

# X67IF1121-1

## 1 General information

Serial interfaces (barcode scanners for example) are often scattered throughout systems. This interface module from the remote X67 system is the optimal choice for this area of application: RS232 and RS485/RS422 connection options directly where they are needed, distributed on the machine or system.

In addition, there are digital inputs and outputs on the same module so that corresponding 24 V sensors/actuators can also be connected.

- RS232 and RS485/RS422 usable in parallel
- 2 digital channels, configurable as inputs or outputs
- 2 digital inputs
- Connection of barcode readers, ID systems and sensors on one module

### 1.1 Other applicable documents

For additional and supplementary information, see the following documents.

#### Other applicable documents

Document name	Title
MAX67	<a href="#">X67 system user's manual</a>
MAEMV	<a href="#">Installation / EMC guide</a>

## 2 Order data


Order number	Short description	Figure
	<b>Communication modules</b>	
X67IF1121-1	X67 interface module, 1 RS232 interface, 1 RS422/485 interface, 2 digital channels configurable as inputs or outputs, 24 VDC, 0.5 A, configurable input filter, 2 inputs, 24 VDC, sink, configurable input filter	

Table 1: X67IF1121-1 - Order data

Required accessories
For a general overview, see section "Accessories - General overview" of the X67 system user's manual.

### 3 Technical description

#### 3.1 Technical data

Order number	X67IF1121-1
Short description	
Communication module	1x RS232 or 1x RS485/RS422, 2 digital inputs, 2 digital channels configurable as inputs or outputs using software
General information	
Insulation voltage between channel and bus	500 V <sub>eff</sub>
B&R ID code	0xA90F
Sensor/Actuator power supply	0.5 A summation current
Status indicators	RS232, RS485/RS422, I/O function per channel, supply voltage, bus function
Diagnostics	
Outputs	Yes, using LED status indicator and software
I/O power supply	Yes, using LED status indicator and software
RS232	Yes, using LED status indicator
RS485/RS422	Yes, using LED status indicator
Connection type	
X2X Link	M12, B-coded
Interfaces and inputs/outputs	4x M12, A-coded
I/O power supply	M8, 4-pin
Power consumption	
Internal I/O	2.4 W
X2X Link power supply	0.75 W
Certifications	
CE	Yes
ATEX	Zone 2, II 3G Ex nA IIA T5 Gc IP67, Ta = 0 - Max. 60°C TÜV 05 ATEX 7201X
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	RS232
Max. distance	900 m
Transfer rate	Max. 115.2 kbit/s
FIFO buffer	Software
Handshake lines	No
Interface IF2	
Signal	RS485/RS422
Max. distance	1200 m
Transfer rate	Max. 115.2 kbit/s
FIFO buffer	Software
Terminating resistor	Integrated in module
I/O power supply	
Nominal voltage	24 VDC
Voltage range	18 to 30 VDC
Integrated protection	Reverse polarity protection
Power consumption	
Sensor/Actuator power supply	Max. 12 W <sup>1)</sup>
Sensor/Actuator power supply	
Voltage	I/O power supply minus voltage drop for short-circuit protection
Voltage drop for short-circuit protection at 0.5 A	Max. 2 VDC
Summation current	Max. 0.5 A
Short-circuit proof	Yes
Digital inputs	
Quantity	Up to 4 if the 2 digital channels are used as digital inputs
Nominal voltage	24 VDC
Input characteristics per EN 61131-2	Type 1
Input voltage	18 to 30 VDC
Input current at 24 VDC	Typ. 3.5 mA
Input circuit	Sink
Input filter	
Hardware	≤100 µs
Software	Default 0 ms, configurable between 0 and 25 ms in 0.2 ms intervals
Input resistance	Typ. 6.67 kΩ


