the signal. The total distance of the network is referred to as the capacitive cable length (the sum of all segment lengths).

Information:

The maximum permissible line resistance (for the longest loop) is 250 Ω.

The maximum permissible line capacitance for the entire bus is 500 nF.

1.4.3 Transmission current and bit threshold

The bit threshold on the master is typically 7.5 mA. Therefore, a slave transmission current of 15 mA results in the least amount of signal distortion while the highest amount occurs at 11 or 20 mA.

1.4.4 Transfer rate

A lower transfer rate decreases the influence of the signal distortion caused by cable capacity and bit threshold.

Information:

Starting with a total bus length >1 km, the slaves must be operated at a baud rate <9600 bit/s.

1.4.5 Calculating the resistive bus length

The resistive cable length must be calculated in order to ensure a sufficient power supply of 12 V on the M-Bus. What matters most here is the longest segment between the master and slave.

The resistive bus length is calculated using the following formula without taking an increased bus current caused by a defective receiver into account:

$$L_{res} = \frac{V_{I/O} - (n * 0.0015 + 0.02) * 6 - 12.6}{(n * 0.0015 + 0.02) * R_L} * 1000$$

L_{res} ... Resistive bus length [m]
n ... Number of slaves (all at end of line)
R_L ... Line resistance (loop resistance [Ω/km])
$V_{I/O}$ ... I/O supply voltage [V]

Examples for calculating the maximum resistive bus length:

No.	Example	Maximum resistive bus length
1	• 64 slaves (all at end of line) • 19.2 V I/O supply voltage • 0.5 mm² wire cross-section	675 m
2	• 64 slaves (all at end of line) • 28.8 V I/O supply voltage • 1.5 mm² wire cross-section	5340 m

1.4.6 Accounting for the capacitive bus length

The total distance of the network is referred to as the capacitive bus length (the sum of all segment lengths). The capacitive bus length depends on two factors:

- Distributed capacitance of cable
- Transfer rate

Distributed capacitance of cable

A lower distributed capacitance on a cable means a higher capacitive bus length.

Transfer rate

A lower transfer rate on an M-Bus system means a higher capacitive bus length.

Example of a cable with a distributed capacitance of 50 nF/km:

Transfer rate	Capacitive bus length
9600 bit/s	1 km
2400 bit/s	4 km
300 bit/s	10 km

1.4.7 Bus installation

Cables with twisted pair wires and a cross-section of 0.5 mm² to 1.5 mm² are normally used for bus installation (according to standard: J-Y(ST)Y nx2x0.8). The shield on shielded cables only has to be grounded to the module on one side. On the slaves, the shielding must be high resistance for DC and low frequency signals.

1.4.8 Repeater

Repeaters can be used to further expand the M-Bus network.

1.5 Register description

1.5.1 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 X20 system user's manual.

1.5.2 Function model 0 - Standard

Register	Name	Data type	Read		Write	
			Cyclic	Non-cyclic	Cyclic	Non-cyclic
Module configuration						
774	CfO_FunctionModel	UINT				•
M-Bus - Configuration						
Index * 16 + 767	CfO_LengthData1 to CfO_LengthData8	USINT				•
Index * 16 + 775	CfO_BaudData1 to CfO_BaudData8	USINT				•
Index * 16 + 761	CfO_PAdrData1 to CfO_PAdrData8	USINT				•
Index * 16 + 765	CfO_IndexData1 to CfO_IndexData8	USINT				•
Index * 16 + 773	CfO_ReqTimeData1 to CfO_ReqTimeData8	USINT				•
Index * 16 + 770	CfO_MBusModeData1 to CfO_MBusModeData8	UINT				•
Index * 16 + 763	CfO_ToutOffData1 to CfO_ToutOffData8	USINT				•
Index * 8 + 1009	CfO_ReplData1 to CfO_ReplData8	(U)SINT				•
Index * 8 + 1010	CfO_ReplData1 to CfO_ReplData8	(U)INT				•
Index * 8 + 1012	CfO_ReplData1 to CfO_ReplData8	(U)DINT REAL				•
M-Bus - Communication						
513	MBusCommand	USINT			•	•
263	MBusOperation	USINT	•			
257	MBusState	USINT	•			
259	ValidDataByte	USINT	•			
	ValidData1	Bit 0	•			
	...	...	•			
	ValidData8	Bit 7	•			
261	InvalidDataByte	USINT	•			
	InvalidData1	Bit 0	•			
	...	...	•			
	InvalidData8	Bit 7	•			
Index * 8 + 265	Data1 to Data8	(U)SINT	•			
Index * 8 + 266	Data1 to Data8	(U)INT	•			
Index * 8 + 268	Data1 to Data8	(U)DINT REAL	•			
337	ChangedSNByte	USINT	•			
Index * 8 + 900	SNData1 to SNData8	UDINT		•		
FlatStream						
2051	InputMTU	USINT				•
2049	OutputMTU	USINT				•
2113	InputSequence	USINT	•			
Index * 2 + 2113	RxByte1 to RxByte15	USINT	•			
2177	OutputSequence	USINT			•	
Index * 2 + 2177	TxByte1 to TxByte15	USINT			•	
2053	FlatstreamMode	USINT				•
2055	Forward	USINT				•
2057	ForwardDelay	UINT				•

1.5.3 Function model 254 - Bus controller

Register	Offset [1]	Name	Data type	Read		Write	
				Cyclic	Non-cyclic	Cyclic	Non-cyclic
Module configuration							
774	-	CfO_FunctionModel	UINT				•
M-Bus - Configuration							
Index * 16 + 767	-	CfO_LengthData1 to CfO_LengthData8	USINT				•
Index * 16 + 775	-	CfO_BaudData1 to CfO_BaudData8	USINT				•
Index * 16 + 761	-	CfO_PAdrData1 to CfO_PAdrData8	USINT				•
Index * 16 + 765	-	CfO_IndexData1 to CfO_IndexData8	USINT				•
Index * 16 + 773	-	CfO_ReqTimeData1 to CfO_ReqTimeData8	USINT				•
Index * 16 + 770	-	CfO_MBusModeData1 to CfO_MBusMode-Data8	UINT				•
Index * 16 + 763	-	CfO_ToutOffData1 to CfO_ToutOffData8	USINT				•
Index * 8 + 1009	-	CfO_ReplData1 to CfO_ReplData8	(U)SINT				•
Index * 8 + 1010	-	CfO_ReplData1 to CfO_ReplData8	(U)INT				•
Index * 8 + 1012	-	CfO_ReplData1 to CfO_ReplData8	(U)DINT REAL				•
M-Bus - Communication							
8	8	MBusCommand	USINT			•	•
11	11	MBusOperation	USINT	•			
8	8	MBusState	USINT	•			
9	9	ValidDataByte	USINT	•			
10	10	InvalidDataByte	USINT	•			
Index * 4 + 5	Index * 4 + 8	Data1 to Data8	(U)SINT	•			
Index * 4 + 6	Index * 4 + 8	Data1 to Data8	(U)INT	•			
Index * 4 + 8	Index * 4 + 8	Data1 to Data8	(U)DINT REAL	•			
337	-	ChangedSNByte	USINT		•		
Index * 8 + 900	-	SNData1 to SNData8	UDINT		•		
FlatStream							
2051	-	InputMTU	USINT				•
2049	-	OutputMTU	USINT				•
0	0	InputSequence	USINT	•			
Index * 1 + 0	Index * 1 + 0	RxByte1 to RxByte7	USINT	•			
0	0	OutputSequence	USINT			•	
Index * 1 + 0	Index * 1 + 0	TxByte1 to TxByte7	USINT			•	
2053	-	FlatstreamMode	USINT				•
2055	-	Forward	USINT				•
2057	-	ForwardDelay	UINT				•

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

1.5.3.1 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).

1.5.3.2 CAN I/O bus controller

The module occupies 3 analog logical slots with CAN I/O.

1.5.4 General information

The M-Bus standard is a serial bus system that handles half-duplex or asynchronous communication. The high level of variability provided by this protocol enables a wide range of information to be handled via the same interface. In basic M-Bus networks, the master communicates with up to 250 slaves via the "primary address". In later stages of development, the secondary address (4 bytes) was then also specified. This made it possible to significantly increase the number of slaves in a network.

Important information about the module

- Generally: Primary address used (1 to 250)
- Work with the secondary address is supported only by Flatstream.
- Bus can supply 64 slaves with power

1.5.5 Module configuration

The flexible design of the M-Bus protocol can quickly add up to a lot of configuration work. That's why B&R offers two different user interfaces for the module: "Standard" and "FlatStream". The user-friendly B&R Standard interface allows users to view up to eight values requested cyclically from the M-Bus network. In FlatStream mode, the module acts as a bridge between the PLC and the M-Bus slave, which makes all M-Bus functions available.

Information:

The B&R Standard interface is statically configured and based on cyclic registers. Because X2X Link can only transfer a certain number of values cyclically, the user must make his selection accordingly.

1.5.5.1 Settings for operation

Name:
CfO_FunctionModel

This register can be used to enable either the Standard or FlatStream interface, which makes the module much more efficient.

Bits 8 to 15 are only evaluated if bit 0 (B&R default interface) is enabled.

Data type	Values	Bus controller default setting
USINT	See the bit structure.	1825

Bit structure:

Bit	Description	Value	Information
0	B&R standard interface	0	Disabled
		1	Enabled (bus controller default setting)
1	Flatstream	0	Disabled (bus controller default setting)
		1	Enabled
3 - 7	Reserved	0	
8	Data1	0	Disabled
		1	Enabled (bus controller default setting)
...		...	
10	Data 3	0	Disabled
		1	Enabled (bus controller default setting)
11	Data 4	0	Disabled (bus controller default setting)
		1	Enabled
...		...	
15	Data8	0	Disabled (bus controller default setting)
		1	Enabled

1.5.6 M-Bus - Configuration

Separate configuration registers are provided for each value to read. These must be configured correctly in order to call up a counter value from the M-Bus network. The user must know the following values from the slave:

- Transfer rate configured on the slave
- Primary address configured on the slave (value: 1 to 250, otherwise only point-to-point connection is possible)
- Data type / data length of value
- How the slave's memory is structured

Information:

The following section "M-Bus - Configuration" is based solely on the B&R standard interface.

1.5.6.1 Data length

Name:
CfO_LengthData1 to CfO_LengthData8

The Standard interface is able to request data from the M-Bus slave with different lengths. When using Automation Studio the value of the "Length" register is a result of the data type defined for the X2X Link. All common data types with up to 4 bytes in length are supported.

Data type	Value	Bus controller default setting
USINT	See the bit structure.	8

Bit structure:

Bit	Name	Value	Information
0 - 5	Data length code	00 0000	USINT
		00 0001	SINT
		00 0010	UINT
		00 0100	INT
		00 1000	UDINT (bus controller default setting)
		01 0000	DINT
		10 0000	REAL
6 - 7	Reserved	0	

1.5.6.2 Transfer rate

Name:
CfO_BaudData1 to CfO_BaudData8

This register can be used to define the transfer rate for retrieving the desired values.

Data type	Value	Bus controller default setting
USINT	See the bit structure.	4

Bit structure:

Bit	Name	Value	Information
0 - 3	Baud rate (Code)	0000	Reserved!
		0001	300 bit/s
		0010	600 bit/s
		0011	1200 bit/s
		0100	2400 bits/s (bus controller default setting)
		0101	4800 bit/s
		0110	9600 bit/s
		0111	19200 bit/s
		1000	38400 bit/s
4 - 7	Reserved	0	

1.5.6.3 Address

Name:
CfO_PAdrData1 to CfO_PAdrData8

This register can be used to define the address where the desired values will be requested from.

Data type	Value	Information
USINT	1 to 250	Bus controller default setting: 254

Special addresses:

Value	Information
251 to 253	Reserved (in accordance with M-Bus specification)
254	Broadcast address (response from all connected slaves - risk of collision)

1.5.6.4 Index

Name:
CfO_IndexData1 to CfO_IndexData8

This register is used to specify the ordinal number of the value (independent of the medium). This value results from the order of values in the slave. The value is then transferred to the data register.

Data type	Value	Information
USINT	1 to 255	Bus controller default setting: 1

1.5.6.5 Specifies the refresh time

Name:
CfO_ReqTimeData1 to CfO_ReqTimeData8

The slave information can be queried manually or time-based. A time-controlled query must include the value for the refresh time. The associated unit is defined in register "M-Bus mode" on page 13.

Data type	Value	Information
USINT	1 to 255	In [s, min]. Bus controller default setting: 0

1.5.6.6 M-Bus mode

Name:
CfO_MBusModeData1 to CfO_MBusModeData8

To speed up the module's boot procedure, various configuration details that define the module's behavior have been combined in this register.

Data type	Values	Bus controller default setting
UINT	See the bit structure.	2

Bit structure:

Bit	Description	Value	Information
0 - 2	Byte offset	0 - 7	Data types. Bus controller default setting: 2
3 - 4	Reserved	0	
5	InitFrame	0	No additional frame (bus controller default setting)
		1	Transmit additional frame
6	ApplicationResetFrame	0	No additional frame (bus controller default setting)
		1	Transmit additional frame
7	Replacement value strategy	0	Hold last valid value (bus controller default setting)
		1	Replace with static value
8	Time-based reading	0	Disabled (bus controller default setting)
		1	Enabled
9	Manually triggered reading	0	Disabled (bus controller default setting)
		1	Reading via register "MBusCommand" on page 15
10	Unit of periodic reading	0	[s] - Seconds (bus controller default setting)
		1	[min] - Minutes
11 - 15	Reserved	0	

Byte offset

The M-Bus specification allows for many individual data types. In order to also read these counter values with up to 64 bits, a slave value may have to be read using 2 data registers. The byte offset can be defined to select a desired section of the information.

1.5.6.7 Timeout offset

Name:
CfO_ToutOffData1 to CfO_ToutOffData8

The timeout for the M-Bus communication generally depends on the currently defined transfer rate. The user can also define an offset value in addition to the calculated standard timeout.

Timeout = standard timeout + (timeout offset * 10 ms)

Data type	Value	Information
USINT	0 to 255	Resolution 10 ms. Bus controller default setting: 0

1.5.6.8 Static replacement value

Name:
CfO_ReplData1 to CfO_ReplData8

This register defines the static replacement value if replacement value strategy "Replace with static value" was enabled in register "CfO_MBusModeData" on page 13. The data register takes on this value if an invalid input value is detected.

Data type	Value	Information
(U)SINT (U)INT (U)DINT REAL	According to data type	Bus controller default setting: 0

1.5.7 M-Bus - Communication

Three important control and status bytes are provided in the B&R interface for communication with the M-Bus slaves. Register "MBusCommand" on page 15 switches UART on/off, for example, in order to increase the system's energy efficiency.

Up to 8 cyclic input registers are registered depending on the configuration. Manually configured data must be requested via register "MBusCommand" on page 15. Registers "ValidDataByte" on page 16 and "Invalid-DataByte" on page 16 can be used to determine the quality of the value currently read.

Information:

The following section "M-Bus - Communication" is based solely on the B&R standard interface.

1.5.7.1 M-Bus commands

Name:
MBusCommand

This register can be used to apply different commands to the module. The module only responds to positive edges.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	Activate UART	0 → 1	Execute command
1	Read manually triggered values	0 → 1	Execute command
2	Acknowledge the "MBusState" register	0 → 1	Execute command
3 - 6	Reserved	0	
7	Deactivate UART	0 → 1	Execute command

Bit 0 and 7

The level converter is switched on by default when the module boots. This bit can be used to switch it on or off from the application to save electricity, for example.

1.5.7.2 M-Bus operation

Name:
MBusOperation

This register shows the user which task the module is currently processing. The LSB is always set when the UART is active. Manual commands are indicated by an increase of one in this byte while being processed.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	UART	0	Inactive
		1	Active
1	Read values	0	-
		1	Command being processed
2	Refresh/reset the "MBusState" register	0	-
		1	Command being processed[1)]
3 - 6	Reserved	0	
7	UART	0	Inactive
		1	Active

1) Bit 2 is only set for one X2X cycle. Requesting this bit is not recommended when operating the module behind a bus controller.

1.5.7.3 M-Bus state

Name:
MBusState

This register contains the current M-Bus network error state. All bits are managed in nonvolatile memory. This means they must be reset via register "MBusCommand" on page 15.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Collision detection	0	Error-free addressing
		1	Multiple addresses on the bus
1	Read error (at least once)	0	Configured information OK
		1	Could not read information
2	Checksum	0	Received checksum OK
		1	Error in input direction
3	M-Bus load	0	Power supply OK
		1	Load too high on M-Bus network
4	Communication aborted due to overflow	0	Everything OK
		1	The master is overloaded and cannot take on any additional requests. Corrective measure: Repeat the request.[1)]
5	Communication aborted due to level converter	0	Everything OK
		1	Level converter is OFF (aborted at runtime or not started).
6	Data exchange since startup	0	Valid data not yet received
		1	Valid data at least once
7	UART off MBUS ENABLE	0	M-Bus driver, level converter inactive
		1	Module ready for communication

1) Communication is reestablished automatically as soon as the pending communication jobs have been processed.

1.5.7.4 Valid data

Name:
ValidDataByte

ValidData1 to ValidData8

This register indicates (by bit) which of the max. 8 read values are valid.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	ValidData1	0	Value 1 invalid
		1	Value 1 valid
...	...	...	...
7	ValidData8	0	Value 8 invalid
		1	Value 8 valid

1.5.7.5 Invalid data

Name:
InvalidDataByte

InvalidData1 to InvalidData8

The validity of the read values can be checked redundantly. This register indicates (by bit) which of the max. 8 read values are invalid.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	InvalidData1	0	Value 1 valid
		1	Value 1 invalid
...	...	...	...
7	InvalidData8	0	Value 8 valid
		1	Value 8 invalid

1.5.7.6 Data

Name:
Data1 to Data8

Each cyclic data register contains the respective pre-configured value from the M-Bus network. The data type of the data register was designed to be variable and must be specified by the user during configuration.

Information:

Because X2X Link can only transfer a certain number of bytes cyclically, the user must make his selection accordingly.

Data type	Value
(U)SINT (U)INT (U)DINT REAL	According to data type

1.5.7.7 Change the serial number of an M-bus slave

Name:
ChangedSNByte

This register indicates (by bit) whether one of the M-Bus slave serial numbers on the bus has changed. Only the serial numbers of the slaves accessed via the B&R interface are checked. The respective bit is toggled if a change is detected.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	SN (Slave 1)	0 -> 1	Slave1: Serial number changed
		1 -> 0	
...	...	...	...
7	SN (Slave 8)	0 -> 1	Slave8: Serial number changed
		1 -> 0	

1.5.7.8 M-Bus slave serial numbers

Name:
SNData1 to SNData8

These registers contain the serial numbers of the M-Bus slaves that are queried via the B&R interface. They are implemented acyclically and can be read using library AsIOAcc.

Data type	Values
UDINT	0 to 4,294,967,295

1.5.8 Flatstream communication

1.5.8.1 Introduction

B&R offers an additional communication method for some modules. "Flatstream" was designed for X2X and POWERLINK networks and allows data transmission 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.

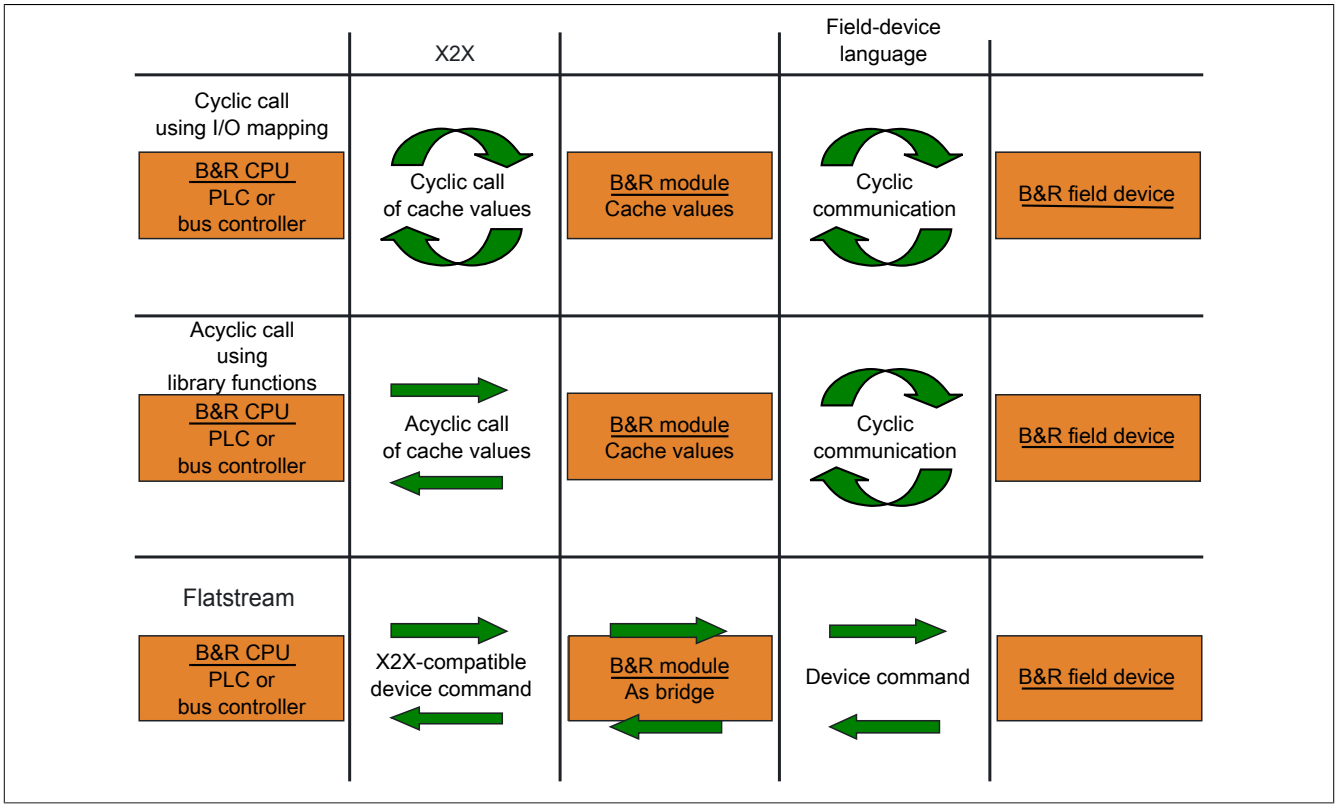


Figure 1: 3 types of communication

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

1.5.8.2 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 put back 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 transmission was faulty and that all affected sequences must be repeated.

1.5.8.3 The Flatstream principle

Requirement

Before Flatstream can be used, the respective communication direction must be synchronized, i.e. both communication partners cyclically query the sequence counter on the opposite 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 opposite 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.

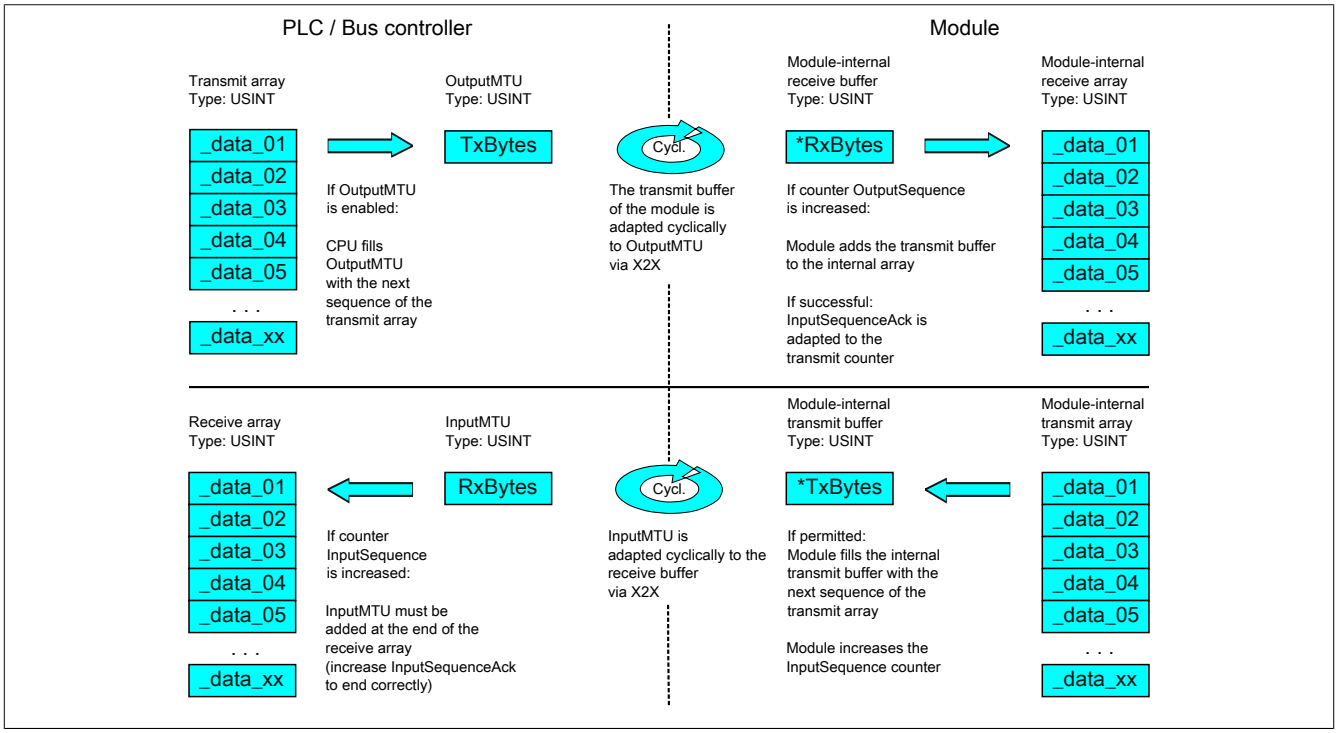


Figure 2: Flatstream communication

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 opposite 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.

1.5.8.4 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 CPU 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.

1.5.8.4.1 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.

1.5.8.4.1.1 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.

Information:

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

Data type	Values
USINT	See the module-specific register overview (theoretically: 3 to 27).

1.5.8.4.2 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 and ensure that communication is taking place properly, i.e. the transmitter issues the directive that the data should be accepted and the receiver acknowledges that a sequence has been transferred successfully.

1.5.8.4.2.1 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.

1.5.8.4.2.2 Transport of 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.
In the user program, only the Tx and Rx bytes from the CPU 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 CPU.

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

Data type	Values
USINT	0 to 255

1.5.8.4.2.3 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

1.5.8.4.2.4 Communication status of the CPU

Name:
OutputSequence

Register "OutputSequence" contains information about the communication status of the CPU. It is written by the CPU 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 disabled
		1	Output direction enabled
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 CPU. The CPU uses OutputSequenceCounter to direct the module to accept a sequence (the output direction must be synchronized when this happens).

OutputSyncBit

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

InputSequenceAck

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

InputSyncAck

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

1.5.8.4.2.5 Communication status of the module

Name:
InputSequence

Register "InputSequence" contains information about the communication status of the module. It is written by the module and should only be read by the CPU.

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 CPU 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 CPU. This indicates that the module is ready to receive data.

1.5.8.4.2.6 Relationship between OutputSequence and InputSequence

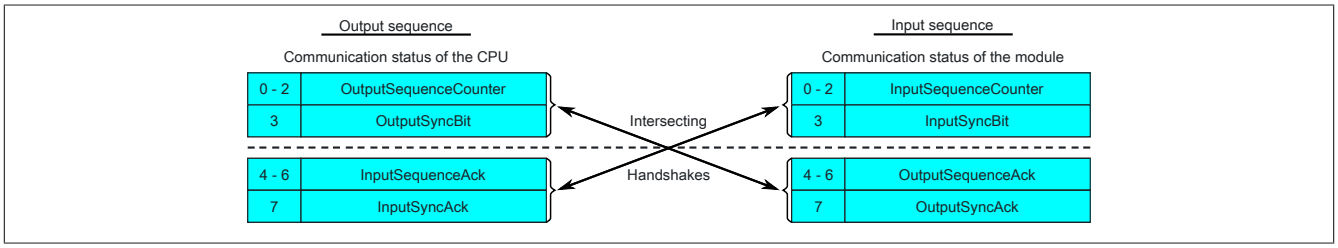


Figure 3: Relationship between OutputSequence and InputSequence

Registers "OutputSequence" and "InputSequence" are logically composed of 2 half-bytes. The low part signals to the opposite station whether a channel should be opened or if 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 opposite 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 opposite 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.

1.5.8.4.3 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 (CPU 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 CPU to the module.

Algorithm

1) The CPU must write 000 to OutputSequenceCounter and reset OutputSyncBit. The CPU 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 CPU registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The CPU 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 CPU registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The CPU 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 CPU can transmit data to the module.

Synchronization in the input direction (CPU 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 CPU.

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 CPU is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The CPU has to 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 CPU is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The CPU has to 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 CPU 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 CPU.

1.5.8.4.4 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.

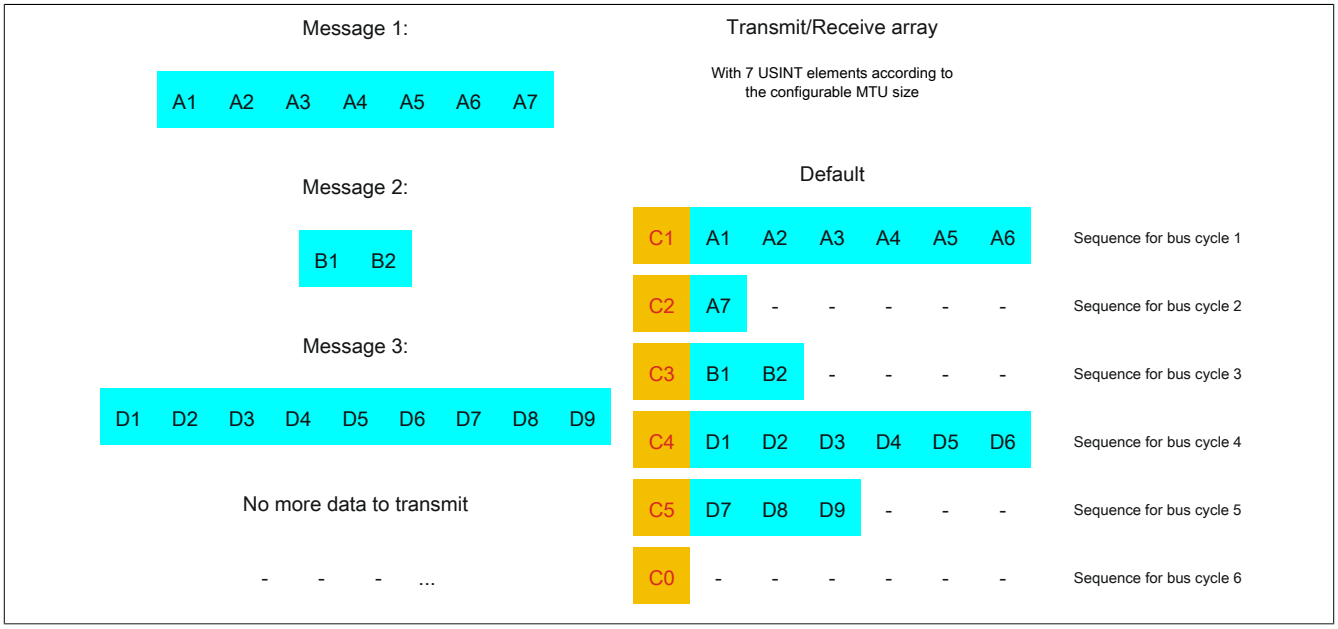


Figure 4: Transmit/Receive array (default)

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)

1.5.8.4.5 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.

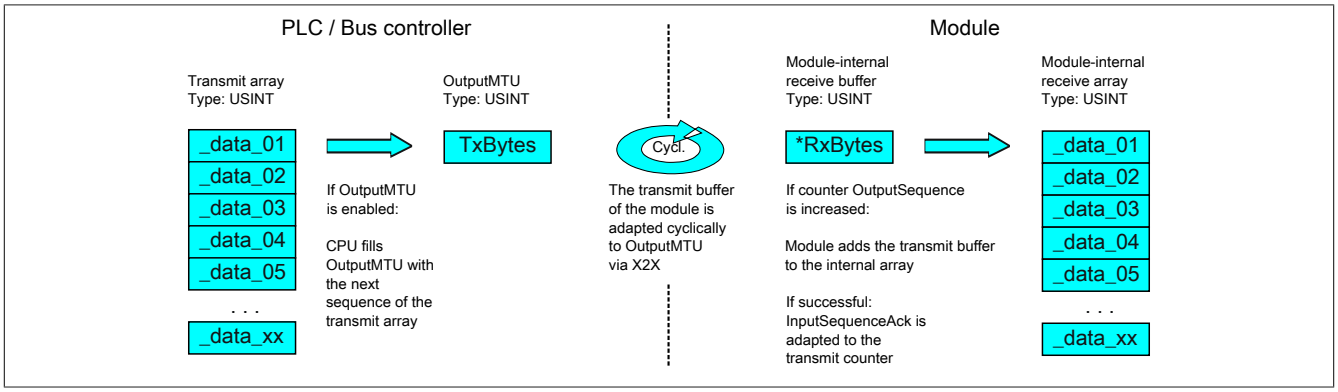


Figure 5: Flatstream communication (output)

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 CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU 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 CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU 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 CPU 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 CPU 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 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

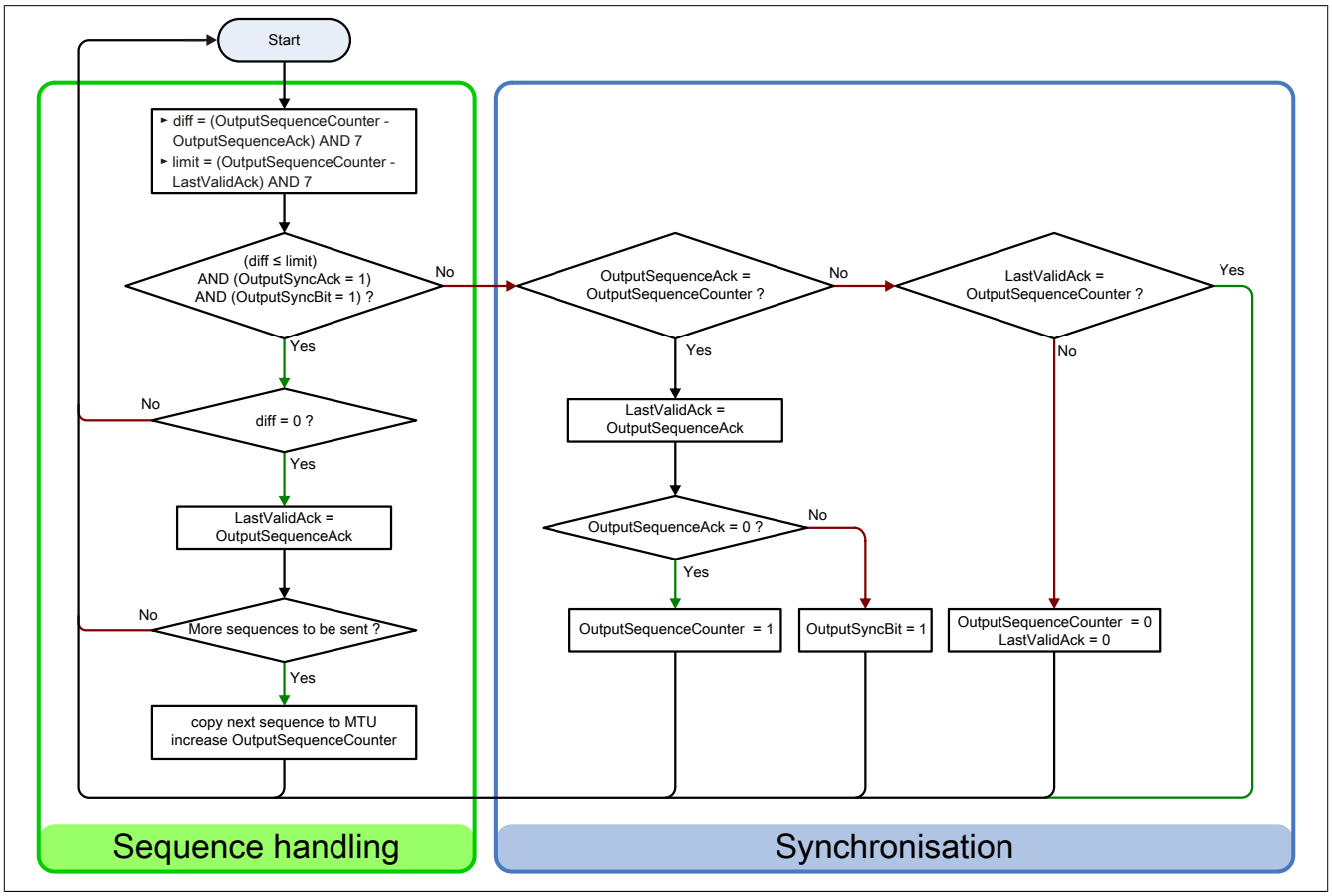


Figure 6: Flowchart for the output direction

1.5.8.4.6 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.

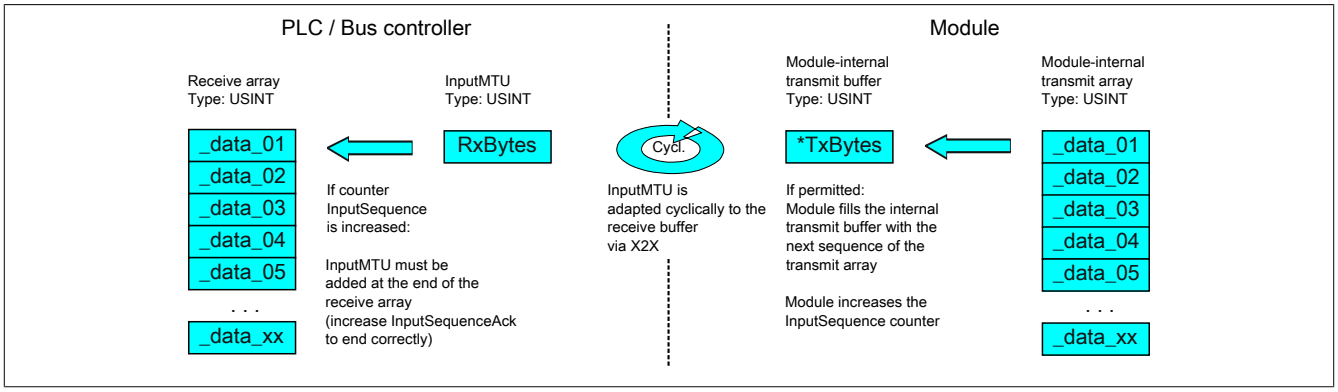


Figure 7: Flatstream communication (input)

Algorithm

0) Cyclic status query: - The CPU 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 CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU 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

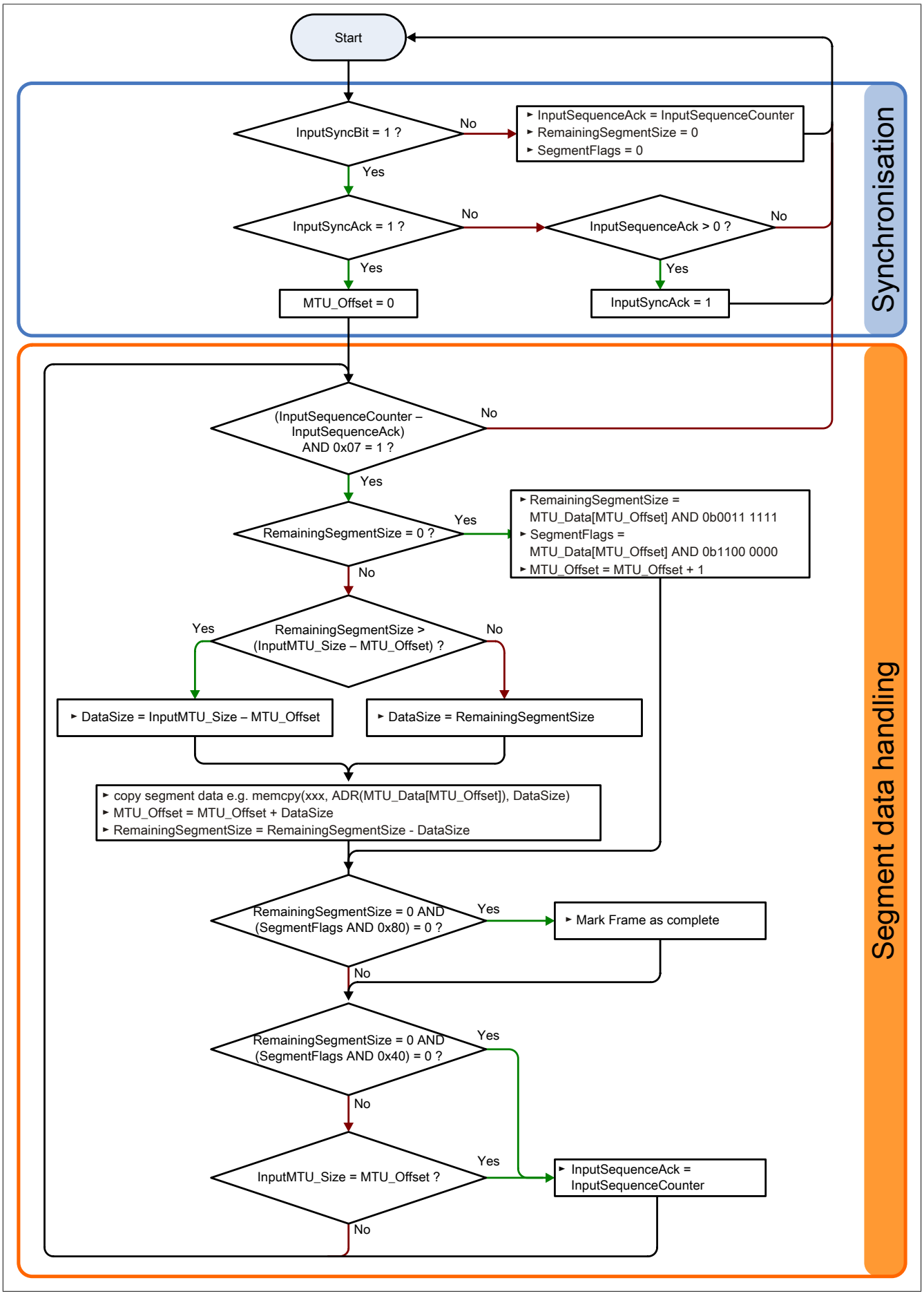


Figure 8: Flowchart for the input direction

1.5.8.4.7 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.

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

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 opposite station's SequenceAck and continue the transfer from this point.

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.

1.5.8.4.8 Flatstream mode

Name:
FlatstreamMode

In the input direction, the transmit array is generated automatically. This register offers 2 options to the user that allow an incoming data stream to have a more compact arrangement. 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.

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		

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.

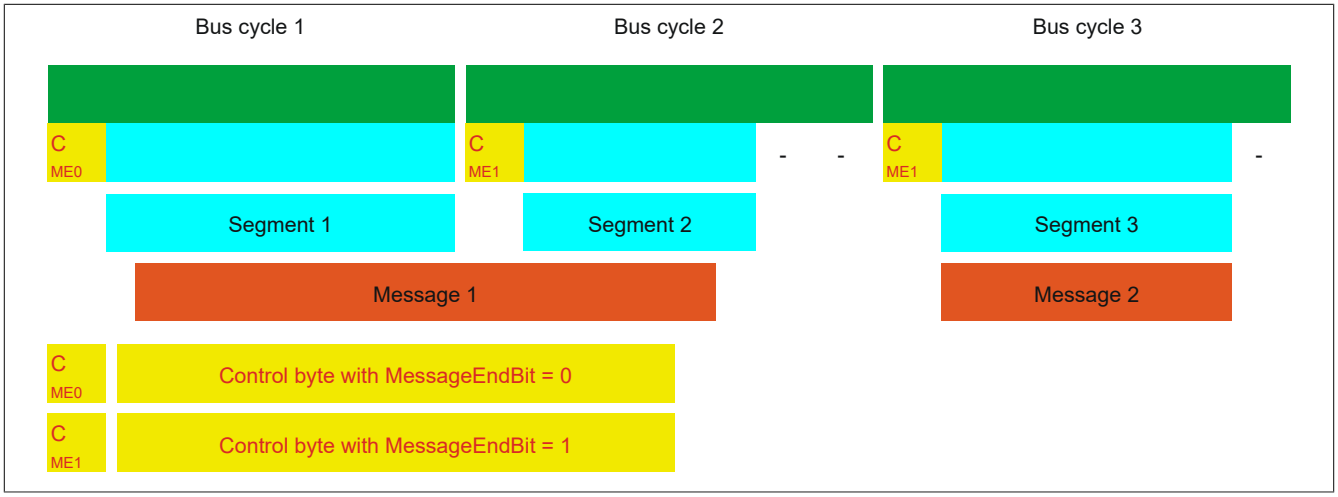


Figure 9: Message arrangement in the MTU (default)

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.

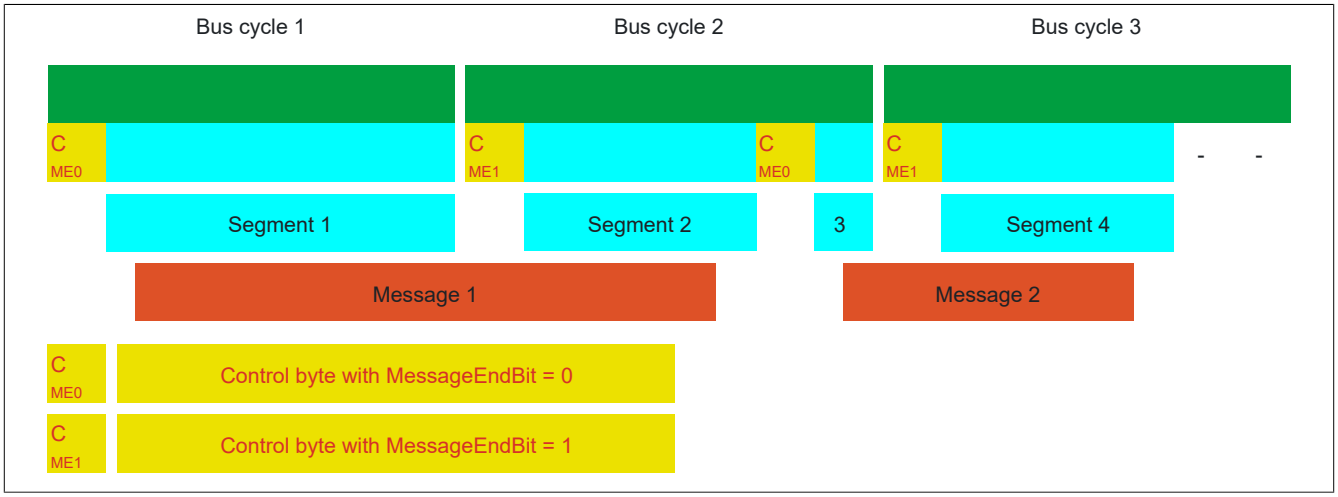


Figure 10: Arrangement of messages in the MTU (MultiSegmentMTUs)

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.