Table 2: X67IF1121-1 - Technical data

Order number	X67IF1121-1
Switching threshold	
Low	<5 V
High	>15 VDC
<b>Digital outputs</b>	
Quantity	Up to 2 if the 2 digital channels are used as digital outputs
Variant	Current-sourcing FET
Nominal voltage	24 VDC
Switching voltage	I/O power supply minus residual voltage
Nominal output current	0.5 A
Total nominal current	1 A
Output circuit	Source
Output protection	Thermal shutdown in the event of overcurrent or short circuit, integrated protection for switching inductive loads, reverse polarity protection of the output power supply
Diagnostic status	Output monitoring with 10 ms delay
Leakage current when the output is switched off	<120 µA
Peak short-circuit current	<2.1 A
Switch-on in the event of overload shutdown or short-circuit shutdown	Approx. 1 ms (depends on the module temperature)
Switching delay	
0 → 1	<100 µs
1 → 0	<150 µs
Switching frequency	
Resistive load	Max. 1000 Hz
Inductive load	See section "Switching inductive loads".
Braking voltage when switching off inductive loads	Typ. 68 VDC
<b>Electrical properties</b>	
Electrical isolation	Bus isolated from channel and IF Channel not isolated from channel and IF
<b>Operating conditions</b>	
Mounting orientation	
Any	Yes
Installation elevation above sea level	
0 to 2000 m	No limitation
>2000 m	Reduction of ambient temperature by 0.5°C per 100 m
Degree of protection per EN 60529	IP67
<b>Ambient conditions</b>	
Temperature	
Operation	-25 to 60°C
Derating	-
Storage	-40 to 85°C
Transport	-40 to 85°C
<b>Mechanical properties</b>	
Dimensions	
Width	53 mm
Height	85 mm
Depth	42 mm
Weight	195 g
Torque for connections	
M8	Max. 0.4 Nm
M12	Max. 0.6 Nm

Table 2: X67IF1121-1 - Technical data

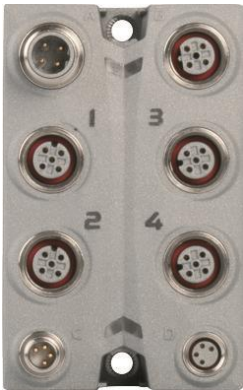
- 1) The power consumption of the sensors and actuators connected to the module is not permitted to exceed 12 W.

### 3.2 LED status indicators

Figure	LED	Description
 <p>Status indicator 1: Left: Green, Right: Red</p> <p>Status indicator 2: Left: Green, Right: Red</p>	Status indicator 1	Status indicator - X2X Link.
	Green	Red
	Off	Off
	On	Off
	Off	On
	On	On
	1	Orange
	3	Orange
	2	Status indicator for input/output 3 and digital input 1.
	LED	Description
	Orange	Input/output status of channel 3
	Green	Input status of channel 1
	Orange and green <sup>1)</sup>	Both channel 1 and channel 3 are active.
	4	Status indicator for input/output 4 and digital input 2.
	LED	Description
	Orange	Input/output status of channel 4
	Green	Input status of channel 2
	Orange and green <sup>1)</sup>	Both channel 2 and channel 4 are active.
	Status indicator 2	Status indicator for module function.
	LED	Status
	Green	Off
	Green	Single flash
	Green	Blinking
	Green	On
	Red	Off
	Red	On
	Red	Single flash
	Red	Double flash

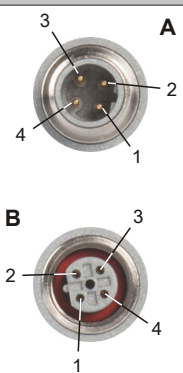
1) Both LEDs are lit, but a mixed color is seen because there is only one light conductor.

### 3.3 Connection elements

	X2X Link Connection A: Input Connection B: Output
	Connection 1: RS232 Connection 3: RS485/RS422
	Connection 2: Input 1 and input/output 3 Connection 4: Input 2 and input/output 4
	24 VDC I/O power supply Connection C: Supply Connection D: Routing

### 3.4 X2X Link

This module is connected to X2X Link using pre-assembled cables. The connection is made using M12 circular connectors.