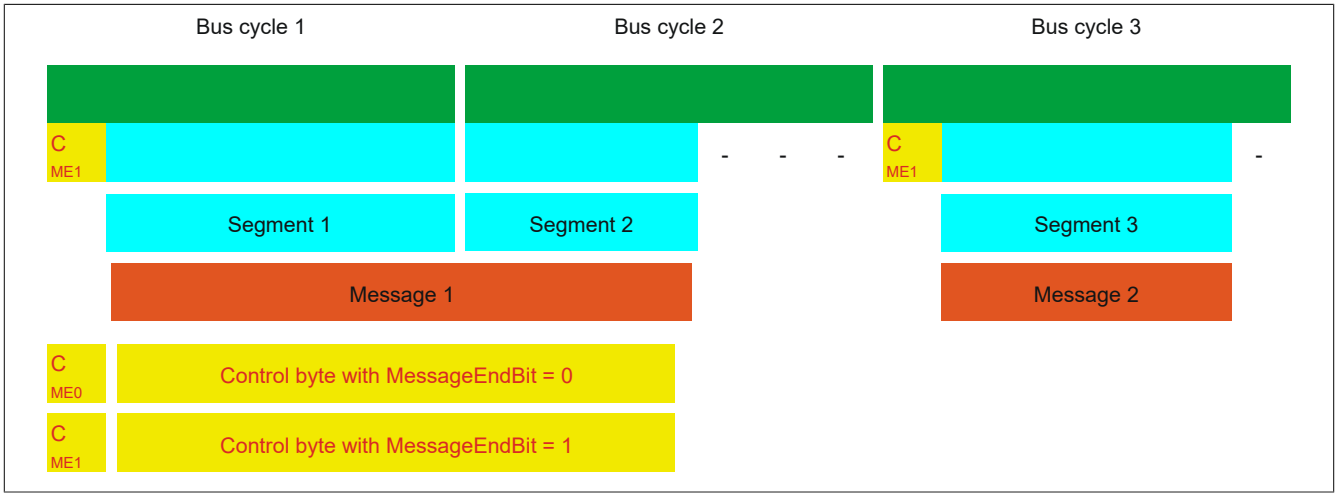


Figure 11: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

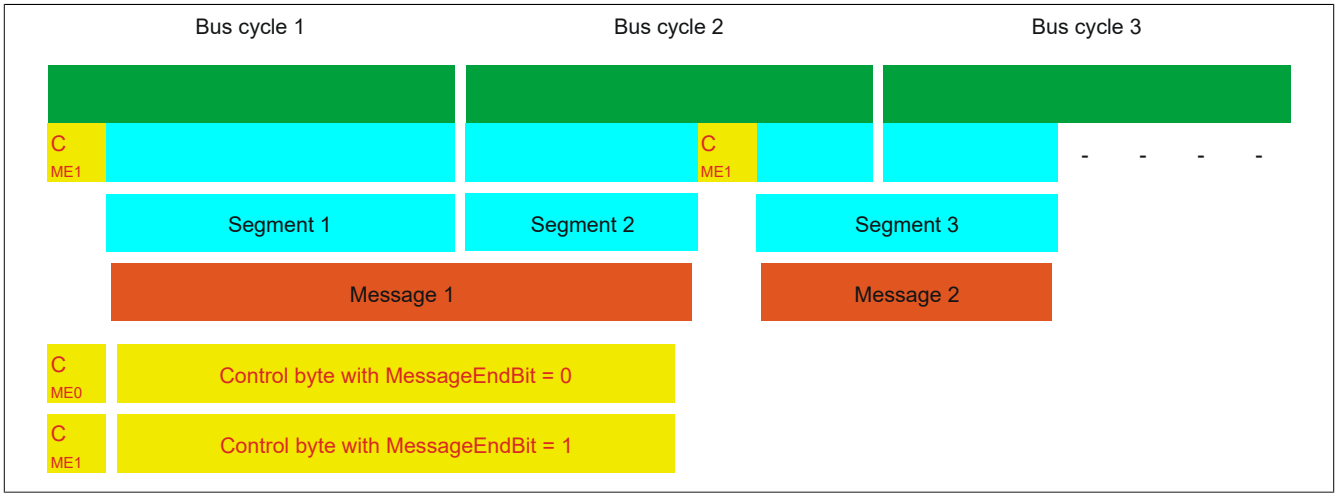


Figure 12: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

1.5.8.4.9 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.

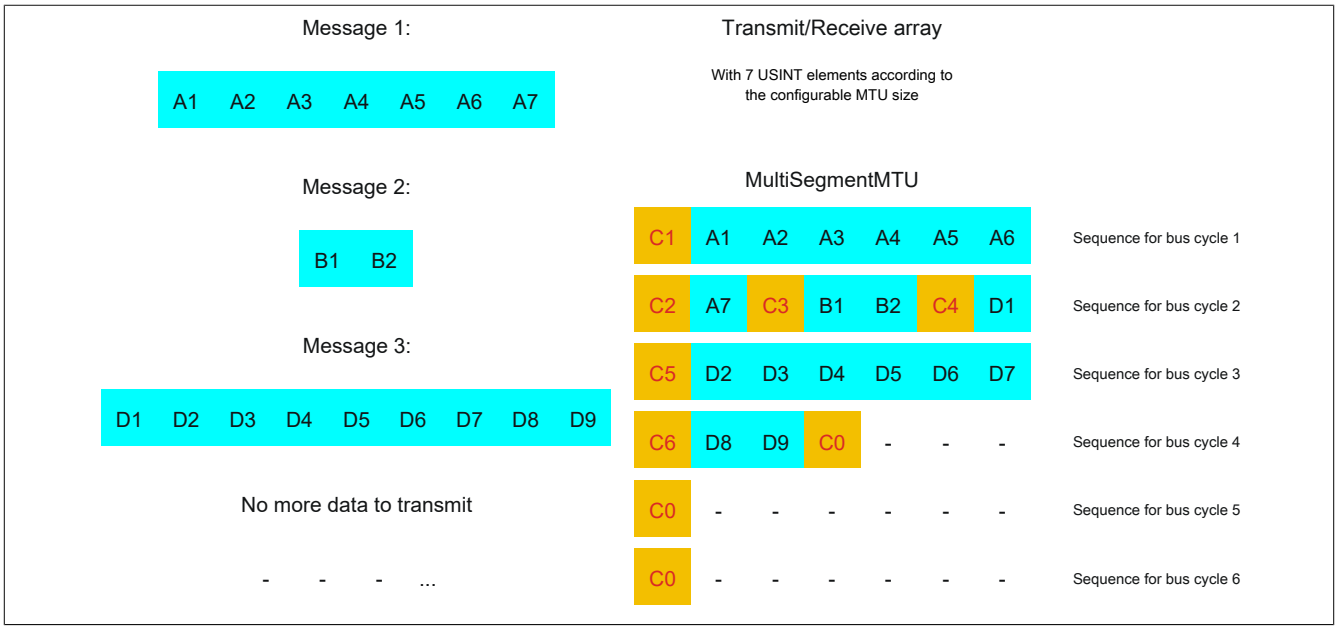


Figure 13: Transmit/receive array (MultiSegmentMTUs)

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 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.

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.

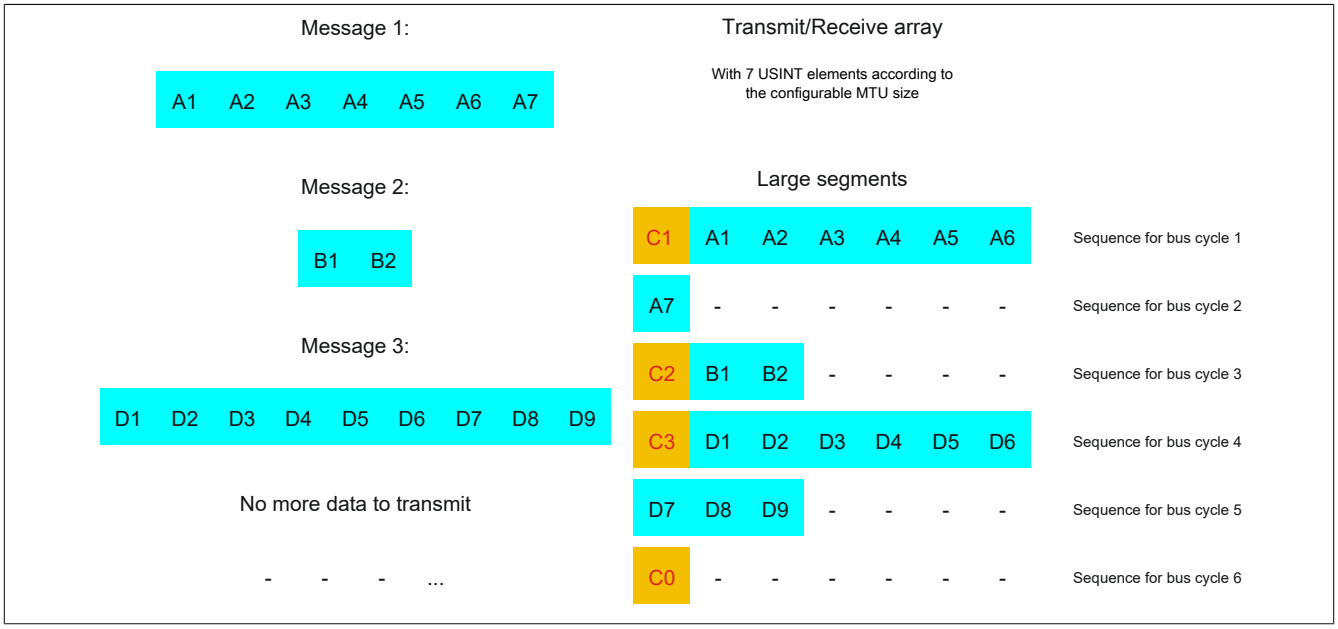


Figure 14: Transmit/receive array (large segments)

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 7: 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.

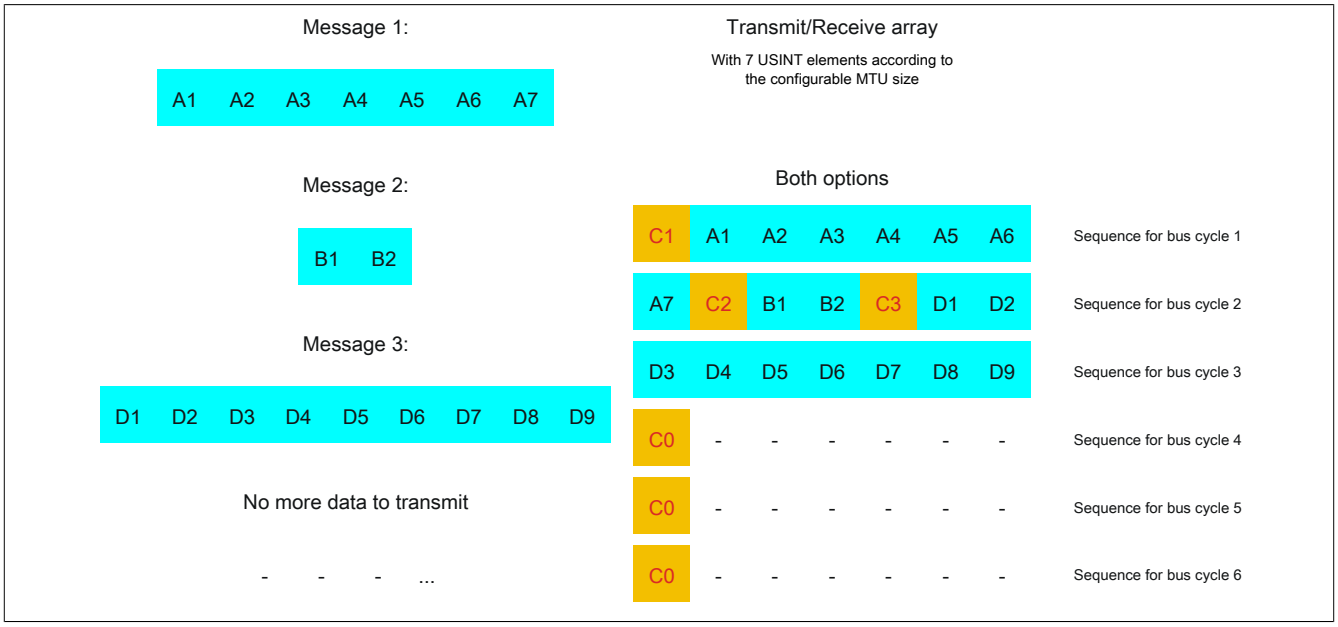


Figure 15: Transmit/receive array (large segments and MultiSegmentMTUs)

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

1.5.8.5 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.

1.5.8.5.1 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)

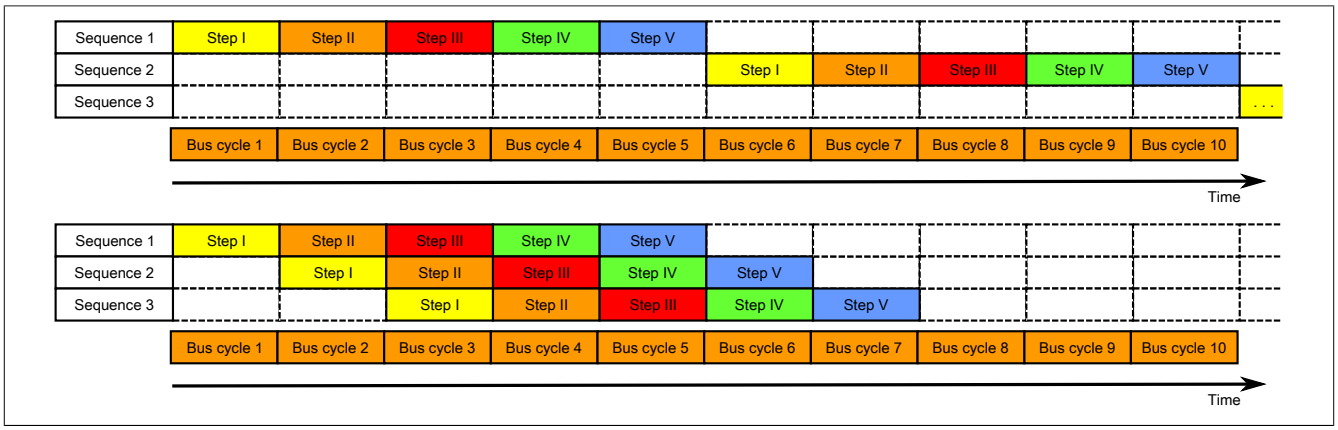


Figure 16: Comparison of transfer without/with Forward

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 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.

1.5.8.5.2 Configuration

The Forward function must only be enabled for the input direction. 2 additional configuration registers are available for doing so. 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.

1.5.8.5.2.1 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

1.5.8.5.2.2 Delay time

Name:
ForwardDelay

Register "ForwardDelay" is used to specify the delay time 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.

Data type	Values
UINT	0 to 65535 [µs] Default: 0

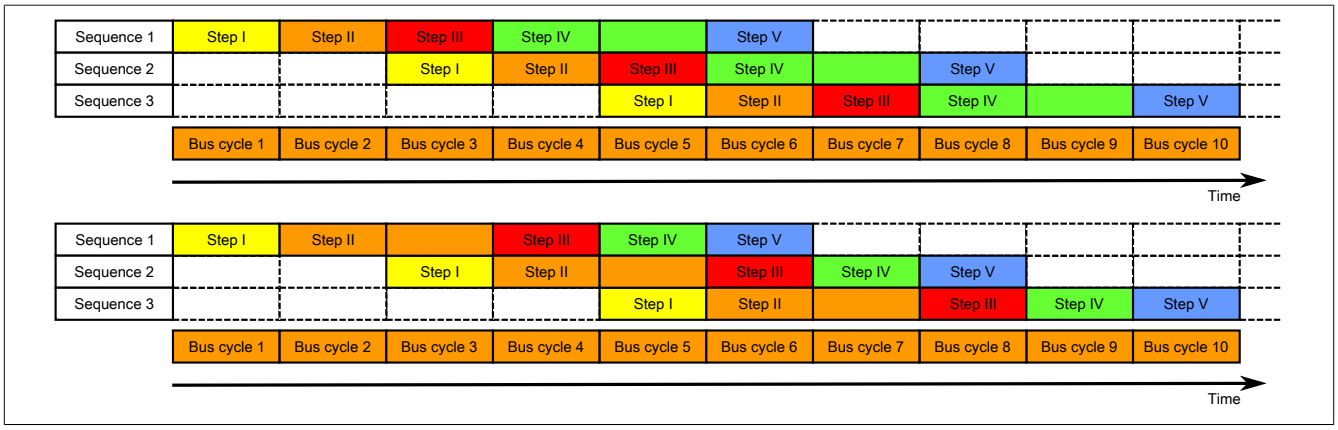


Figure 17: Effect of ForwardDelay when using Flatstream communication with the Forward function

In the program, it is important to make sure that the CPU 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 CPU has more time to process the incoming InputSequence or InputMTU.

1.5.8.5.3 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 CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU 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 CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU must transfer the current part of the transmit array to OutputMTU. - The CPU must increase OutputSequenceCounter for the sequence to be accepted by the module. - The CPU 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 CPU 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 CPU 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 CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU 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 Sequence-Counter 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 CPU 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.

1.5.8.5.4 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).

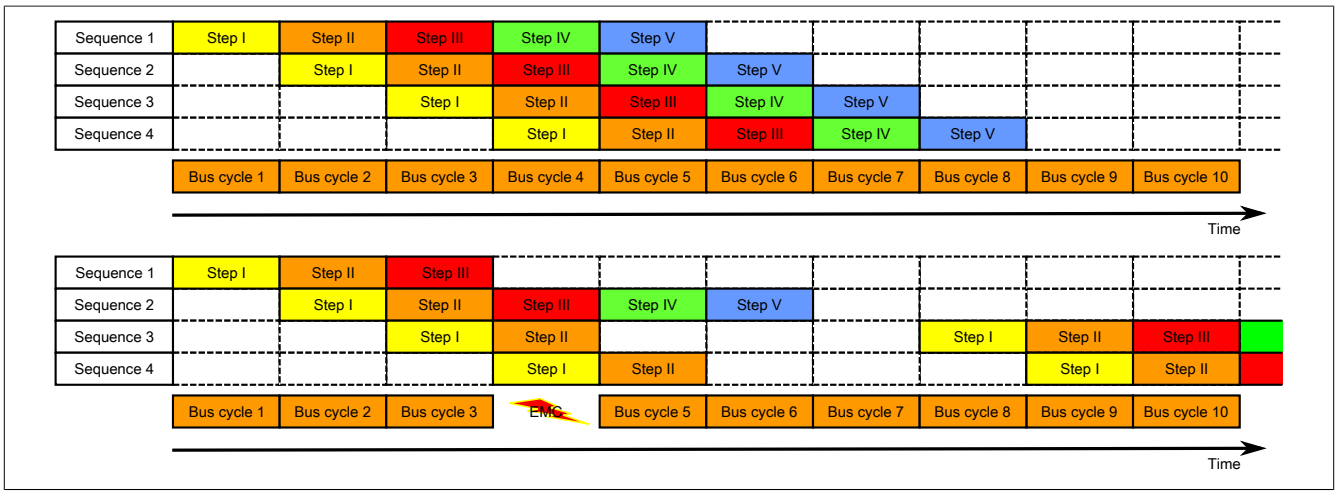


Figure 18: Effect of a lost bus cycle

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.

1.5.9 M-Bus with FlatStream

When using FlatStream communication, the module acts as a bridge between the X2X 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 bus systems. Specific algorithms such as timeout and checksum monitoring are usually managed automatically. During normal operation, the user does not have direct access to these details.

Operation

The M-Bus specification recognizes four different frame types. From the application standpoint, only "long frames" are generated and transferred when using the M-Bus via FlatStreams. Due to the flexible design of the M-Bus protocol, the user must include the corresponding slave configuration with each request.

FlatStream structure		
In-/OutputSequence	Rx-/TxBytes	
(unchanged)	Control byte (unchanged)	M-Bus data (FlatStream)

1.5.9.1 FlatStream in output direction

FlatStream query

This standard protocol specifies that a data query via FlatStream consists of a main part and two index records. An index record is made up of an introduction containing various information and followed by a parameter block.

1.5.9.1.1 Introduction

The primary role of the main part is to assign a synchronization number and register the protocol type.

Note 1
When registering an undefined protocol type, the module works with the standard protocol.

Note 2
Because there is currently only one protocol type defined, the corresponding configuration bytes should be set to 1. This will allow the protocol to be expanded later without becoming incompatible with existing projects.

Bytes	Name	Value	Description
1	Frame number: For synchronization in the application	0 - 255	The frame number is repeated in the module's response. This allows the later response from the module to be distinctly attributed to the request.
2	Index Record Count "i"	2!	Number of subsequent index records
3	Protocol type	0	Native M-Bus (level converter mode) - see "Native M-Bus"
		1	Data query (raw data / parameters)
4	Reserved	1!	
...	Index record (configuration)		
...	Index record (data query)		

Native M-Bus

The "Native M-Bus" protocol type provides universal communication within the M-Bus network. It can be used to assemble and send M-Bus frames in the application.

A conventional data query is possible using a raw data or parameter query.

1.5.9.1.2 Index record 0

Configuration block

The interface parameters for defining the module's behavior in the M-Bus network must be chosen configuration part.

Information:

With the standard protocol, the index record must be resent with each request for configuration.

Introduction

Bytes	Name	Value	Description
1	Index record type	0!	Module interface configuration
2	Counter (config parameter)	5!	Number of subsequent M-Bus parameters
3	Length of parameter block - Low	19!	Length of index record description
4	Length of parameter block - High	0!	Length of index record description

Parameter block

Configuration parameter 0 - Addressing type

Bytes	Name	Value	Description
1	Parameter number	0!	
2	Length	1!	
3	Addressing type	1	Addressing via primary address
		2	Addressing via secondary address

Configuration parameter 1 - Address

Bytes	Name	Value	Description
4	Parameter number	1!	
5	Length	4!	
6	Address - LowLow	1 - 255	Primary address
7	Address - LowHigh	0 - 255	0!, if primary addressing
8	Address - HighLow	0 - 255	0!, if primary addressing
9	Address - HighHigh	0 - 255	0!, if primary addressing

Configuration parameter 2 - Transfer rate

Bytes	Name	Value	Description
10	Parameter number	2!	
11	Length	2!	
12	Transfer rate Low	0 - 255	Verified transfer rates 300 bit/s, 2400 bit/s, 9600 bit/s
13	Transfer rate High	0 - 255	

Configuration parameter 3 - Timeout offset

Bytes	Name	Value	Description
14	Parameter number	3!	
15	Length	1!	
16	TimeoutOffset	0 - 255	Additional time for timeout monitoring on the M-Bus (Resolution: 10 ms)

Configuration parameter 4 - Extra frames

Bytes	Name	Value	Description
17	Parameter number	4!	
18	Length	1!	
19	M-Bus options	Bit 0 … 1	Send Init frame
		Bit 1 … 1	Send application reset
		Bit 6 … 1	Set frame count bit[1)]
		Bit 7 … 1	Request media and version

1) Some M-Bus slaves use this bit to switch to a another data set.

1.5.9.1.3 Index record 1

Data query block

The M-Bus parameters to be retrieved from the memory of the M-Bus slaves are requested in the request part. The user can request certain parameters from the slave or the entire slave memory.

Introduction

Bytes	Name	Value	Description
1	Index record type	1!	Data request for M-Bus slave
2	Counter (data parameter) = (d + 1)	0	• Communication via native M-Bus • Read out M-Bus raw data
		1 - 20	Number of parameters to read out
3	Length of subsequent block - Low	0 - 255	• Length of M-Bus frame to be sent • Length of the parameter block • 0! with raw data query
4	Length of subsequent block - High	0 - 255	• Length of M-Bus frame to be sent • Length of the parameter block • 0! with raw data query
...	Depending on request: • Native M-Bus frame • Parameter block		*Not needed if raw data query = 0*

M-Bus frame

M-Bus frame to be sent

Bytes	Name	Value	Description
1	TxByte 1	0 - 255	Byte 1 in output direction
2	TxByte 2	0 - 255	Byte 2 in output direction
n	TxByte n	0 - 255	Byte n in output direction

Parameter block

Data parameter 0

Bytes	Name	Value	Description
1	Parameter number	0!	
2	Data index	1 - 48	Data index in the M-Bus frame

Data parameter 1

Bytes	Name	Value	Description
2	Parameter number	1!	
3	Data index	1 - 48	Data index in the M-Bus frame

Data parameter d

Bytes	Name	Value	Description
...	Parameter number	d!	
...	Data index	1 - 48	Data index in the M-Bus frame

1.5.9.2 FlatStream in input direction

FlatStream response

The standard protocol has three different responses according to the request.

1.5.9.2.1 Error response

The error response occurs when the module receives an invalid or incomplete request.

Byte	Name	Value	Description
1	Frame number: For synchronization in the application	0 - 255	This frame number is repeated in the module's response. In this way, the response of the module can be clearly assigned to the request.
2	Error code - LowLow	0 - 255	See the error code table.
3	Error code - LowHigh	0 - 255	See the error code table.
4	Error code - HighLow	0 - 255	See the error code table.
5	Error code - HighHigh	0 - 255	See the error code table.
6	Additional information - LowLow	0 - 255	Optional
7	Additional information - LowHigh	0 - 255	Optional
8	Additional information - HighLow	0 - 255	Optional
9	Additional information - HighHigh	0 - 255	Optional

Error codes

Error code and name	Error description
0x11111111	An M-Bus counter is not responding to a data request. This can have different causes: • The counter is not connected. • The counter is defective. • A counter with the selected addressing parameters does not exist on the bus.
0x22222222	This error code is transmitted if addressing via the secondary address and the selected counter does not respond.
0x33333333	If an invalid transfer rate is sent along with the stream, it is not evaluated. No M-Bus frame is sent; the Flatstream interface responds directly with this error code.
0x44444444	If a collision occurs on the bus while querying the data, the data request is ended and this error code is returned.
0x55555555	Communication aborted due to overflow (see bit 4 in section "M-Bus state" on page 16)
0x66666666	Before the data is evaluated by the M-Bus counter, the checksum of the M-Bus frame is checked. If this is not correct, the received data is not processed further; instead, the error code is transmitted to the CPU.
0x77777777	The stream (CPU → IOM) is not correct. It is possible that a parameter number is not correct. The stream is checked very carefully, so an incorrect stream is never used.
0x88888888	Overload during M-Bus communication
0x99999999	Communication aborted due to level converter (see bit 5 in section "M-Bus state" on page 16)
0xAAAAAAAA	Interpretation of slave data not possible. The M-Bus slave being used is not compatible with the parameter query. The M-Bus slave must be implemented using the native M-Bus protocol or a raw data query.

Additional information

Additional information	Error description
0x00000001	Number of index records less than 2
0x00000002	Invalid stream length
0x00000004	Invalid index numbers
0x00000008	Incorrect number of parameters per index record
0x00000010	Index length too small
0x00000020	Incorrect parameter number for index record 0
0x00000040	Incorrect parameter length for index record 0
0x00000080	Invalid addressing type
0x00000100	Invalid address
0x00000200	Invalid transfer rate
0x00000400	Invalid timeout offset
0x00000800	Invalid additional frame configuration

1.5.9.2.2 Response - native M-Bus

This response corresponds with a successfully transferred M-Bus frame created within the application.

Bytes	Name	Value	Description
1	Frame number: For synchronization in the application	0 - 255	The frame number is repeated in the module's response. This allows the response from the module to be distinctly attributed to the request.
2	Reserved	0	
...	Response		

Response

Bytes	Name	Value	Description
1	RxByte 1	0 - 255	Byte 1 in input direction
2	RxByte 2	0 - 255	Byte 2 in input direction
n	RxByte n	0 - 255	Byte n in input direction

1.5.9.2.3 Response - Raw data

The raw data response is sent if the M-Bus slave's entire memory is requested.

Bytes	Name	Value	Description
1	Frame number: For synchronization in the application	0 - 255	The frame number is repeated in the module's response. This allows the response from the module to be distinctly attributed to the request.
2	M-Bus status	0 - 255	Status info from M-Bus header
3	Raw data frame	0 - 255	Includes all bytes sent by the M-Bus slave.
...		0 - 255	

1.5.9.2.4 Response - Parameters

The parameter response is sent if one or more parameters from an M-Bus slave have been requested.

Bytes	Name	Value	Description
1	Frame number: For synchronization in the application	0 - 255	The frame number is repeated in the module's response. This allows the response from the module to be distinctly attributed to the request.
2	M-Bus status	0 - 255	Status info from M-Bus header
3	Parameter count "p"	0 - 255	Number of parameters received
4	M-Bus address	0 - 255	Primary address
5	Serial number - LowLow	0 - 255	Secondary address
6	Serial number - LowHigh	0 - 255	Secondary address
7	Serial number - HighLow	0 - 255	Secondary address
8	Serial number - HighHigh	0 - 255	Secondary address
9	VendorID – Low \ Version	0 - 255	See "MBus option (IndexRecord 0)" on page 47
10	VendorID – High \ Medium	0 - 255	See "MBus option (IndexRecord 0)" on page 47
11	Data structure (M-Bus)	1	Fixed data structure
		2	Variable data structure
...	Received parameter 1 through p		*Not needed if parameter count = 0*

Received parameter

Bytes	Name	Value	Description
1	Medium	0 - 255	Medium of subsequent counter value
2	Index	0 - 255	Index of subsequent counter value
3	Data length	1 - 8	Length of the counter value
		255	If the parameter number is invalid
4	DIF	0 - 255	0!, if fixed data structure
5	VIF	0 - 255	0!, if fixed data structure
6	Counter value	0 - 255	LowLowLowLowLowLowLowLow
...		...	...
13		0 - 255	HighHighHighHighHighHighHigh

1.5.10 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 µs

1.5.11 Minimum I/O update time

The minimum I/O update time defines how far the bus cycle can be reduced while still allowing an I/O update to take place in each cycle.

Minimum I/O update time
1 s

2 X20CS1013

2.1 General information

The module is a DALI control device with an integrated power supply. Up to 64 operating devices can be connected.

DALI stands for Digital Addressable Lighting Interface and enables easy and safe control of light fixtures using a standardized digital operating device interface. The DALI bus conforms with EN 62386 series of standards and is now supported by many electronic ballast manufacturers.

- Integrated power supply
- Up to 64 operating devices (individual addresses)
- Up to 16 groups (group addresses)
- Up to 16 scenes (scene lighting values)

2.1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX20	X20 System user's manual
MAEMV	Installation / EMC guide

2.2 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS1013	X20 interface module, 1 DALI master interface	
	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	
	Terminal blocks	
X20TB06	X20 terminal block, 6-pin, 24 VDC keyed	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	

Table 9: X20CS1013 - Order data

2.3 Technical description

2.3.1 Technical data

Order number	X20CS1013
Short description	
Communication module	DALI master
General information	
B&R ID code	0xDE85
Diagnostics	
Module status	Yes, using LED status indicator and software
Bus status	Yes, using LED status indicator and software
Power consumption	
Bus	0.2 W
Internal I/O	0.4 W
Additional power dissipation caused by actuators (resistive) [W]	4 W
Insulation voltages	
Channel - Bus	510 VAC / 1 minute
Certifications	
CE	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
EAC	Yes
DALI bus	
Insulation system	Basic insulation
Open-circuit voltage	16.5 V ±5%
Short-circuit proof	Yes (current limiting)
Signal voltage	
Low	-6.5 V to 6.5 V (typically 0 V)
High	11.5 V to 20.5 V (typically 16 V)
Signal current	
Low	≤250 mA (internally limited)
High	≤130 mA at voltage ≥11.5 V
Transfer rate	1200 baud
Maximum number of slaves	64
Data signal slew rate (Manchester bi-phase)	
Falling edge	10 µs ≤ t_{fall} ≤ 100 µs
Rising edge	10 µs ≤ t_{rise} ≤ 100 µs
Electrical properties	
Electrical isolation	Channel isolated from bus
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
Ambient conditions	
Temperature	
Operation	
Horizontal mounting orientation	-25 to 60°C
Vertical mounting orientation	-25 to 50°C
Derating	-
Storage	-40 to 85°C
Transport	-40 to 85°C
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order 1x terminal block X20TB06 or X20TB12 separately. Order 1x bus module X20BM11 separately.
Pitch	$12.5^{+0.2}$ mm

Table 10: X20CS1013 - Technical data

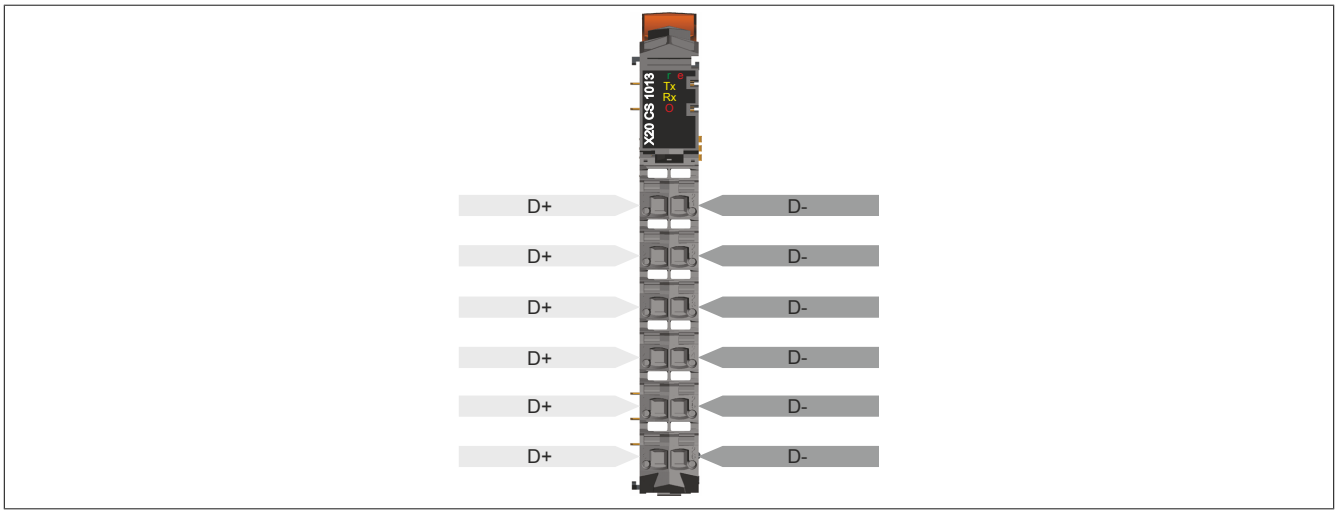
2.3.2 LED status indicators

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
			On	RUN mode
			Double flash	BOOT mode (during firmware update)[1)]
	e	Red	Off	No power to module or everything OK
			On	Error status
	Tx	Yellow		Control device (master) transmitting
	Rx	Yellow		Operating device (slave) responding
	O	Red		Error status: Overload or short circuit

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

2.3.3 Pinout



2.3.4 Using an external power supply

Since the internal DALI power supply provides sufficient power for a configuration with up to 64 slaves, the module is not designed for an external power supply.

Warning!

Using an additional DALI power supply may result in damage to the module.

2.4 Register description

2.4.1 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.

2.4.2 Function model 0 - default

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
258	Dali_State	UINT	•			
263	Dali_RequestCounter	USINT	•			
261	Dali_AnswerCounter	USINT	•			
265	Dali_Answer	USINT	•			
257	Dali_Enable	USINT			•	
262	Dali_Control	UINT			•	
265	Dali_Address	USINT			•	
267	Dali_Command	USINT			•	

2.4.3 Function model 254 - Bus controller

Register	Offset[1)]	Name	Data type	Read		Write	
				Cyclic	Acyclic	Cyclic	Acyclic
258	0	Dali_State	UINT	•			
263	3	Dali_RequestCounter	USINT	•			
261	2	Dali_AnswerCounter	USINT	•			
265	4	Dali_Answer	USINT	•			
257	0	Dali_Enable	USINT			•	
262	2	Dali_Control	UINT			•	
265	4	Dali_Address	USINT			•	
267	5	Dali_Command	USINT			•	

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

2.4.3.1 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).

2.4.3.2 CAN I/O bus controller

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

2.4.4 General information

DALI stands for Digital Addressable Lighting Interface and is mainly used to control lighting systems. The communication standard is intended for building automation systems and described in the EN 62386 series of standards.

2.4.4.1 The DALI protocol

The DALI standard specifies bidirectional communication based on the "request and answer" principle. A DALI network may contain multiple masters. The serial asynchronous interface transmits voltage signals at a transfer rate of 1200 bits/s.
According to the DALI standard, up to 64 individual addresses can be assigned on the network. In addition, all of the slaves in the network can be addressed via broadcast and group addresses. 16 different group addresses can be assigned independently of the individual slave addresses. This makes it possible to send a command to multiple slaves at the same time.

2.4.5 DALI - Communication

The module provides the user with a channel for communicating with and controlling DALI slaves in a DALI network.

The multi-master mode described in the DALI standard is accepted by the module but not actively supported.

2.4.5.1 Communication in the DALI network

The module supports all commands defined in the DALI standard.

Communication in the DALI network takes place using the 2 bytes of the following registers:

- "Address of the DALI slave" on page 56
- "Direct or indirect command for receiver" on page 56

Some commands are described in the DALI specification with the structure "YAAA AAAS XXXX XXXX". In order to translate this representation to the B&R interface, the two registers "DALI_Address" and "DALI_Command" must be viewed with "Byte" as the unit.

Dali_Address								Dali_Command							
MSB	6	5	4	3	2	1	0	LSB	6	5	4	3	2	1	0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Y	A	A	A	A	A	A	S	X	X	X	X	X	X	X	X

Key

Y	Address type
A	Address
S	Command type
X	Command

2.4.5.1.1 Address of the DALI slave

Name:
Dali_Address

This register provides the module with the address of the DALI slave being addressed. It also defines the address type (single or group address) and command type (direct or indirect command).

Data type	Value	
USINT	0 to 159	Single or group address for direct or indirect command
	254	Broadcast address for direct DALI command
	255	Broadcast address for indirect DALI command

Bit structure:

Bit	Name	Value	Information
0	Type of the subsequent command	0	Direct DALI command
		1	Indirect DALI command
1 - 6	Address	0 to 63	Address of an individual slave
		0 to 15	Address of a group of slaves
7	Type of the subsequent address	0	Addressing of an individual slave
		1	Addressing of a group of slaves

2.4.5.1.2 Direct or indirect command for receiver

Name:
Dali_Command

This register provides the module with the direct or indirect command for the receiver in the DALI network.

Data type	Value	Information
USINT	0 to 255	DALI or slave-specific command

2.4.5.2 Status in the DALI network

Name:
Dali_State

This register is used to indicate the current status of the DALI network.

Data type	Value
UINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	Enables/Disables the level converter	0	Communication off
		1	Communication on
1	Status of the last request	0	Valid request not yet sent
		1	Transmit procedure successful
2	Status of the last response	0	No response since the last request
		1	Receive procedure successful
3	Collision (multi-master)	0	No collision
		1	Collision in the DALI network
4 - 7	Reserved	-	
8	Transmit error	0	No error
		1	Transmit procedure failed
9	Receive error	0	No error
		1	Invalid response received
10	TX busy	0	No transmission activity
		1	Transmission taking place
11	RX busy	0	No receiving activity
		1	Receiving taking place
12 - 15	Reserved	-	

2.4.5.3 Transmission counter

Name:
Dali_RequestCounter

This register provides the user with information about how many DALI messages have already been sent by the module.

Data type	Value
USINT	0 to 255

2.4.5.4 Response counter

Name:
Dali_AnswerCounter

This register provides the user with information about how many DALI messages have already been received by the module.

Data type	Value
USINT	0 to 255

2.4.5.5 Response from DALI network

Name:
Dali_Answer

This register provides the user with access to the last valid response from the downstream DALI network.

Data type	Value
USINT	0 to 255

2.4.5.6 Enabling the communication channel

Name:
Dali_Enable

This register is used to enable or disable the communication channel.

Data type	Value
USINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	Turn communication on/off (via software)	0	Turn communication channel off
		1	Turn communication channel on
1	Turn power saving mode on/off	0	Supply DALI network
		1	Turn internal power supply for the module off
2 - 7	Reserved	-	

Information:

for communication in the DALI network, the internal power supply in the module must be turned on.

2.4.5.7 Controlling the DALI module

Name:
Dali_Control

This register is used to control the module. The respective command is transported via X2X Link and then executed by the module. The register is edge-triggered (i.e. this type of command is only triggered if the state of the respective bit changes).

Data type	Value
UINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	Requests command (pos. edge)	0	No action
		1	Transmits request in the DALI network
1	Reserved	-	
2	Acknowledges the status byte (pos. edge)	0	No action
		1	Resets the status byte
3	Acknowledges the transmission counter (pos. edge)	0	No action
		1	Resets the transmission counter
4	Acknowledges the response counter (pos. edge)	0	No action
		1	Resets the response counter
5 - 15	Reserved	-	

2.4.6 Excerpt from the DALI specification

2.4.6.1 General

The DALI standard involves 2 different command types. Direct commands control the brightness of the lights on the DALI slave being addressed. This type of communication runs only from the master to the slave.

Setting the LSB in the address register will use the included command for independent digital communication. The commands are also transferred from the master to the slave. Some requests require a response from the slave. In this case, communication from the slave to the master must also be possible.

2.4.6.2 Direct DALI commands (ARC)

These commands can be used to directly set the brightness of each DALI slave. The statements 1 to 254 correspond to a brightness of the connected DALI slave based on the following formula:

$$P = 10^{\frac{\text{Value} - 1}{253 / 3}} * \frac{P_{max}}{1000}$$

Command 0 can also be transmitted to switch off a DALI slave. In this case, the brightness decreases slowly at first and then shuts off when a critical power level is crossed.

Command 255 serves as an internal mask value. It is not applied by the DALI slave, which means it has no effect on its behavior.

2.4.6.3 Indirect DALI commands for lamp wattage

Indirect commands make digital communication possible on the DALI network. In addition to the commands defined in the DALI standard, some manufacturers of DALI slaves also define their own commands.

Selected standardized DALI commands

Source: EN 62386-102:2009

Code (dec.)	Function
Indirect control commands	
0	Switches off the light immediately – No smooth transition
1	200 ms dimming up – Possible to configure the dimming speed separately – No further change once maximum is reached – Command ignored when light is off
2	200 ms dimming down – Possible to configure the dimming speed separately – No further change once minimum is reached – Command does not turn light off
3	Increases the brightness by one step – No smooth transition – No further change once maximum is reached – Command ignored when light is off
4	Decreases the brightness by one step – No smooth transition – No further change once minimum is reached – Command does not turn light off
5	Maximum brightness – No smooth transition – Turns the light on
6	Minimum brightness – No smooth transition – Turns the light on
7	Decrease brightness by one step (including switching off) – No smooth transition – Command can turn light off
8	Increase brightness by one step (including switching on) – No smooth transition – Switches on a switched-off light
9	Commence DACP sequence – Starts direct power control – Dimming speed adjusted dynamically by the control device – DACP sequence required at the end
10 - 15	Reserved
16 - 31	Enables scene 0 to 15 – Power regulated to the level stored in the scene

2.4.6.4 Indirect DALI commands for configuration

Indirect commands make digital communication possible on the DALI network. In addition to the commands defined in the DALI standard, some manufacturers of DALI slaves also define their own commands.

Information:

Some indirect DALI commands must be repeated within 100 ms. The module does not evaluate specified addresses and commands, which means this repetition must be ensured by the application.

Selected standardized DALI commands

Source: EN 62386-102:2009

Code (dec.)	Function	Response
Configuration commands[1)]		
32	Resets nonvolatile memory – DALI slave requires up to 300 ms for execution	
33	Reads out the current power level – Stores the current power value in the DTR – Command code 152 required	
34 - 41	Reserved	
Save DTR value[1)]		
42	Save as maximum power value	
43	Save as minimum power value	
44	Save power value as value for event of error	
45	Save power value as switch-on value	
46	Save value as dimming time	
47	Save value as dimming speed	
48 - 63	Reserved	
Used for setting system parameters[1)]		
64 - 79	Save DTR value as selected scene 0 to 15 – Scene number = Command number - 64	
80 - 95	Removes DALI slave from scene 0 to 15 – Scene number = Command number - 80	
96 - 111	Adds DALI slave to group 0 to 15 – Group number = Command number - 96	
112 - 127	Removes DALI slave from group 0 to 15 – Group number = Command number - 112	
128	Save DTR value as short address	
129 - 143	Reserved	
Request commands		
144	Checks the general status	**Bit / Value / Function** 0 / 0 / DALI slave status OK 0 / 1 / DALI slave status not OK 1 / 0 / Light status OK 1 / 1 / Light status not OK 2 / 0 / Light off 2 / 1 / Light on 3 / 0 / Last requested power level permissible 3 / 1 / Last requested power level not permissible 4 / 0 / Last dimming procedure complete 4 / 1 / Dimming procedure not yet complete 5 / 0 / DALI slave not in reset state 5 / 1 / DALI slave in reset state 6 / 0 / DALI slave has short address 6 / 1 / DALI slave has no short address 7 / 0 / Reset or control command not yet received by DALI slave 7 / 1 / Reset or control command received by DALI slave
145	Checks communication readiness	Yes/No
146	Checks for light failure	Yes/No
147	Checks whether the light is currently switched on	Yes/No
148	Checks whether the last requested power value was applied	Yes/No
149	Checks whether the DALI slave is in reset state	Yes/No
150	Checks whether the DALI slave has a short address	Yes/No
151	Checks whether the DALI slave has a version number	The response depends on the DALI slave: • Yes/No (DALI slave has a version number or not) • Version number
152	Checks the DTR value	DTR value
153	Checks the device type	DALI-specific code for categorizing DALI slaves
154	Checks the physical minimum level (greater than 0)	Value of physical minimum level
155	Checks for power failure	Yes/No

<table>
<tr><th>Code (dec.)</th><th>Function</th><th>Response</th></tr>
<tr><td>156 - 159</td><td>Reserved</td><td></td></tr>
<tr><td>160</td><td>Checks the current power level</td><td>Current power level or 255 if the light is being warmed up</td></tr>
<tr><td>161</td><td>Checks the maximum value</td><td>Maximum value</td></tr>
<tr><td>162</td><td>Checks the minimum value</td><td>Minimum value</td></tr>
<tr><td>163</td><td>Checks the switch-on power level</td><td>Switch-on power level</td></tr>
<tr><td>164</td><td>Checks the power level in the event of error</td><td>Power level in the event of error</td></tr>
<tr><td>165</td><td>Checks the dimming time and dimming speed</td><td><table><tr><th>Bit</th><th>Function</th></tr><tr><td>0 - 3</td><td>Dimming speed</td></tr><tr><td>4 - 7</td><td>Dimming time</td></tr></table></td></tr>
<tr><td>166 - 175</td><td>Reserved</td><td></td></tr>
<tr><td>176 - 191</td><td>Checks the light level for scene 0 to 15</td><td></td></tr>
<tr><td>192</td><td>Checks whether the DALI slave member is part of group 0 to 7</td><td><table><tr><th>Bit</th><th>Val-ue</th><th>Function</th></tr><tr><td rowspan="2">0</td><td>0</td><td>Slave not in group 0</td></tr><tr><td>1</td><td>Slave in group 0</td></tr><tr><td>...</td><td></td><td></td></tr><tr><td rowspan="2">7</td><td>0</td><td>Slave not in group 7</td></tr><tr><td>1</td><td>Slave in group 7</td></tr></table></td></tr>
<tr><td>193</td><td>Checks whether the DALI slave member is part of group 8 to 15</td><td><table><tr><th>Bit</th><th>Val-ue</th><th>Function</th></tr><tr><td rowspan="2">0</td><td>0</td><td>Slave not in group 8</td></tr><tr><td>1</td><td>Slave in group 8</td></tr><tr><td>...</td><td></td><td></td></tr><tr><td rowspan="2">7</td><td>0</td><td>Slave not in group 15</td></tr><tr><td>1</td><td>Slave in group 15</td></tr></table></td></tr>
<tr><td>194</td><td>Checks a 24-bit random address (H)</td><td>Random address (higher 8 bits)</td></tr>
<tr><td>195</td><td>Checks a 24-bit random address (M)</td><td>Random address (middle 8 bits)</td></tr>
<tr><td>196</td><td>Checks a 24-bit random address (L)</td><td>Random address (lower 8 bits)</td></tr>
<tr><td>197 - 223</td><td>Reserved</td><td></td></tr>
<tr><td>224 - 255</td><td>Checks application-specific defined commands</td><td></td></tr>
</table>

1) Any command in the range 32 to 129 must be repeated within the next 100 ms. No other commands can be transmitted to the DALI slave being addressed during this time.

2.4.6.5 DALI special commands

In the DALI standard, special commands are described as a bit structure represented by the arrangement YAAAAAAS XXXXXXXX (see also "Communication in the DALI network" on page 56). This section contains information about he most important special commands from the DALI specification.

Leave special modes

TERMINATE

YAAAAAAS XXXXXXXX 10100001 00000000

Switches all DALI slaves on the bus in normal mode

Write DTR

DATA TRANSFER REGISTER (DTR)

YAAAAAAS XXXXXXXX 10100011 xxxxxxxx

Writes the bit pattern xxxxxxxx to the Data Transfer Register (DTR)

Special addressing for address assignment

INITIALISE

YAAAAAAS XXXXXXXX 10100101 xxxxxxxx

Allows commands for special addressing within the next 15 minutes.

Information:

- **The command must be sent twice within 100 ms.**
- **"TERMINATE" can be used to exit initialization early.**
  Reinitializing (before the 15 minutes is completed) extends initialization by a further 15 minutes.

RANDOMISE

YAAAAAAS XXXXXXXX 10100111 00000000

Information:

The command must be sent twice within 100 ms.

SEARCHADDRH
SEARCHADDRM
SEARCHADDRL

YAAAAAAS XXXXXXXX 10110001 hhhhhhhh
10110011 mmmmmmmm
10110101 llllllll

"hhhhhhhh", "mmmmmmmm" and "llllllll" represent the currently "selected" 24-bit address in the DALI network.

COMPARE

YAAAAAAS XXXXXXXX 10101001 00000000

All slaves in the DALI network with a 24-bit address less than or equal to hhhhhhhh mmmmmmmm llllllll respond with YES. By repeatedly assigning new search addresses and using "COMPARE", it is possible to select the currently initialized slave with the smallest 24-bit address.

PROGRAM SHORT ADDRESS

YAAAAAAS XXXXXXXX 10110111 0aaaaaa1

The selected slave takes on the short address assigned to aaaaaa.

QUERY SHORT ADDRESS

YAAAAAAS XXXXXXXX = 10111011 00000000

The selected slave responds with its current short address. If no short address has been assigned, it responds with 255. This can be used to check the possible success of address assignment.

VERIFY SHORT ADDRESS

YAAAAAAS XXXXXXXX 10111001 0aaaaaa1

The selected slave responds with YES if the value specified on aaaaaa corresponds to its short address. This can be used to check the possible success of address assignment.

WITHDRAW

YAAAAAAS XXXXXXXX 10101011 00000000

The selected slave is excluded from the subsequent search with "COMPARE" statements but remains initialized and can be selected.

PHYSICAL SELECTION

YAAAAAAS XXXXXXXX = 10111101 00000000

The selected slave is excluded from the subsequent search with "COMPARE" statements, no longer initialized and can no longer be selected.

Additional special commands

Additional special commands can be found in the DALI standard.

2.4.7 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
100 µs

2.4.8 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
30 ms

3 X20(c)CS1020

3.1 General information

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.

- RS232 interface for serial, remote connection of complex devices to the X20 system

3.2 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

3.3 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS1020	X20 interface module, 1 RS232 interface, max. 115.2 kbit/s	
X20cCS1020	X20 interface module, coated, 1 RS232 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 11: X20CS1020, X20cCS1020 - Order data

3.4 Technical data

Order number	X20CS1020	X20cCS1020
Short description		
Communication module	1x RS232	
General information		
B&R ID code	0x1FCF	0xE7F2
Status indicators	Data transfer, operating status, module status	
Diagnostics		
Module run/error	Yes, using status LED and software	
Data transfer	Yes, using status LED	
Power consumption		
Bus	0.01 W	
Internal I/O	1.44 W	
Additional power dissipation caused by actuators (resistive) [W]	-	
Certifications		
CE	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 GL	Temperature: **B** (0 - 55°C) Humidity: **B** (up to 100%) Vibration: **B** (4 g) EMC: **B** (bridge and open deck)	
LR	ENV1	
KR	Yes	
ABS	Yes	
EAC	Yes	
KC	Yes	-
Interfaces		
Interface IF1		
Signal	RS232	
Variant	Connection made using 12-pin X20TB12 terminal block	
Max. distance	900 m	
Transfer rate	Max. 115.2 kbit/s	
FIFO buffer	1 kB	
Handshake lines	RTS, CTS	
Controller	UART type 16C550 compatible	
Electrical properties		
Electrical isolation	RS232 (IF1) isolated from bus RS232 (IF1) not isolated from I/O power supply	
Operating conditions		
Mounting orientation		
Horizontal	Yes	
Vertical	Yes	
Installation elevation above sea level		
0 to 2000 m	No limitations	
>2000 m	Reduction of ambient temperature by 0.5°C per 100 m	
Degree of protection per EN 60529	IP20	
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 X20TB06 or X20T-B12 terminal block separately Order 1x X20BM11 bus module separately	Order 1x X20TB06 or X20T-B12 terminal block separately Order 1x X20cBM11 bus module separately
Pitch	$12.5^{+0.2}$ mm	

Table 12: X20CS1020, X20cCS1020 - Technical data

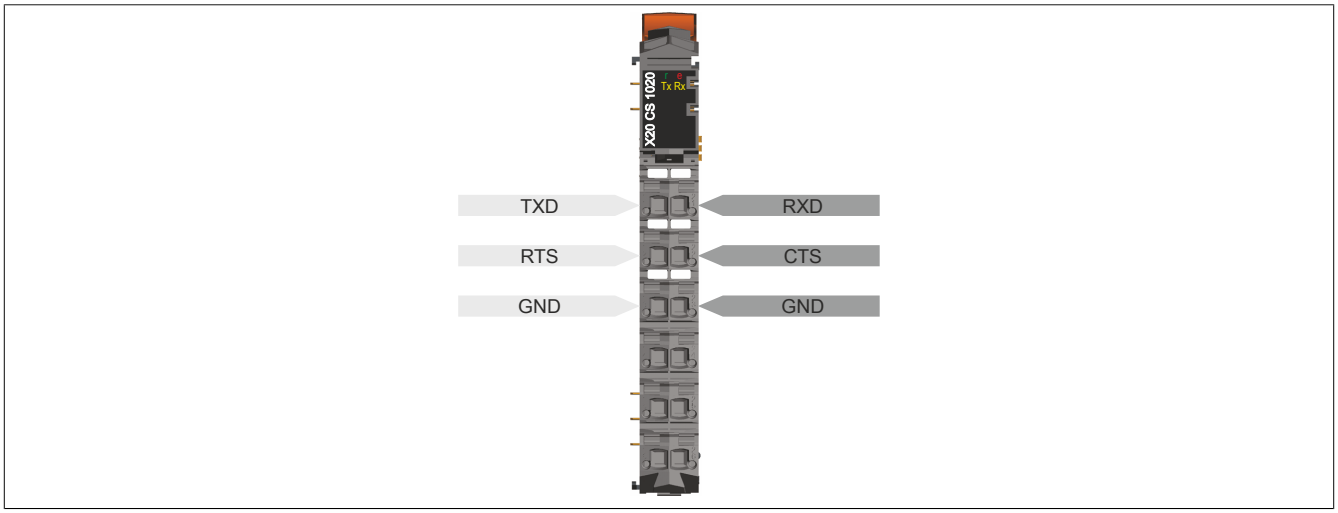
3.5 LED status indicators

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 77
			On	Error or reset status
	e + r	Red on / Green single flash		Invalid firmware
	Tx	Yellow	On	The module transmits data via the RS232 interface.
	Rx	Yellow	On	The module receives data via the RS232 interface.

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

3.6 Pinout

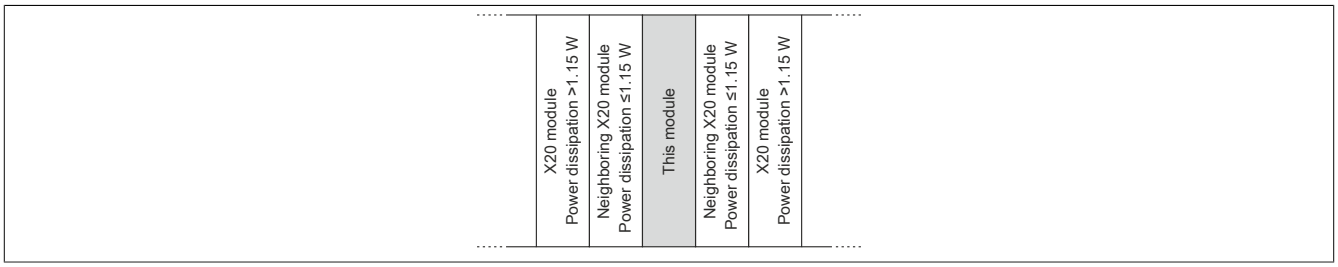


3.7 Derating

There is no derating when operated below 55°C.

During operation over 55°C, the power dissipation of the modules to the left and right of this module is not permitted to exceed 1.15 W!

For an example of calculating the power dissipation of I/O modules, see section "Mechanical and electrical configuration - Power dissipation of I/O modules" in the X20 user's manual.



3.8 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.**

3.9 Register description

3.9.1 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.

3.9.2 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 relatively 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_RxStateIgnoreMask	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				

3.9.3 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 the data stream (see "Flatstream communication" on page 78).

When using function model Flatstream, the user can choose whether to use library "AsFltGen" in AS 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	rxlLock	UINT				•
70	rxlUnlock	UINT				•
34	hssXOn	UINT				•
38	hssXOff	UINT				•
42	hssPeriod	UINT				•
19	hshTxF	USINT				•
29	hshRxF	USINT				•
27	hshSet	USINT				•
25	hshClr	USINT				•
17	hshInv	USINT				•
Frame – Configuration						
74	rxCto	UINT				•
106	txCto	UINT				•
78	rxEomSize	UINT				•
110	txEomSize	UINT				•
Index * 4 + 82	rxEomCharN (index N = 0 to 3)	UINT				•
Index * 4 + 114	txEomCharN (index N = 0 to 3)	UINT				•
Status messages – Configuration						
50	CfO_RxStateIgnoreMask	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	•			
Index + 128	RxByteN (index N = 1 to 27)	USINT	•			
160	OutputSequence	USINT			•	
Index + 160	TxByteN (index N = 1 to 27)	USINT			•	

3.9.4 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	Acyclic	Cyclic	Acyclic
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				•
275	-	hshTxF_CANIO	USINT				•
285	-	hshRxF_CANIO	USINT				•
281	-	hshClr_CANIO	USINT				•
283	-	hshSet_CANIO	USINT				•
287	-	hshFrm_CANIO	USINT				•
273	-	hshInv_CANIO	USINT				•
Frame – Configuration							
330	-	rxCto_CANIO	UINT				•
362	-	txCto_CANIO	UINT				•
334	-	rxEomSize_CANIO	UINT				•
366	-	txEomSize_CANIO	UINT				•
Index*4 + 338	-	rxEomCharN (N = 0 to 3)	UINT				•
Index*4 + 370	-	txEomCharN (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	•			
Index + 128	Index	RxByteN (index N = 1 to 7)	USINT	•			
160	0	OutputSequence	USINT			•	
Index + 160	Index	TxByteN (index N = 1 to 7)	USINT			•	

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

3.9.4.1 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).

3.9.4.2 CAN I/O bus controller

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

3.9.5 Serial interface - Configuration

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

3.9.5.1 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	RS232 interface disabled (default)
	2	RS232 interface enabled

3.9.5.2 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

3.9.5.3 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)

3.9.5.4 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

3.9.5.5 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

3.9.6 Handshake - Configuration

In order to guarantee that serial communication runs smoothly, the size of the receive buffer in the module must be known. In addition, the user can configure a software or hardware handshake algorithm.

3.9.6.1 Locking the receive buffer

Name:
rxlLock
rxlLock_CANIO

This register is used to configure the upper threshold of 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".

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

3.9.6.2 Unlocking the receive buffer

Name:
rxlUnlock
rxlUnlock_CANIO

This register is used to configure the lower threshold of 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".

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

3.9.6.3 RTS evaluation

Name:
hshRxF
hshRxF_CANIO

These registers can be used to configure how the hardware handshake line RTS is controlled depending on the fill level of the receive buffer.

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.

Data type	Value	Description
USINT	0	RTS line freely available for other flow control methods (bus controller default setting)
	16	RTS line is controlled by the fill level of the receive buffer

3.9.6.4 CTS evaluation

Name:
hshTxF
hshTxF_CANIO

This register is used to configure how the CTS hardware handshake line is evaluated. Make sure wiring to the peer station is correct when CTS query is enabled.

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.

Data type	Value	Description
USINT	0	CTS line ignored, transmission can take place at any time (bus controller default setting)
	1	CTS line active and is being used for flow control, transmit enable from the peer station

3.9.6.5 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.

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.

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

3.9.6.6 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.

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.

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

3.9.6.7 Handshake repetition

Name:
hssPeriod
hssPeriod_CANIO

When using a software handshake, some applications require periodic repetition of the current status. The repeat time can be defined in this register in ms 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

3.9.6.8 Enable handshake manually

Name:
hshSet
hshSet_CANIO

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

These registers can be used to force the output level of the RTS hardware handshake line to remain active.

Information:

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

Data type	Value	Description
USINT	0	RTS line freely available for other flow control methods (bus controller default setting)
	16	RTS line enabled

3.9.6.9 Disable handshake manually

Name:
hshClr
hshClr_CANIO

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

These registers can be used to force the output level of the RTS hardware handshake line to remain passive.

Information:

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

Data type	Value	Description
USINT	0	RTS line freely available for other flow control methods (bus controller default setting)
	16	RTS line disabled

3.9.6.10 Frame detection

Name:
hshFrm
hshFrm_CANIO

This register generally enables hardware-based frame detection. The RTS line is enabled as long as data is being transmitted. This Tx framing mode can be used to control external interface converters.

Information:

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

Data type	Values	Explanation
USINT	0	RTS line freely available for other flow control methods (bus controller default setting)
	16	Tx framing enabled for RTS line
	80	Tx framing enabled for RTS line (without echo)

3.9.6.11 Inverting RTS/CTS

Name:
hshInv
hshInv_CANIO

This register can be used to create a logical inverse of the RTS/CTS signals.

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

Bit structure:

Bit	Description	Value	Information
0	CTS signal	0	Inversion off (bus controller default setting)
		1	Inversion on
1 - 3	Reserved	0	
4	RTS signal	0	Inversion off (bus controller default setting)
		1	Inversion on
5 - 7	Reserved	0	

3.9.7 Frame - Configuration

Different message termination codes can be specified in order to correctly create transmitted Tx frames and correctly interpret received Rx frames.

3.9.7.1 Terminating when a receive timeout occurs

Name:
rxCto
rxCto_CANIO

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

The message is considered to be terminated when nothing is transfered 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.

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

3.9.7.2 Terminating when a transmit timeout occurs

Name:
txCto
txCto_CANIO

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

The message is considered to be terminated when nothing is transfered 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.

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

3.9.7.3 Maximum number of bytes received

Name:
rxEomSize
rxEomSize_CANIO

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

The message is considered to be ended as soon as a frame with the specified size in bytes is transferred. The longest possible frame length is the size of the 4096-byte receive buffer. Larger frames cause the Receive Overrun error.

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

3.9.7.4 Maximum number of bytes transmitted

Name:
txEomSize
txEomSize_CANIO

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

The message is considered to be ended as soon as a frame with the specified size in bytes is transferred. The longest possible frame length is the size of the 4096-byte transmit buffer. The configured transmit timeout is maintained after the frame has been sent.

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

3.9.7.5 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)

3.9.7.6 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)

3.9.8 Status messages - Configuration

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

3.9.8.1 Error detection setting

Name:
CfO_RxStateIgnoreMask
CfO_RxStateIgnoreMask_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

3.9.8.2 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		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	

3.9.9 Status messages - Communication

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

3.9.9.1 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.

3.9.9.2 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	

3.9.10 Flatstream communication

3.9.10.1 Introduction

B&R offers an additional communication method for some modules. "Flatstream" was designed for X2X and POWERLINK networks and allows data transmission 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.

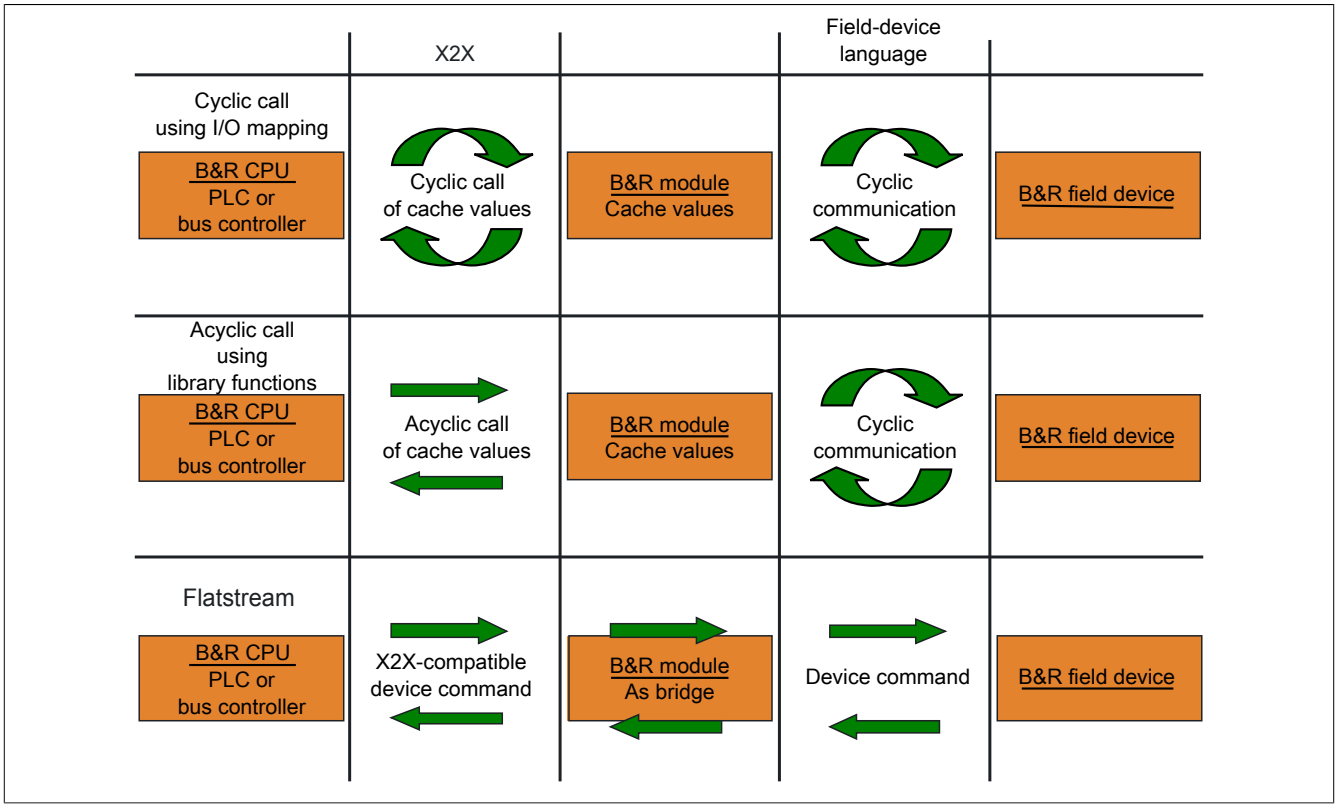


Figure 19: 3 types of communication

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

3.9.10.2 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 put back 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 transmission was faulty and that all affected sequences must be repeated.

3.9.10.3 The Flatstream principle

Requirement

Before Flatstream can be used, the respective communication direction must be synchronized, i.e. both communication partners cyclically query the sequence counter on the opposite 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 opposite 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.

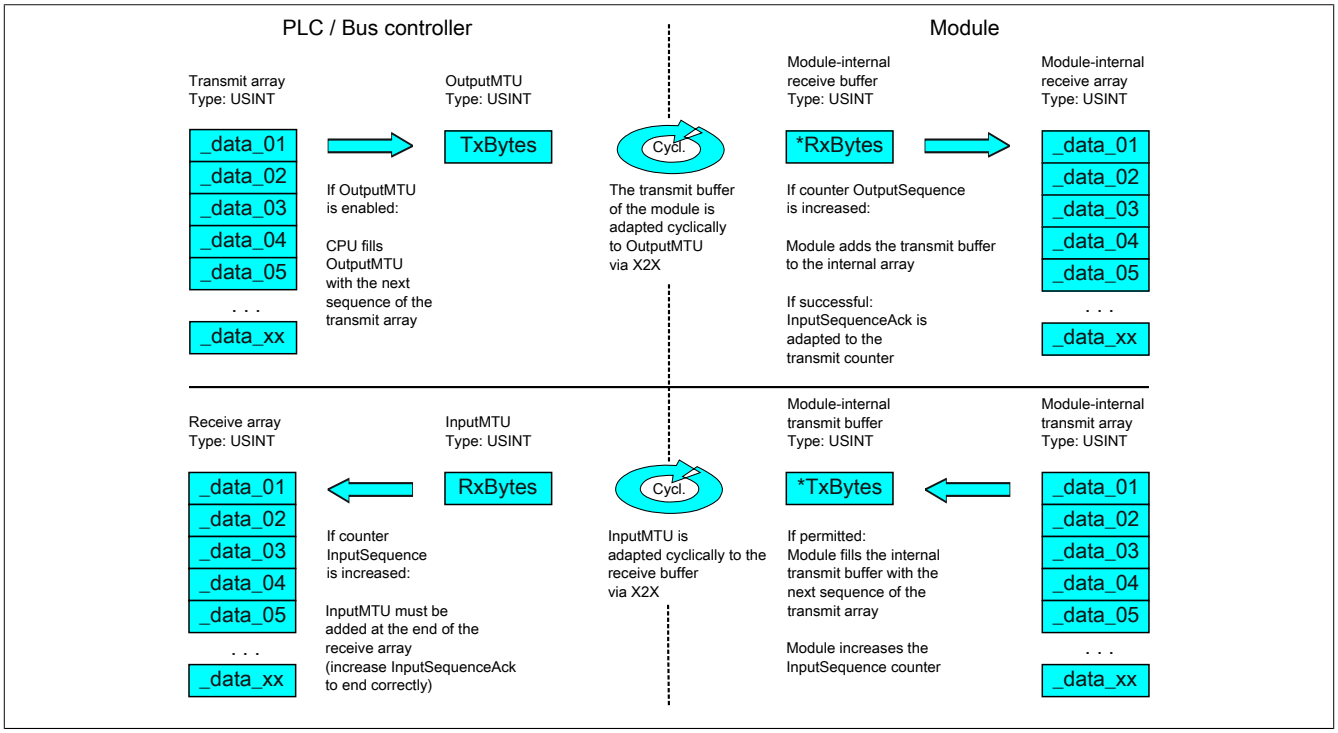


Figure 20: Flatstream communication

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 opposite 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.

3.9.10.4 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 CPU 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.

3.9.10.4.1 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.

3.9.10.4.1.1 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.

Information:

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

Data type	Values
USINT	See the module-specific register overview (theoretically: 3 to 27).

3.9.10.4.2 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 and ensure that communication is taking place properly, i.e. the transmitter issues the directive that the data should be accepted and the receiver acknowledges that a sequence has been transferred successfully.

3.9.10.4.2.1 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.

3.9.10.4.2.2 Transport of 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.
In the user program, only the Tx and Rx bytes from the CPU 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 CPU.

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

Data type	Values
USINT	0 to 255

3.9.10.4.2.3 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

3.9.10.4.2.4 Communication status of the CPU

Name:
OutputSequence

Register "OutputSequence" contains information about the communication status of the CPU. It is written by the CPU 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 disabled
		1	Output direction enabled
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 CPU. The CPU uses OutputSequenceCounter to direct the module to accept a sequence (the output direction must be synchronized when this happens).

OutputSyncBit

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

InputSequenceAck

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

InputSyncAck

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

3.9.10.4.2.5 Communication status of the module

Name:
InputSequence

Register "InputSequence" contains information about the communication status of the module. It is written by the module and should only be read by the CPU.

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 CPU 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 CPU. This indicates that the module is ready to receive data.

3.9.10.4.2.6 Relationship between OutputSequence and InputSequence

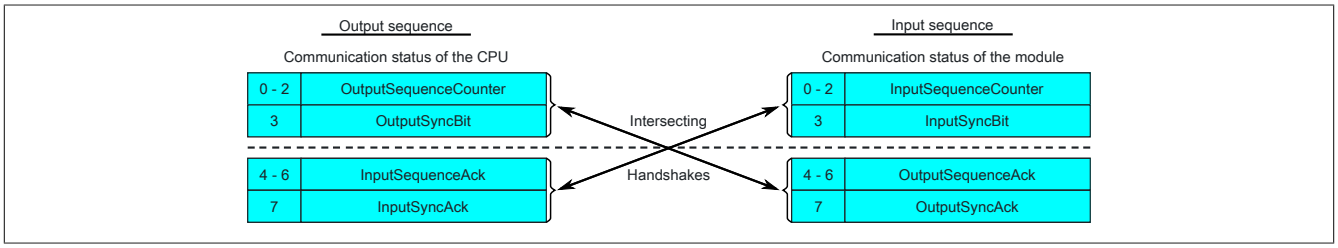


Figure 21: Relationship between OutputSequence and InputSequence

Registers "OutputSequence" and "InputSequence" are logically composed of 2 half-bytes. The low part signals to the opposite station whether a channel should be opened or if 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 opposite 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 opposite 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.

3.9.10.4.3 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 (CPU 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 CPU to the module.

Algorithm

1) The CPU must write 000 to OutputSequenceCounter and reset OutputSyncBit. The CPU 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 CPU registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The CPU 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 CPU registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The CPU 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 CPU can transmit data to the module.

Synchronization in the input direction (CPU 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 CPU.

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 CPU is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The CPU has to 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 CPU is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The CPU has to 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 CPU 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 CPU.

3.9.10.4.4 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.

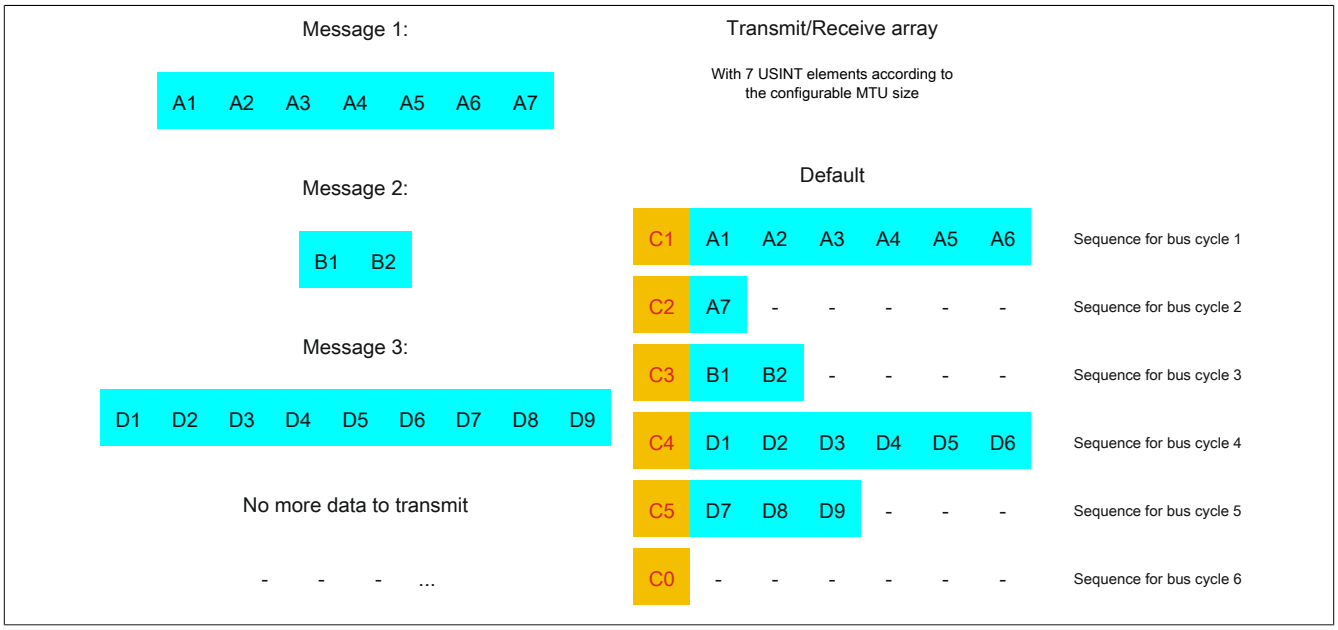


Figure 22: Transmit/Receive array (default)

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 13: 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 14: Flatstream determination of the control bytes for the default configuration example (part 2)

3.9.10.4.5 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.

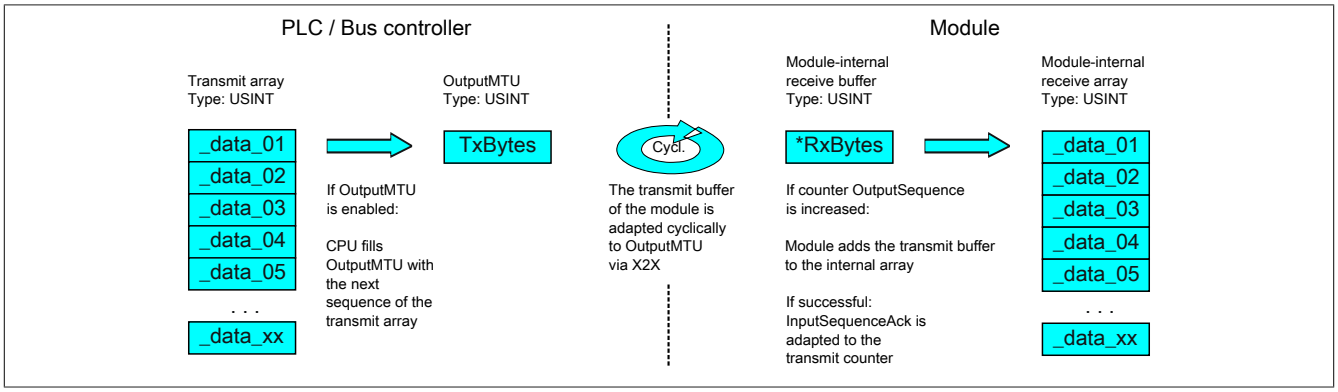


Figure 23: Flatstream communication (output)

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 CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU 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 CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU 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 CPU 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 CPU 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 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

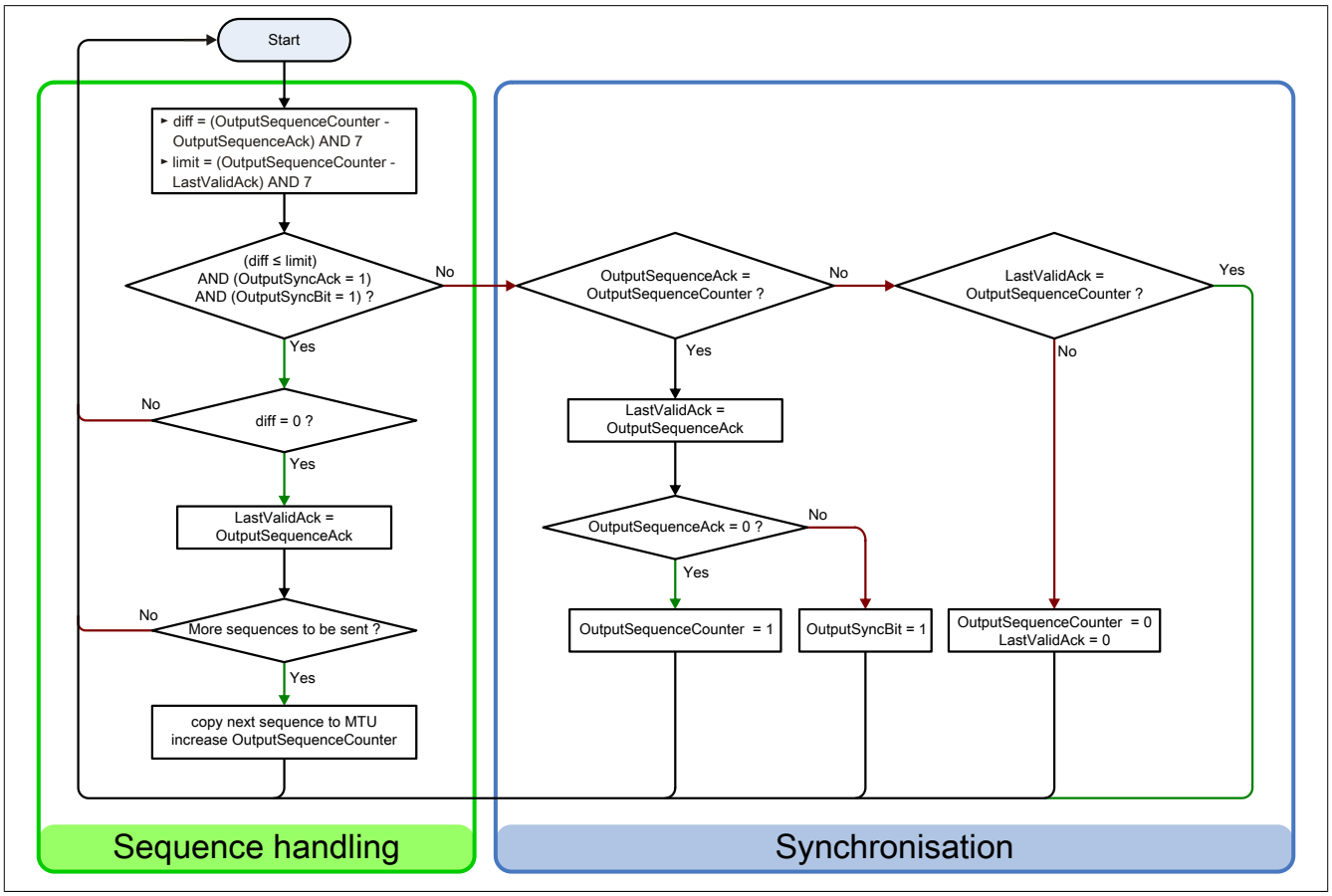


Figure 24: Flowchart for the output direction

3.9.10.4.6 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.

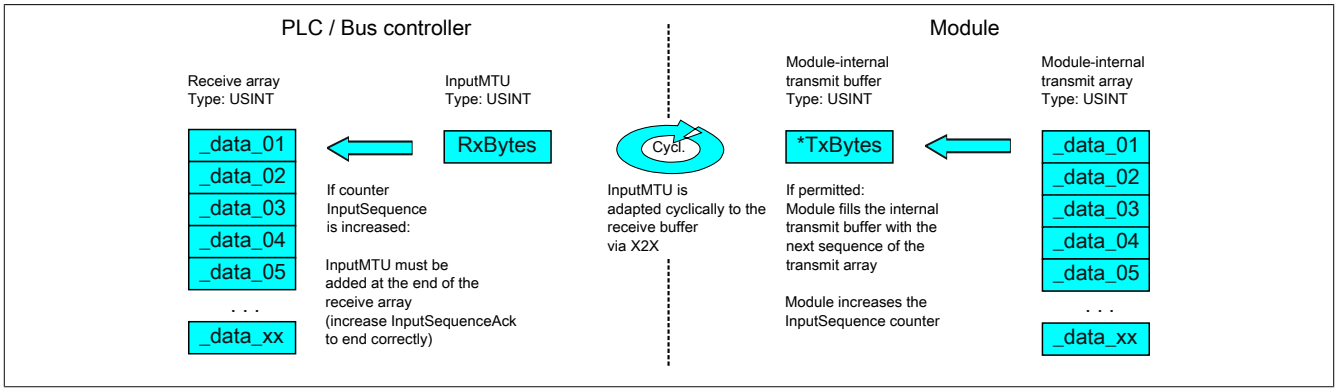


Figure 25: Flatstream communication (input)

Algorithm

0) Cyclic status query: - The CPU 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 CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU 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

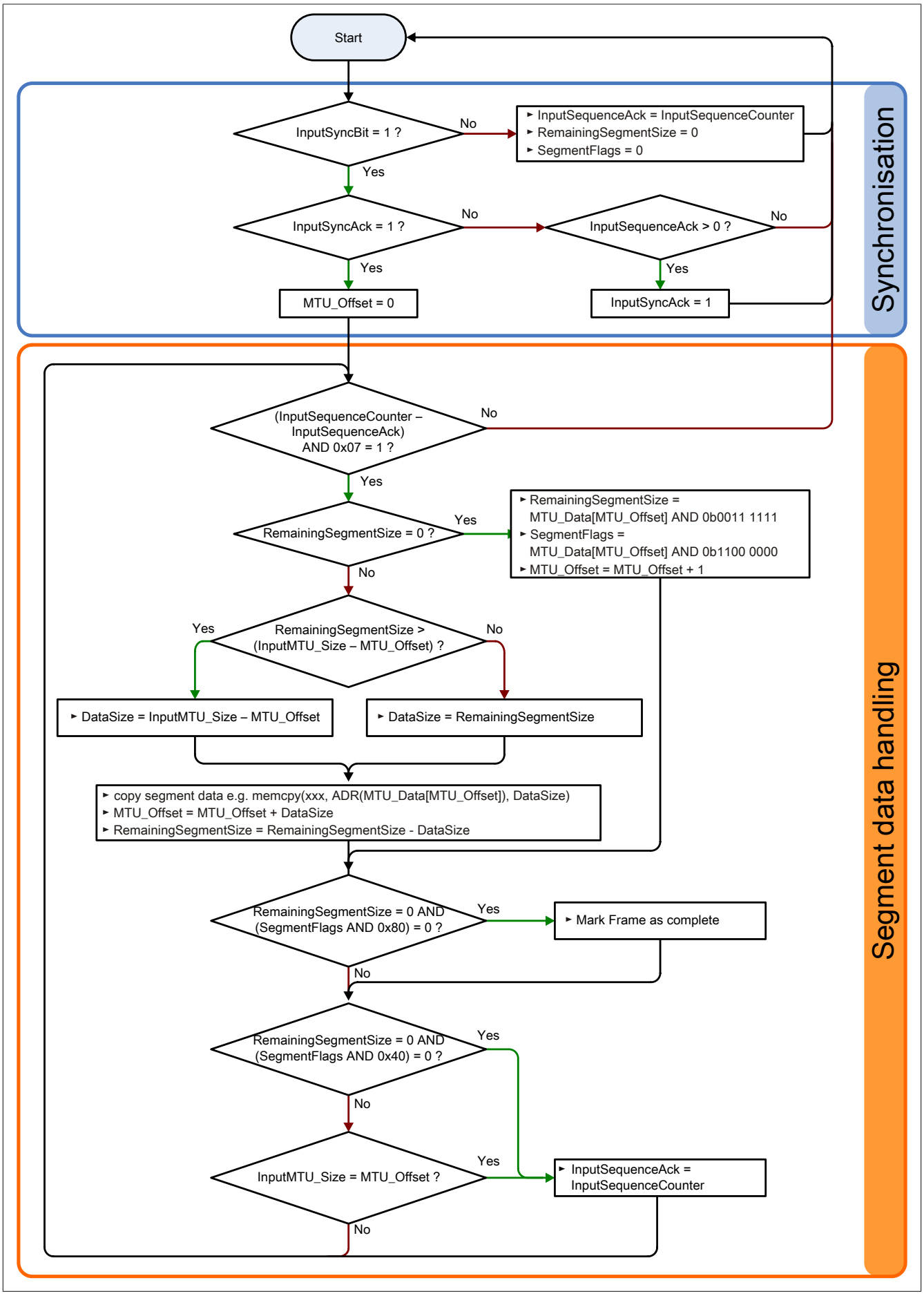


Figure 26: Flowchart for the input direction

3.9.10.4.7 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.

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

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 opposite station's SequenceAck and continue the transfer from this point.

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.

3.9.10.4.8 Flatstream mode

Name:
FlatstreamMode

In the input direction, the transmit array is generated automatically. This register offers 2 options to the user that allow an incoming data stream to have a more compact arrangement. 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.

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		

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.

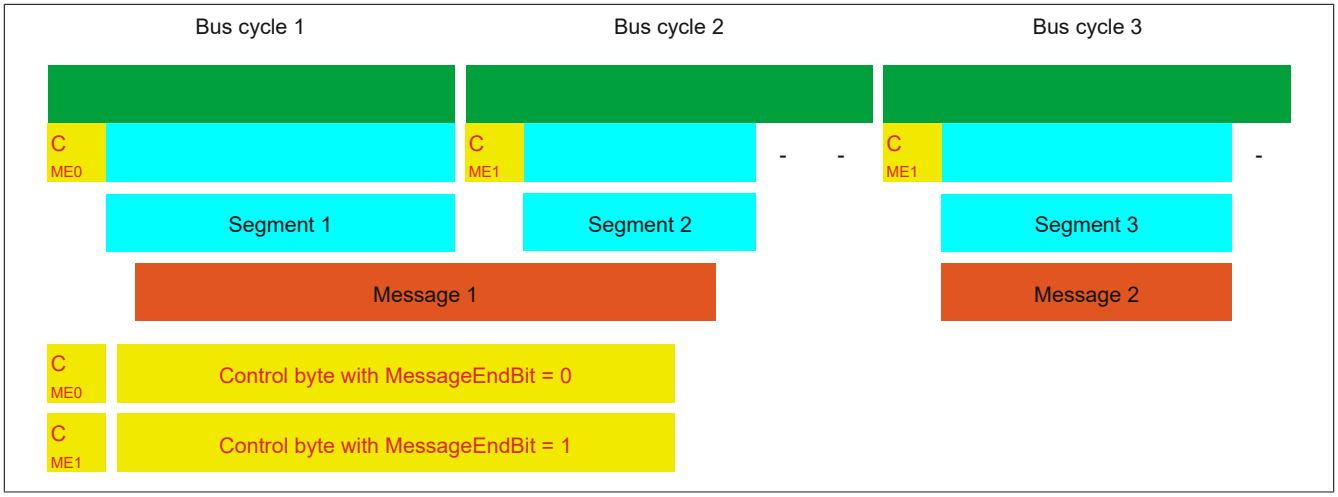


Figure 27: Message arrangement in the MTU (default)

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.

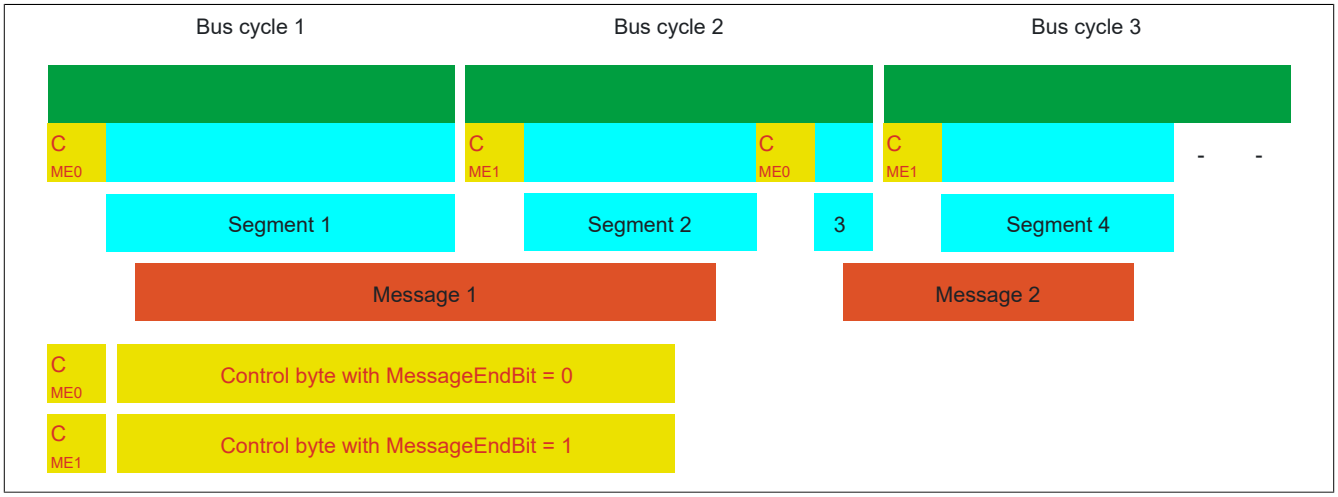


Figure 28: Arrangement of messages in the MTU (MultiSegmentMTUs)

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.

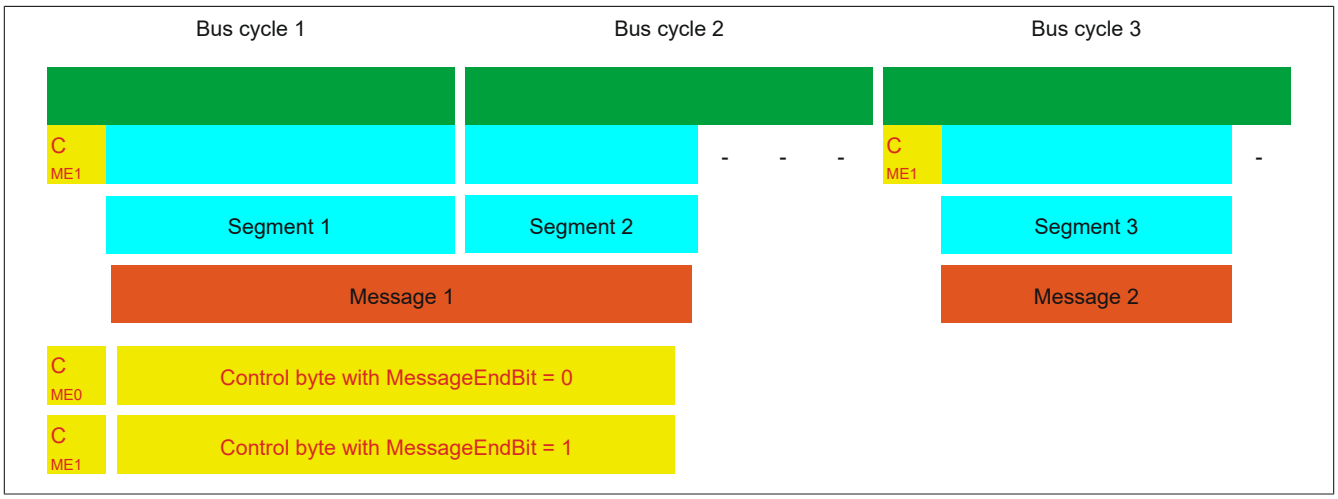


Figure 29: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

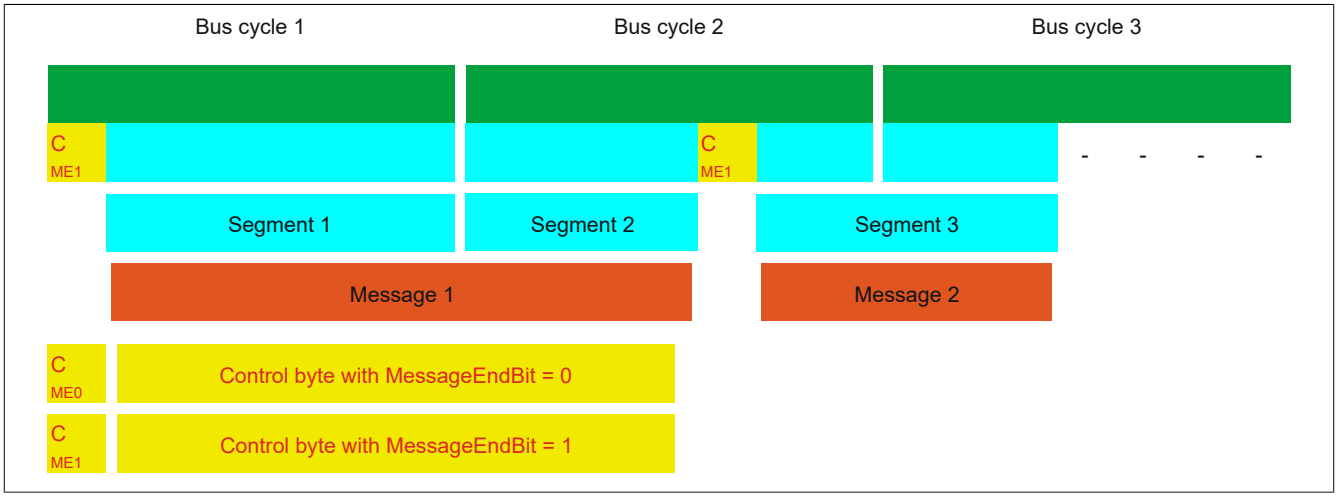


Figure 30: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

3.9.10.4.9 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.

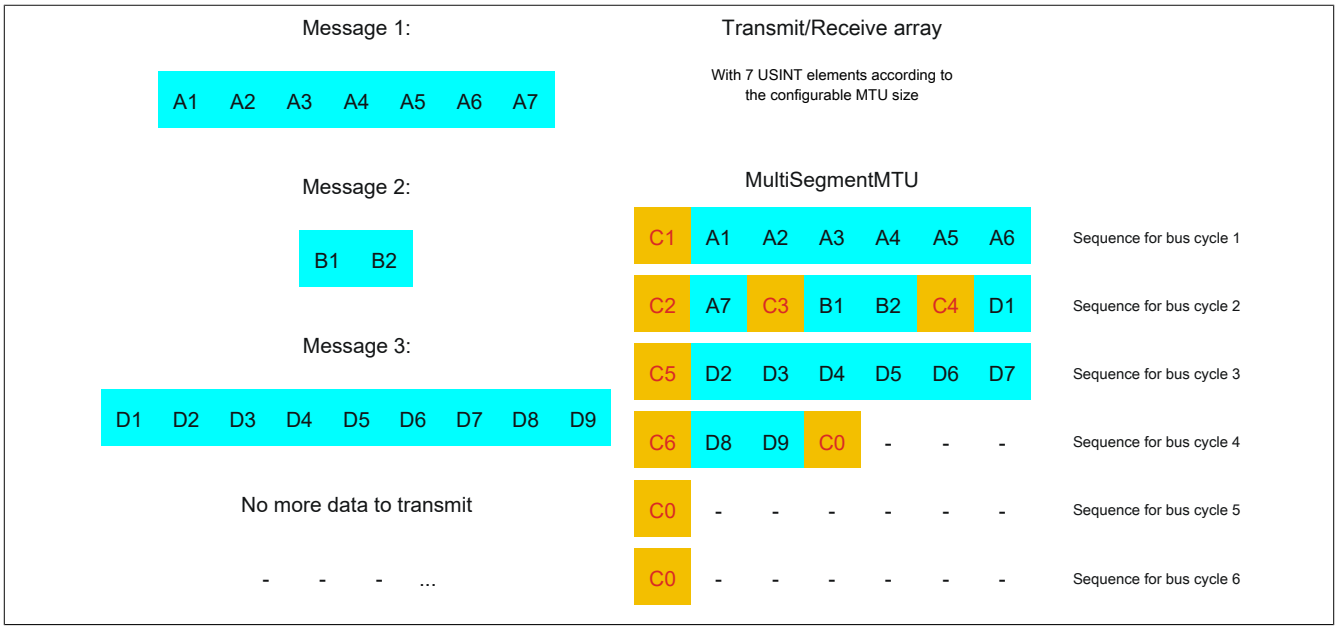


Figure 31: Transmit/receive array (MultiSegmentMTUs)

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 15: 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 16: 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.