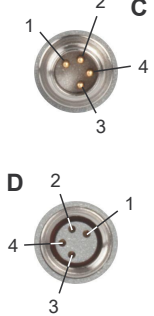
Connection	Pinout
	Pin
	Name
	1
	2
	3
	4
	Shield connection made via threaded insert in the module.
	A → B-coded (male), input
	B → B-coded (female), output

### 3.5 24 VDC I/O power supply

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

#### Information:

**The maximum permissible current for the I/O power supply is 8 A (4 A per connection pin)!**

Connection	Pinout	
	Pin	Name
	1	24 VDC
	2	24 VDC
	3	GND
	4	GND
	C → Connector (male) in module, feed for I/O power supply D → Connector (female) in module, relay of I/O power supply	

### 3.6 Pinout

X1  
M12 ①

Shield	
1	Reserved
2	TxD
3	GND
4	RxD
5	Shield

X2 and X4  
M12 ①

Shield	
1	+24 VDC
2	AI-1
3	GND
4	AI-2
5	Shield

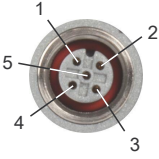
X3  
M12 ①

Shield	
1	TxD\
2	TxD
3	RxD\
4	RxD
5	GND

- ① X67CA0A41.xxxx: M12 sensor cable, straight  
 X67CA0A51.xxxx: M12 sensor cable, angled

#### 3.6.1 RS232 interface

The input is connected using M12 circular connectors.

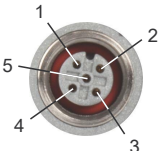
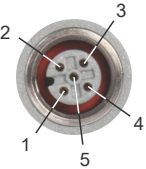
Connection	Pinout	
	Pin	Name
	1	Reserved
	2	TxD
	3	GND
	4	RxD
	5	Shield
	Shield connection made via threaded insert in the module X1 → A-keyed (female), output	

3.6.2 RS485/RS422 interface

The input is connected using M12 circular connectors.

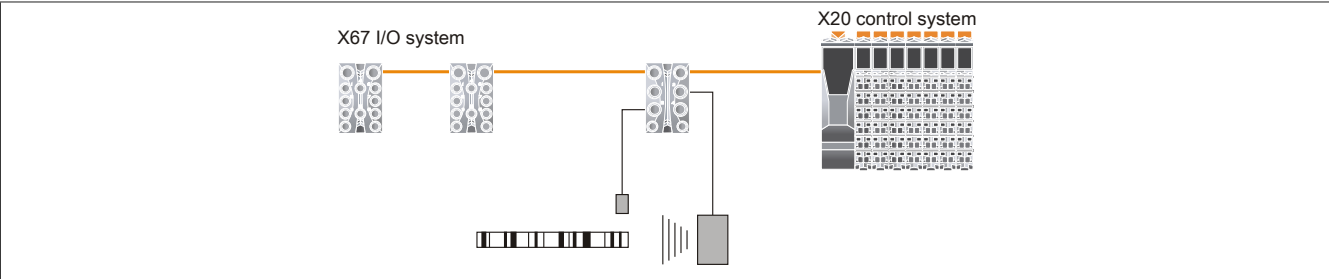
Connection	Pinout		
<div>X3</div>	Pin	RS422	RS485
	1	TxD\	Reserved
	2	TxD	Reserved
	3	RxD\	Data\
	4	RxD	Data
	5	GND	GND
Shield connection made via threaded insert in the module			
X3 → A-keyed (female), output			