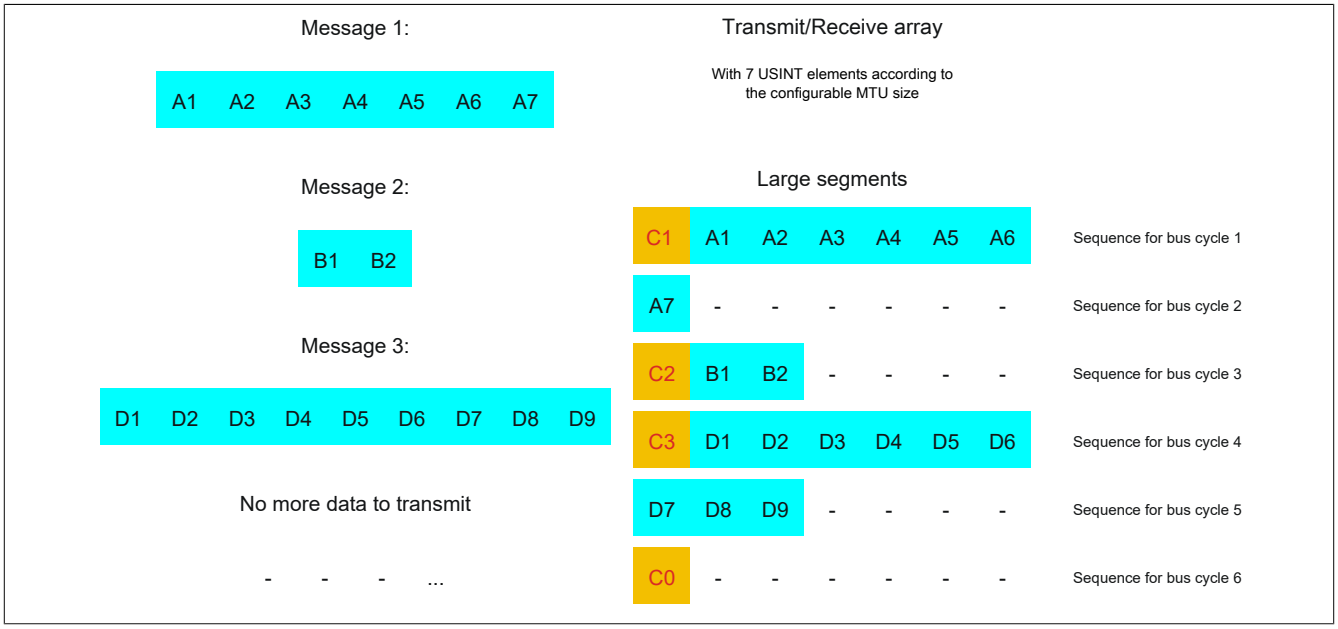


Figure 32: Transmit/receive array (large segments)

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 17: 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.

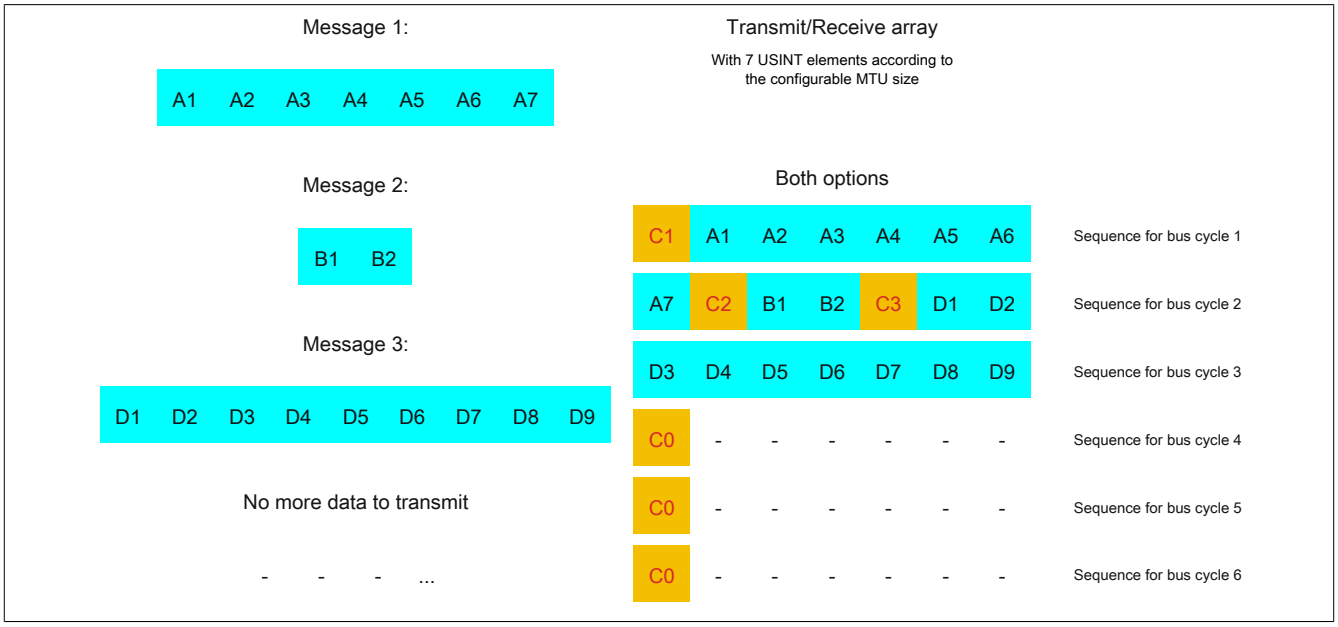


Figure 33: Transmit/receive array (large segments and MultiSegmentMTUs)

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

3.9.10.5 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.

3.9.10.5.1 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)

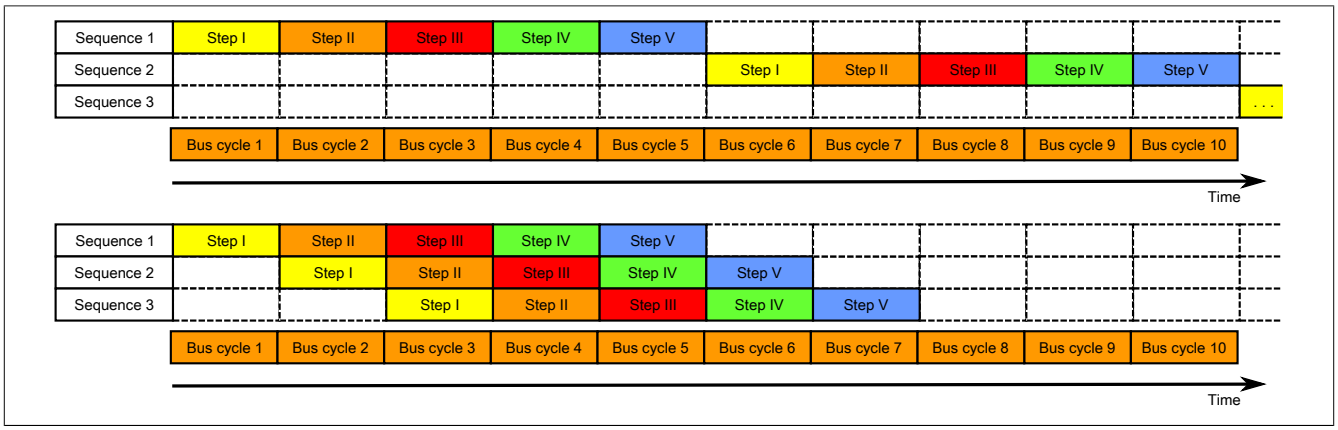


Figure 34: Comparison of transfer without/with Forward

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.

3.9.10.5.2 Configuration

The Forward function must only be enabled for the input direction. 2 additional configuration registers are available for doing so. 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.

3.9.10.5.2.1 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

3.9.10.5.2.2 Delay time

Name:
ForwardDelay

Register "ForwardDelay" is used to specify the delay time 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.

Data type	Values
UINT	0 to 65535 [µs] Default: 0

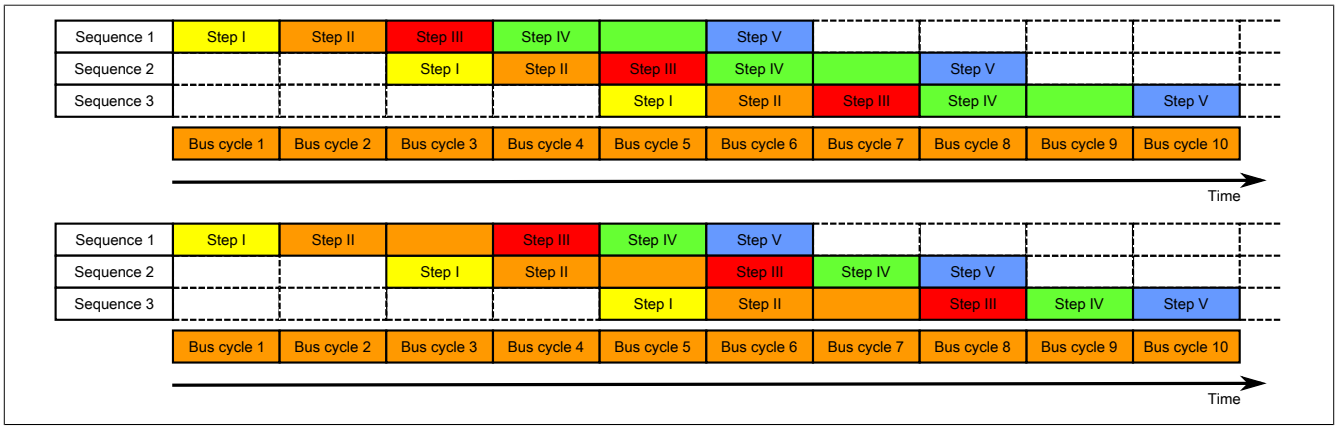


Figure 35: Effect of ForwardDelay when using Flatstream communication with the Forward function

In the program, it is important to make sure that the CPU 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 CPU has more time to process the incoming InputSequence or InputMTU.

3.9.10.5.3 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 CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU 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 CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU must transfer the current part of the transmit array to OutputMTU. - The CPU must increase OutputSequenceCounter for the sequence to be accepted by the module. - The CPU 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 CPU 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 CPU 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 CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU 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 Sequence-Counter 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 CPU 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.

3.9.10.5.4 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).

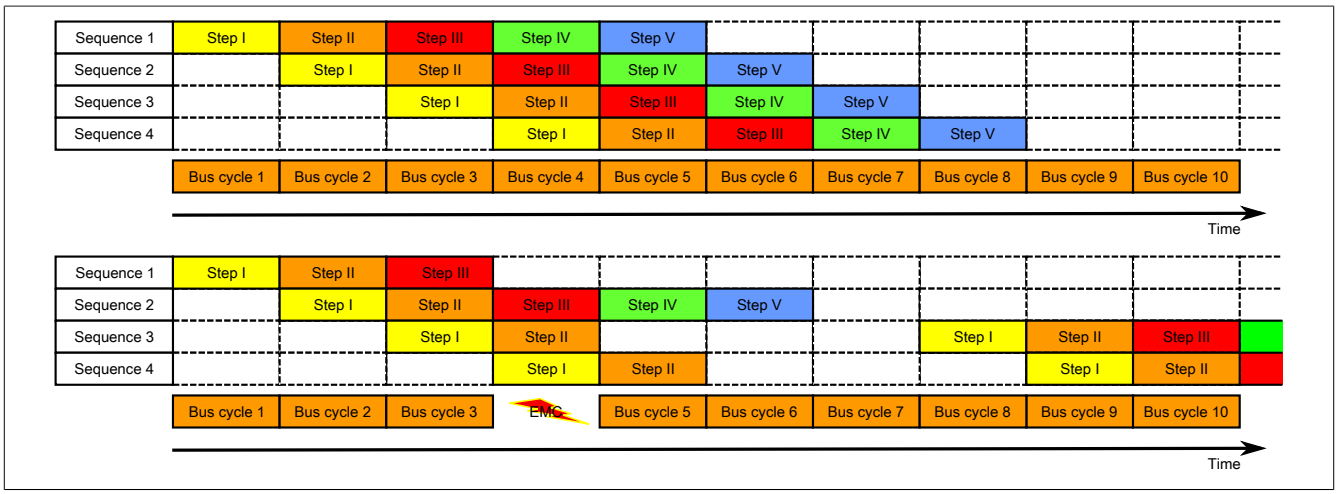


Figure 36: Effect of a lost bus cycle

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.

3.9.11 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 hand-shake or similar measures)

3.9.12 Acyclic frame size

Name:
AsynSize

When the stream is used, data is exchanged internally between the module and CPU. For this purpose, a defined amount of acyclic bytes is reserved for this slot.
Increasing the acyclic frame size leads to increased data throughput on this slot.

Information:

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

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

3.9.13 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 µs

3.9.14 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 µs

4 X20(c)CS1030

4.1 General information

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.

- RS485/RS422 interface for serial, remote connection of complex devices to the X20 system
- Integrated terminating resistor

4.2 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

4.3 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 19: X20CS1030, X20cCS1030 - Order data

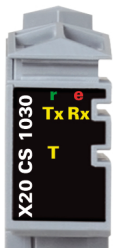
4.4 Technical data

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	
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 GL	Temperature: **B** (0 - 55°C) Humidity: **B** (up to 100%) Vibration: **B** (4 g) EMC: **B** (bridge and open deck)	
LR	ENV1	
KR	Yes	
ABS	Yes	
EAC	Yes	
KC	Yes	-
Interfaces		
Interface IF1		
Signal	RS485/RS422	
Variant	Connection made using 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	
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 20: X20CS1030, X20cCS1030 - Technical data

4.5 LED status indicators

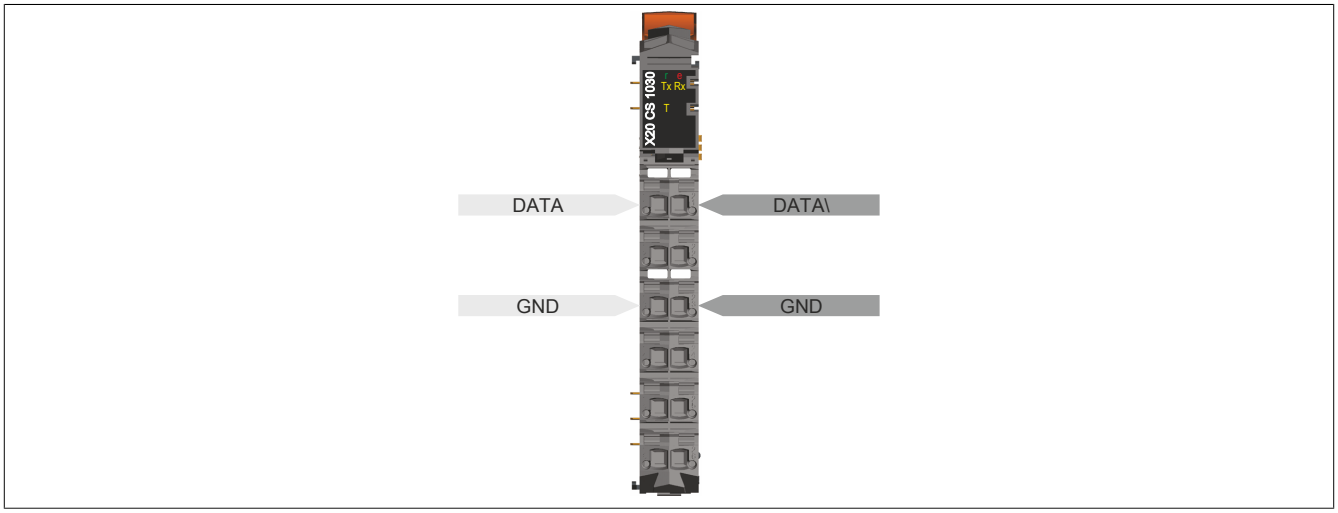
For a description of the various operating modes, see section "Additional information - Diagnostic LEDs" in the X20 system user's manual.

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 120
			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

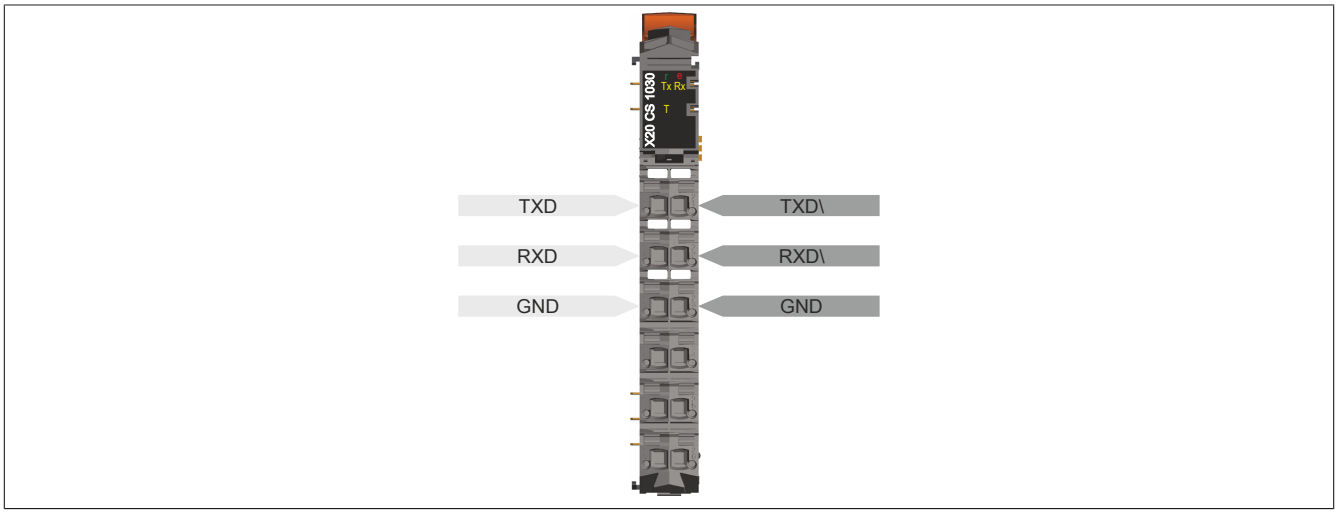
1) Depending on the configuration, a firmware update can take up to several minutes.

4.6 Pinout

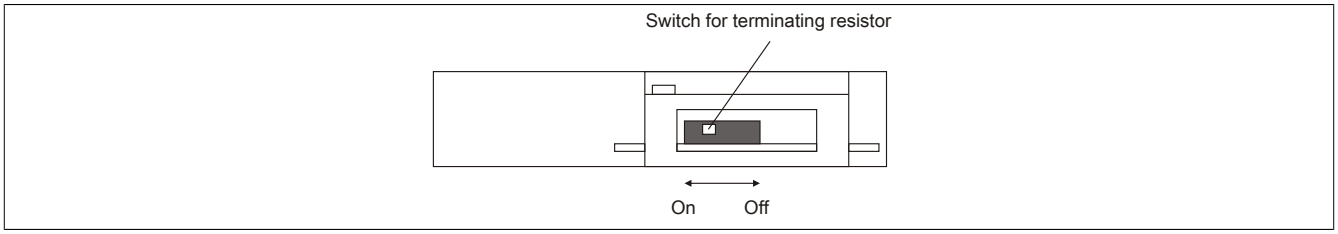
RS485 mode



RS422 mode



4.7 Terminating resistor



A terminating resistor is already integrated in the communication module. The terminating resistor can be switched on or off in RS485 bus mode with a switch on the bottom of the housing. An enabled terminating resistor is indicated by LED "T".

Information:

In RS422 bus mode, the terminating resistor between RXD and RXD\ is not switched off. This must be taken into account in the cable routing.

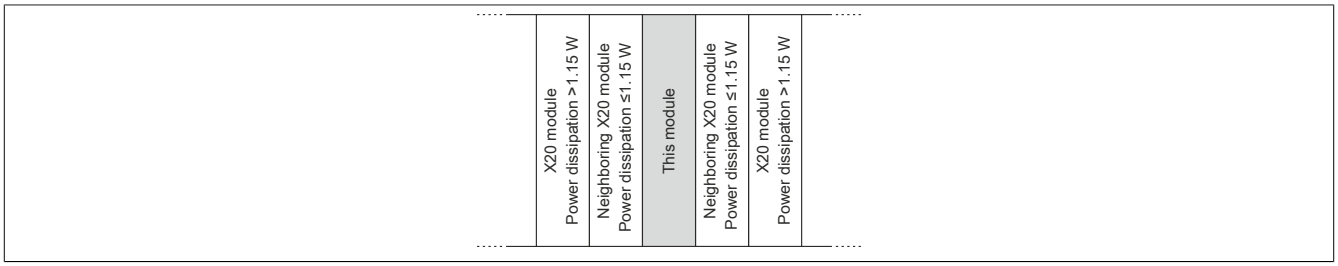
No limitations in all other modes.

4.8 Derating

There is no derating when operated below 55°C.

During operation over 55°C, the power dissipation of the modules to the left and right of this module is not permitted to exceed 1.15 W!

For an example of calculating the power dissipation of I/O modules, see section "Mechanical and electrical configuration - Power dissipation of I/O modules" in the X20 user's manual.



4.9 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.

4.10 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.**

4.11 Register description

4.11.1 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.

4.11.2 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 relatively 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_RxStateIgnoreMask	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				

4.11.3 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 the data stream (see "Flatstream communication" on page 121).

When using function model Flatstream, the user can choose whether to use library "AsFltGen" in AS 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	rxlLock	UINT				•
70	rxlUnlock	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_RxStateIgnoreMask	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			•	

4.11.4 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.

4.11.4.1 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).

4.11.4.2 CAN I/O bus controller

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

4.11.5 Serial interface - Configuration

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

4.11.5.1 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.

4.11.5.2 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

4.11.5.3 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)

4.11.5.4 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

4.11.5.5 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

4.11.6 Handshake - Configuration

In order to guarantee that serial communication runs smoothly, the size of the receive buffer in the module must be known. In addition, the user can configure a software or hardware handshake algorithm.

4.11.6.1 Locking the receive buffer

Name:
rxlLock
rxlLock_CANIO

This register is used to configure the upper threshold of 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".

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

4.11.6.2 Unlocking the receive buffer

Name:
rxlUnlock
rxlUnlock_CANIO

This register is used to configure the lower threshold of 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".

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

4.11.6.3 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.

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.

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

4.11.6.4 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.

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.

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

4.11.6.5 Handshake repetition

Name:
hssPeriod
hssPeriod_CANIO

When using a software handshake, some applications require periodic repetition of the current status. The repeat time can be defined in this register in ms 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

4.11.7 Frame - Configuration

Different message termination codes can be specified in order to correctly create transmitted Tx frames and correctly interpret received Rx frames.

4.11.7.1 Terminating when a receive timeout occurs

Name:
rxCto
rxCto_CANIO

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

The message is considered to be terminated when nothing is transfered 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.

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

4.11.7.2 Terminating when a transmit timeout occurs

Name:
txCto
txCto_CANIO

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

The message is considered to be terminated when nothing is transfered 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.

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

4.11.7.3 Maximum number of bytes received

Name:
rxEomSize
rxEomSize_CANIO

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

The message is considered to be ended as soon as a frame with the specified size in bytes is transferred. The longest possible frame length is the size of the 4096-byte receive buffer. Larger frames cause the Receive Overrun error.

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

4.11.7.4 Maximum number of bytes transmitted

Name:
txEomSize
txEomSize_CANIO

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

The message is considered to be ended as soon as a frame with the specified size in bytes is transferred. The longest possible frame length is the size of the 4096-byte transmit buffer. The configured transmit timeout is maintained after the frame has been sent.

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

4.11.7.5 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)

4.11.7.6 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)

4.11.8 Status messages - Configuration

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

4.11.8.1 Error detection setting

Name:
CfO_RxStateIgnoreMask
CfO_RxStateIgnoreMask_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

4.11.8.2 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		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	

4.11.9 Status messages - Communication

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

4.11.9.1 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.

4.11.9.2 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	

4.11.10 Flatstream communication

4.11.10.1 Introduction

B&R offers an additional communication method for some modules. "Flatstream" was designed for X2X and POWERLINK networks and allows data transmission 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.

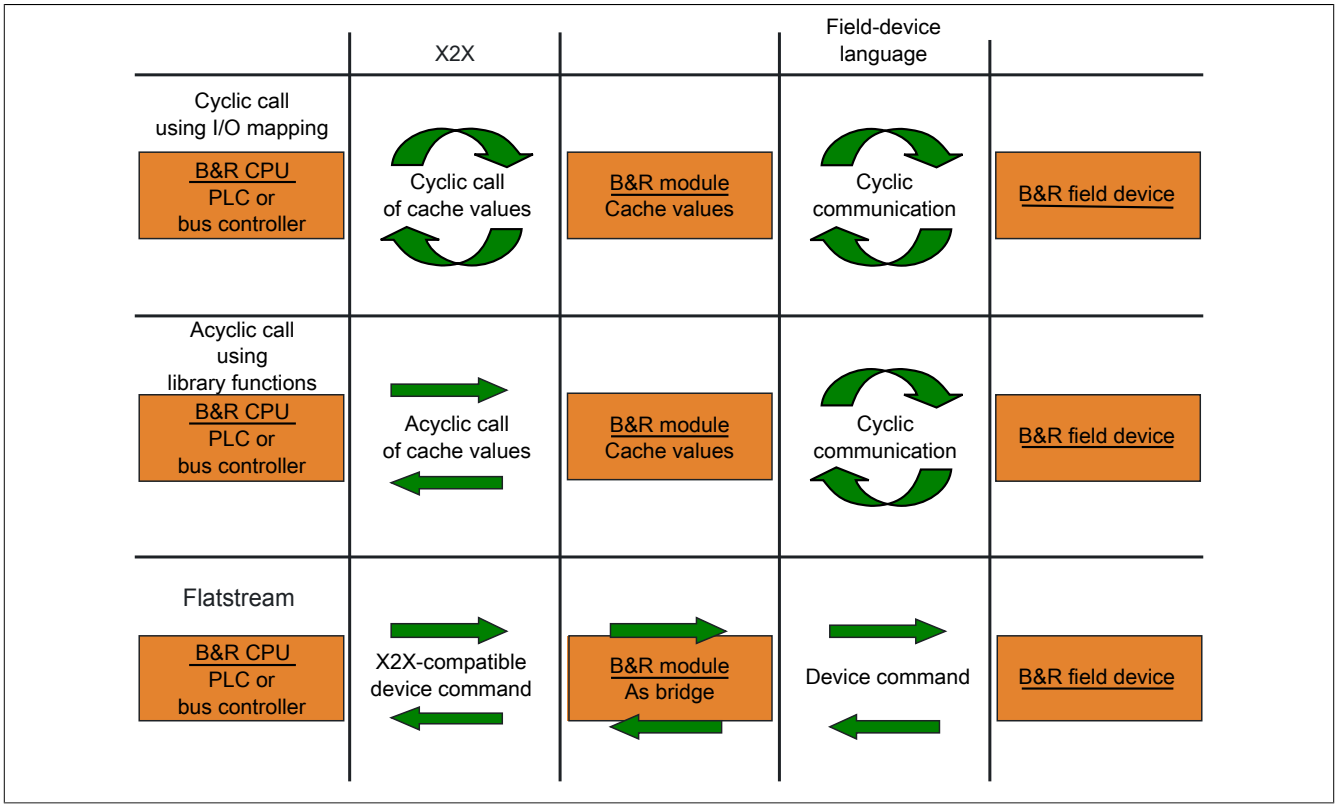


Figure 37: 3 types of communication

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

4.11.10.2 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 put back 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 transmission was faulty and that all affected sequences must be repeated.

4.11.10.3 The Flatstream principle

Requirement

Before Flatstream can be used, the respective communication direction must be synchronized, i.e. both communication partners cyclically query the sequence counter on the opposite 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 opposite 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.

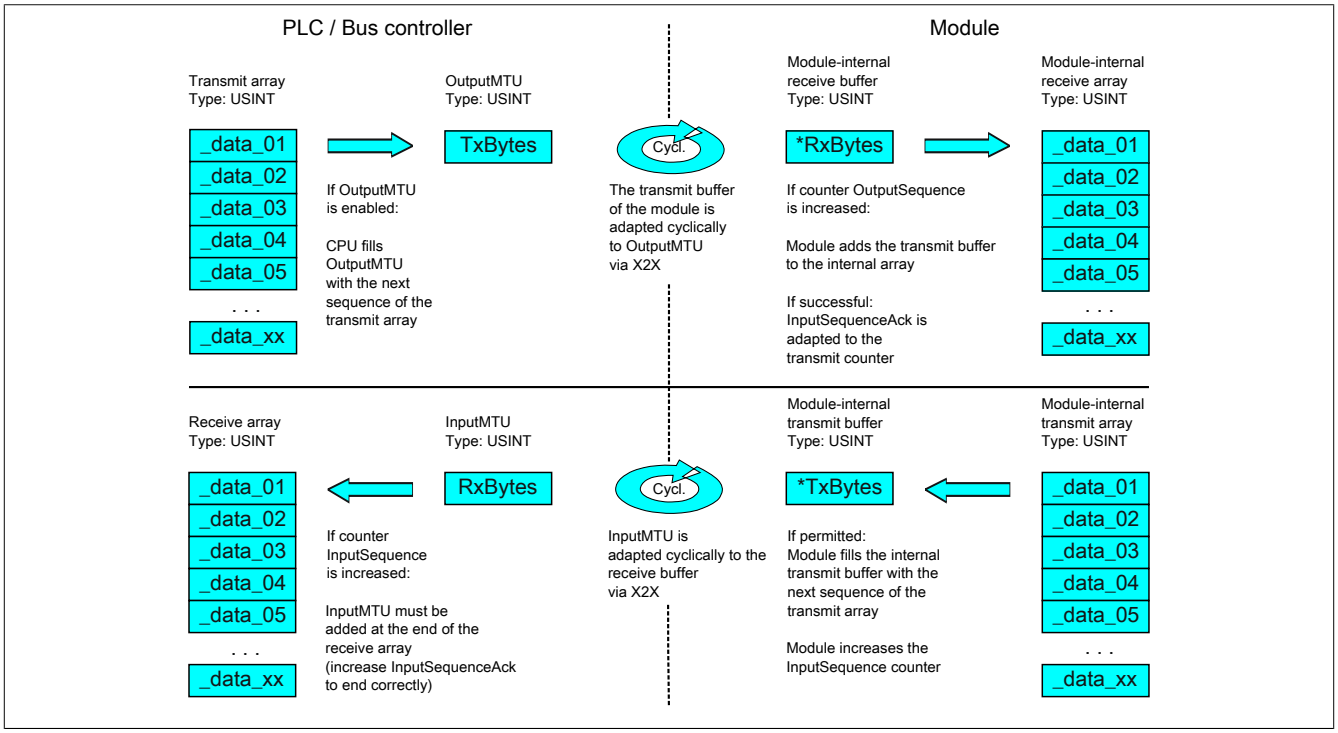


Figure 38: Flatstream communication

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 opposite 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.

4.11.10.4 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 CPU 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.

4.11.10.4.1 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.

4.11.10.4.1.1 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.

Information:

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

Data type	Values
USINT	See the module-specific register overview (theoretically: 3 to 27).

4.11.10.4.2 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 and ensure that communication is taking place properly, i.e. the transmitter issues the directive that the data should be accepted and the receiver acknowledges that a sequence has been transferred successfully.

4.11.10.4.2.1 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.

4.11.10.4.2.2 Transport of 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.
In the user program, only the Tx and Rx bytes from the CPU 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 CPU.

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

Data type	Values
USINT	0 to 255

4.11.10.4.2.3 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

4.11.10.4.2.4 Communication status of the CPU

Name:
OutputSequence

Register "OutputSequence" contains information about the communication status of the CPU. It is written by the CPU 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 disabled
		1	Output direction enabled
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 CPU. The CPU uses OutputSequenceCounter to direct the module to accept a sequence (the output direction must be synchronized when this happens).

OutputSyncBit

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

InputSequenceAck

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

InputSyncAck

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

4.11.10.4.2.5 Communication status of the module

Name:
InputSequence

Register "InputSequence" contains information about the communication status of the module. It is written by the module and should only be read by the CPU.

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 CPU 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 CPU. This indicates that the module is ready to receive data.

4.11.10.4.2.6 Relationship between OutputSequence and InputSequence

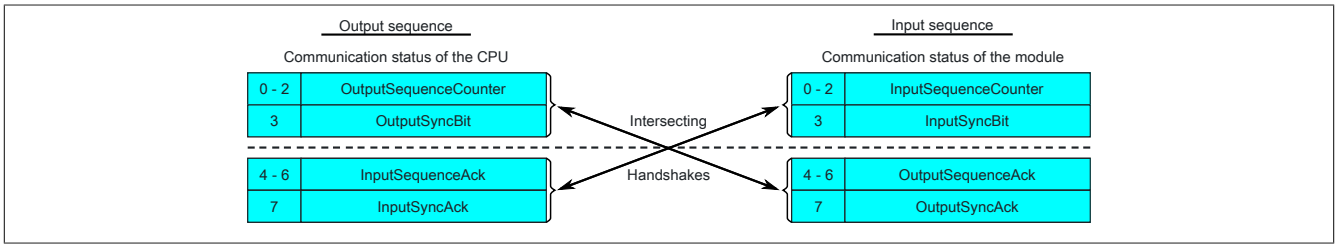


Figure 39: Relationship between OutputSequence and InputSequence

Registers "OutputSequence" and "InputSequence" are logically composed of 2 half-bytes. The low part signals to the opposite station whether a channel should be opened or if 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 opposite 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 opposite 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.

4.11.10.4.3 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 (CPU 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 CPU to the module.

Algorithm

1) The CPU must write 000 to OutputSequenceCounter and reset OutputSyncBit. The CPU 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 CPU registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The CPU 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 CPU registers the expected values in OutputSequenceAck and OutputSyncAck, it is permitted to increment OutputSequenceCounter. The CPU 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 CPU can transmit data to the module.

Synchronization in the input direction (CPU 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 CPU.

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 CPU is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The CPU has to 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 CPU is not permitted to accept the current contents of InputMTU since the channel is not yet synchronized. The CPU has to 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 CPU 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 CPU.

4.11.10.4.4 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.

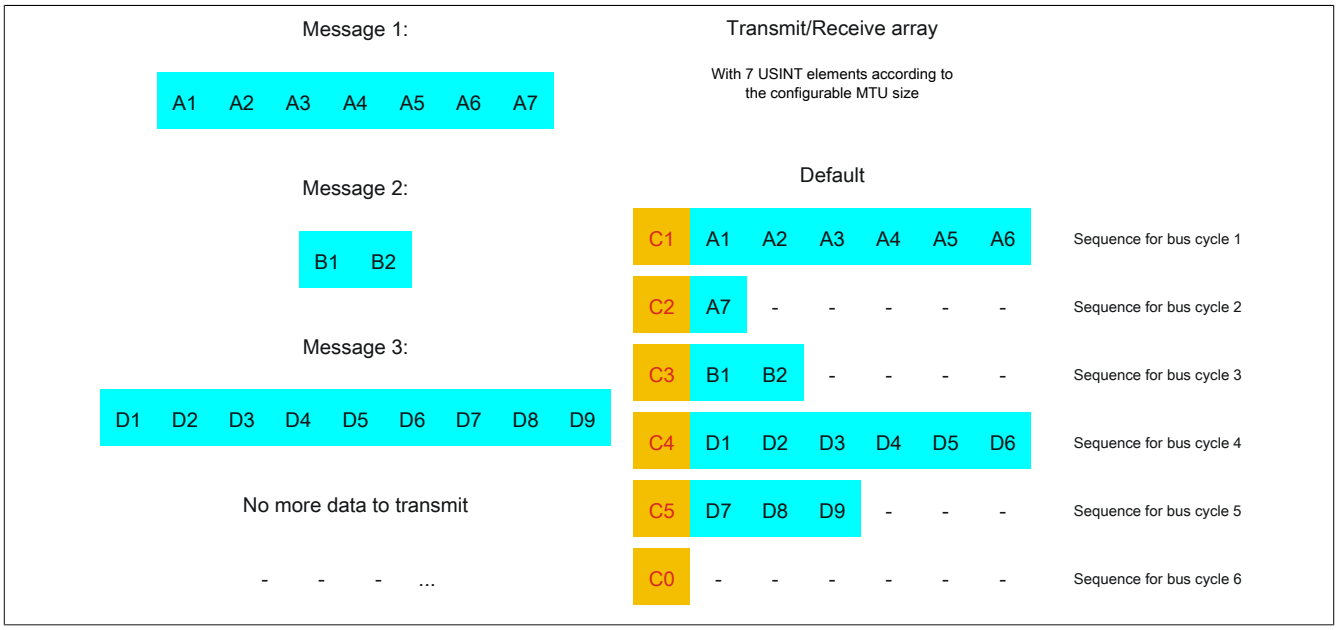


Figure 40: Transmit/Receive array (default)

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 21: 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 22: Flatstream determination of the control bytes for the default configuration example (part 2)

4.11.10.4.5 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.

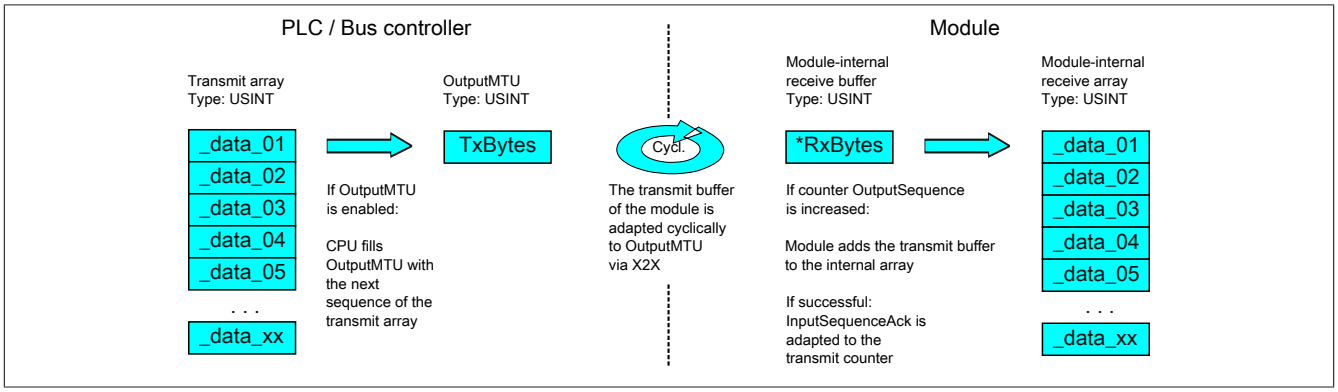


Figure 41: Flatstream communication (output)

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 CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU 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 CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU 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 CPU 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 CPU 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 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

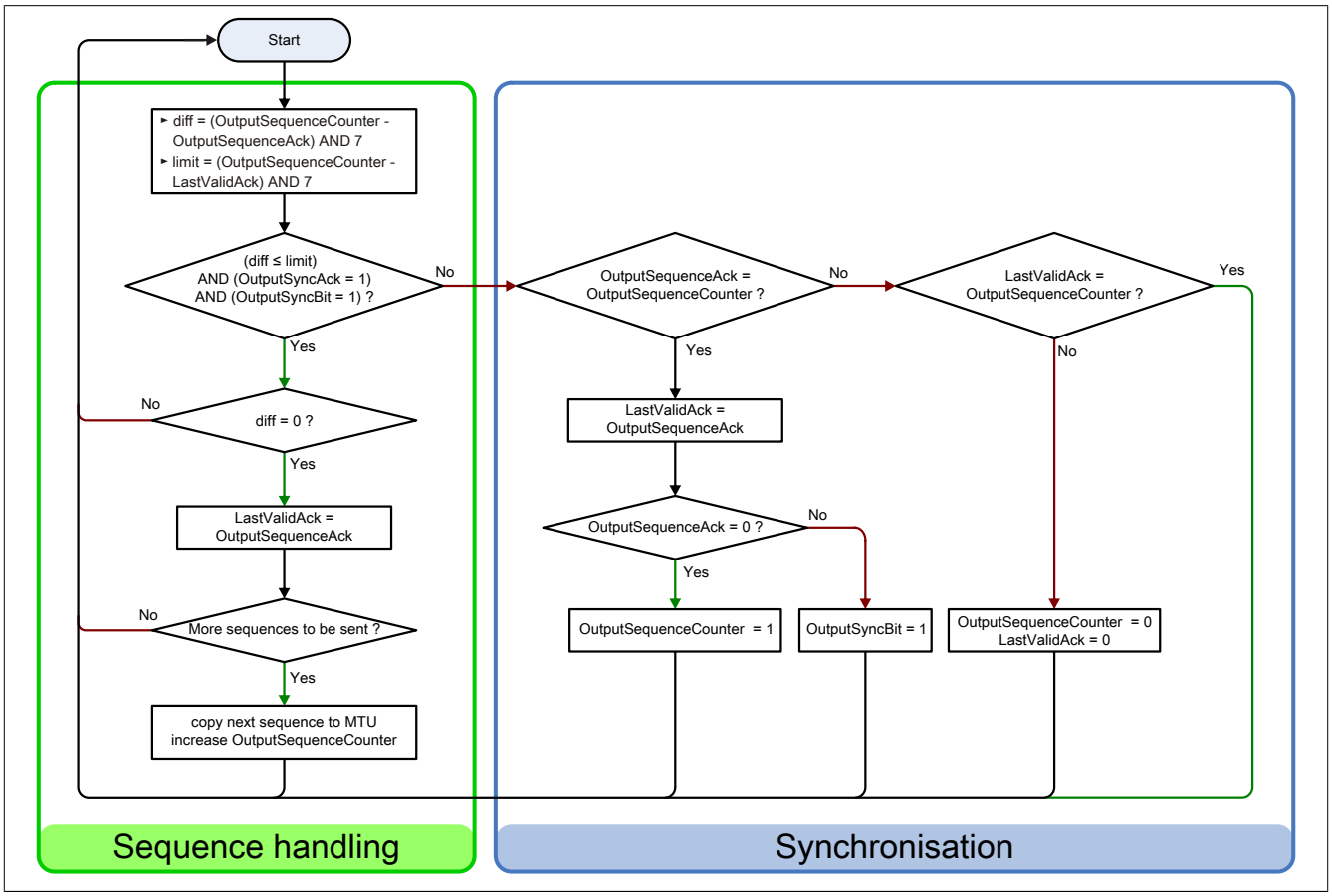


Figure 42: Flowchart for the output direction

4.11.10.4.6 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.

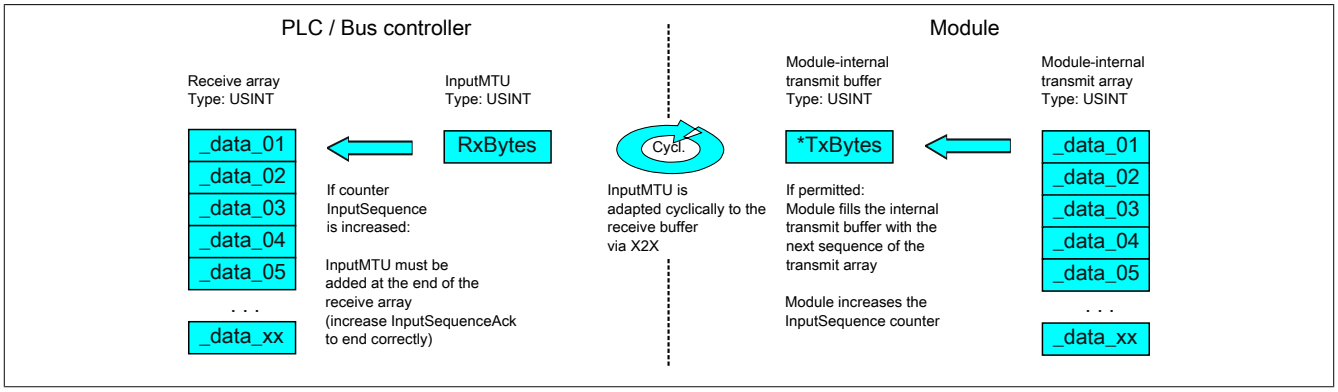


Figure 43: Flatstream communication (input)

Algorithm

0) Cyclic status query: - The CPU 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 CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU 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

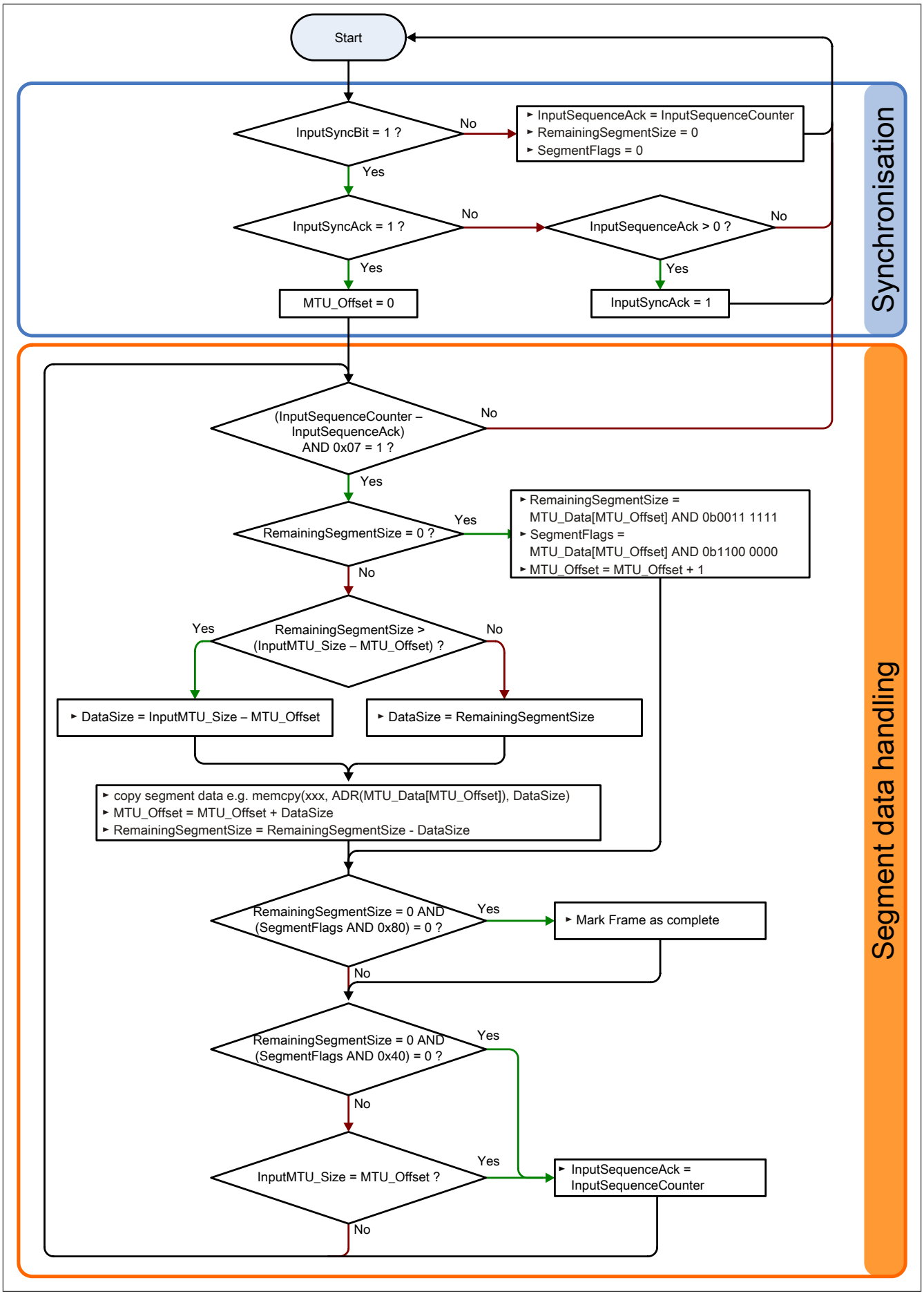


Figure 44: Flowchart for the input direction

4.11.10.4.7 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.

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

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 opposite station's SequenceAck and continue the transfer from this point.

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.

4.11.10.4.8 Flatstream mode

Name:
FlatstreamMode

In the input direction, the transmit array is generated automatically. This register offers 2 options to the user that allow an incoming data stream to have a more compact arrangement. 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.

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		

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.

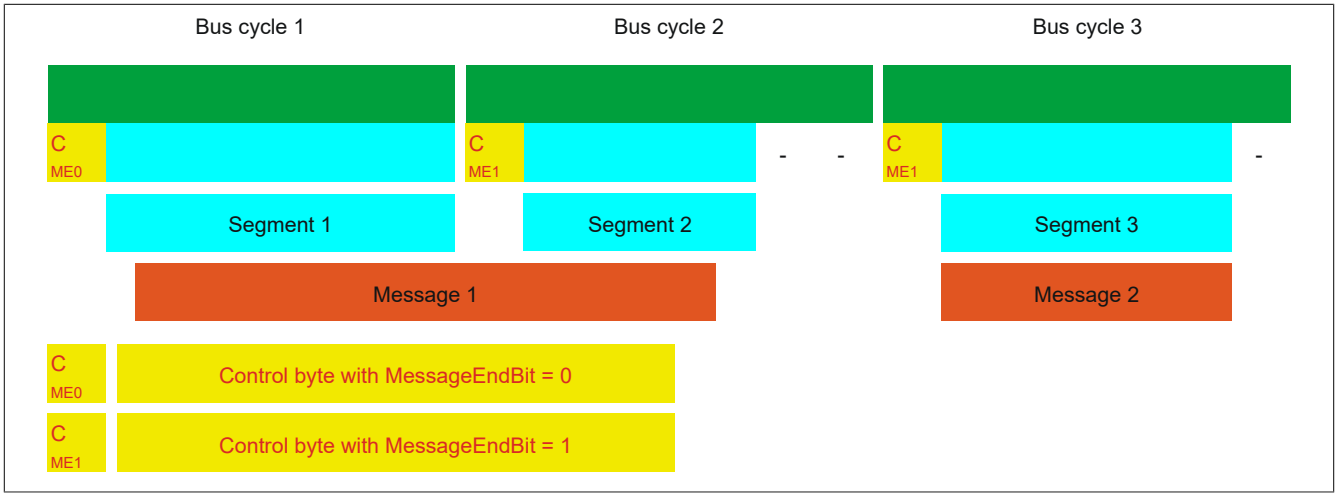


Figure 45: Message arrangement in the MTU (default)

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.

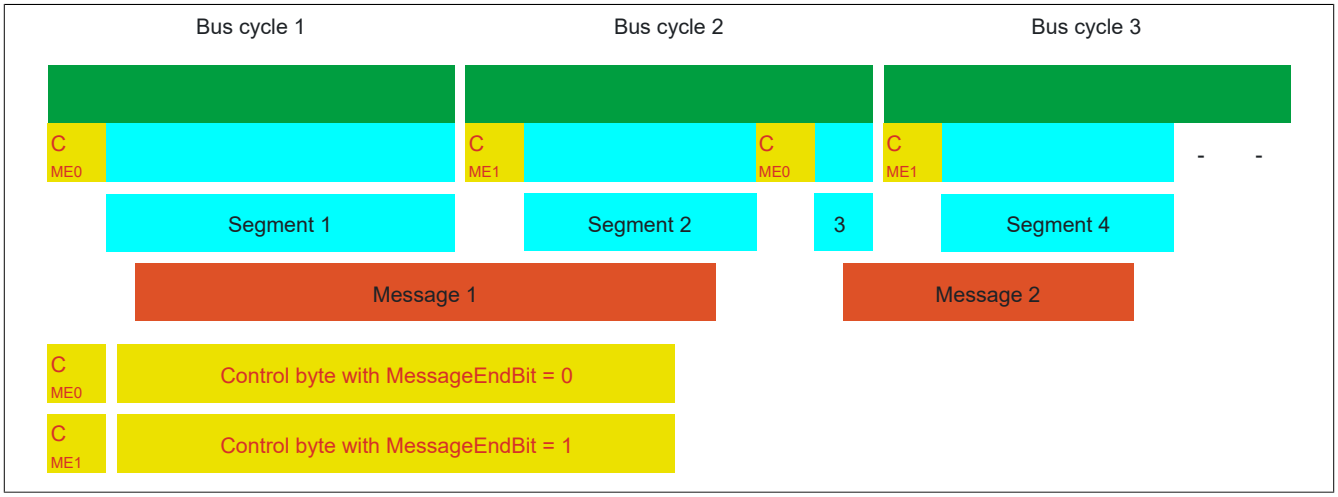


Figure 46: Arrangement of messages in the MTU (MultiSegmentMTUs)

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.

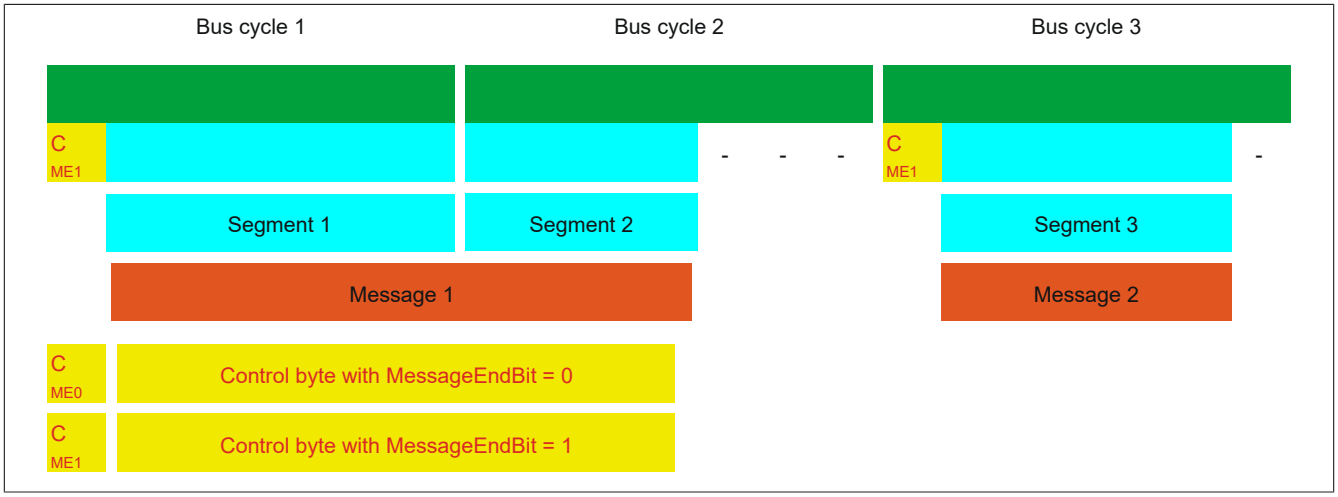


Figure 47: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

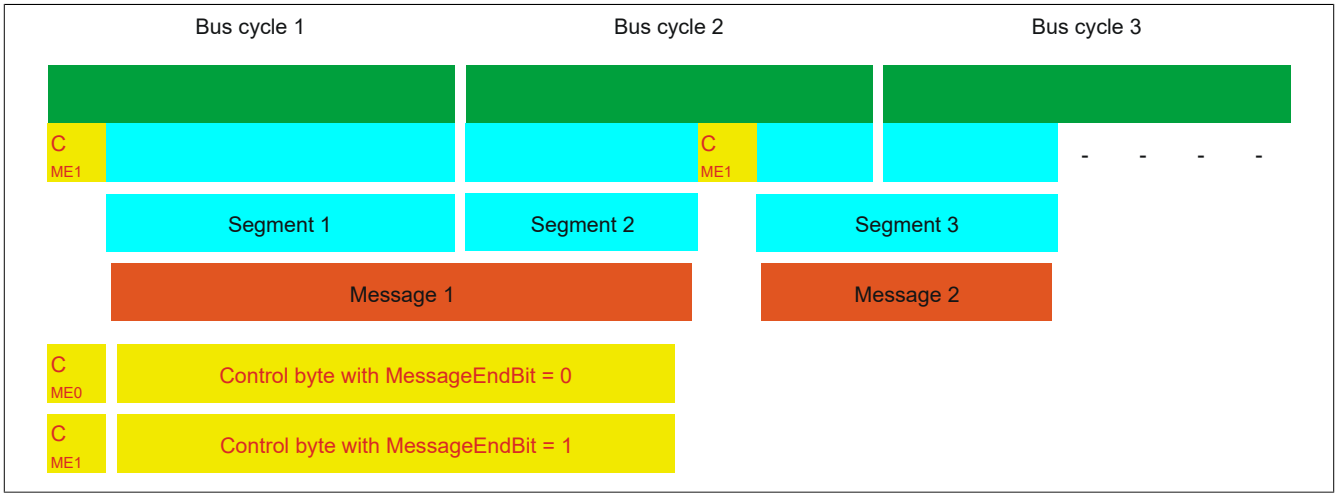


Figure 48: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

4.11.10.4.9 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.

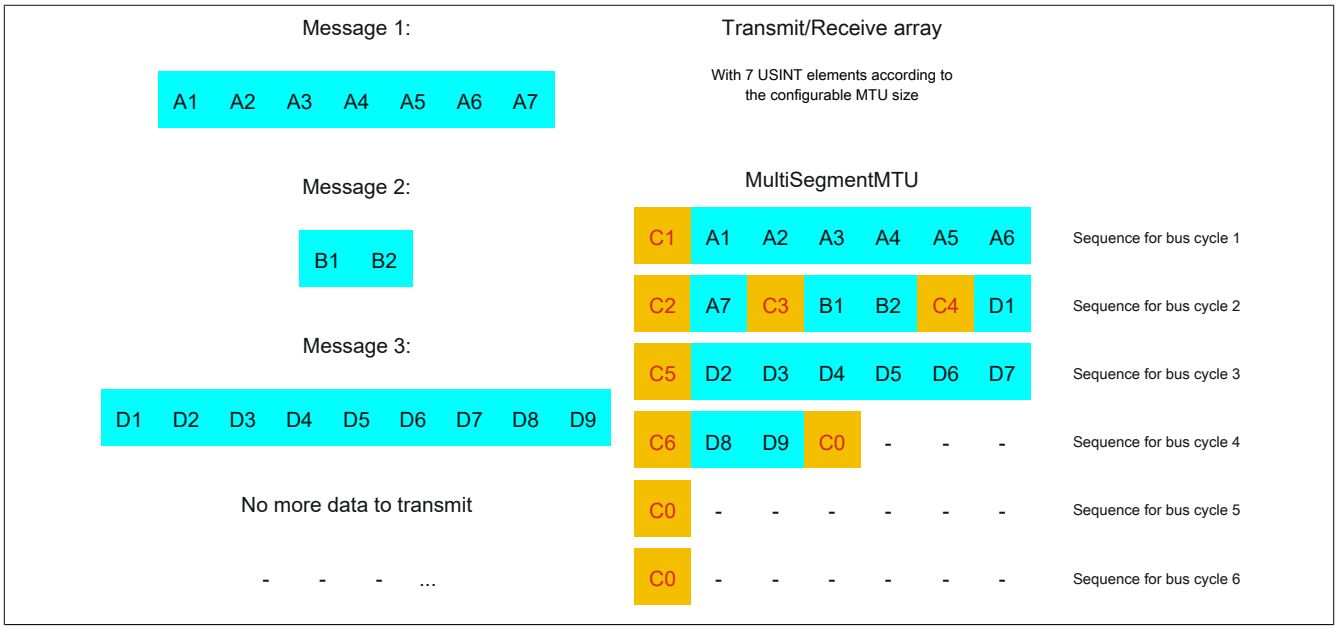


Figure 49: Transmit/receive array (MultiSegmentMTUs)

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 23: 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 24: 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.

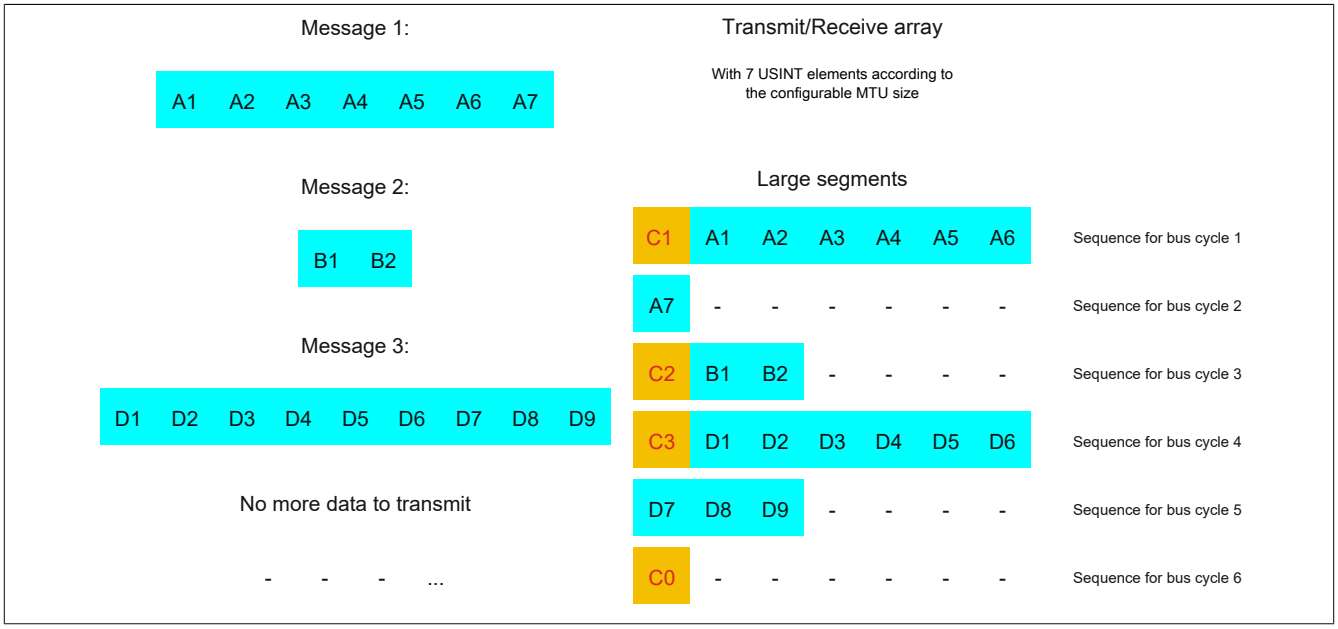


Figure 50: Transmit/receive array (large segments)

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 25: 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.

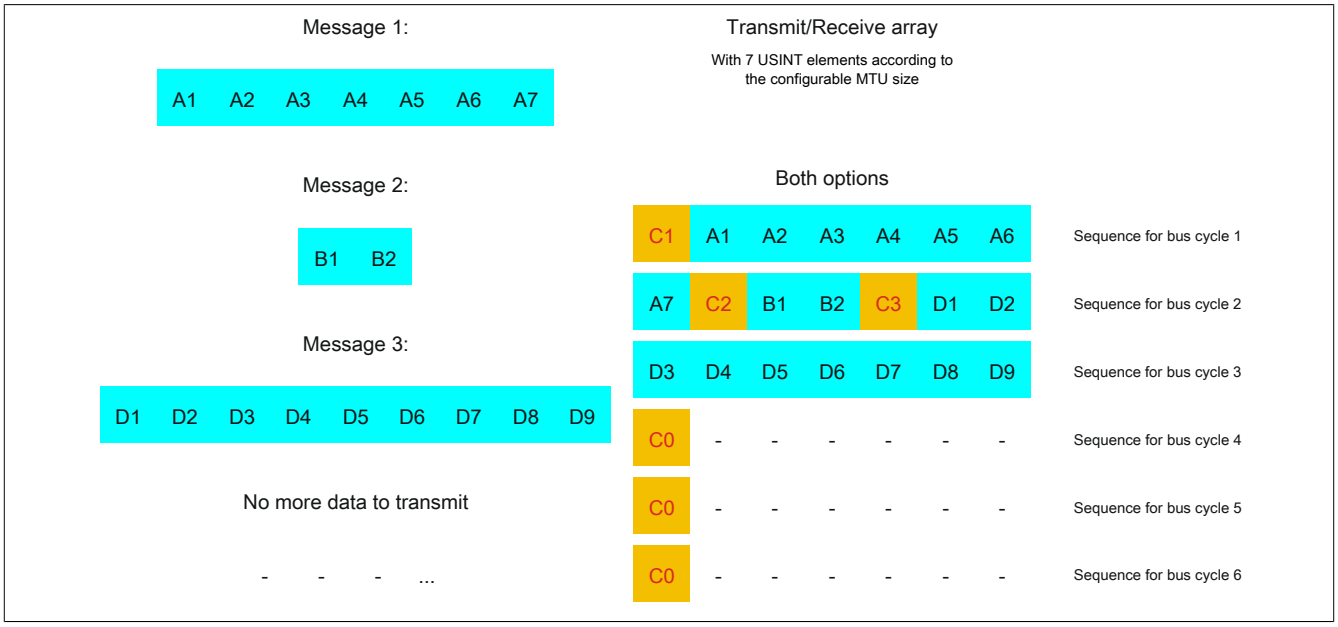


Figure 51: Transmit/receive array (large segments and MultiSegmentMTUs)

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

4.11.10.5 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.

4.11.10.5.1 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)

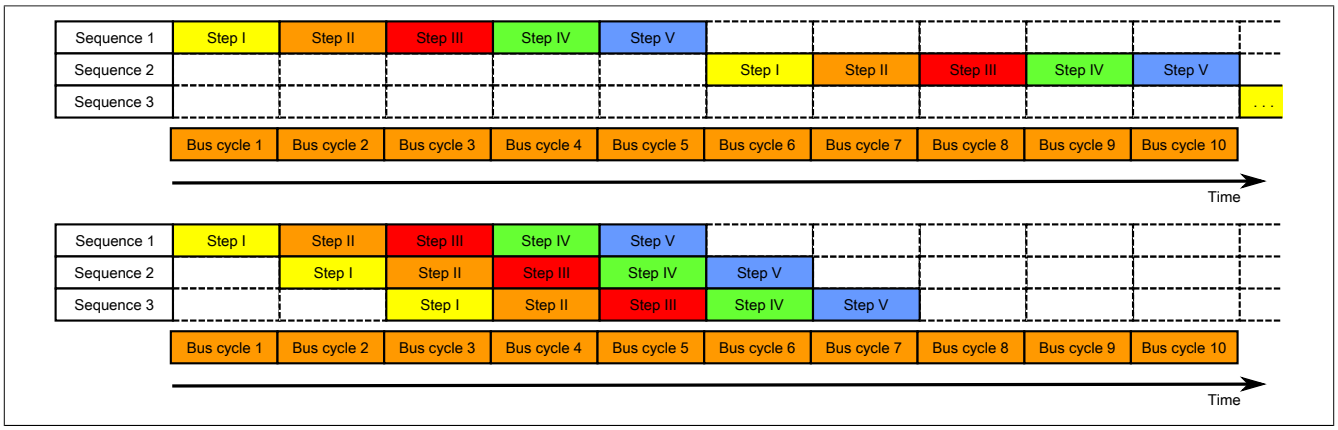


Figure 52: Comparison of transfer without/with Forward

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.

4.11.10.5.2 Configuration

The Forward function must only be enabled for the input direction. 2 additional configuration registers are available for doing so. 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.

4.11.10.5.2.1 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

4.11.10.5.2.2 Delay time

Name:
ForwardDelay

Register "ForwardDelay" is used to specify the delay time 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.

Data type	Values
UINT	0 to 65535 [µs] Default: 0

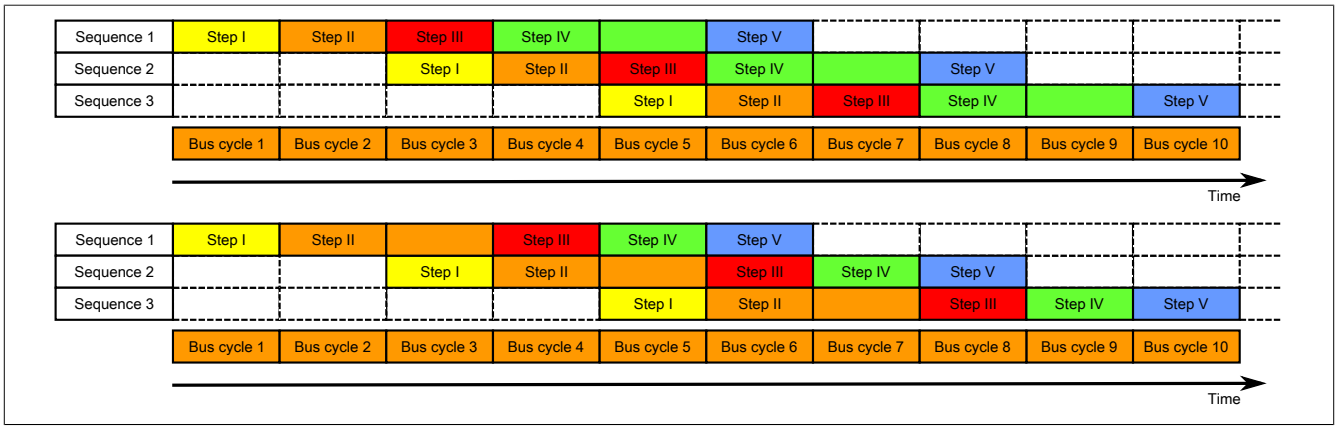


Figure 53: Effect of ForwardDelay when using Flatstream communication with the Forward function

In the program, it is important to make sure that the CPU 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 CPU has more time to process the incoming InputSequence or InputMTU.

4.11.10.5.3 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 CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU 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 CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU must transfer the current part of the transmit array to OutputMTU. - The CPU must increase OutputSequenceCounter for the sequence to be accepted by the module. - The CPU 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 CPU 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 CPU 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 CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU 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 Sequence-Counter 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 CPU 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.

4.11.10.5.4 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).

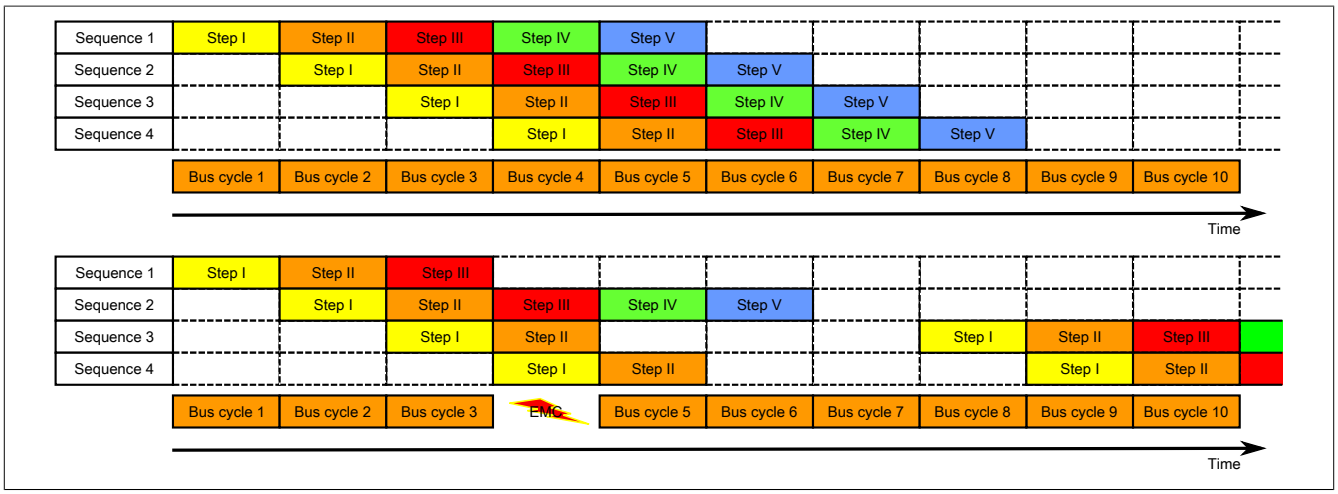


Figure 54: Effect of a lost bus cycle

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.

4.11.11 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 hand-shake or similar measures)

4.11.12 Acyclic frame size

Name:
AsynSize

When the stream is used, data is exchanged internally between the module and CPU. For this purpose, a defined amount of acyclic bytes is reserved for this slot.
Increasing the acyclic frame size leads to increased data throughput on this slot.

Information:

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

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

4.11.13 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 µs

4.11.14 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 µs

5 X20CS1070

5.1 General information

5.1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX20	X20 System user's manual
MAEMV	Installation / EMC guide

5.1.2 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS1070	X20 interface module, 1 CAN bus interface, max. 1 Mbit/s, object buffer in the transmit and receive directions	
	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	
	Terminal blocks	
X20TB06	X20 terminal block, 6-pin, 24 VDC keyed	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	

Table 27: X20CS1070 - Order data

5.1.3 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.

- CAN bus interface for serial, remote connection of complex devices to the X20 system
- Integrated terminating resistor

5.2 Technical description

5.2.1 Technical data

Order number	X20CS1070
Short description	
Communication module	1x CAN bus
General information	
B&R ID code	0x1FD1
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)
LR	ENV1
KR	Yes
ABS	Yes
BV	**EC33B** Temperature: 5 - 55°C Vibration: 4 g EMC: Bridge and open deck
EAC	Yes
KC	Yes
Interfaces	
Interface IF1	
Signal	CAN bus
Variant	Connection via 12-pin terminal block X20TB12
Max. distance	1000 m
Transfer rate	Max. 1 Mbit/s
Terminating resistor	Integrated in module
Controller	SJA 1000
Electrical properties	
Electrical isolation	CAN (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
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

Table 28: X20CS1070 - Technical data

Order number	X20CS1070
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order 1x terminal block X20TB06 or X20TB12 separately. Order 1x bus module X20BM11 separately.
Pitch	12.5$^{+0.2}$ mm

Table 28: X20CS1070 - Technical data

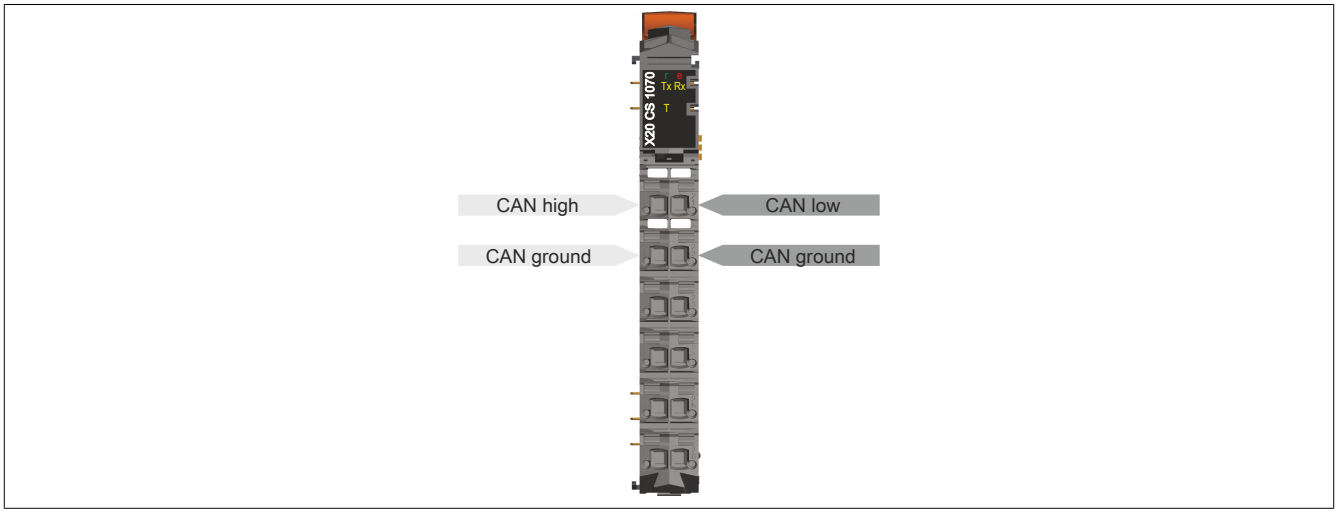
5.2.2 LED status indicators

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

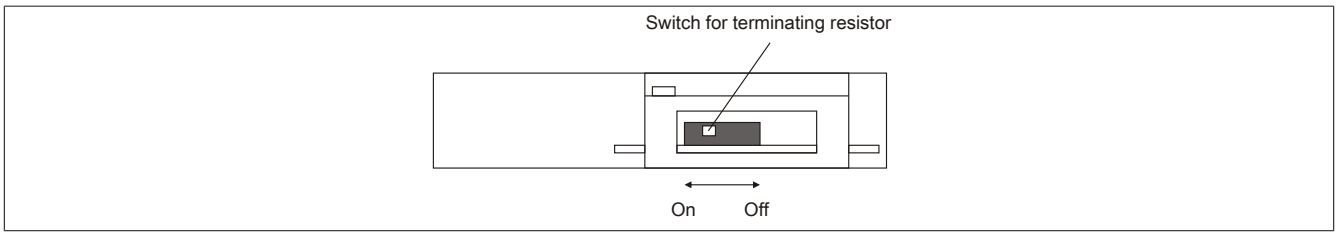
Figure	LED	Color	Status	Description
	r	Green	Off	No power to module
			Single flash	RESET mode
			Double flash	BOOT mode (during firmware update)[1]
			Blinking	PREOPERATIONAL mode
			On	RUN mode
	e	Red	Off	No power to module or everything OK
			Single flash	I/O error occurred • CAN bus: Warning, passive or off • Buffer overflow
			On	Error or reset status
	e + r	Red on / Green single flash		Invalid firmware
	Tx	Yellow	On	The module is sending data via the CAN bus interface
	Rx	Yellow	On	The module is receiving data via the CAN bus 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.

5.2.3 Pinout



5.2.4 Terminating resistor



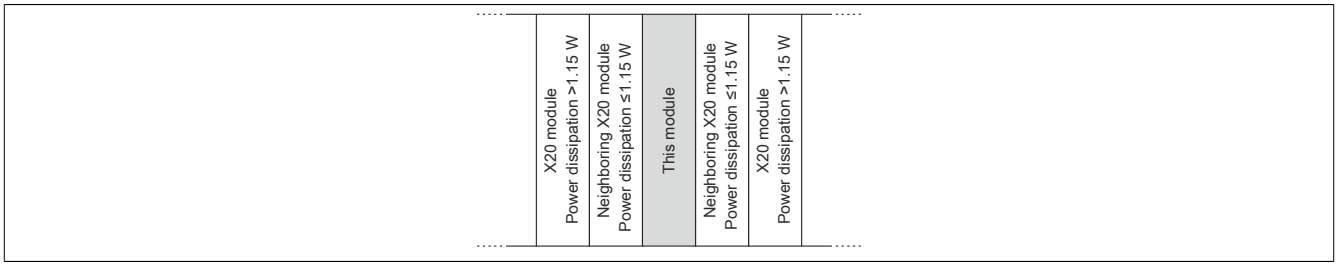
A terminating resistor is integrated in the communication module. It can be turned on and off with a switch on the bottom of the housing. An active terminating resistor is indicated by the "T" LED.

5.2.5 Derating

There is no derating when operated below 55°C.

When operated above 55°C, the modules to the left and right of this module are permitted to have a maximum power dissipation of 1.15 W!

For an example of calculating the power dissipation of I/O modules, see section "Mechanical and electrical configuration - Power dissipation of I/O modules" in the X20 user's manual.



5.3 Function description

5.3.1 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.

5.3.1.1 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:**
  - **CAN payload data length = Frame length - 4**
  - **Frame length = CAN payload data length + 4**

5.3.2 Flatstream communication

5.3.2.1 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.

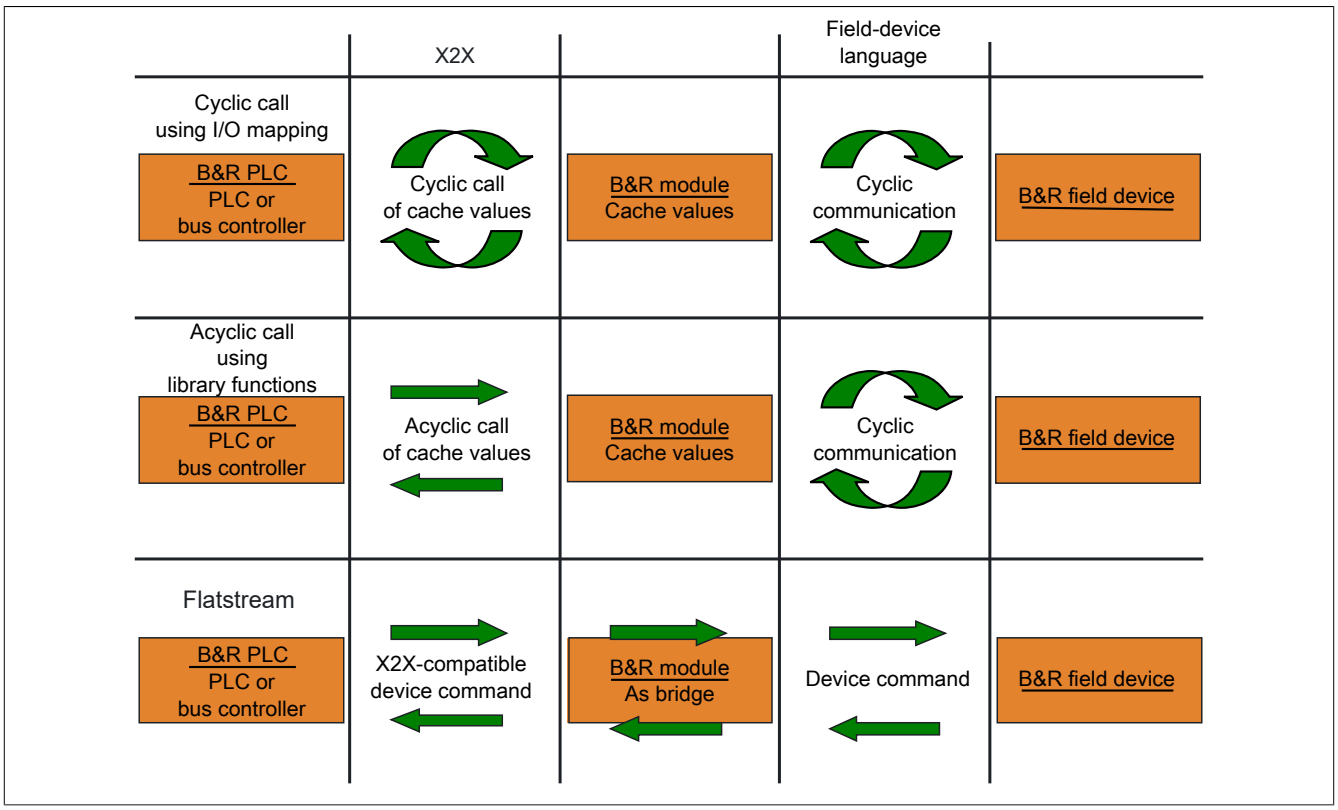


Figure 55: 3 types of communication

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.

5.3.2.2 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.

5.3.2.3 The Flatstream principle

Requirement

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.

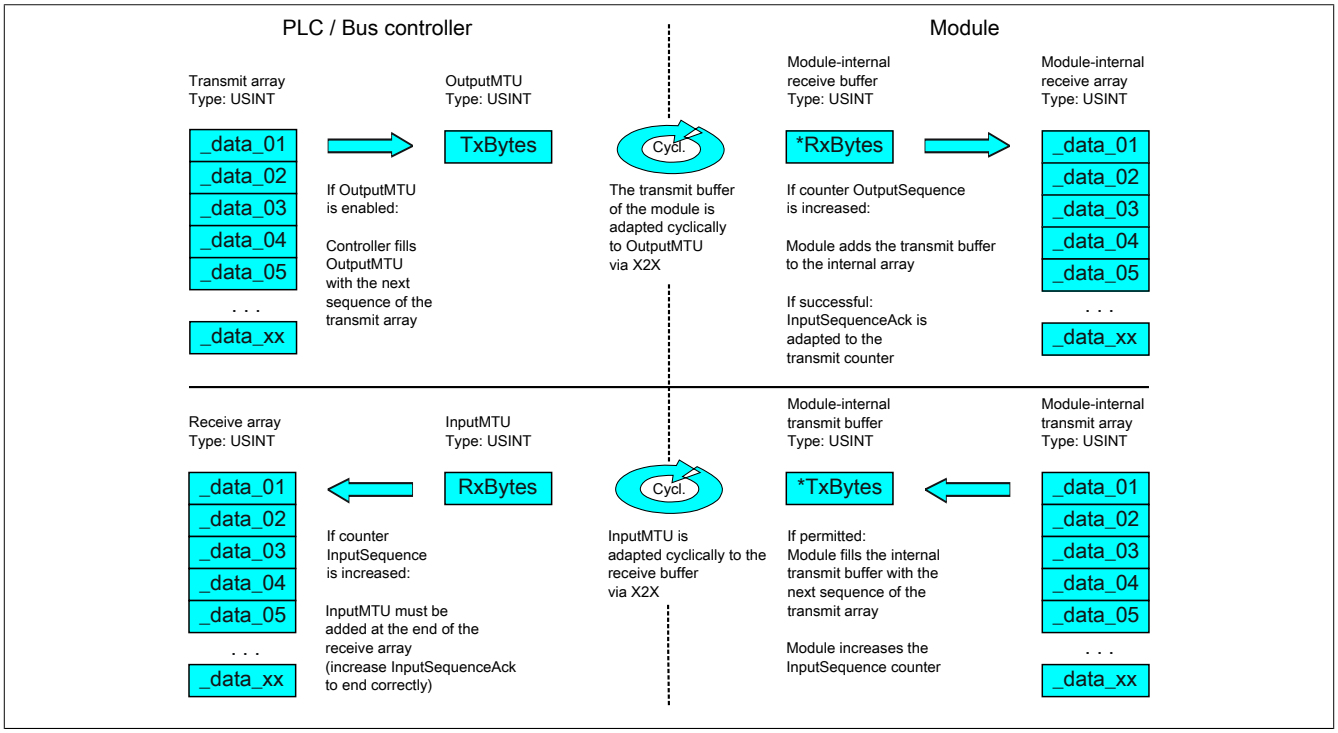


Figure 56: Flatstream communication

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.

5.3.2.4 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.

5.3.2.4.1 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 193.

5.3.2.4.2 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 193.

5.3.2.4.2.1 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.

5.3.2.4.2.2 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.

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

5.3.2.4.2.3 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.

Relationship between OutputSequence and InputSequence

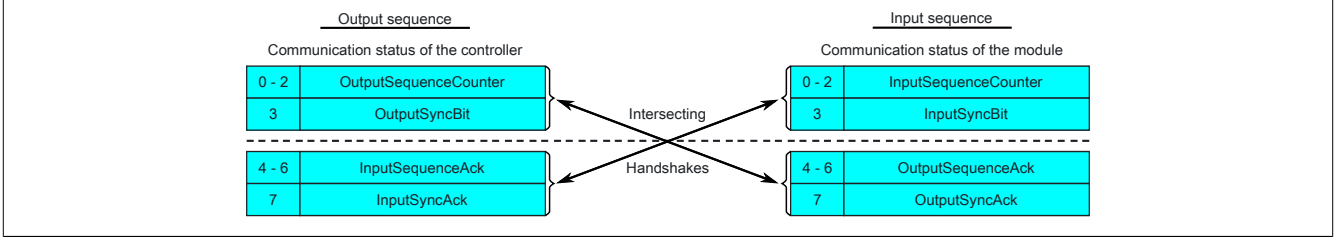


Figure 57: Relationship between OutputSequence and InputSequence

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.

5.3.2.4.3 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.

5.3.2.4.4 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.

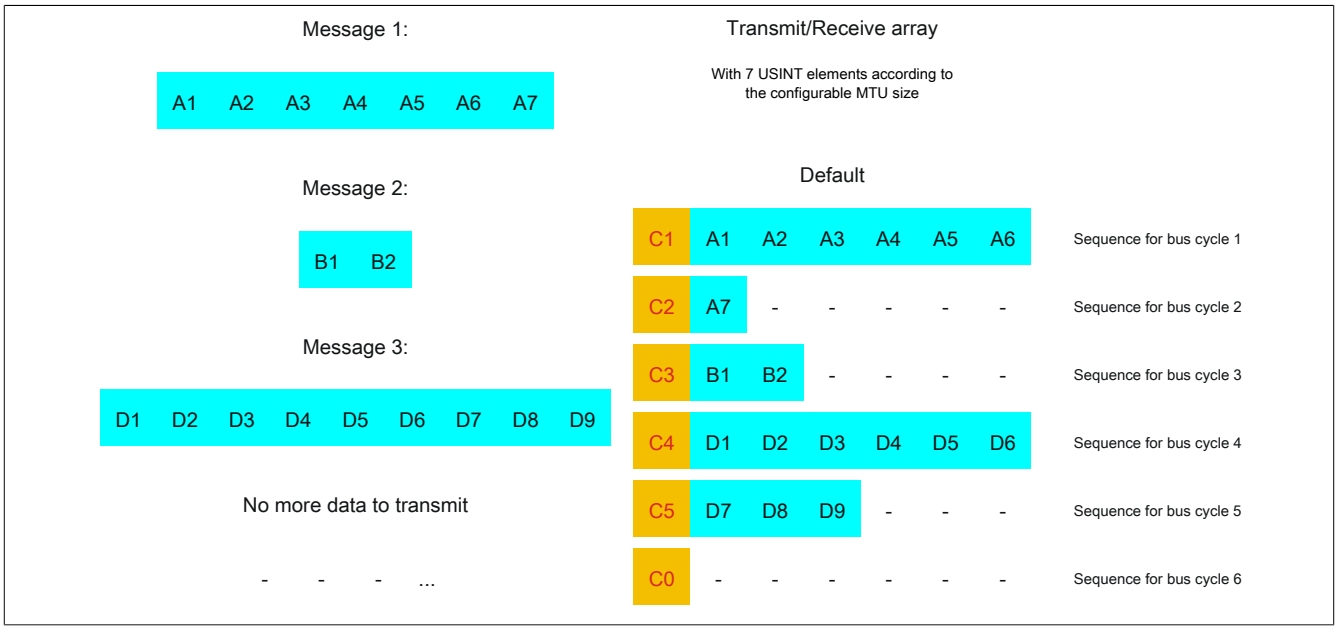


Figure 58: Transmit/Receive array (default)

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 29: 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 30: Flatstream determination of the control bytes for the default configuration example (part 2)

5.3.2.4.4.1 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.

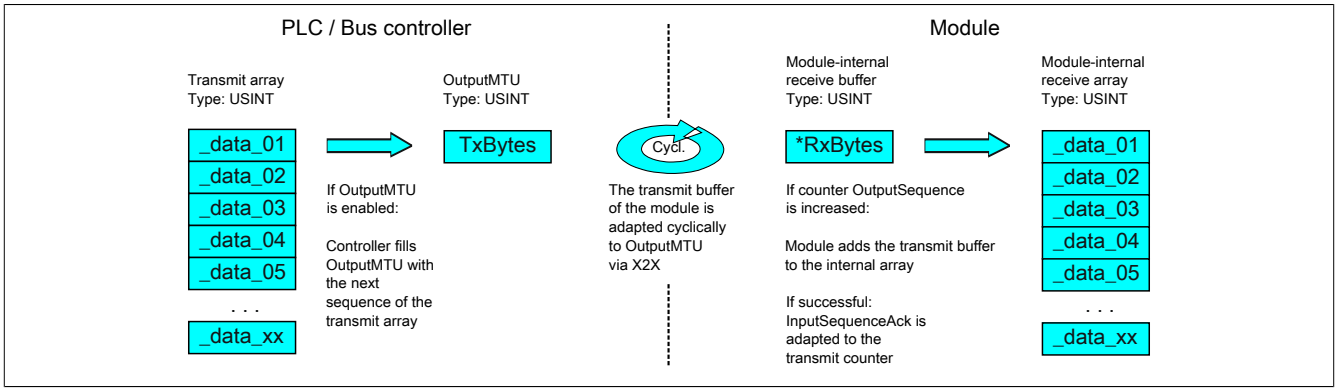


Figure 59: Flatstream communication (output)

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

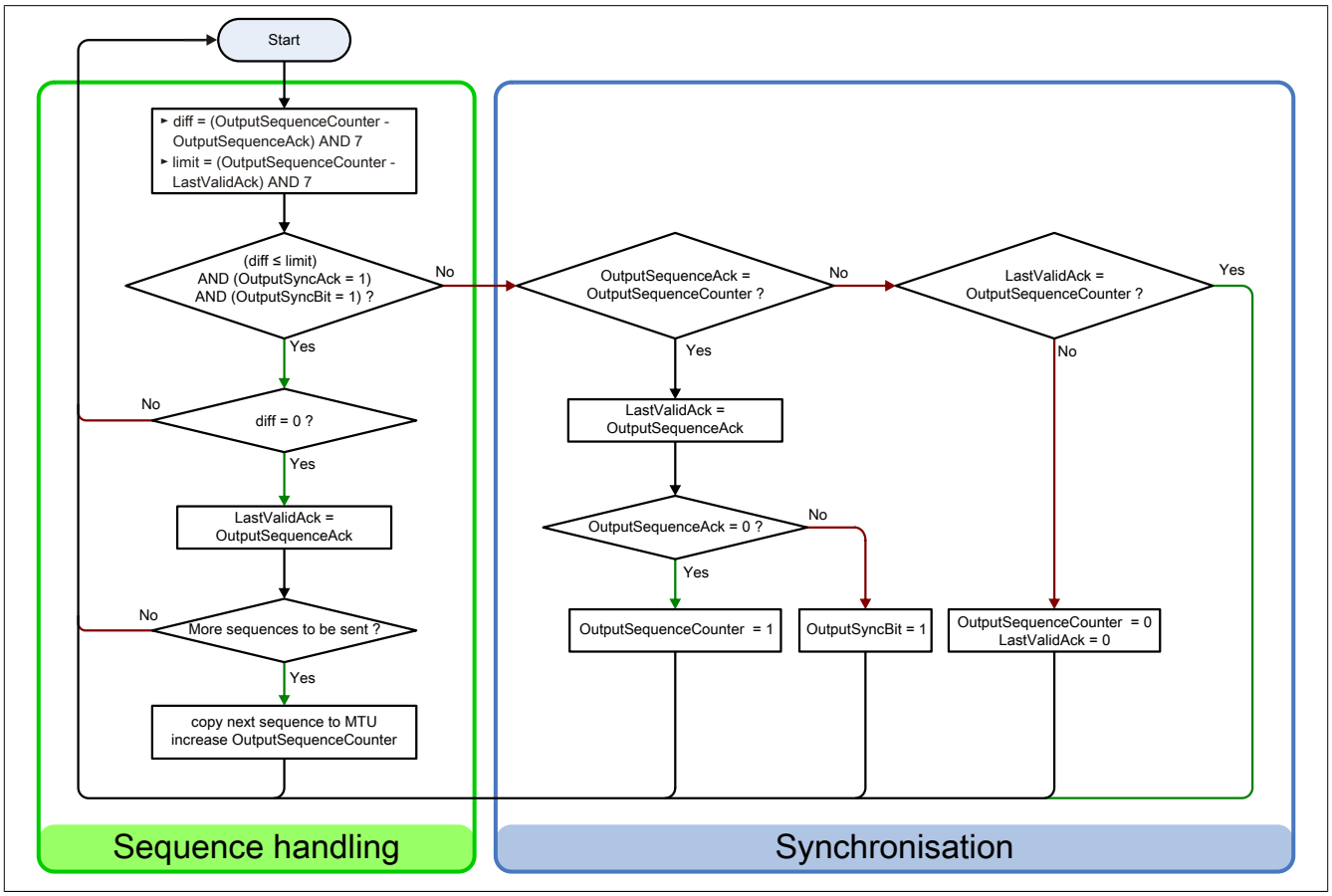


Figure 60: Flowchart for the output direction

5.3.2.4.4.2 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.