3.6.3 Connections X2 and X4

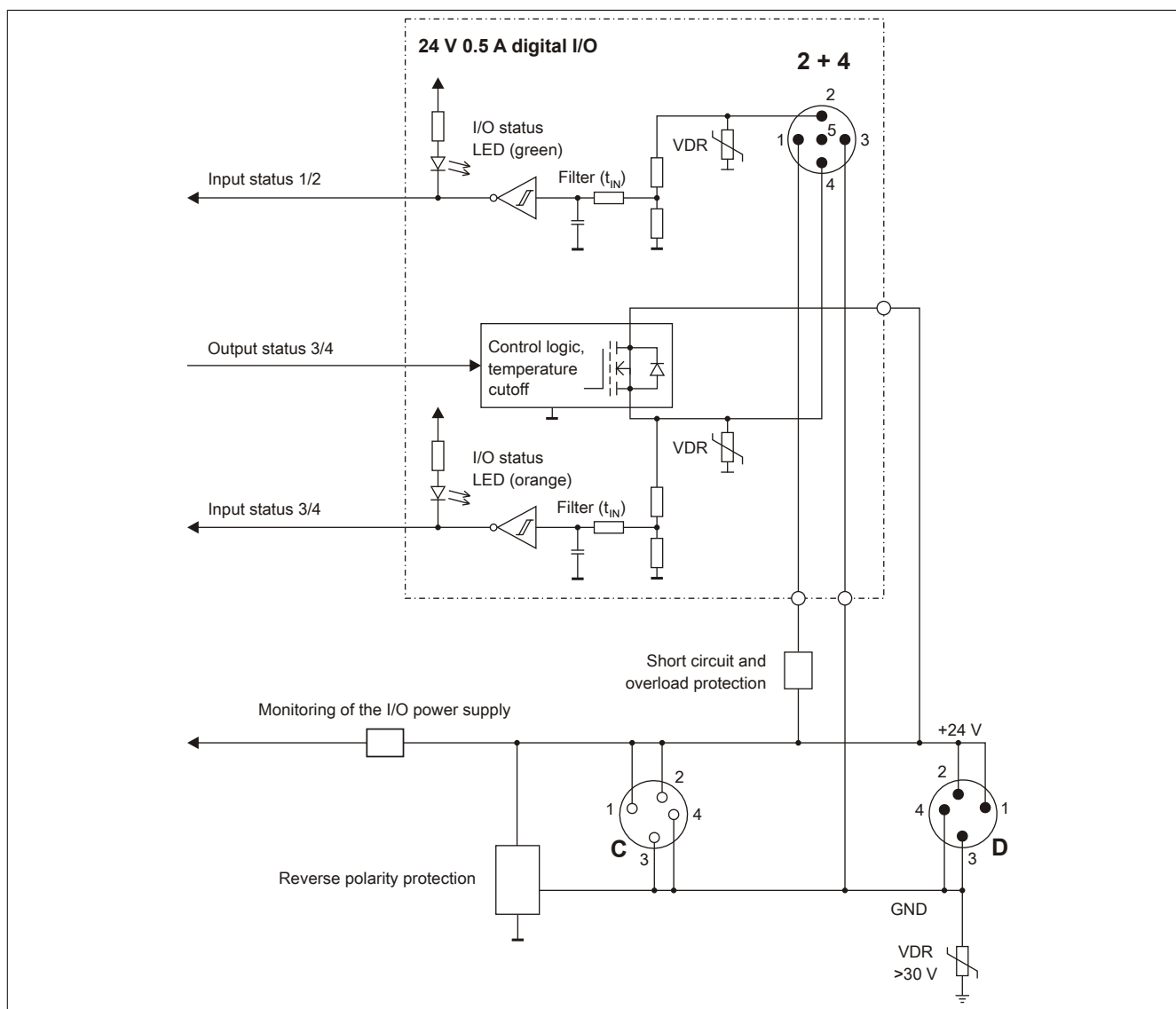
M12, 5-pin	Pinout	
Connection 2	Pin	Name
	1	24 VDC sensor/actuator supply <sup>1)</sup>
	2	DI 1 or 2
	3	GND
	4	DI/DO 3 or 4
	5	Shield <sup>2)</sup>
	1) Sensors/Actuators are not permitted to be supplied externally.	
	2) Shielding also provided by threaded insert in the module.	
Connection 4	X2 → A-keyed (female), input/output	