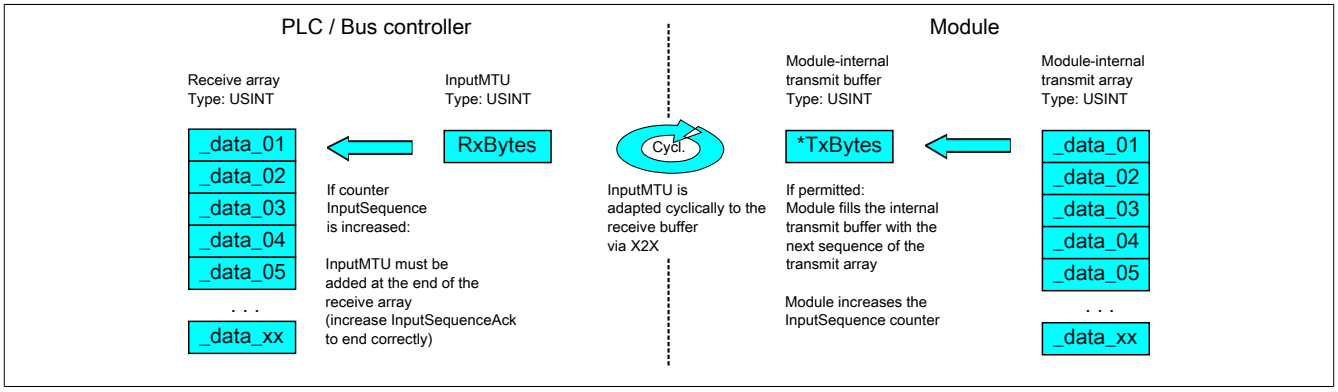


Figure 61: Flatstream communication (input)

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

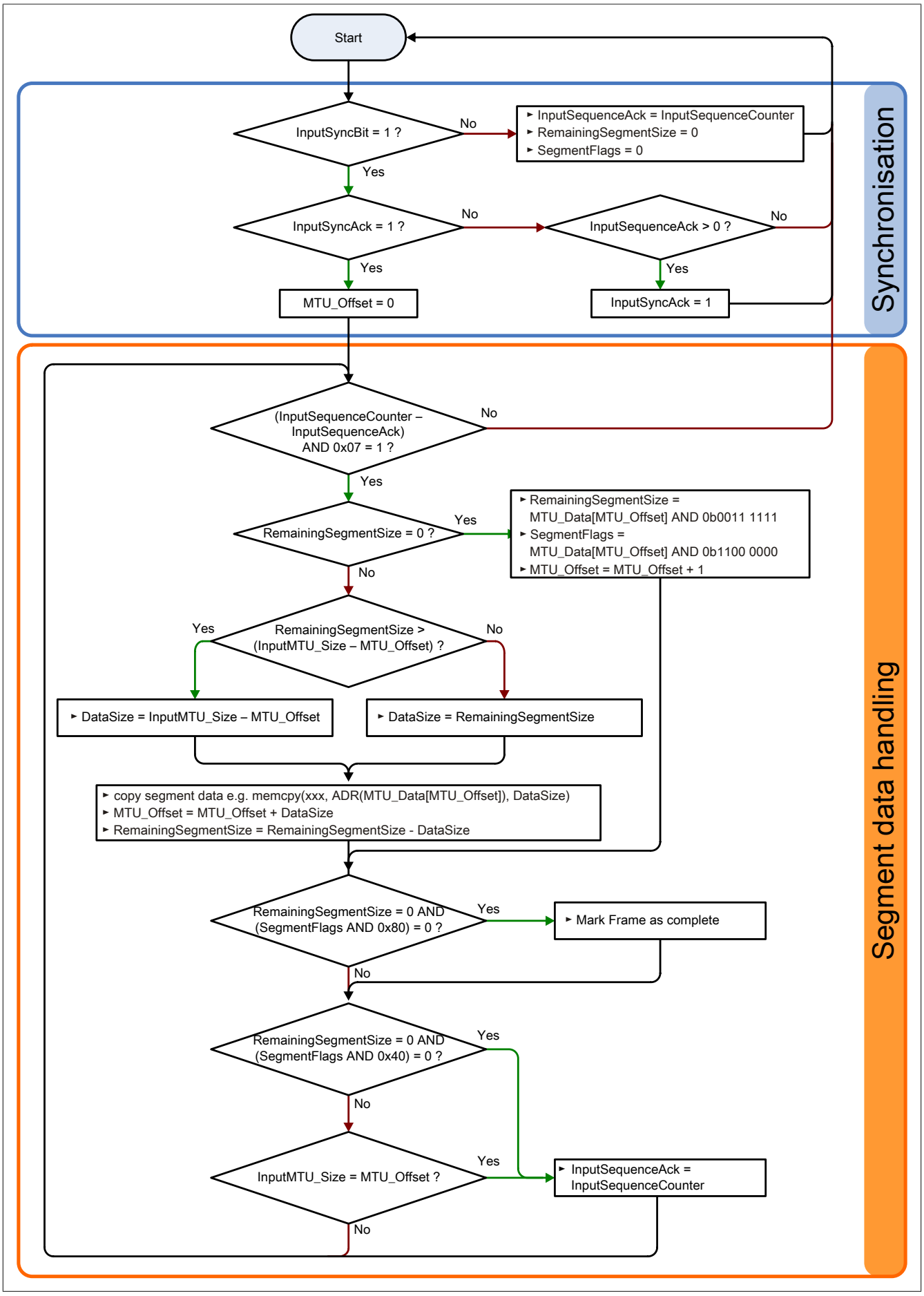


Figure 62: Flowchart for the input direction

5.3.2.4.4.3 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.

5.3.2.4.5 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.

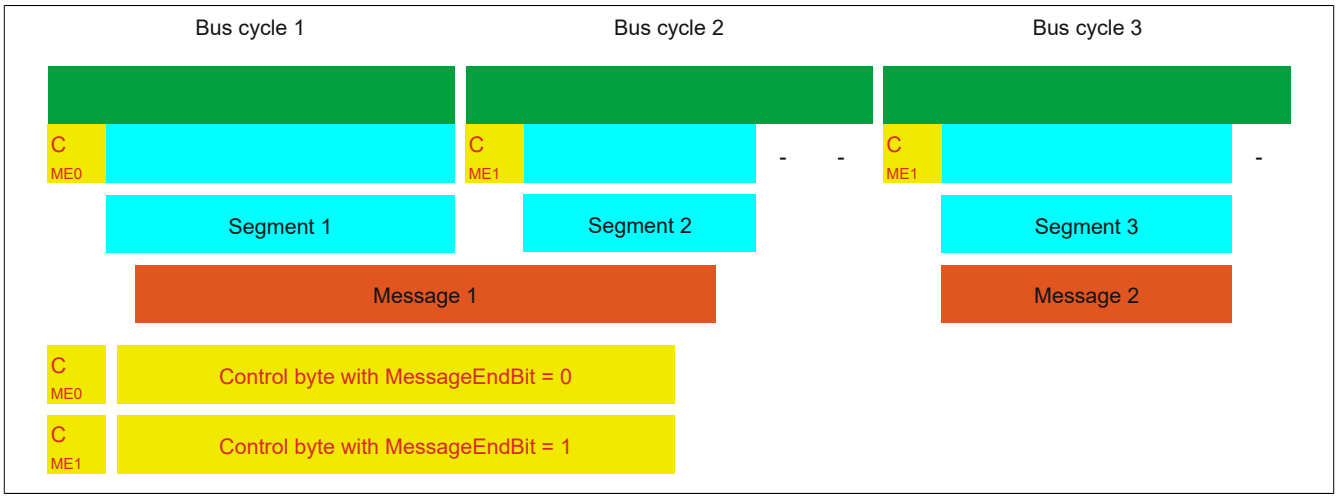


Figure 63: Message arrangement in the MTU (default)

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.

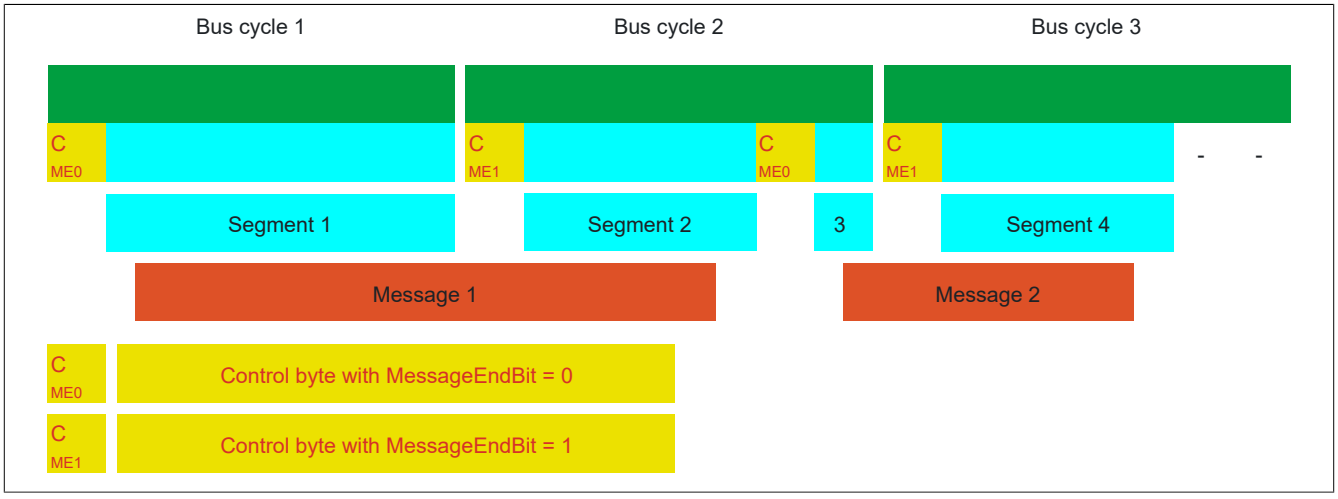


Figure 64: Arrangement of messages in the MTU (MultiSegmentMTU)

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.

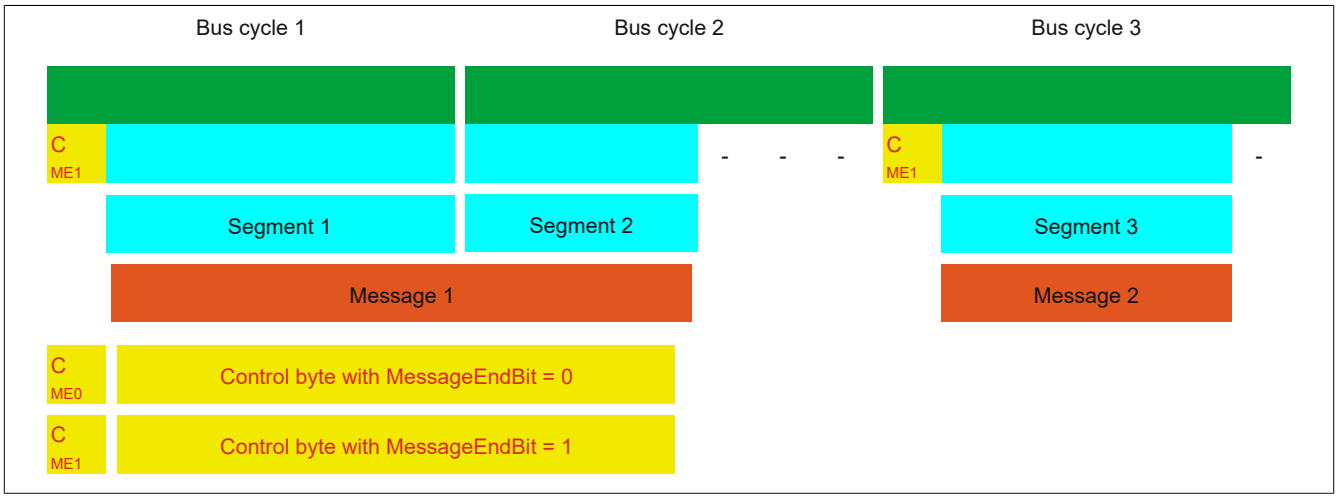


Figure 65: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

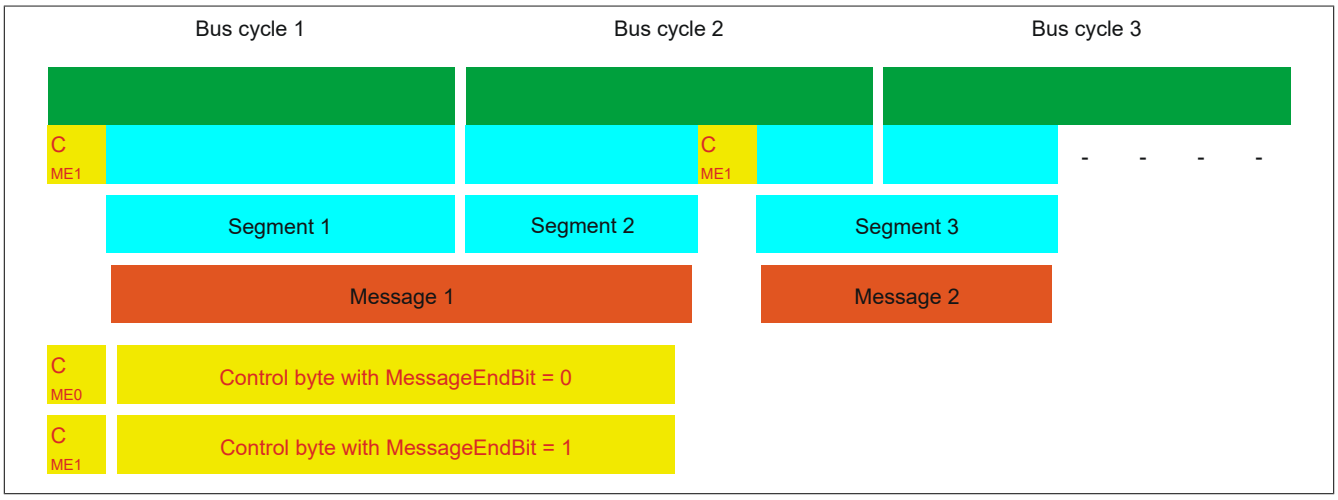


Figure 66: Arrangement of messages in the MTU (large segments and MultiSegmentMTU)

5.3.2.4.6 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 "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.

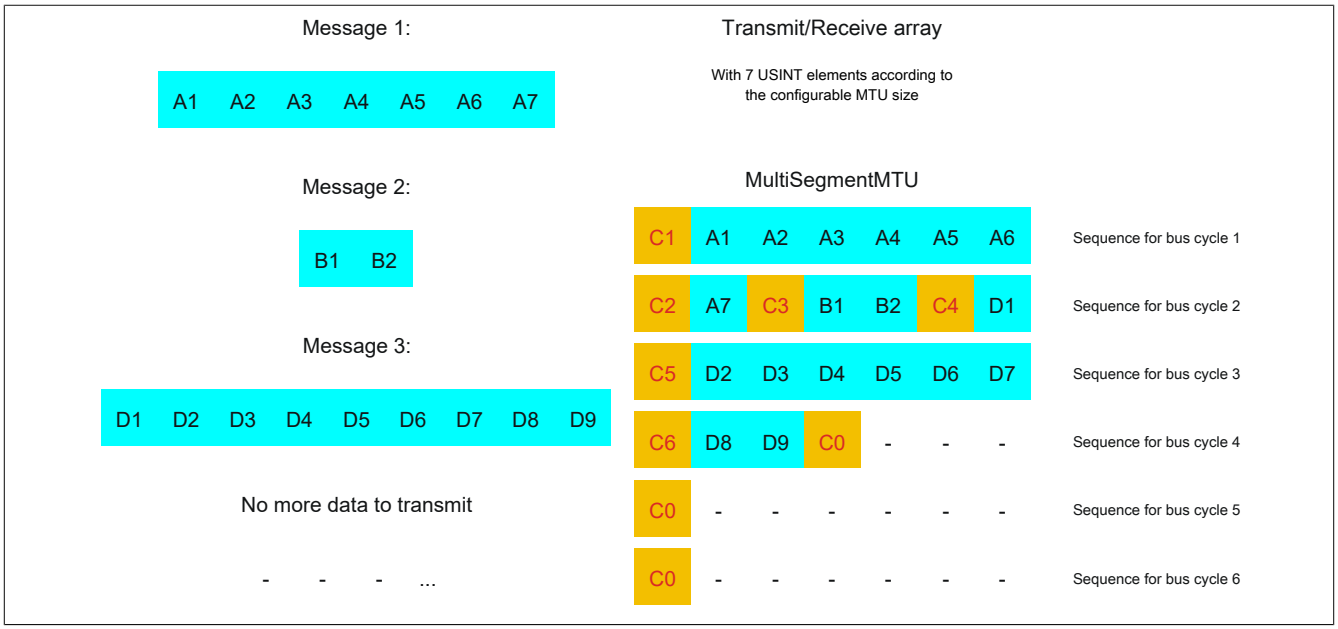


Figure 67: Transmit/Receive array (MultiSegmentMTU)

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 31: 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 32: 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.

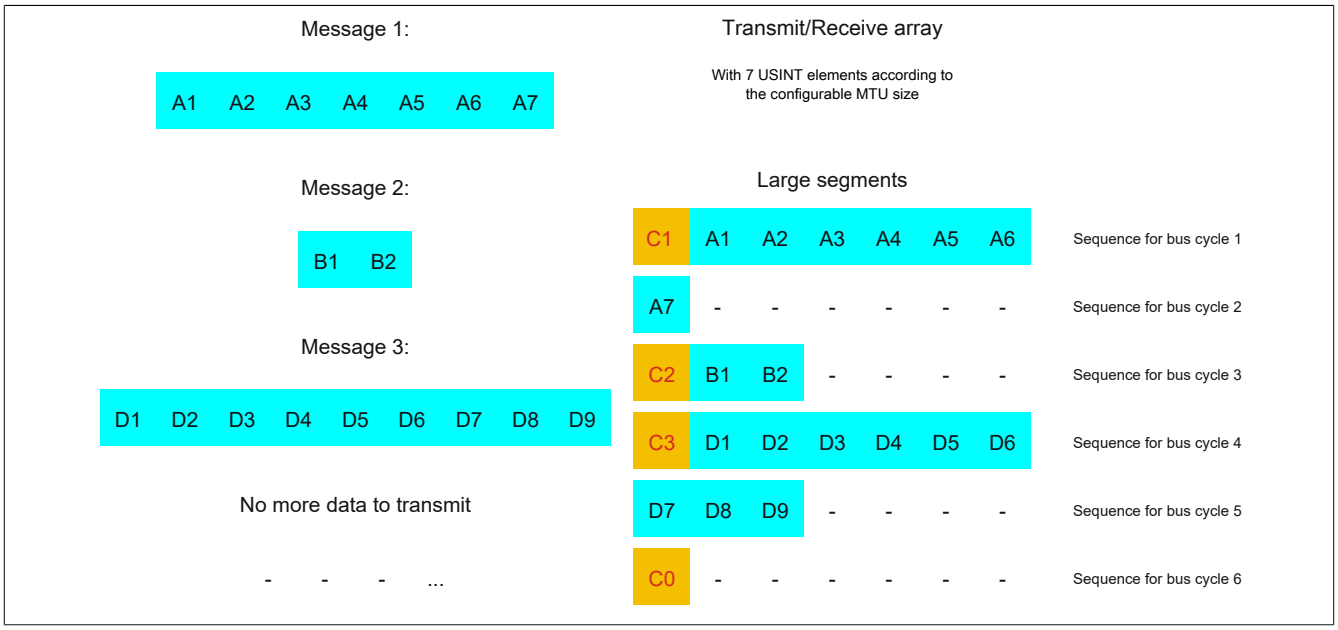


Figure 68: Transmit/receive array (large segments)

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 33: 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.

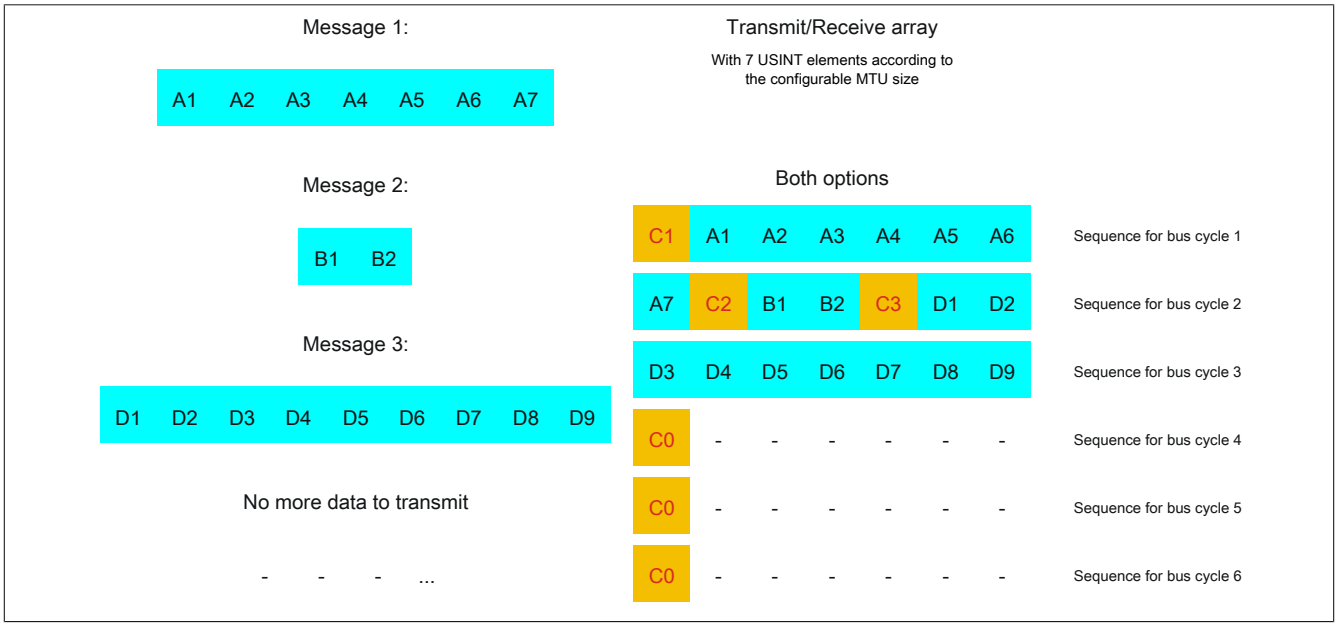


Figure 69: Transmit/Receive array (large segments and MultiSegmentMTU)

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

5.3.2.5 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.

5.3.2.5.1 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 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)

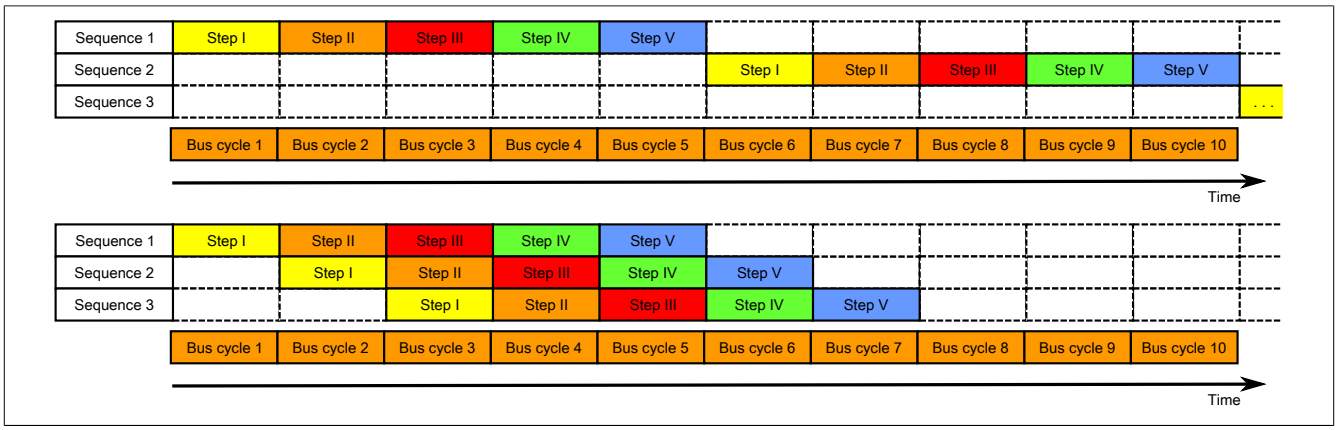


Figure 70: Comparison of transfer without/with Forward

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.

5.3.2.5.2 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 193.

5.3.2.5.2.1 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.

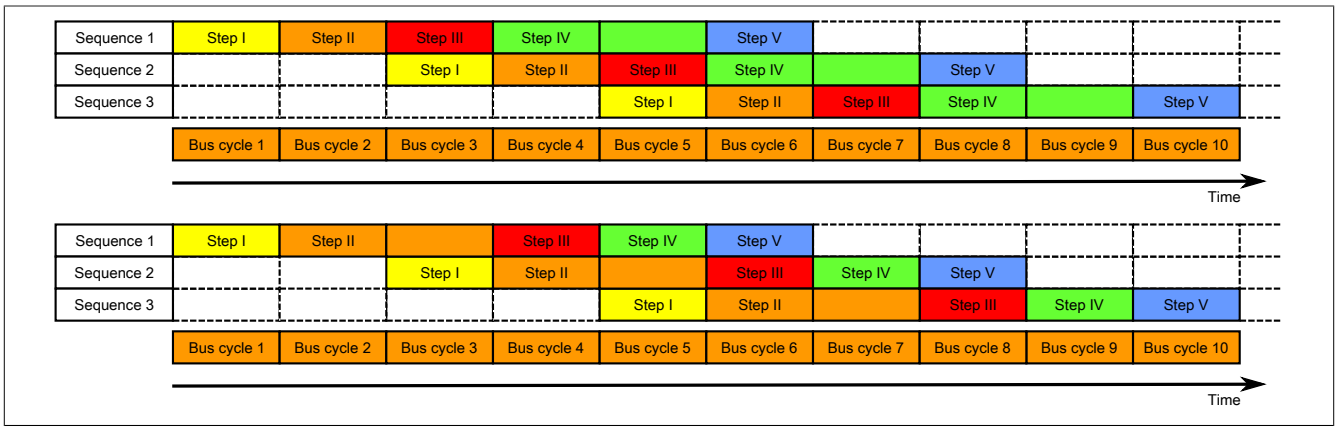


Figure 71: Effect of ForwardDelay when using Flatstream communication with the Forward function

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.

5.3.2.5.3 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.

5.3.2.5.4 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).

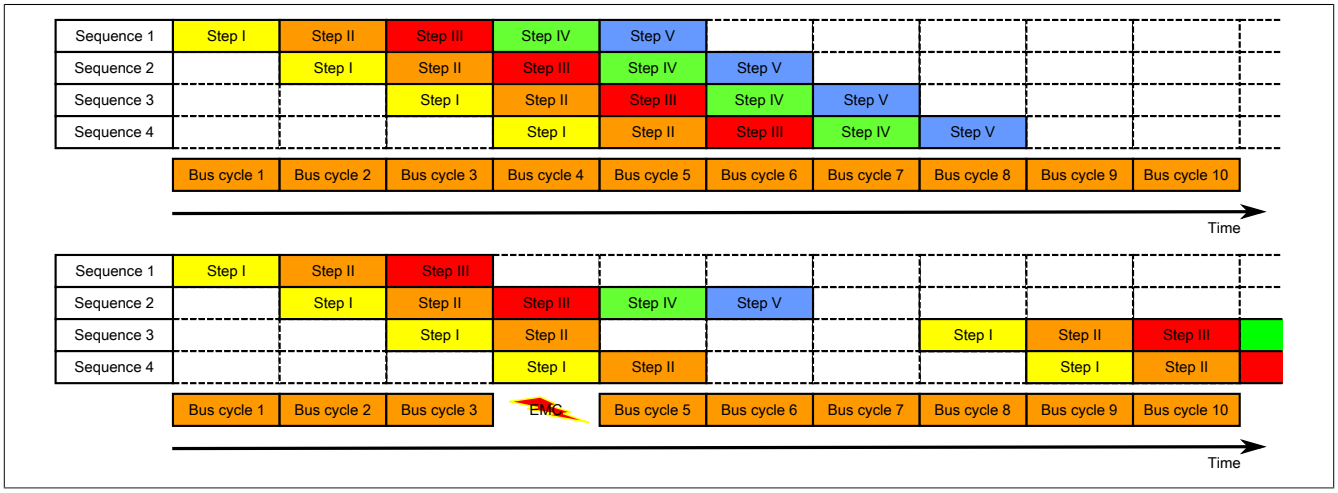


Figure 72: Effect of a lost bus cycle

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.

5.4 Commissioning

5.4.1 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.

5.4.2 Using this module with SGC target systems

Information:

This module can only be used with SGC target systems if the function model is set to "Flatstream" or "Flat".

5.5 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.**

5.6 Register description

5.6.1 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.

5.6.2 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 154).

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				•
270	CfO_Fifosize_01	UINT				•
673	Cfo_FIFOTXlimit	USINT				•
677	Cfo_TXRXinfoFlags	USINT				•
Stream filter - configuration						
385	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
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	•			

5.6.3 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.**

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Module - Configuration						
-	AsynSize	-				
Interface - Configuration						
270	CfO_Fifosize_01	UINT				•
6273	CfO_ErrorID0007	USINT				•
Stream filter - configuration						
385	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
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				

5.6.4 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 155).

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				•
270	CfO_Fifosize_01	UINT				•
6273	CfO_ErrorID0007	USINT				•
Stream filter - configuration						
385	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
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				
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			•	

5.6.5 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	Acyclic	Cyclic	Acyclic
Interface - Configuration							
257	-	ConfigBaudrate	USINT				•
259	-	ConfigSJW	USINT				•
261	-	ConfigSPO	USINT				•
266	-	ConfigTXtrigger	UINT				•
270	-	CfO_Fifosize_01	UINT				•
6273	-	CfO_ErrorID0007	USINT				•
Stream filter - configuration							
385	-	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	-	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	-	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
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				
Flatstream - Configuration							
193	-	outputMTU	USINT				•
195	-	inputMTU	USINT				•
197	-	mode	USINT				•
199	-	forward	USINT				•
206	-	forwardDelay	UINT				•
Flatstream - Communication							
0	0	InputSequence	USINT	•			
Index * 1 + 0	Index * 1 + 0	RxByte1 to RxByte7	USINT	•			
32	0	OutputSequence	USINT			•	
Index * 1 + 32	Index * 1 + 0	TxByte1 to TxByte7	USINT			•	

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

5.6.5.1 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).

5.6.5.2 CAN I/O bus controller

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

5.6.6 Interface - Configuration

5.6.6.1 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
		8	800 kbit/s
		9	1000 kbit/s
4 - 7	Reserved	-	

5.6.6.2 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

5.6.6.3 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

5.6.6.4 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

5.6.6.5 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	-	

5.6.6.6 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

5.6.6.7 FIFO memory size

Name:
CfO_Fifosize_01

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

Determines the size of the FIFO memory for the respective interface.

Data type	Values	Explanation
UINT	20 to 4096	Size of the FIFO memory in bytes

5.6.6.8 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 190 and "TXCountLatchReadBack" on page 190 are use to read back "TXCount".
		1	The lower 4 bits of registers "TXCountReadBack" on page 190 and "TXCountLatchReadBack" on page 190 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 190 and "RXCountLatch" on page 191 are used to indicate the number of telegrams that have been received.
		1	The lower 4 bits of registers "RXCount" on page 190 and "RXCountLatch" on page 191 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	-	

5.6.6.9 Stream filter

Up to 4 stream filters can be configured per CAN interface. These determine which CAN IDs are forwarded to the controller via the cyclic stream.
The filters are run through in numerical order. The first filter matching the incoming CAN message is used; all other filters are ignored. If no filter matches the incoming CAN message, a global configuration determines whether the message is rejected or accepted (default: accept message).
Each filter has a configurable ID and configurable filter mask. Only those bits of the ID are compared that are set to 0 in the mask.

5.6.6.9.1 CANFilterMode

Name:
CfO_IF1DefaultCANFilterMode

These registers specify the default settings for IDs that do not match any of the set filters.

Data type	Values	Information
USINT	0	No filter response, the CAN frame is discarded.
	1	No filter response, the CAN frame is transferred via the stream.

5.6.6.9.2 CAN filter

Name:
CfO_IF1CANFilter01 to CfO_IF1CANFilter04

The filter properties are defined in these registers.

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0 to 28	Filter ID	x	Identifier value for filtering.[1)]
29	Frame format	0	Standard frame format (SFF) with 11-bit identifier. Possible filter ID values: 0 to 2047 (0x7FF)
		1	Extended frame format (EFF) with 29-bit identifier. Possible filter ID values: 0 to 536 870 911 (0x1FFFFF)
30	Reserved	-	
31	Enable	0	Filter inactive
		1	Filter active

1) This value is linked with the identifier value and mask value (see example).

Examples

Example 1

The following example shows the correlation between filter mask, filter ID and the actual received 11-bit CAN messages.

Filter mask[1)]	Filter ID	CAN message ID	Information
000 0011 1110	110 0100 0000	110 0110 1010	Relevant bits of filter ID and CAN message are identical. → The filter responds, and the frame is discarded or forwarded according to the mode setting.
000 0011 1110	110 0100 0000	110 0110 1011	Relevant bits not identical → Next filter or default mode is executed.
000 0011 1111	110 0100 0000	110 0110 1011	Relevant bits of filter ID and CAN message are identical. → The filter responds, and the frame is discarded or forwarded according to the mode setting.
000 0001 1111	110 0100 0000	110 0110 1011	Relevant bits not identical → Next filter or default mode is executed.

1) Red = Relevant bits

Example 2

Configuration of 2 filters with different filter mode.

	Mode	Filter ID	Filter mask	Description
Filter 1	ignore	16#300	16#07F	CANopen PDO2
Filter 2	accept	16#005	16#780	NodeID 5
Default	ignore			

With CANopen, the 11-bit CAN IDs (COB IDs) are composed of a 4-bit function code and 7-bit NodeID. All CANopen PDO2 objects are initially rejected here. After that, only the frames from the CAN device with NodeID 5 are accepted.

5.6.6.9.3 CANFilterMask

Name:
CfO_IF1CANFilterMask01 to CfO_IF1CANFilterMask04

The filter mask and filter mode are defined in these registers.

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0 to 28	Filter mask	x	Comparison bit pattern for filter ID[1)]
29	Frame format mask	0	The frame format of the received message must match the configuration in "CfO_IF1CANFilter" on page 187.
		1	The filter applies to both frame formats. 11-bit and 29-bit identifiers are filtered.
30	Reserved	-	
31	Mode	0	The CAN frame is transferred when the filter responds.
		1	The CAN frame is discarded when the filter responds.

1) Only those bits of the ID are compared that are set to 0 in the mask (see "Examples" on page 188).

5.6.7 Interface - Communication

5.6.7.1 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 189).

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.

5.6.7.2 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	QuitCANbussoff	0	No acknowledgment
		1	Acknowledge CANbusoff error on IF1
3	QuitCANRXoverrun	0	No acknowledgment
		1	Acknowledge CANRXoverrun error on IF1
4 - 7	Reserved	-	

5.6.7.3 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

5.6.7.4 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 186.

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.

5.6.7.5 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 192).

The meaning of the value depends on bit "TxFifoInfo" in register "Cfo_TXRXinfoFlags" on page 186.

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

5.6.7.6 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 186.

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	

5.6.7.7 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 192).

The meaning of the value depends on bit "RxFifoInfo" in register "Cfo_TXRXinfoFlags" on page 186.

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	

5.6.8 Transmit buffer

5.6.8.1 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 192).

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

5.6.8.2 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.

5.6.8.3 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

5.6.8.4 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.

Registers for counter feedback:
- "TXCountReadBack" on page 190
- "TXCountLatchReadBack" on page 190

5.6.9 Receive buffers 0 and 1

5.6.9.1 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 189.

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

5.6.9.2 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.

5.6.9.3 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

5.6.10 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" on page 155.

5.6.10.1 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.

5.6.10.2 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

5.6.10.3 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.

5.6.10.4 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.

5.6.10.5 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		

5.6.10.6 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

5.6.10.7 Delay time

Name:
ForwardDelay

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

Data type	Values
UINT	0 to 65535 [µs] Default: 0

5.6.11 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

5.6.12 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 µs

5.6.13 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 µs

6 X20CS2770

6.1 General information

6.1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX20	X20 System user's manual
MAEMV	Installation / EMC guide

6.1.2 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS2770	X20 interface module, 2 CAN bus interfaces, max. 1 Mbit/s, object buffer in the transmit and receive directions	
	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	
	Terminal blocks	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	

Table 35: X20CS2770 - Order data

6.1.3 Module description

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

Functions

- CAN interfaces
- Flatstream

CAN interface

The Controller Area Network (CAN) bus is a serial transmission interface. It allows systems, devices and controllers to communicate with each other within a network.

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.

6.2 Technical description

6.2.1 Technical data

Order number	X20CS2770
Short description	
Communication module	2x CAN bus
General information	
B&R ID code	0xA009
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	0.55 W (Rev. ≤D0 1.5 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
EAC	Yes
KC	Yes
Interfaces	
Interface IF1	
Signal	CAN bus
Variant	Connection via 12-pin terminal block X20TB12
Max. distance	1000 m
Transfer rate	Max. 1 Mbit/s
Terminating resistor	Integrated in module
Controller	SJA 1000
Interface IF2	
Signal	CAN bus
Variant	Connection via 12-pin terminal block X20TB12
Max. distance	1000 m
Transfer rate	Max. 1 Mbit/s
Terminating resistor	Integrated in module
Controller	SJA 1000
Electrical properties	
Electrical isolation	CAN (IF1, IF2) isolated from bus and I/O power supply CAN (IF1, IF2) not isolated from each other
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
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

Table 36: X20CS2770 - Technical data

Order number	X20CS2770
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order 1x terminal block X20TB12 separately. Order 1x bus module X20BM11 separately.
Pitch	12.5$^{+0.2}$ mm

Table 36: X20CS2770 - Technical data

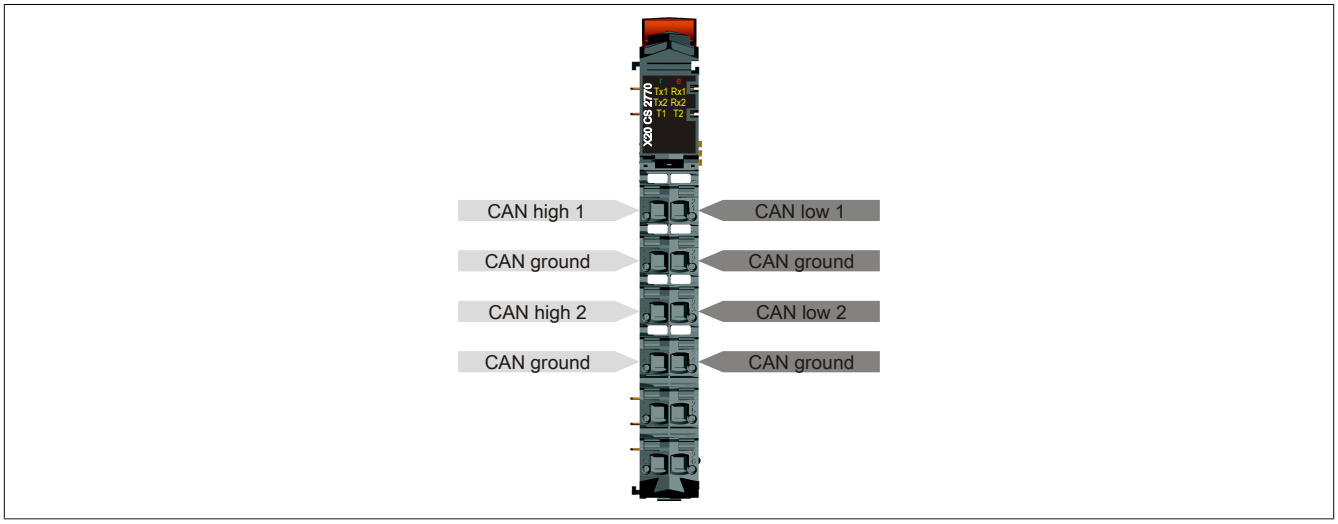
6.2.2 LED status indicators

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

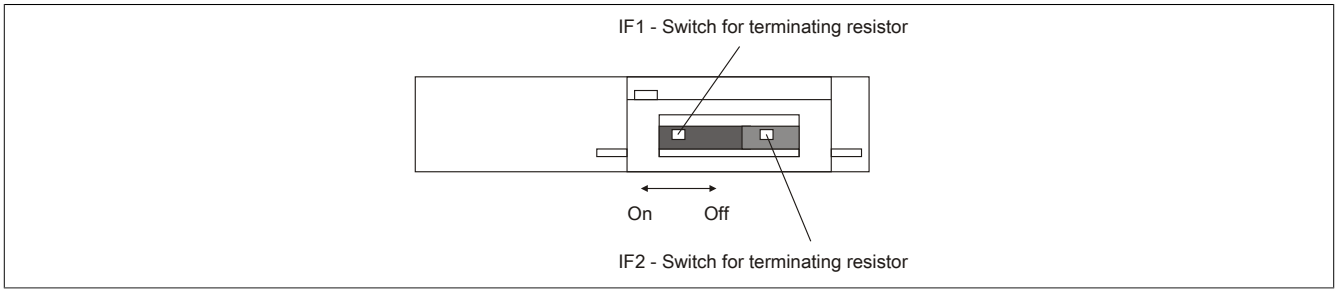
Figure	LED	Color	Status	Description
	r	Green	Off	No power to module
			Single flash	RESET mode
			Double flash	BOOT mode (during firmware update)[1]
			Blinking	PREOPERATIONAL mode
			On	RUN mode
	e	Red	Off	No power to module or everything OK
			Single flash	I/O error occurred • CAN bus: Warning, passive or off • Buffer overflow
			On	Error or reset status
	e + r	Red on / Green single flash		Invalid firmware
	Tx1/2	Yellow	On	The module is sending data via the CAN bus interface IF1/IF2
	Rx1/2	Yellow	On	The module is receiving data via the CAN bus interface IF1/IF2
	T1/2	Yellow	On	The integrated terminating resistor for the CAN bus interface IF1/IF2 is turned on

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

6.2.3 Pinout



6.2.4 Terminating resistors



Two terminating resistors are integrated in the communication module. The respective resistor can be turned on and off with a switch on the bottom of the housing. An active terminating resistor is indicated by the "T1" or "T2" LED.

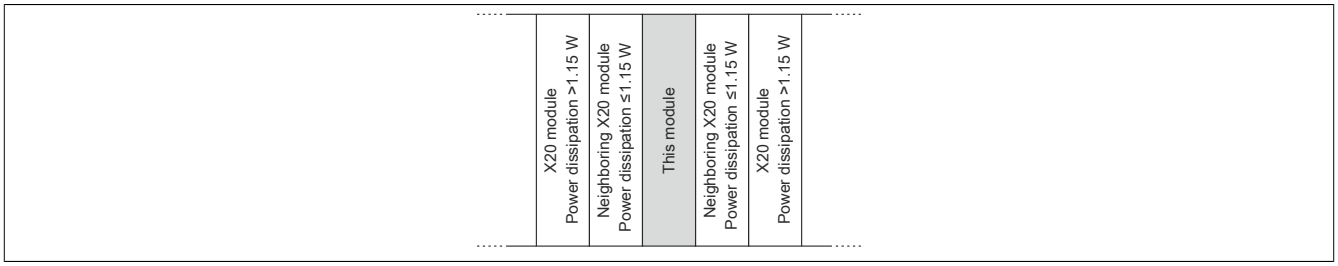
6.2.5 Derating

Up to hardware revision ≤D0

There is no derating when operated below 55°C.

When operated above 55°C, the modules to the left and right of this module are permitted to have a maximum power dissipation of 1.15 W!

For an example of calculating the power dissipation of I/O modules, see section "Mechanical and electrical configuration - Power dissipation of I/O modules" in the X20 user's manual.



Hardware revision >D0 and later

No derating

6.3 Function description

6.3.1 CAN interfaces

The module is equipped with 2 CAN interfaces that can be used to transmit and receive CAN objects.

6.3.1.1 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.

6.3.1.2 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.

6.3.1.3 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:**
  - **CAN payload data length = Frame length - 4**
  - **Frame length = CAN payload data length + 4**

6.3.1.4 CAN filters

Up to 4 stream filters can be configured per CAN interface. These determine which CAN IDs are forwarded to the controller via the cyclic stream.
The filters are run through in numerical order. The first filter matching the incoming CAN message is used; all other filters are ignored. If no filter matches the incoming CAN message, a global configuration determines whether the message is rejected or accepted (default: accept message).
Each filter has a configurable ID and configurable filter mask. Only those bits of the ID are compared that are set to 0 in the mask.

Example 1

The following example shows the correlation between filter mask, filter ID and the actual received 11-bit CAN messages.

Filter mask[1)]	Filter ID	CAN message ID	Information
000 0011 1110	110 0100 0000	110 0110 1010	Relevant bits of filter ID and CAN message are identical. → The filter responds, and the frame is discarded or forwarded according to the mode setting.
000 0011 1110	110 0100 0000	110 0110 1011	Relevant bits not identical → Next filter or default mode is executed.
000 0011 1111	110 0100 0000	110 0110 1011	Relevant bits of filter ID and CAN message are identical. → The filter responds, and the frame is discarded or forwarded according to the mode setting.
000 0001 1111	110 0100 0000	110 0110 1011	Relevant bits not identical → Next filter or default mode is executed.

1) Red = Relevant bits

Example 2

Configuration of 2 filters with different filter mode.

	Mode	Filter ID	Filter mask	Description
Filter 1	ignore	16#300	16#07F	CANopen PDO2
Filter 2	accept	16#005	16#780	NodeID 5
Default	ignore			

With CANopen, the 11-bit CAN IDs (COB IDs) are composed of a 4-bit function code and 7-bit NodeID. All CANopen PDO2 objects are initially rejected here. After that, only the frames from the CAN device with NodeID 5 are accepted.

Information:

The registers are described in "Stream filter" on page 235.

6.3.1.5 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 register is described under "TXCountReadBack" on page 238.

6.3.2 Flatstream communication

6.3.2.1 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.

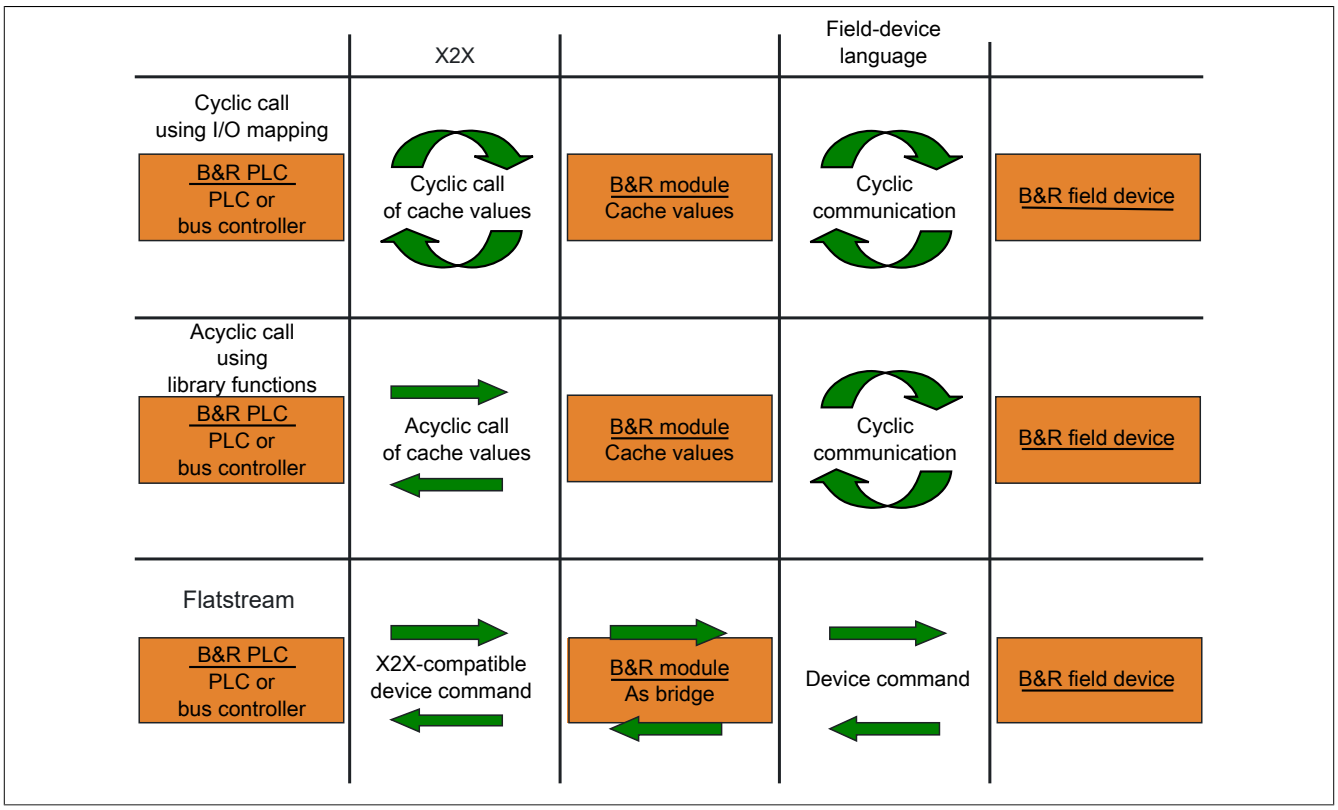


Figure 73: 3 types of communication

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.

6.3.2.2 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.

6.3.2.3 The Flatstream principle

Requirement

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.

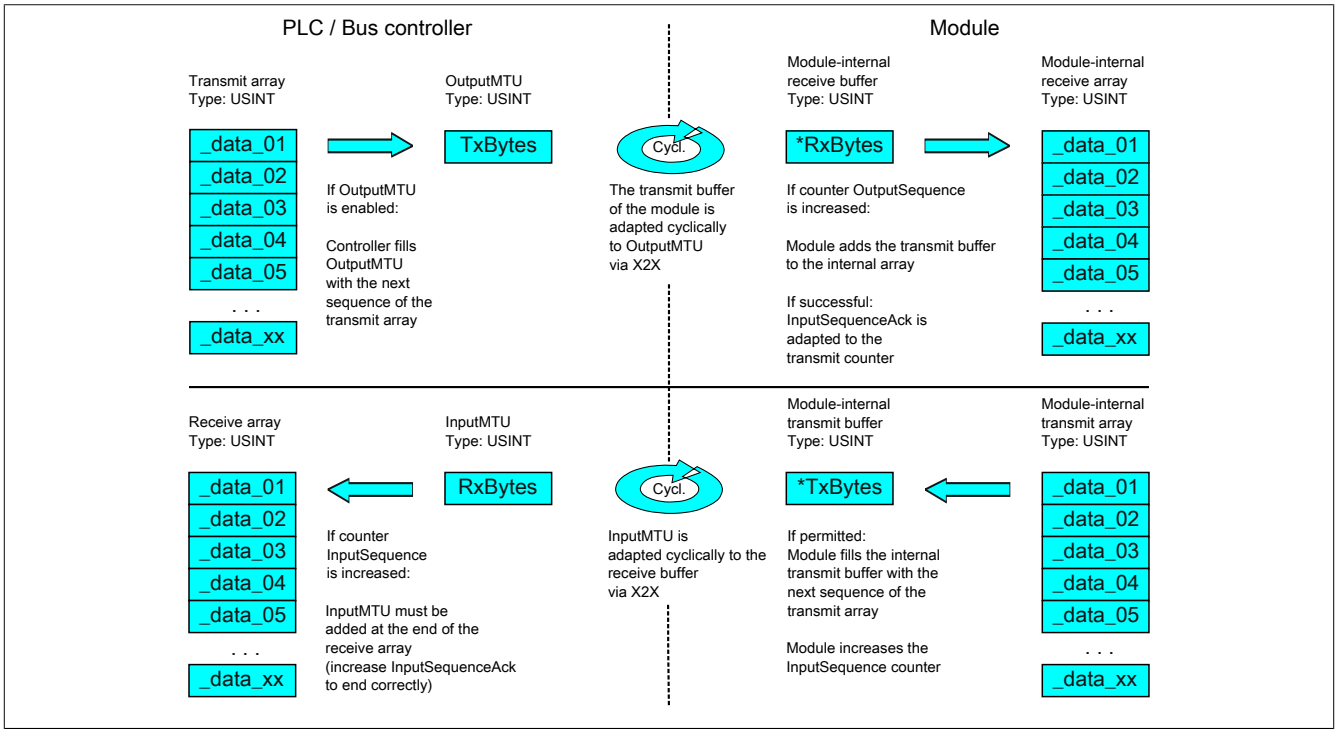


Figure 74: Flatstream communication

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.