3.7 Usage example

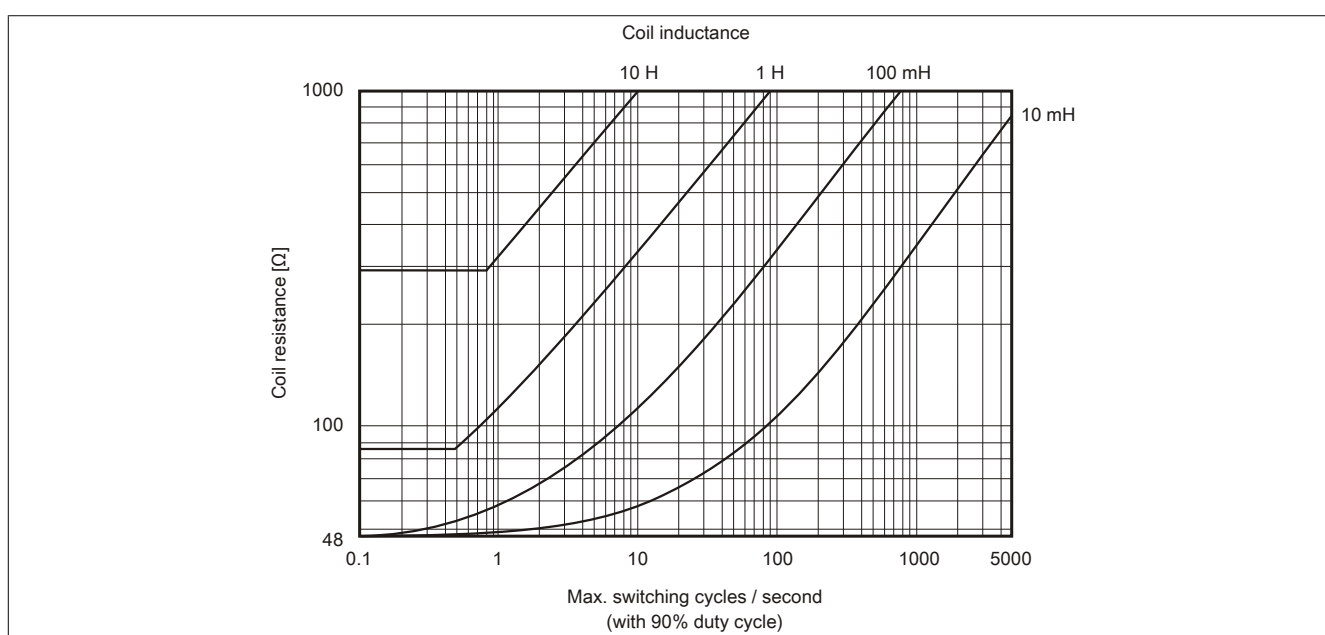
In the application displayed here, the proximity sensor and barcode reader are connected with the communication module. The sensor activates the barcode scanner when a corresponding product arrives in the read area.



### 3.8 Input/Output circuit diagram



### 3.9 Switching inductive loads



## 4 Register description

### 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 X67 system user's manual.

### 4.2 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	-				
Configuration						
1294	InputFilter	UINT				•
1281	OutputEnable	USINT				•
6273	CfO_ErrorID0007	USINT				•
Communication						
135	Input state of digital inputs 1 to 4	USINT	•			
	DigitalInput01	Bit 0				
	...	...				
	DigitalInput04	Bit 3				
129	Output status of the digital outputs	USINT			•	
	DigitalOutput03	Bit 2				
	DigitalOutput04	Bit 3				
133	Status of the digital outputs	USINT	•			
	StatusDigitalOutput03	Bit 2				
	StatusDigitalOutput04	Bit 3				
137	Status of the operating limits	USINT		•		
	StatusSupplyVoltage	Bit 0				