6.3.2.4 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.

6.3.2.4.1 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 241.

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

6.3.2.4.2 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 241.

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

6.3.2.4.2.1 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.

6.3.2.4.2.2 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.

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

6.3.2.4.2.3 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.

Relationship between OutputSequence and InputSequence

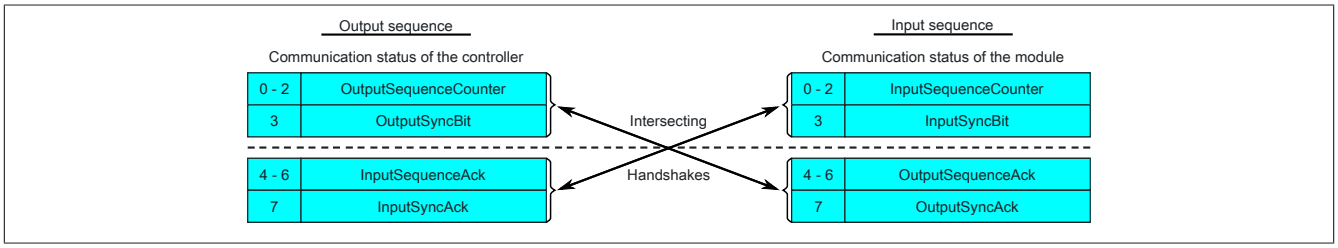


Figure 75: Relationship between OutputSequence and InputSequence

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.

6.3.2.4.3 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.

6.3.2.4.4 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.

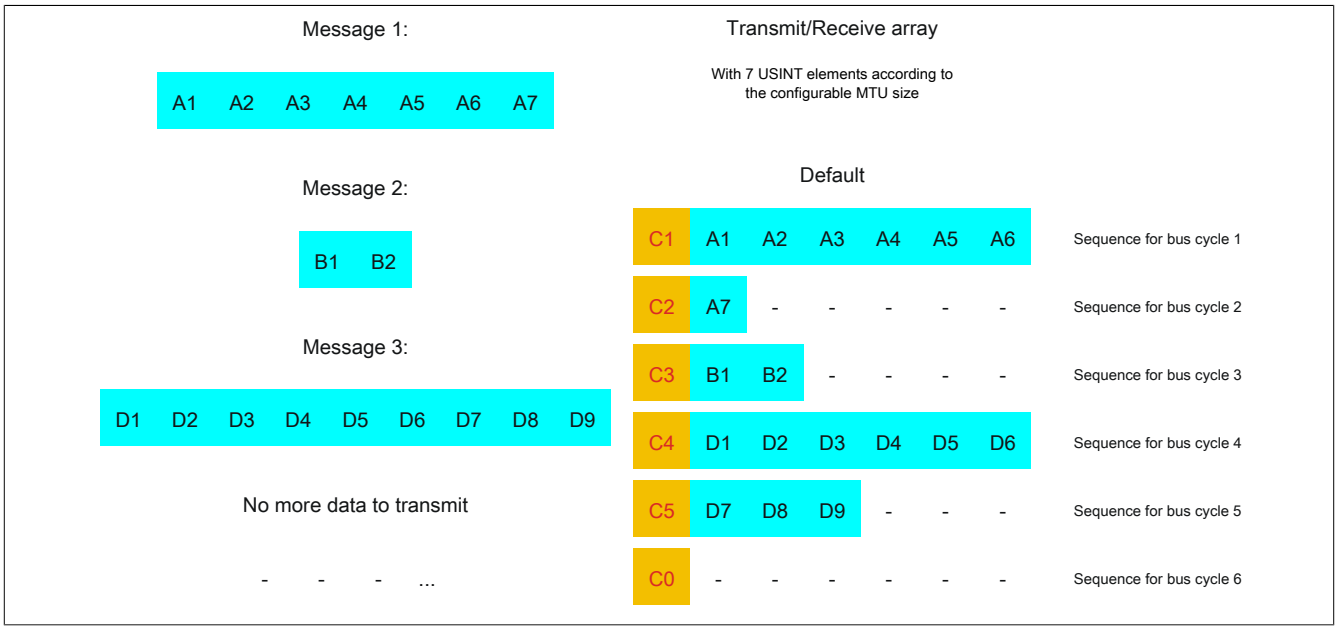


Figure 76: Transmit/Receive array (default)

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 37: 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 38: Flatstream determination of the control bytes for the default configuration example (part 2)

6.3.2.4.4.1 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.

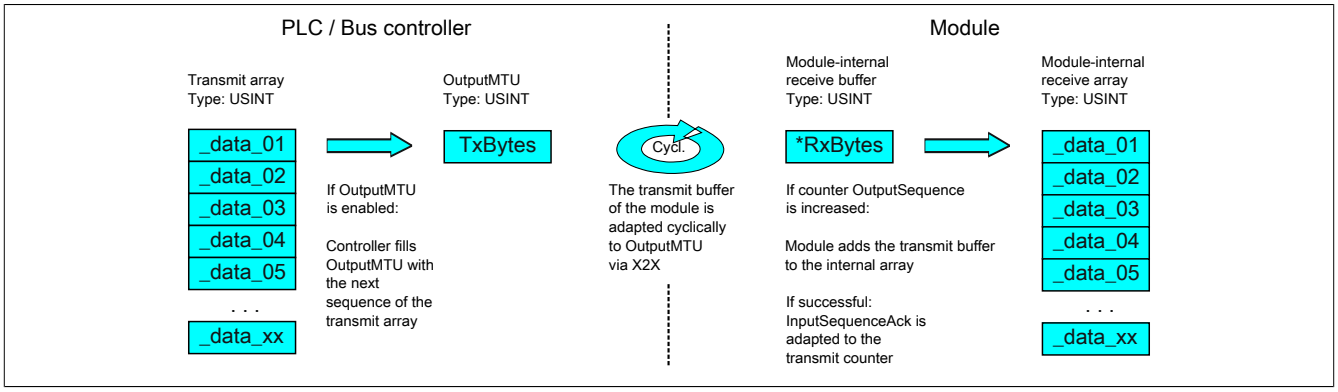


Figure 77: Flatstream communication (output)

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

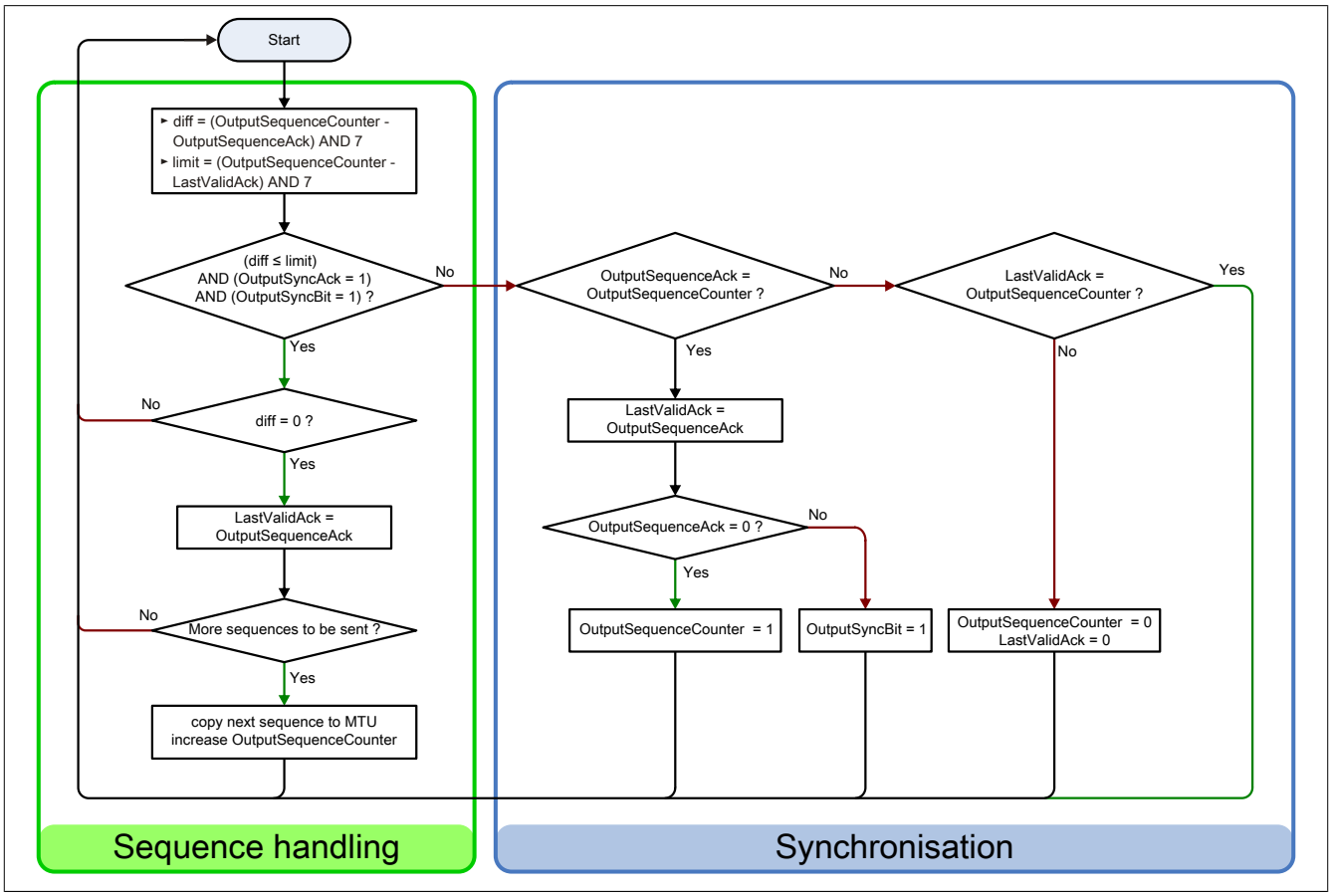


Figure 78: Flowchart for the output direction

6.3.2.4.4.2 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.

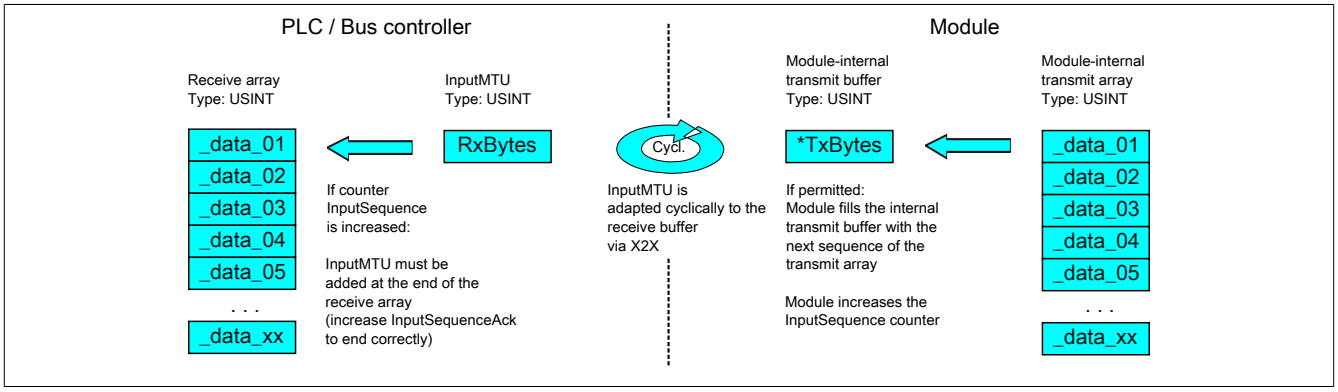


Figure 79: Flatstream communication (input)

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

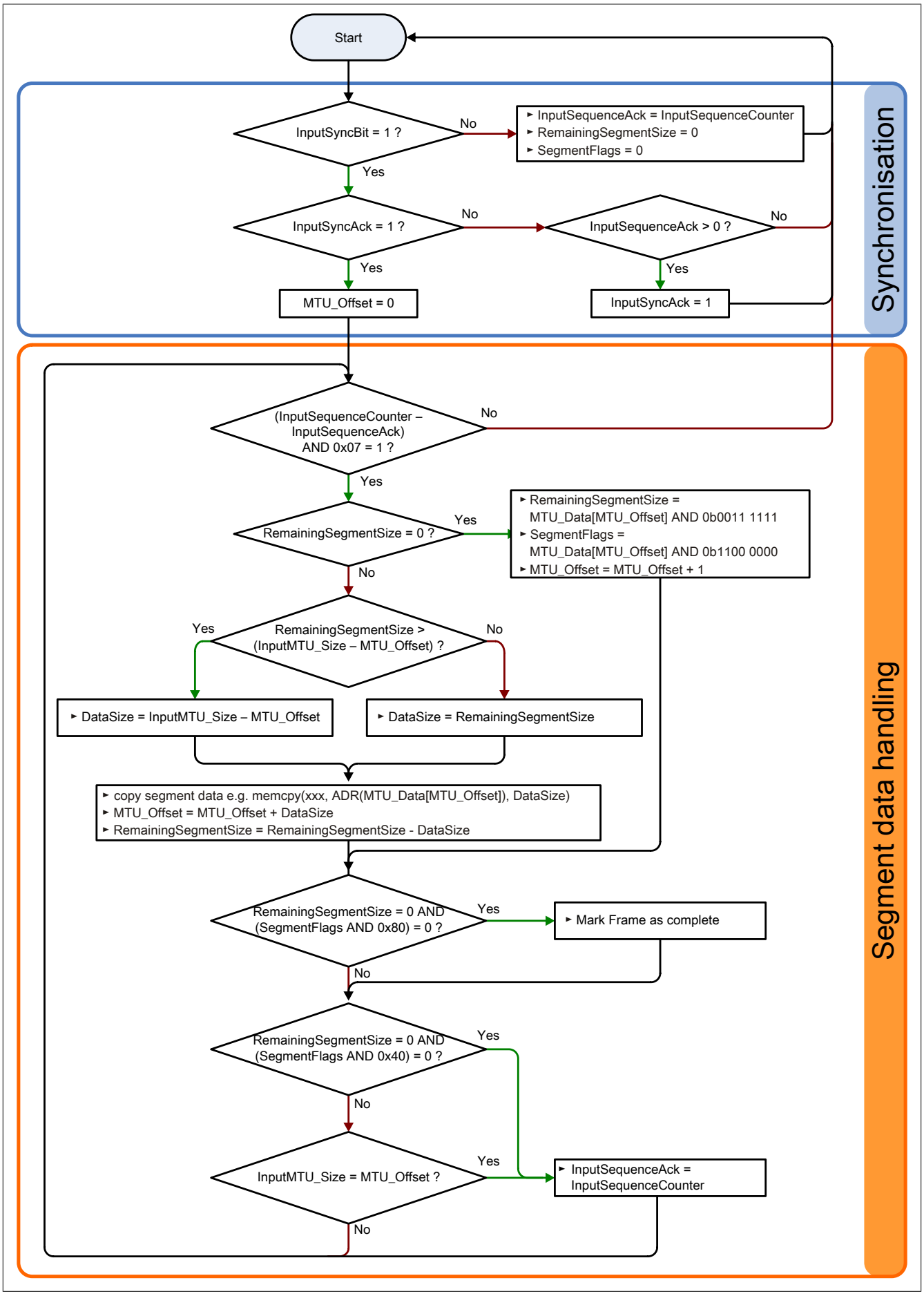


Figure 80: Flowchart for the input direction

6.3.2.4.4.3 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.

6.3.2.4.5 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.

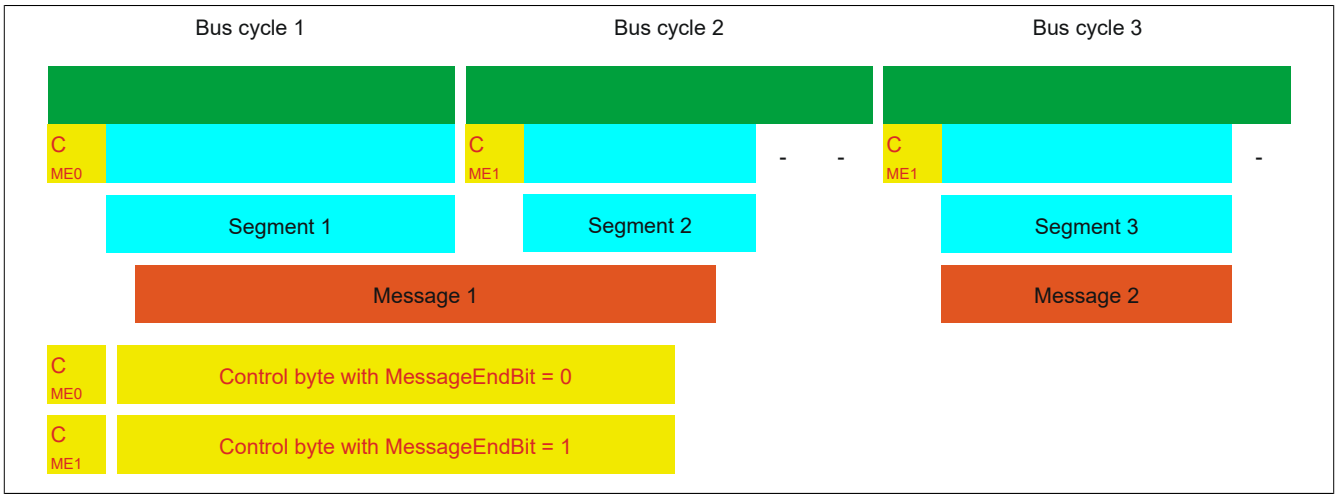


Figure 81: Message arrangement in the MTU (default)

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.

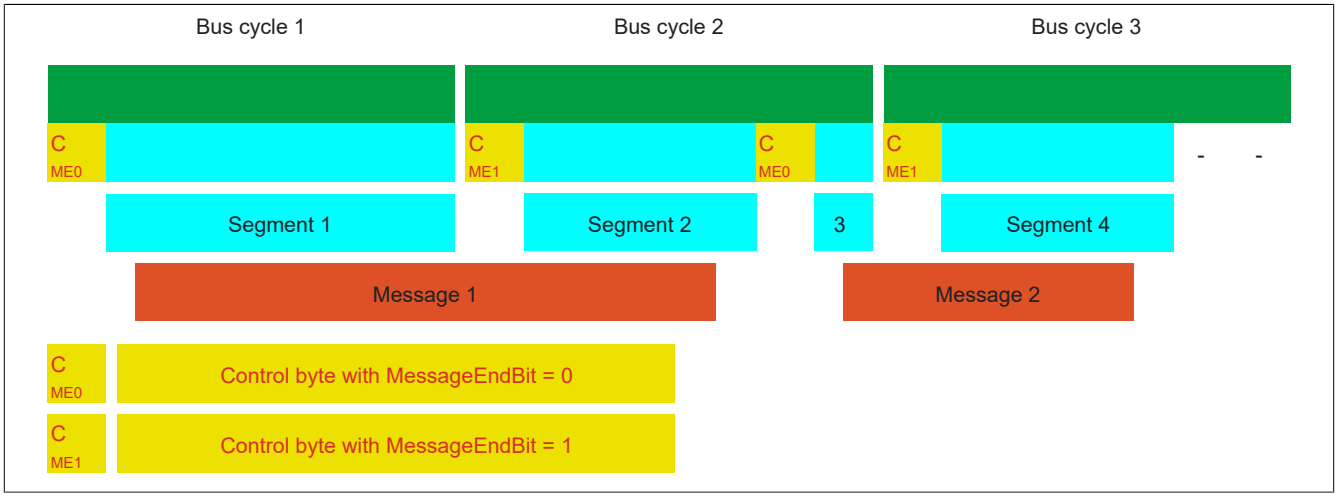


Figure 82: Arrangement of messages in the MTU (MultiSegmentMTU)

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.

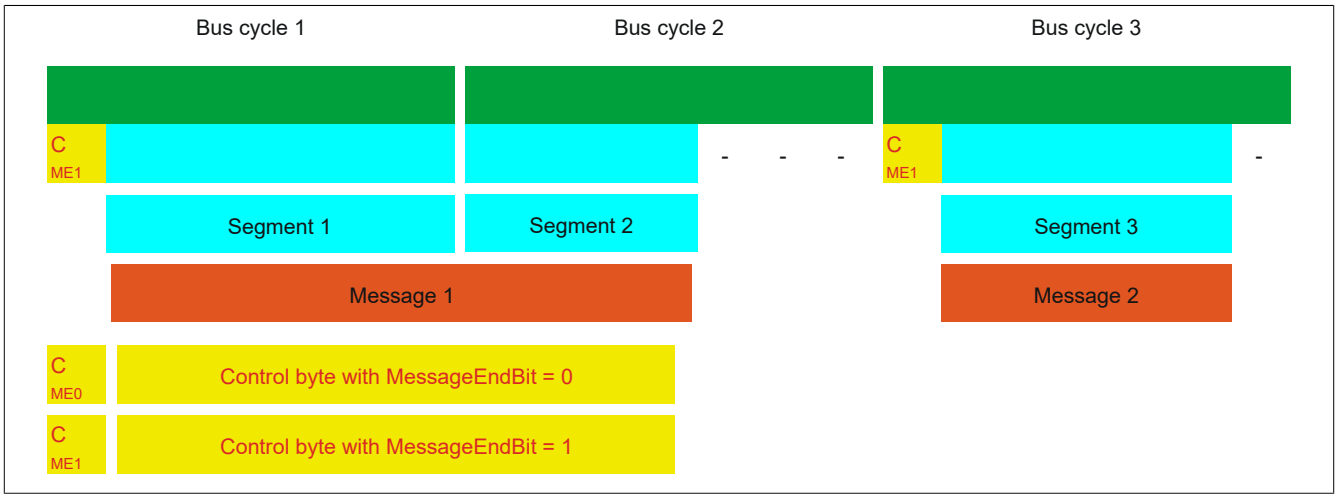


Figure 83: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

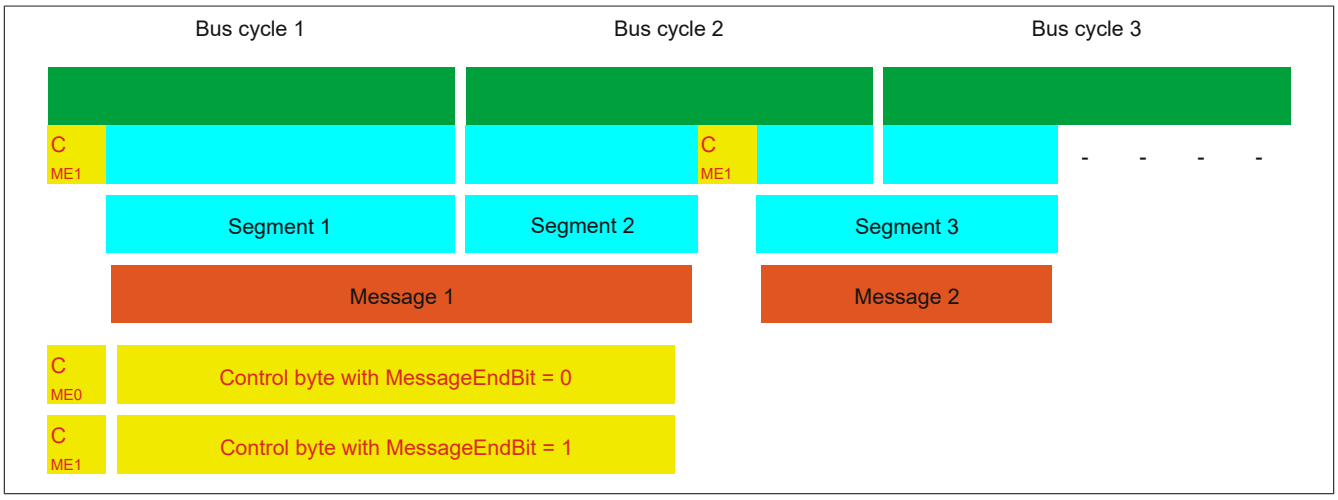


Figure 84: Arrangement of messages in the MTU (large segments and MultiSegmentMTU)

6.3.2.4.6 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 "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.

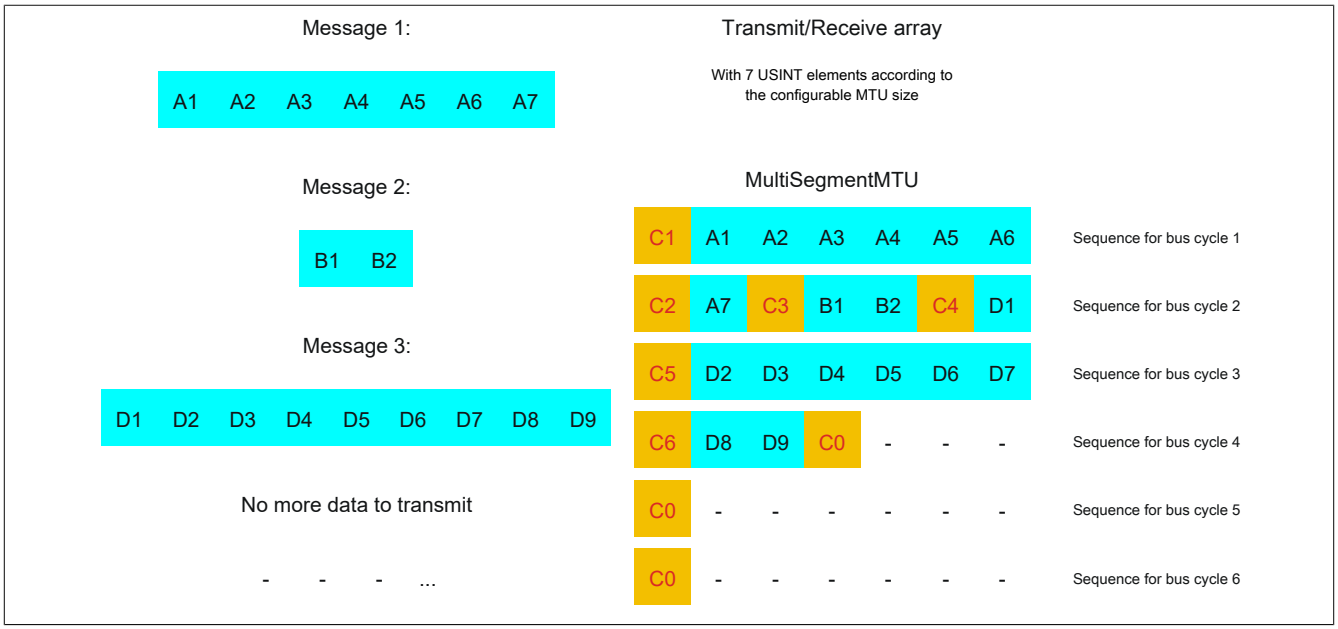


Figure 85: Transmit/Receive array (MultiSegmentMTU)

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 39: 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 40: 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.

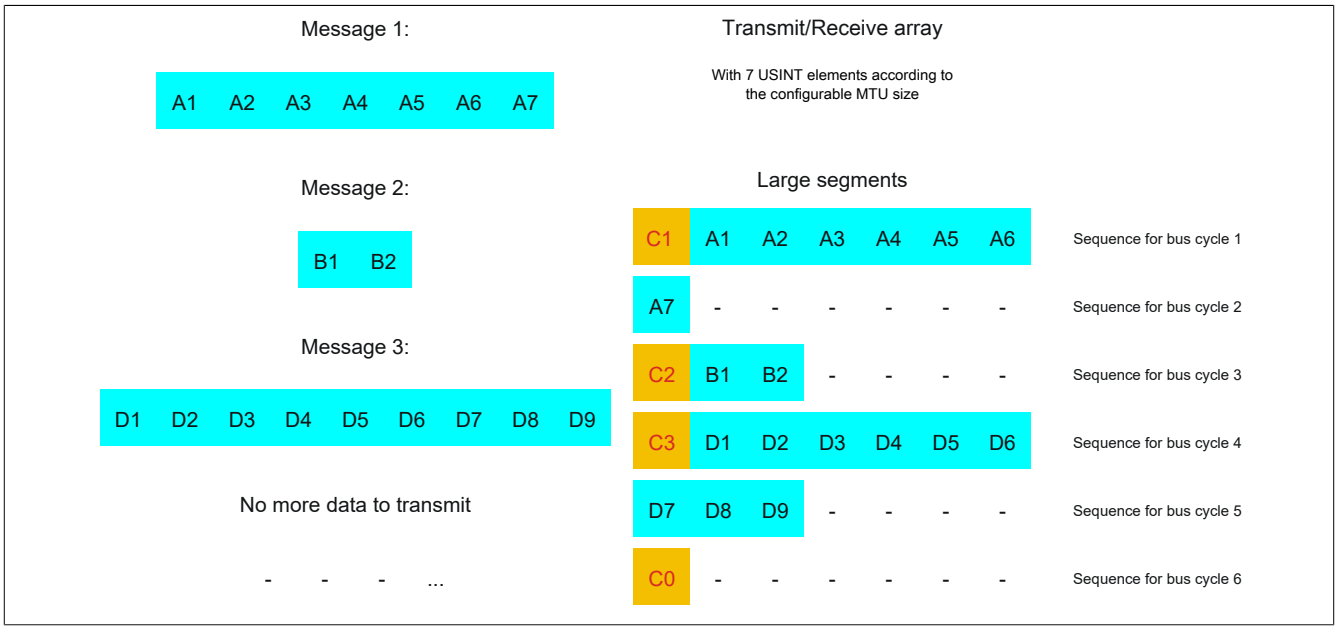


Figure 86: Transmit/receive array (large segments)

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 41: 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.

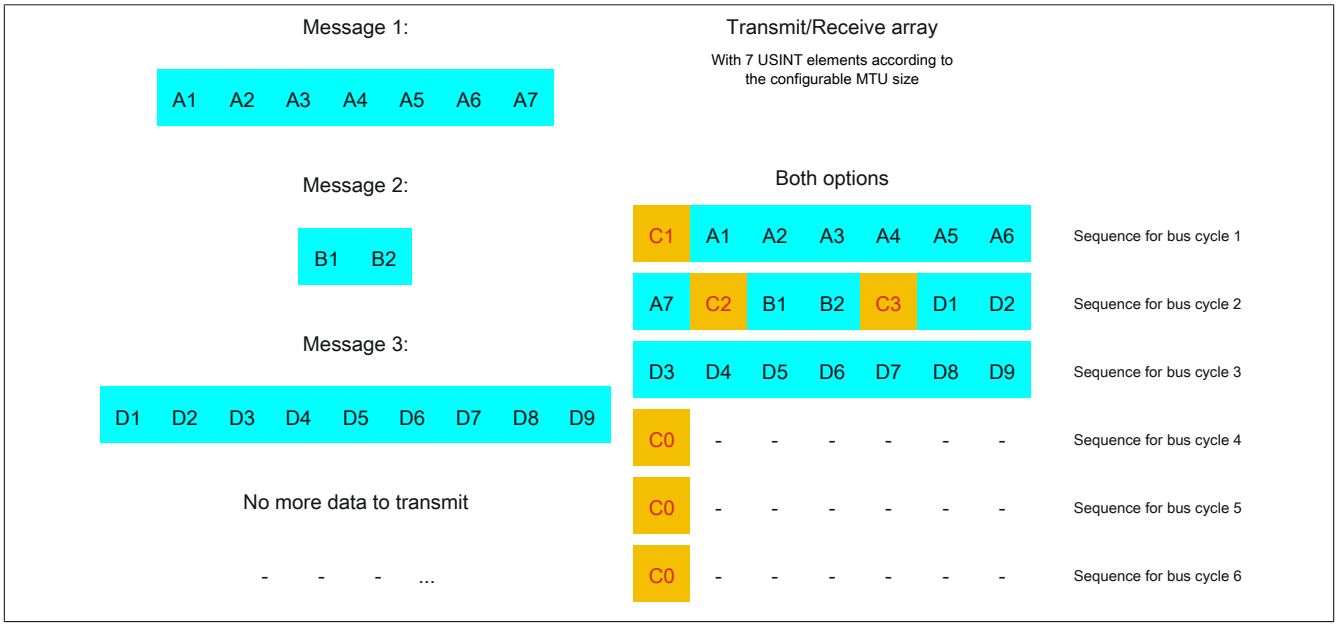


Figure 87: Transmit/Receive array (large segments and MultiSegmentMTU)

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

6.3.2.5 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.

6.3.2.5.1 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 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)

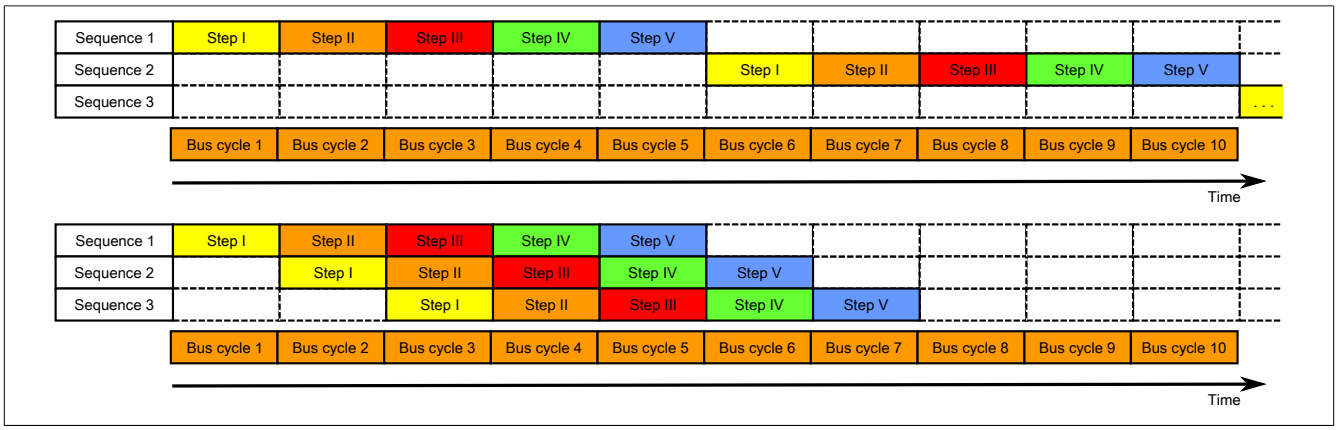


Figure 88: Comparison of transfer without/with Forward

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.

6.3.2.5.2 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 241.

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

6.3.2.5.2.1 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.

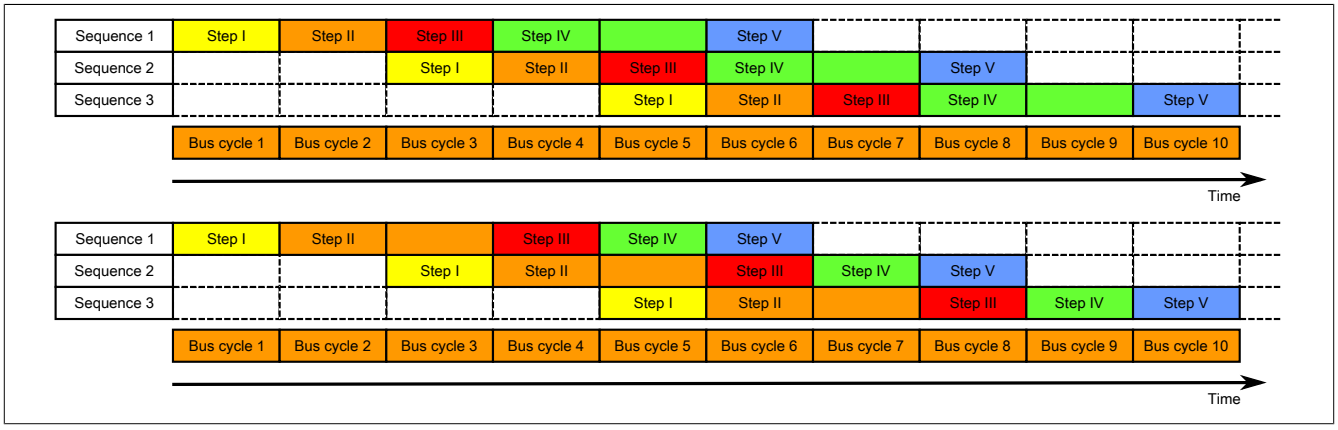


Figure 89: Effect of ForwardDelay when using Flatstream communication with the Forward function

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.

6.3.2.5.3 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.

6.3.2.5.4 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).

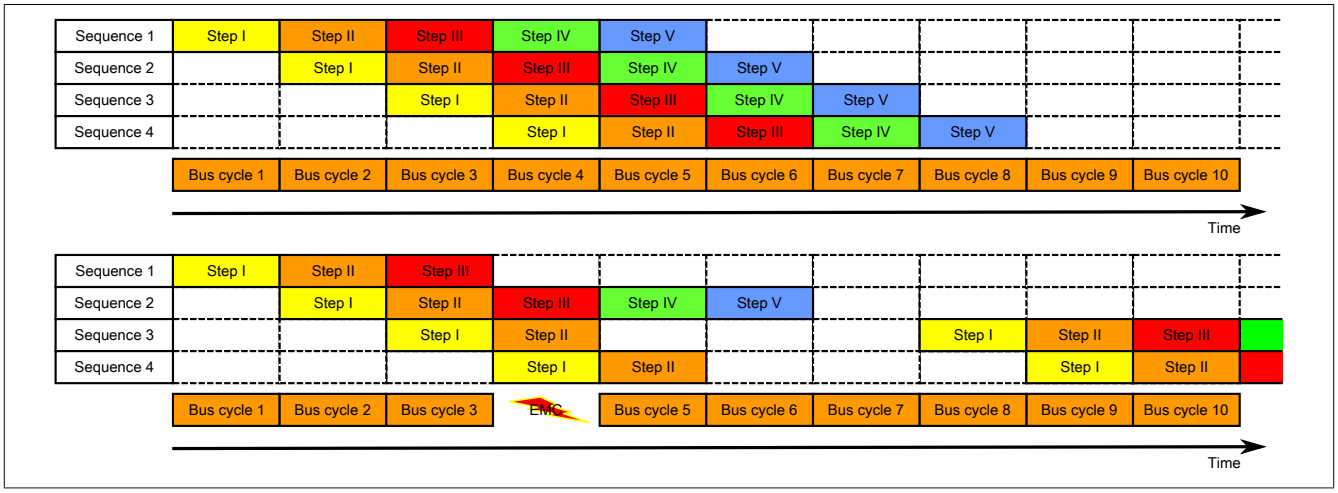


Figure 90: Effect of a lost bus cycle

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.

6.4 Commissioning

6.4.1 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.

6.4.2 Using this module with SGC target systems

> **Information:**
>
> **This module can only be used with SGC target systems if the function model is set to "Flatstream" or "Flat".**

6.4.3 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).

6.4.3.1 CAN I/O bus controller

The module occupies 2 analog logical slots on CAN I/O.

6.5 Register description

6.5.1 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 X20 System user's manual.

6.5.2 Function model 0 - Flat

In the "Flat" function model, 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 201).

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. Then, "TX0[x]Count" is increased to send the transmission. The data is held in the module's internal buffer (max. 18 objects) and transmitted over the CAN network at the next available opportunity.

Receiving information from the CAN network uses the same algorithm. The module saves the CAN messages in its internal buffer along with the respective identifiers. Then the CAN identifier, the CAN data (max. 8 bytes) and the number of bytes to be processed are written to the cyclic I/O data points. RX0[x]Count tells the application how much new data must be taken from these input data points.

Information:

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

Register	Name	Data type	Read		Write	
			Cyclic	Non-cyclic	Cyclic	Non-cyclic
Interface - Configuration						
257	Config01Baudrate	USINT				•
259	Config01SJW	USINT				•
261	Config01SPO	USINT				•
266	Config01TXtrigger	UINT				•
270	CfO_Fifosize_01	UINT				•
673	Cfo_FIFOTXlimit01	USINT				•
677	Cfo_TXRXinfoFlags01	USINT				•
769	Config02Baudrate	USINT				•
771	Config02SJW	USINT				•
773	Config02SPO	USINT				•
778	Config02TXtrigger	UINT				•
782	CfO_Fifosize_02	UINT				•
1185	Cfo_FIFOTXlimit02	USINT				•
1189	Cfo_TXRXinfoFlags02	USINT				•
Stream filter - configuration						
385	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
897	CfO_IF2DefaultCANFilterMode	USINT		•		•
392 + N*16	CfO_IF2CANFilter0N (index N = 1 to 4)	UDINT		•		•
900 + N*16	CfO_IF2CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
Interface - Communication						
641	TX01Count	USINT			•	
513	TX01CountReadBack	USINT	•			
515	RX01Count	USINT	•			
1153	TX02Count	USINT			•	
1025	TX02CountReadBack	USINT	•			
1027	RX02Count	USINT	•			
Transmit buffer IF1						
645	TX01DataSize	USINT			•	
652	TX01Ident	UDINT			•	
Index * 2 + 657	TX01DataByte0 to TX01DataByte7	USINT			•	
Index * 4 + 658	TX01DataWord0 to TX01DataWord3	UINT			•	
Index * 8 + 660	TX01DataLong0 to TX01DataLong1	UDINT			•	
Receive buffer IF1						
517	RX01DataSize	USINT	•			
524	RX01Ident	UDINT	•			
Index * 2 + 529	RX01DataByte0 to RX01DataByte7	USINT	•			
Index * 4 + 530	RX01DataWord0 to RX01DataWord3	UINT	•			

Register	Name	Data type	Read		Write	
			Cyclic	Non-cyclic	Cyclic	Non-cyclic
Index * 8 + 532	RX01DataLong0 to RX01DataLong1	UDINT	•			
Transmit buffer IF2						
1157	TX02DataSize	USINT			•	
1164	TX02Ident	UDINT			•	
Index * 2 + 1170	TX02DataByte0 to TX02DataByte7	USINT			•	
Index * 4 + 658	TX02DataWord0 to TX02DataWord3	UINT			•	
Index * 8 + 1172	TX02DataLong0 to TX02DataLong1	UDINT			•	
Receive buffer IF2						
1029	RX02DataSize	USINT	•			
1036	RX02Ident	UDINT	•			
Index * 2 + 1041	RX02DataByte0 to RX02DataByte7	USINT	•			
Index * 4 + 1042	RX02DataWord0 to RX02DataWord3	UINT	•			
Index * 8 + 1044	RX02DataLong0 to RX02DataLong1	UDINT	•			

6.5.3 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, 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	-				
Interface - Configuration						
270	CfO_Fifosize_01	UINT				•
782	CfO_Fifosize_02	UINT				•
6273	CfO_ErrorID0007	USINT				•
Stream filter - configuration						
385	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
897	CfO_IF2DefaultCANFilterMode	USINT		•		•
892 + N*16	CfO_IF2CANFilter0N (index N = 1 to 4)	UDINT		•		•
900 + N*16	CfO_IF2CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
Interface - Communication						
6145	CAN error status	USINT	•			
	CANIF1warning	Bit 0				
	CANIF1passive	Bit 1				
	CANIF1busoff	Bit 2				
	CANIF1RXoverrun	Bit 3				
	CANIF2warning	Bit 4				
	CANIF2passive	Bit 5				
	CANIF2busoff	Bit 6				
	CANIF2RXoverrun	Bit 7				
6209	CAN error acknowledgment	USINT			•	
	QuitCANIF1warning	Bit 0				
	QuitCANIF1passive	Bit 1				
	QuitCANIF1bussoff	Bit 2				
	QuitCANIF1RXoverrun	Bit 3				
	QuitCANIF2warning	Bit 4				
	QuitCANIF2passive	Bit 5				
	QuitCANIF2bussoff	Bit 6				
	QuitCANIF2RXoverrun	Bit 7				

6.5.4 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 203).

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 the "Flat" function model.**

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Interface - Configuration						
257	Config01Baudrate	USINT				•
259	Config01SJW	USINT				•
261	Config01SPO	USINT				•
266	Config01TXtrigger	UINT				•
270	CfO_Fifosize_01	UINT				•
769	Config02Baudrate	USINT				•
771	Config02SJW	USINT				•
773	Config02SPO	USINT				•
778	Config02TXtrigger	UINT				•
782	CfO_Fifosize_01	UINT				•
6273	CfO_ErrorID0007	USINT				•
Stream filter - configuration						
385	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
897	CfO_IF2DefaultCANFilterMode	USINT		•		•
392 + N*16	CfO_IF2CANFilter0N (index N = 1 to 4)	UDINT		•		•
900 + N*16	CfO_IF2CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
Interface - Communication						
6145	CAN error status	USINT	•			
	CANIF1warning	Bit 0				
	CANIF1passive	Bit 1				
	CANIF1busoff	Bit 2				
	CANIF1RXoverrun	Bit 3				
	CANIF2warning	Bit 4				
	CANIF2passive	Bit 5				
	CANIF2busoff	Bit 6				
	CANIF2RXoverrun	Bit 7				
6209	CAN error acknowledgment	USINT			•	
	QuitCANIF1warning	Bit 0				
	QuitCANIF1passive	Bit 1				
	QuitCANIF1bussoff	Bit 2				
	QuitCANIF1RXoverrun	Bit 3				
	QuitCANIF2warning	Bit 4				
	QuitCANIF2passive	Bit 5				
	QuitCANIF2bussoff	Bit 6				
	QuitCANIF2RXoverrun	Bit 7				
Flatstream - Configuration						
193	output01MTU	USINT				•
195	input01MTU	USINT				•
197	mode01	USINT				•
199	forward01	USINT				•
206	forwardDelay01	UINT				•
209	output02MTU	USINT				•
211	input02MTU	USINT				•
213	mode02	USINT				•
215	forward02	USINT				•
222	forwardDelay02	UINT				•
Flatstream - Communication						
0	Input01Sequence	USINT	•			
64	Input02Sequence	USINT	•			
Index * 1 + 0	Rx01Byte1 to Rx01Byte27	USINT	•			
Index * 1 + 64	Rx02Byte1 to Rx02Byte27	USINT	•			

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
32	Output01Sequence	USINT			•	
96	Output02Sequence	USINT			•	
Index * 1 + 32	Tx01Byte1 to Tx01Byte27	USINT			•	
Index * 1 + 96	Tx02Byte1 to Tx02Byte27	USINT			•	

6.5.5 Function model 254 - Bus controller

The "Bus controller" function model is a reduced form of the "FlatStream" function model. 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
Interface - Configuration							
257	-	Config01Baudrate	USINT				•
259	-	Config01SJW	USINT				•
261	-	Config01SPO	USINT				•
266	-	Config01TXtrigger	UINT				•
270	-	CfO_Fifosize_01	UINT				•
769	-	Config02Baudrate	USINT				•
771	-	Config02SJW	USINT				•
773	-	Config02SPO	USINT				•
778	-	Config02TXtrigger	UINT				•
782	-	CfO_Fifosize_02	UINT				•
6273	-	CfO_ErrorID0007	USINT				•
Stream filter - configuration							
385	-	CfO_IF1DefaultCANFilterMode	USINT		•		•
380 + N*16	-	CfO_IF1CANFilter0N (index N = 1 to 4)	UDINT		•		•
388 + N*16	-	CfO_IF1CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
897	-	CfO_IF2DefaultCANFilterMode	USINT		•		•
392 + N*16	-	CfO_IF2CANFilter0N (index N = 1 to 4)	UDINT		•		•
900 + N*16	-	CfO_IF2CANFilterMask0N (index N = 1 to 4)	UDINT		•		•
Interface - Communication							
6145	-	CAN error status	USINT		•		
		CANIF1warning	Bit 0				
		CANIF1passive	Bit 1				
		CANIF1busoff	Bit 2				
		CANIF1RXoverrun	Bit 3				
		CANIF2warning	Bit 4				
		CANIF2passive	Bit 5				
		CANIF2busoff	Bit 6				
		CANIF2RXoverrun	Bit 7				
6209	-	CAN error acknowledgment	USINT				•
		QuitCANIF1warning	Bit 0				
		QuitCANIF1passive	Bit 1				
		QuitCANIF1bussoff	Bit 2				
		QuitCANIF1RXoverrun	Bit 3				
		QuitCANIF2warning	Bit 4				
		QuitCANIF2passive	Bit 5				
		QuitCANIF2bussoff	Bit 6				
		QuitCANIF2RXoverrun	Bit 7				
Flatstream - Configuration							
193	-	output01MTU	USINT				•
195	-	input01MTU	USINT				•
197	-	mode01	USINT				•
199	-	forward01	USINT				•
206	-	forwardDelay01	UINT				•
209	-	output02MTU	USINT				•
211	-	input02MTU	USINT				•
213	-	mode02	USINT				•
215	-	forward02	USINT				•
222	-	forwardDelay02	UINT				•
Flatstream - Communication							
0	0	Input01Sequence	USINT	•			
64	8	Input02Sequence	USINT	•			
Index * 1 + 0	Index * 1 + 0	Rx01Byte1 to Rx01Byte7	USINT	•			
Index * 1 + 64	Index * 1 + 8	Rx02Byte1 to Rx02Byte7	USINT	•			
32	0	Output01Sequence	USINT			•	
96	8	Output02Sequence	USINT			•	
Index * 1 + 32	Index * 1 + 0	Tx01Byte1 to Tx01Byte7	USINT			•	
Index * 1 + 96	Index * 1 + 8	Tx02Byte1 to Tx02Byte7	USINT			•	

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

6.5.6 Interface - Configuration

6.5.6.1 Transfer rate

Name:
Config01Baudrate to Config02Baudrate

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

Configuration of the CAN transfer rate for the respective 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
		8	800 kbit/s
		9	1000 kbit/s
4 - 7	Reserved	-	

6.5.6.2 Synchronization Jump Width

Name:
Config01SJW to Config02SJW

"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.

A detailed description of the SJW can be found in the CAN specification.

Data type	Value	Meaning
USINT	0 to 4	Synchronization jump width. Bus controller default setting: 3

6.5.6.3 Offset for the sampling instant

Name:
Config01SPO to Config02SPO

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

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

A detailed description of the SPO can be found in the CAN specification.

Data type	Value	Meaning
USINT	0 to 1	Sample point offset. Bus controller default setting: 0

6.5.6.4 Start of transmission

Name:
Config01TXtrigger to Config02TXtrigger

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

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

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

6.5.6.5 Configuration of error messages

Name:
CfO_ErrorID0007

This register must be used first to configure the error messages that have to be transferred. If the corresponding enable bit is not set, no error status will be sent 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	CANIF1warning	0	Disabled (bus controller default setting)
		1	Enabled
1	CANIF1passive	0	Disabled (bus controller default setting)
		1	Enabled
2	CANIF1bussoff	0	Disabled (bus controller default setting)
		1	Enabled
3	CANIF1RXoverrun	0	Disabled (bus controller default setting)
		1	Enabled
4	CANIF2warning	0	Disabled (bus controller default setting)
		1	Enabled
5	CANIF2passive	0	Disabled (bus controller default setting)
		1	Enabled
6	CANIF2bussoff	0	Disabled (bus controller default setting)
		1	Enabled
7	CANIF2RXoverrun	0	Disabled (bus controller default setting)
		1	Enabled

6.5.6.6 Size of the transmit buffer

Name:
Cfo_FIFOTXlimit01 to Cfo_FIFOTXlimit02

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

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

Data type	Value	Meaning
USINT	0 to 18	Size of the transmit buffer

6.5.6.7 FIFO memory size

Name:
CfO_Fifosize_01 to CfO_Fifosize_02

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

Determines the size of the FIFO memory for the respective interface.

Data type	Values	Explanation
UINT	20 to 4096	Size of the FIFO memory in bytes

6.5.6.8 Display of unprocessed elements remaining in transmit/receive buffer

Name:
Cfo_TXRXinfoFlags01 to Cfo_TXRXinfoFlags02

This register can be used to specify that the number of unprocessed elements in the transmit and receive buffers is indicated in the upper 4 bits of the "TX0[x]CountReadBack" and "RX0[x]Count" registers for the respective interface.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	TxFifoInfo "Mode of channel TX0[x]CountReadBack" in the Automation Studio I/O configuration	0	The "TX0[x]Count" is read in the "TX0[x]CountReadBack" on page 238 register.
		1	The "TX0[x]Count" is read in the "TX0[x]CountReadBack" on page 238 register. 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 RX0[x]Count" in the Automation Studio I/O configuration	0	The number of received telegrams is shown in the "RX0[x]Count" on page 239 register.
		1	The number of received telegrams is shown in the lower 4 bits of the "RX0[x]Count" on page 239 register. The upper 4 bits are used to indicate the number of received but not acknowledged telegrams in the receive buffer.
2 - 7	Reserved	-	

6.5.6.9 Stream filter

Up to 4 stream filters can be configured per CAN interface.

6.5.6.9.1 CANFilterMode

Name:
CfO_IF1DefaultCANFilterMode to CfO_IF2DefaultCANFilterMode

These registers specify the default settings for IDs that do not match any of the set filters.

Data type	Values	Information
USINT	0	No filter response, the CAN frame is discarded.
	1	No filter response, the CAN frame is transferred via the stream.

6.5.6.9.2 CAN filter

Name:
CfO_IF1CANFilter01 to CfO_IF1CANFilter04
CfO_IF2CANFilter01 to CfO_IF2CANFilter04

The filter properties are defined in these registers.

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0 to 28	Filter ID	x	Identifier value for filtering.[1)]
29	Frame format	0	Standard frame format (SFF) with 11-bit identifier. Possible filter ID values: 0 to 2047 (0x7FF)
		1	Extended frame format (EFF) with 29-bit identifier. Possible filter ID values: 0 to 536 870 911 (0x1FFFFF)
30	Reserved	-	
31	Enable	0	Filter inactive
		1	Filter active

1) This value is linked with the identifier value and mask value (see example).

6.5.6.9.3 CANFilterMask

Name:
CfO_IF1CANFilterMask01 to CfO_IF1CANFilterMask04
CfO_IF2CANFilterMask01 to CfO_IF2CANFilterMask04

The filter mask and filter mode are defined in these registers.

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0 to 28	Filter mask	x	Comparison bit pattern for filter ID[1)]
29	Frame format mask	0	The frame format of the received message must match the configuration in "CfO_IFxCANFilter" on page 235.
		1	The filter applies to both frame formats. 11-bit and 29-bit identifiers are filtered.
30	Reserved	-	
31	Mode	0	The CAN frame is transferred when the filter responds.
		1	The CAN frame is discarded when the filter responds.

1) Only those bits of the ID are compared that are set to 0 in the mask (see "Examples" on page).

6.5.7 Interface - Communication

6.5.7.1 CAN error status

Name:
CAN error status

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 238).

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	CANIF1warning	0	No error
		1	CANwarning Error occurred on IF1
1	CANIF1passive	0	No error
		1	CANpassive Error occurred on IF1
2	CANIF1busoff	0	No error
		1	CANbusoff Error occurred on IF1
3	CANIF1RXoverrun	0	No error
		1	CANRXoverrun Error occurred on IF1
4	CANIF2warning	0	No error
		1	CANwarning Error occurred on IF2
5	CANIF2passive	0	No error
		1	CANpassive Error occurred on IF2
6	CANIF2busoff	0	No error
		1	CANbusoff Error occurred on IF2
7	CANIF2RXoverrun	0	No error
		1	CANRXoverrun Error occurred on IF2

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.

6.5.7.2 CAN error acknowledgment

Name:
CAN error acknowledgment

Setting the bits in this register acknowledges the error assigned to the bit and clears the corresponding bit in the "CAN error status" register. The application thus informs the module that it has recognized the error state.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	QuitCANIF1warning	0	No acknowledgment
		1	Acknowledge CANwarning error on IF1
1	QuitCANIF1passive	0	No acknowledgment
		1	Acknowledge CANpassive error on IF1
2	QuitCANIF1bussoff	0	No acknowledgment
		1	Acknowledge CANbusoff error on IF1
3	QuitCANIF1RXoverrun	0	No acknowledgment
		1	Acknowledge CANRXoverrun error on IF1
4	QuitCANIF2warning	0	No acknowledgment
		1	Acknowledge CANwarning error on IF2
5	QuitCANIF2passive	0	No acknowledgment
		1	Acknowledge CANpassive error on IF2
6	QuitCANIF2bussoff	0	No acknowledgment
		1	Acknowledge CANbusoff error on IF2
7	QuitCANIF2RXoverrun	0	No acknowledgment
		1	Acknowledge CANRXoverrun error on IF2

6.5.7.3 New CAN telegram for transmit buffer

Name:
TX01Count to TX02Count

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

Data type	Value
USINT	0 to 255

6.5.7.4 Read "TX0[x]Count"

Name:
TX01CountReadBack to TX02CountReadBack

The value of "TX0[x]Count" is copied from the module into this register. This makes it possible for the application task to verify that the CAN telegram data was transfered from the module correctly.

The meaning of the value depends on the "TxFifoInfo" bit. This is located in the register "Cfo_TXRXinfoFlags0[x]" on page 235.

Data type	Value	"TxFifoInfo" bit	Meaning
USINT	0 to 255	0	Read "TX0[x]Count"
		1	See bit structure.

Bit structure:

Bit	Function	Value	Information
0 - 3	Read "TX0[x]Count"	0 to 15	Only the lower 4 bits
4 - 7	Number of frames in the transmit buffer that have not been transmitted	0 to 15	If this number exceeds the 15 (a maximum of 18 possible), the value 15 is returned.

6.5.7.5 Counter for received CAN telegrams

Name:
RX01Count to RX02Count

This counter is increased by 1 with each CAN telegram. The application task can thus detect when new data is received and get it from the corresponding "RX0[x]Data" registers.

The meaning of the value depends on the "Cfo_TXRXinfoFlags0[x]" on page 235 bit in the "Cfo_TXRXinfoFlags" register.

Data type	Value	"RxFifoInfo" bit	Meaning
USINT	0 to 255	0	Counter for received telegrams
		1	See bit structure.

Bit structure:

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

6.5.8 Transmit buffer for IF1 and IF2

6.5.8.1 Number of CAN payload data bytes

Name:
TX01DataSize to TX02DataSize

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 202).

Data type	Value	Meaning
USINT	-128 to 8	Amount of CAN payload data to be transmitted

6.5.8.2 Identifier of the CAN telegram to be transmitted.

Name:
TX01Ident to TX02Ident

Identifier of the CAN telegram to be transmitted. The frame format and the 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.

6.5.8.3 Configuration of the CAN payload data being sent

Name:
TX0[x]DataByte0 to TX0[x]DataByte7
TX0[x]DataWord0 to TX0[x]DataWord3
TX0[x]DataLong0 to TX0[x]DataLong1

CAN payload data in the transmit direction. The 8 payload data bytes for a telegram can be used as data points with 8 individual bytes, 4 words or 2 longs as needed.

Data type	Value	Description
USINT	0 to 255	CAN payload data transmitted as bytes
UINT	0 to 65,535	CAN payload data transmitted as words
UDINT	0 to 4.294.967.295	CAN payload data transmitted as longs

6.5.9 Receive buffer for IF1 and IF2

6.5.9.1 Number of valid CAN payload data bytes

Name:
RX01DataSize to RX02DataSize

Number of valid CAN payload data bytes.

This register also uses the value -1 (0xFF) to indicate a general error or gap in the input data stream. Details regarding the error that has occurred can be seen in the "CAN error status" on page 237 register.

Data type	Value	Meaning
USINT	1 to 8	Number CAN payload data
	-1	Error

6.5.9.2 Identifier of the received data

Name:
RX01Ident to RX02Ident

Identifiers assigned to the received data. The frame format and the 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.

6.5.9.3 Configuration of the CAN payload data to be received

Name:
RX0[x]DataByte0 to RX0[x]DataByte7
RX0[x]DataWord0 to RX0[x]DataWord3
RX0[x]DataLong0 to RX0[x]DataLong1

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 avoid losing CAN objects, the application must respond immediately to a change in the "RX0[x]Count" and copies the data from these registers.

The maximum 8 bytes for a CAN telegram can be used as data points with 8 individual bytes, 4 words or 2 longs as needed.

Data type	Value	Description
USINT	0 to 255	Received CAN payload data as bytes
UINT	0 to 65,535	Received CAN payload data as words
UDINT	0 to 4.294.967.295	Received CAN payload data as longs

6.5.10 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" on page 203.

6.5.10.1 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.

6.5.10.2 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

6.5.10.3 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.

6.5.10.4 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.

6.5.10.5 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		

6.5.10.6 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

6.5.10.7 Delay time

Name:
ForwardDelay

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

Data type	Values
UINT	0 to 65535 [µs] Default: 0

6.5.11 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

6.5.12 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 µs

6.5.13 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 µs

7 For reference only

Information:

The data sheets in this section are for reference only when the modules are already in use.

7.1 X20CS1011

7.1.1 General information

SmartWire from the company Moeller makes it possible to very easily integrate switching devices such as contactors or motor protection switches in the X20 system without extensive wiring. It replaces the control circuit wiring between the controller and switching devices with pluggable, pre-assembled connection cables.

Although SmartWire is an intelligent connection, this changes almost nothing for the machine programmer. Integration in the X20 system via the interface module cuts down on overall communication. The individual switching devices can simply be viewed as digital inputs and outputs.

Practical applications

SmartWire allows up to 16 switching devices to be connected using pre-assembled cables and attached to the X20 SmartWire interface module. The system can configure itself completely at the push of a button without additional intervention or effort. This replaces the wiring test that was previously necessary.

At the same time, the device configuration is known to the system. If a device is no longer available due to an error or intervention, it will be detected immediately. Once corrected, the system continues to run.

The interface module is designed as a normal electronic module, which means it can be placed anywhere on the remote backplane.

- X2X SmartWire master for controlling up to 16 SmartWire slaves
- Simple connection using pre-assembled connection cables
- Moeller SmartWire modules for Moeller standard switching devices
- Replaces control circuit wiring
- Contactor activation
- Contactor switching status
- Motor circuit breaker status
- 24 VDC control voltage via SmartWire connection cable

7.1.1.1 Other applicable documents

For additional and supplementary information, see the following documents.

Other applicable documents

Document name	Title
MAX20	X20 System user's manual
MAEMV	Installation / EMC guide

7.1.2 Order data

Order number	Short description	Figure
	X20 electronics module communication	
X20CS1011	X20 interface module, 1 Moeller SmartWire interface	
	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	
	Others	
X20CA4S00.0005	SmartWire attachment cable, X20TB12 to SmartWire connector, 0.5 m	
X20CA4S00.0015	SmartWire attachment cable, X20TB12 to SmartWire connector, 1.5 m	

Table 43: X20CS1011 - Order data

7.1.3 Technical description

7.1.3.1 Technical data

Order number	X20CS1011
Short description	
Communication module	1 SmartWire master for controlling up to 16 slaves
General information	
B&R ID code	0xA38D
Status indicators	SmartWire bus function, external supply voltage, operating state, module status
Diagnostics	
Module run/error	Yes, using LED status indicator and software
SmartWire operating state	Yes, using LED status indicator and software
U Aux	Yes, using LED status indicator
Power output	
Internal I/O	6.8 W for power supply of external slaves (corresponds to 16 slaves at 0.425 W each)
Power consumption	
Bus	0.01 W
Internal I/O	1.5 W
Additional power dissipation caused by actuators (resistive) [W]	-
Certifications	
CE	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
EAC	Yes
KC	Yes
Interfaces	
Interface	
Type	SmartWire (LIN bus)
Variant	Connection via 12-pin terminal block X20TB12
Transfer rate	19200 bit/s
SmartWire	
Data format	1 start bit, 8 data bits, 1 parity bit, 1 stop bit
Max. distance	4 m
Configuration button	
Internal	Integrated in the module on the bottom of the housing
External	Connection via 12-pin terminal block Normally open contact, not galvanically isolated (use dry contact)
SWIRE terminal 1 (24 VDC)	
Voltage drop for reverse polarity protection at 3 A	Max. 0.1 VDC
Voltage range	Voltage like supply
Current-carrying capacity	Max. 3 A
Short-circuit proof	No, only with external fuse protection
Monitoring	20 VDC < 24 VDC Aux < 29.4 VDC (per firmware)
SWIRE terminal 2	
Daisy chain signal	5 VDC, CMOS level
SWIRE terminal 5 (bus level)	
Dominant	<2 VDC
Recessive	>14.85 VDC
SWIRE terminal 6 (17 VDC)	
Voltage range	Typ. 16.6 VDC (16.3 VDC to 16.8 VDC)
Summation current	Max. 400 mA for 16 SmartWire slaves
Short-circuit proof	Yes
Monitoring	14.2 VDC < 17 VDC bus < 17.9 VDC (per firmware)
U-Aux (24 VDC aux supply)	
Connection	External via 12-pin terminal block [1)]
Input voltage	24 VDC -15% / +20%
Fuse	Recommended 3 A slow-blow line fuse
Summation current	Max. 3 A for 16 SmartWire slaves
Reverse polarity protection	Yes
Electrical properties	
Electrical isolation	SmartWire isolated from bus SmartWire power supply (17 VDC) not isolated from I/O power supply

Table 44: X20CS1011 - Technical data

Order number	X20CS1011
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
Ambient conditions	
Temperature	
Operation	
Horizontal mounting orientation	0 to 55°C
Vertical mounting orientation	0 to 50°C
Derating	-
Storage	-25 to 70°C
Transport	-25 to 70°C
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order SmartWire attachment cable X20CA4S00.00xx separately. Order 1x bus module X20BM11 separately.
Pitch	12.5$^{+0.2}$ mm

Table 44: X20CS1011 - Technical data

1) It is possible to switch off via EMERGENCY STOP DEVICE or safety relay by using an external power supply.

7.1.3.2 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
			On	Error or reset status
			Single flash	Warning/Error on an I/O channel
	e + r	Red on / Green single flash		Invalid firmware
	S + R	Green		The "S" and "R" LEDs indicate the status of the SmartWire interface.
	A	Green	Off	U-Aux supply missing or too low
			On	U-Aux supply OK

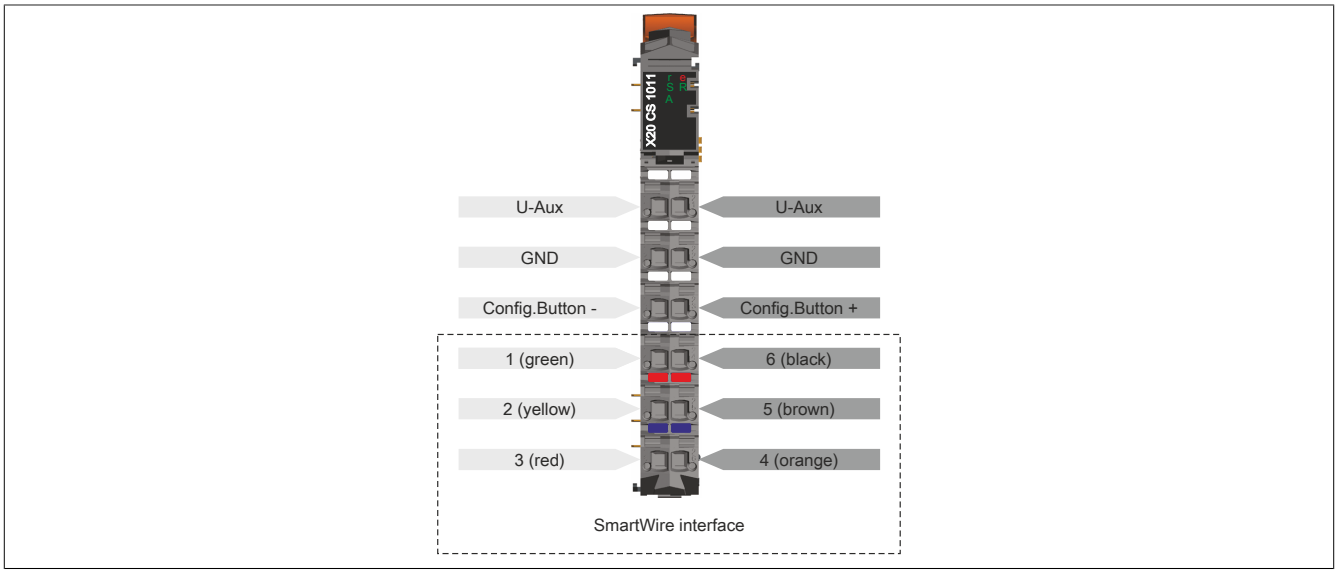
1) Depending on the configuration, a firmware update can take up to several minutes.

"S" and "R" LEDs

The status of the SmartWire interface is indicated by the "S" and "R" LEDs.

S	R	Firmware status	Description
Off	Off	CHECK_INT_FRAM	Initialization
		CHECK_LIN_SUPPLY	Wait until 17 VDC bus is OK
		INT_ERROR_STATE	Internal error has occurred (remains)
Off	On	SET_TRANSCEIVER_MODE	Initialization of transceiver
		RESET_UART	Initialization of UART + 10 ms delay
		READ_REVISION_CNT	Initialization (revision counter from FLASH)
Blinking slowly	Blinking slowly	INIT_LIN_SCAN	Initialization for bus scan
		RUN_LIN_SCAN	Perform bus scan
		INIT_LIN_SETUP	Initialization for RUN_LIN_SETUP
		RUN_LIN_SETUP	Perform bus setup
		STORE_REVISION_CNT	Revision counter in FLASH
		WAIT_FOR_PUSHBUTTON	Wait for configuration button after bus scan and difference with existing configuration
Blinking slowly	Blinking quickly	TIME_DELAY	4 s optical confirmation signal after pressing the configuration button
On	Blinking slowly	DP_CFG_CHECK	Check of the configuration by the higher-level CPU (not currently used)
		SET_SLAVES_TO_OP	Switch SmartWire stack to Operational
		SET_SLAVES_TO_PREOP	Switch SmartWire stack to Preoperational
		INIT_LIN_SCHED	Initialization for RUN_LIN_SCHED
		RUN_LIN_SCHED (PREOP)	RUN SmartWire scheduler (in PREOP)
On	On	RUN_LIN_SCHED (OP)	RUN SmartWire scheduler (in OP)
Blinking slowly	On	IDLE_STATE	RUN SmartWire without scheduler (no slaves connected)
Blinking quickly	On	LIN_ERROR_STATE	ERROR LIN-BUS has occurred (remains)

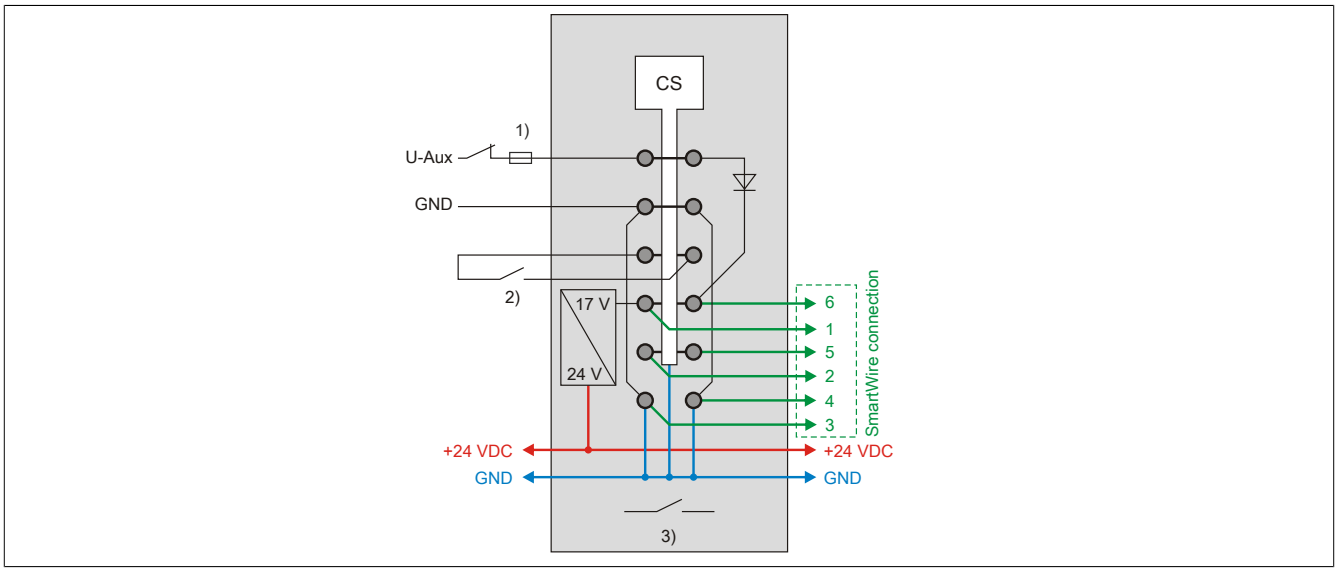
7.1.3.3 Pinout



Information:

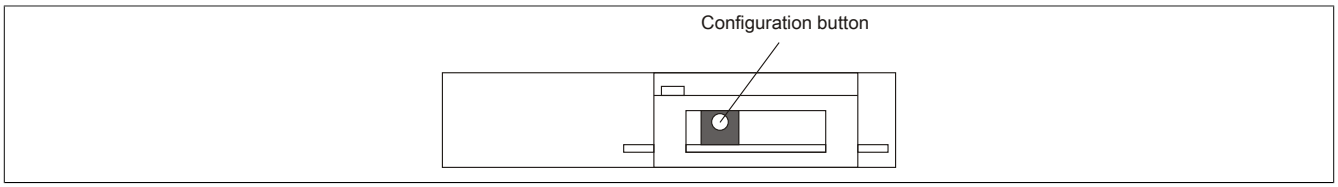
X20CA4S00.00xx SmartWire cables are delivered with the X20TB12 terminal block fully installed.

7.1.3.4 Connection example



1) Fuse, 10 A slow-blow
2) External configuration pushbutton
3) Internal configuration button

7.1.3.5 Configuration button



A configuration button is integrated on the underside of the interface module housing. It can be used to completely configure the entire system.

After adding or removing SmartWire sensors/actuators, pressing the configuration button rescans the SmartWire bus and saves the new configuration in the X20 SmartWire interface module.

In addition to the internal configuration button, it is also possible to connect an external configuration button to the terminal block.

7.1.4 Register description

7.1.4.1 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.

7.1.4.2 Function model 0 - Standard

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Configuration						
5121	FastOutput01_02	USINT			•	
5123	FastOutput03_04	USINT			•	
...	...					
5133	FastOutput13_14	USINT			•	
5135	FastOutput15_16	USINT			•	
257	SmartWireEnable	USINT				•
259	Smart WireMode	USINT				•
8193 + (N-1) * 32	VendorNCfg (Index N = 1 to 16)	USINT				•
8195 + (N-1) * 32	DeviceNCfg (Index N = 1 to 16)	USINT				•
Communication						
557	MasterOperatingState	USINT	•			
550	MasterStatus	UINT	•			
546	SlaveStatus	UINT	•			
4097 + (N-1) * 32	InputN (Index N = 01 to 16)	USINT	•			
513 + (N-1) * 2	SlaveStatusN (Index N = 01 to 16)	USINT		•		
8193 + (N-1) * 32	VendorN (Index N = 1 to 16)	USINT		•		
8195 + (N-1) * 32	DeviceN (Index N = 1 to 16)	USINT		•		

7.1.4.3 Function model 254 - Bus controller

Register	Offset[1)]	Name	Data type	Read		Write	
				Cyclic	Acyclic	Cyclic	Acyclic
Configuration							
5121	0	FastOutput01_02	USINT			•	
5123	1	FastOutput03_04	USINT			•	
...	...	...					
5133	6	FastOutput13_14	USINT			•	
5135	7	FastOutput15_16	USINT			•	
257	-	SmartWireEnable	USINT				•
259	-	Smart WireMode	USINT				•
8193 + (N-1) * 32	-	VendorNCfg (Index N = 1 to 16)	USINT				•
8195 + (N-1) * 32	-	DeviceNCfg (Index N = 1 to 16)	USINT				•
Communication							
77	-	MasterOperatingState	USINT		•		
70	-	MasterStatus	UINT		•		
66	-	SlaveStatus	UINT		•		
4097 + (N-1) * 32	N - 1	InputN (Index N = 01 to 16)	USINT	•			
513 + (N-1) * 2	-	SlaveStatusN (Index N = 01 to 16)	USINT		•		
8193 + (N-1) * 32	-	VendorN (Index N = 1 to 16)	USINT		•		
8195 + (N-1) * 32	-	DeviceN (Index N = 1 to 16)	USINT		•		

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

7.1.4.3.1 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).

7.1.4.3.2 CAN I/O bus controller

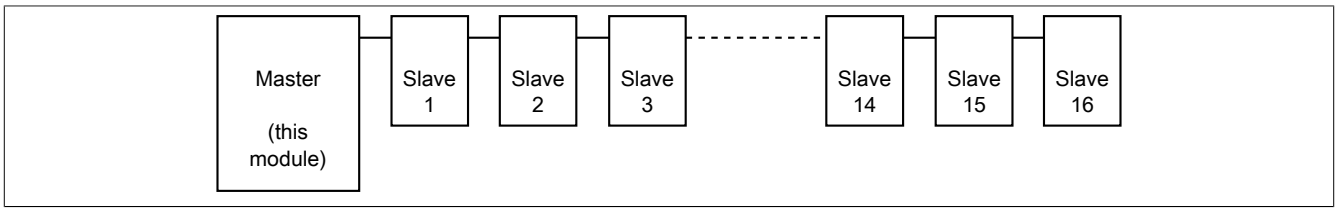
The module occupies 2 analog logical slots on CAN I/O.

7.1.4.4 Communication module Basic Master for SmartWire

SmartWire is essentially a master-slave system.

- All data traffic is initiated by the master, but the system can only contain one master.
- The SmartWire master can control up to 16 SmartWire slaves.
- The total scheduling time is 160 ms (i.e. after 160 ms, all 16 slaves have been queried one time).
- The maximum allowed bus extension is 2.6 m.
- Due to the automatic bus configuration, the numbering of the individual slaves is determined by the line structure of the bus.

This results in the following order:



Node address = Physical position in the bus line

7.1.4.5 Functions

7.1.4.5.1 Scan SmartWire

Automatically started and run after the system is turned on (default settings).

This procedure terminates if

- the set and actual configuration of the bus are identical: system changes to normal operation (i.e cyclic data exchange)
- or if there is a deviation between the set and actual configuration: error present, cyclic data transfer is not started

7.1.4.5.2 Setup SmartWire

Can be activated by pressing the configuration button or using a software command:

- if no configuration is saved
- if a SmartWire scan was just terminated due to error

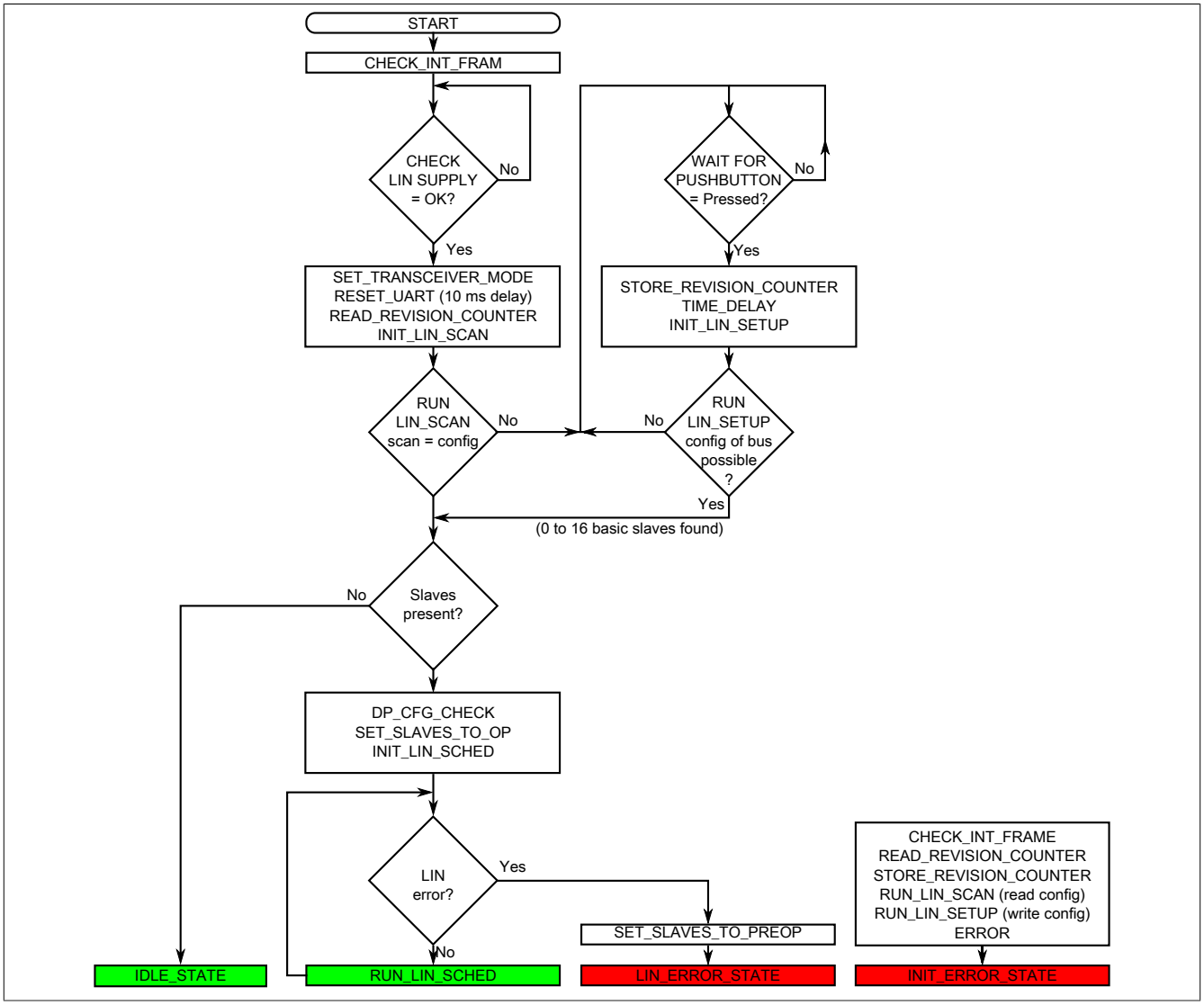
During setup, all connected stations are saved remanently in the master as new set configuration. Valid stations are uniquely identified by the two parameters "Device ID" on page 259 and "Vendor ID" on page 259.

7.1.4.6 Show operating state of the master

The current state of the master state machine is indicated in this register.

Data type	Value	Code	Description
USINT	1	CHECK_INT_FRAM	Init State
	2	CHECK_LIN_SUPPLY	Waiting for 17 V voltage OK
	3	SET_TRANSCEIVER_MODE	Turn on transceiver
	4	RESET_UART UART	Reset
	6	INIT_LIN_SCAN	Init before bus scan
	7	RUN_LIN_SCAN	Bus scan is running
	8	WAIT_FOR_PUSHBUTTON	Scan != Configuration, waiting for Config button
	9	TIME_DELAY	Delay before bus setup
	10	INIT_LIN_SETUP	Init before bus setup
	11	RUN_LIN_SETUP	Bus setup is running (new configuration)
	12	DP_CFG_CHECK	PLC has set "Wait for configuration"
	15	SET_SLAVES_TO_OP	Sets slaves to OP mode (after successful scan or setup)
	16	SET_SLAVES_TO_PREOP	Sets slaves to PREOP mode (after errors have occurred, before LIN_ERROR or INT_ERROR)
	19	INIT_LIN_SCHED	Init bus scheduling
	20	RUN_LIN_SCHED	Bus scheduler is running
	21	LIN_ERROR_STATE	A fatal bus error has occurred (permanent)
	22	INT_ERROR_STATE	A fatal internal error has occurred (permanent)
	23	IDLE_STATE	Idle because there is no slave connected (permanent)

7.1.4.6.1 Flow chart of SmartWire master operating status



The register receives the following value after successfully starting:

Value	Code	Description
20	RUN_LIN_SCHED	Bus scheduler is running

7.1.4.7 Status of the master

Name:
MasterStatus

The current status information for the master is shown in this register.

Data type	Values
UINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	LIN_BUS_SETUP_COMPLETE	0	Saved configuration does not match the actual hardware on the bus
		1	Setup finished: SCAN or SETUP after config button is valid
1	LIN_FATAL_ERROR	0	No error on the bus
		1	SmartWire bus is defective: e.g. short circuit, no echo → more than 10 consecutive communication errors have occurred. **Information:** **A failure of a slave is not a communication error. Instead, it is indicated in the slave status and the scheduler continues to run!**
2	LIN_MASTER_PREOP	0	SmartWire stack not in PREOP mode
		1	SmartWire stack in PREOP mode
3	LIN_MASTER_OP	0	SmartWire stack not in OP mode
		1	SmartWire stack in OP mode
4	LIN_GLOBAL_CONTROL	0	No command sent
		1	Set SmartWire stack to OP mode: The bit is written to the enable bit and can be read back here.
5	Reserved	0	
6	LIN_POWER_SUPPLY_STATE	0	The bus voltage supply is not OK.
		1	Bus voltage supply is OK
7	Reserved	0	
8	DP_CHECK_COMPLETED	0	Not a valid configuration
		1	Configuration check completed (Not Used) (could optionally be written by the PLC, if the SCAN (configuration) is OK and was able to be read back from here)
9	Reserved	0	
10	DP_RECONFIGURATION	0	X2X Reconfiguration → X2X configuration button not pressed
		1	X2X Reconfiguration → X2X configuration button can be read back
11 - 15	Reserved	0	

The register receives the following value after successfully starting:

Equal to the decimal value: 345

Bit	Description	Value	Information
0	LIN_BUS_SETUP_COMPLETE	1	SmartWire setup complete: SCAN or SETUP after config button is valid
3	LIN_MASTER_OP	1	SmartWire stack in OP mode
4	LIN_GLOBAL_CONTROL	1	Set SmartWire stack to OP mode - Command set
6	LIN_POWER_SUPPLY_STATE	1	Bus voltage supply is OK
7	DP_CHECK_COMPLETED	1	Configuration is Ok

7.1.4.8 Status of all slaves

Name:
SlaveStatus

The current state of the slave is indicated collectively in this register.

In the event of an error, the failed slaves are indicated in the respective bits, and in the status registers individually set up for the slaves (see "SlaveStatus1 to SlaveStatus16" on page 259).

Data is exchanged cyclically as long as none of these bits are set. If an error does error, then I/O transfer is stopped. The bus can be started again after the error has been corrected or a setup has been performed again (see "Basic application registers "SmartWireEnable" and "SmartWireMode"" on page 257).

Data type	Values
UINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Slave 1	0	Ok
		1	Errors
...		...	
15	Slave 16	0	Ok
		1	Errors

7.1.4.9 Transfer control bits to slaves

Name:
FastOutput01_02 to FastOutput15_16

In these registers, the control bits are transferred to 2 consecutive slaves. Each slave receives 4 control bits, that must be selected from the 8 data bytes depending on the node address (1 to 16). These 4 control bits are assigned fixed values and utilization of the bits by the slave is optional.

All of the slaves evaluate this telegram. It must be sent cyclically by the master so that the slaves can ensure that the master is still functioning without any problems within the monitoring time (lifeguarding time = 400 ms).

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Slave N	0	Digital output 1 reset
		1	Digital output 1 set
...		..	
3	Slave N	0	Digital output 4 reset
		1	Digital output 4 set
4	Slave N + 1	0	Digital output 1 reset
		1	Digital output 1 set
...		..	
7	Slave N + 1	0	Digital output 4 reset
		1	Digital output 4 set

7.1.4.10 Read input data from slave

Name:
Input01 to Input16

Each slave sends its input data and/or its status to the master.

The data volume consumes 1 byte per slave. Each slave has one diagnostics bit, which it sends to the master with the cyclic data. This bit is a message bit if an application error occurs (on the module). It is always located in the highest value bit.

The master can constantly evaluate this bit. The diagnostic bit is set on the slave if the status of the slave is "Error". Slaves that do not have any input data will still send a byte that is used to make their status data available. This is required because the master also monitors the slaves for proper functionality through the receipt of this byte.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Input state - Digital input 1	0 or 1	
...		...	
3	Input state - Digital input 4	0 or 1	
4 - 6	Reserved	0	
7	Error status	0	No error on the slave
		1	Error on the slave

7.1.4.11 Configure the function breakpoints on the master

Name:
SmartWireEnable

This register can be used to configure function breakpoints that may be implemented in master state machine.

Data type	Values	Bus controller default setting
USINT	See the bit structure.	3

Bit structure:

Bit	Description	Value	Information
0	Enable SmartWire stack	0	Disabled
		1	Enabled (bus controller default setting)
1	Set SmartWire stack mode	0	Sets PREOP mode (scheduler is already running, output data will still be output with 0)
		1	Sets to OP mode (bus controller default setting)
2	Reserved	0	
3	Software "Config button"	0	Not actuated (bus controller default setting)
		1	Pressed (necessary so that slaves can also be reconfigured)
4 - 7	Reserved	0	

7.1.4.12 Configure the operating mode of the master

Name:
SmartWireMode

This register can be used to configure the master operating mode.

Data type	Values	Bus controller default setting
USINT	See the bit structure.	1

Bit structure:

Bit	Description	Value	Information
0 - 1	Operating mode	00	CONFIG from RAM (controller)
		01	Read CONFIG from flash memory (bus controller default setting)
		10	Write CONFIG to flash
		11	Reserved
2 - 7	Reserved	0	

7.1.4.13 Basic application registers "SmartWireEnable" and "SmartWireMode"

By default, the SmartWire bus is started automatically and must at least be configured using an external method (e.g. external button or push-button).

If sensors / actuators are added to or removed from the SmartWire bus, then the configuration procedure must be restarted so that the SmartWire bus will be rescanned and the new configuration saved remanently in the master.

These registers can and must be used for special conditions and for acknowledging errors.

The commands with the library are sent asynchronously on the X2X link network. The following is therefore essential for error-free operation of the module.

- Command register "SmartWireMode" on page 256 is written to first. Writing to register "SmartWireEnable" on page 256 is permitted after the function block reports that it is finished.
- The function block status response is checked in the application.
- The specified response from the master status information must arrive in order for the master state machine to function properly

7.1.4.13.1 Starting the bus when Manual Start has been configured

Status information after startup:

Value (decimal)	Register	Information
1	MasterOperatingState	Init State
0	MasterStatus	
0	SlaveStatus	

If Manual start is selected for the bus in the configuration, then the AsIOAccWrite() function from the AsIOAcc library must be used to write to the two registers in the specified order.

Value (decimal)	Register	Information
1	SmartWireMode	Configuration from rem. memory
3	SmartWireEnable	Command for STACK ON / OPERATIONAL

Status information after error-free startup of the bus:

Value (decimal)	Register	Information
20	MasterOperatingState	"RUN without error if SlaveStatus = 0"
345	MasterStatus	"RUN without error if SlaveStatus = 0"
0	SlaveStatus	No slave errors

7.1.4.13.2 Starting the bus after slave error

Status information after slave error

In this case, a change to MasterOperatingState and MasterStatus cannot be detected at first, although the respective error bits are set in the SlaveStatus. The slaves have failed, and data is no longer being exchanged.

Value (decimal)	Register	Information
20	MasterOperatingState	
345	MasterStatus	
x	SlaveStatus	Bits for the faulty slaves have been set

To set the master to a defined state, the bus must first be stopped with the following write commands.

Value (decimal)	Register	Information
0	SmartWireMode	All off
0	SmartWireEnable	All off

Wait until the commands have been completed successfully, which is indicated in the MasterStatus. Bit 4 is cleared: Response indicating that the bus is no longer operational.

Value (decimal)	Register	Information
20	MasterOperatingState	
329	MasterStatus	
0	SlaveStatus	

The bus can be restarted using the write commands after the errors have been corrected:

Value (decimal)	Register	Information
1	SmartWireMode	Configuration from rem. memory
3	SmartWireEnable	Command for STACK ON / OPERATIONAL

Status information after error-free startup of the bus:

Value (decimal)	Register	Information
20	MasterOperatingState	"RUN without error if SlaveStatus = 0"
345	MasterStatus	"RUN without error if SlaveStatus = 0"
0	SlaveStatus	No slave errors

Different status information can result depending on the present error situation (see "MasterOperatingState" on page 253).

Typical situation when there are hardware configuration differences:

Value (decimal)	Register	Information
0	MasterOperatingState	
80	MasterStatus	
0	SlaveStatus	

Information:

A stop command must be sent before a new start command can be applied!

7.1.4.14 Advanced applications

The following registers are used for advanced diagnostics, for reading back the current configuration and for creating a configuration from the application. The registers that have already been written and their respective contents are, of course, valid.

7.1.4.14.1 Status of the individual slaves

Name:
SlaveStatus1 to SlaveStatus16

These registers display the respective slave status.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0		0	Slave integrated on the bus
		1	Slave failure on the bus
1 - 7	Reserved	0	

7.1.4.14.2 Read slave vendor ID

Name:
Vendor1 to Vendor16

These registers display the respective slave Vendor ID.

Data type	Value	Information
USINT	x	Slave vendor ID

7.1.4.14.3 Read slave device ID

Name:
Device1 to Device16

These registers display the respective slave device ID.

Data type	Value	Information
USINT	x	Slave device ID

7.1.4.14.4 Write slave vendor ID

Name:
Vendor1Cfg to Vendor16Cfg

The desired vendor ID for the slave can be written in these registers.

Data type	Value	Information
USINT	x	Vendor ID of the slave. Bus controller default setting: 1

7.1.4.14.5 Write slave device ID

Name:
Device1Cfg to Device16Cfg

The desired device ID for the slave can be written in these registers.

Data type	Value	Information
USINT	x	Device ID of the slave. Bus controller default setting: 33

7.1.4.14.5.1 Import the configuration without starting the bus

For safety reasons, it is possible to import the configuration for the connected bus without starting cyclic data transfer. This actual configuration can be compared with the set configuration stored in the application. Cyclic data transfer can be started if the configurations are the same. An error is reported if they are not the same.

Manual start is configured, status information after startup:

Value (decimal)	Register	Information
1	MasterOperatingState	Init State
0	MasterStatus	
0	SlaveStatus	

The function AsIOAccWrite() from the library AsIOAcc must be used to write the two registers in the specified order.

Value (decimal)	Register	Information
0	SmartWireMode	RAM memory configuration
9	SmartWireEnable	Command for STACK ON / PREOPERATIONAL and CONFIG-BUTTON

Status information after error-free import of the bus configuration:

Value (decimal)	Register	Information
20	MasterOperatingState	"RUN without error if SlaveStatus = 0"
1349	MasterStatus	"PREOP and no errors"
0	SlaveStatus	No slave errors

Once these commands have been completed, the connected slave modules are imported and stored in the remanent memory for subsequent startups.

The function AsIOAccRead() from the library AsIOAcc must now be used to read all corresponding registers "Vendor1 to Vendor16" on page 259 and "Device1 to Device 16" on page 259 . If the configuration matches, then the bus can now be started using the standard command:

Value (decimal)	Register	Information
1	SmartWireMode	Configuration from rem. memory
3	SmartWireEnable	Command for STACK ON / OPERATIONAL

Status information after error-free startup of the bus:

Value (decimal)	Register	Information
20	MasterOperatingState	"RUN without error if SlaveStatus = 0"
345	MasterStatus	"RUN without error if SlaveStatus = 0"
0	SlaveStatus	No slave errors

7.1.4.14.5.2 Bus configuration settings

Manual start is configured, status information after startup:

Value (decimal)	Register	Information
1	MasterOperatingState	Init State
0	MasterStatus	
0	SlaveStatus	

A running bus can, of course, also be stopped with the standard command and re-configured!

The function AsIOAccWrite() from the library AsIOAcc must now be used to write the respective data to all registers "Vendor1Cfg to Vendor16Cfg" on page 259 and "Device1Cfg to Device16Cfg" on page 259. All vendor and device registers that are not being used must be set to zero. This does not cause a change in the status registers.

To save the data in remanent memory, the function AsIOAccWrite() from the library AsIOAcc must be used to write to the two registers in the specified order.

Value (decimal)	Register	Information
2	SmartWireMode	WRITE rem. memory configuration
1	SmartWireEnable	Command for STACK ON / OPERATIONAL

Status information after error-free configuration:

Value (decimal)	Register	Information
20	MasterOperatingState	"RUN without error if SlaveStatus = 0"
325	MasterStatus	"PREOP without errors"
0	SlaveStatus	No slave errors

The bus can now be started using the standard command for the bus:

Value (decimal)	Register	Information
1	SmartWireMode	Configuration from rem. memory
3	SmartWireEnable	Command for STACK ON / OPERATIONAL

Status information after error-free startup of the bus:

Value (decimal)	Register	Information
20	MasterOperatingState	"RUN without error if SlaveStatus = 0"
345	MasterStatus	"RUN without error if SlaveStatus = 0"
0	SlaveStatus	No slave errors

7.1.4.15 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 µs