### 4.3 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 21).

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
Configuration - I/O and status						
1294	<a href="#">InputFilter</a>	UINT				•
1281	<a href="#">OutputEnable</a>	USINT				•
6273	<a href="#">CfO_ErrorID0007</a>	USINT				•
Configuration - Interface						
260	<a href="#">IF1CfgPhy</a>	UDINT				•
772	<a href="#">IF2CfgPhy</a>	UDINT				•
268	<a href="#">IF1phyBaud</a>	UDINT				•
780	<a href="#">IF2phyBaud</a>	UDINT				•
Configuration - Handshake						
284	<a href="#">IF1hshCfg</a>	UDINT				•
796	<a href="#">IF2hshCfg</a>	UDINT				•
292	<a href="#">IF1hssXOnOff</a>	UDINT				•
804	<a href="#">IF2hssXOnOff</a>	UDINT				•
298	<a href="#">IF1hssPeriod</a>	UINT				•
810	<a href="#">IF2hssPeriod</a>	UINT				•
274	<a href="#">IF1hshInvTxF</a>	UINT				•
786	<a href="#">IF2hshInvTxF</a>	UINT				•
324	<a href="#">IF1rxLockUnlock</a>	UDINT				•
836	<a href="#">IF2rxLockUnlock</a>	UDINT				•
Configuration - Frame						
332	<a href="#">IF1rxCtoEomSize</a>	UDINT				•
844	<a href="#">IF2rxCtoEomSize</a>	UDINT				•
364	<a href="#">IF1txCtoEomSize</a>	UDINT				•
876	<a href="#">IF2txCtoEomSize</a>	UDINT				•
340	<a href="#">IF1rxEomChar01</a>	UDINT				•
852	<a href="#">IF2rxEomChar01</a>	UDINT				•
348	<a href="#">IF1rxEomChar23</a>	UDINT				•
860	<a href="#">IF2rxEomChar23</a>	UDINT				•
372	<a href="#">IF1txEomChar01</a>	UDINT				•
884	<a href="#">IF2txEomChar01</a>	UDINT				•
380	<a href="#">IF1txEomChar23</a>	UDINT				•
892	<a href="#">IF2txEomChar23</a>	UDINT				•
Communication						
135	<a href="#">Input state of digital inputs 1 to 4</a>	USINT	•			
	<a href="#">DigitalInput01</a>	Bit 0				
	...	...				
	<a href="#">DigitalInput04</a>	Bit 3				
129	<a href="#">Output status of the digital outputs</a>	USINT			•	
	<a href="#">DigitalOutput03</a>	Bit 2				
	<a href="#">DigitalOutput04</a>	Bit 3				
6145	<a href="#">Error message status bits</a>	USINT	•			
	<a href="#">IF1StartBitError</a>	Bit 0				
	<a href="#">IF1StopBitError</a>	Bit 1				
	<a href="#">IF1ParityError</a>	Bit 2				
	<a href="#">IF1RXoverrun</a>	Bit 3				
	<a href="#">IF2StartBitError</a>	Bit 4				
	<a href="#">IF2StopBitError</a>	Bit 5				
	<a href="#">IF2ParityError</a>	Bit 6				
	<a href="#">IF2RXoverrun</a>	Bit 7				
6209	<a href="#">Acknowledging the status bits</a>	USINT			•	
	<a href="#">IF1QuitStartBitError</a>	Bit 0				
	<a href="#">IF1QuitStopBitError</a>	Bit 1				
	<a href="#">IF1QuitParityError</a>	Bit 2				
	<a href="#">IF1QuitRXoverrun</a>	Bit 3				
	<a href="#">IF2QuitStartBitError</a>	Bit 4				
	<a href="#">IF2QuitStopBitError</a>	Bit 5				
	<a href="#">IF2QuitParityError</a>	Bit 6				
	<a href="#">IF2QuitRXoverrun</a>	Bit 7				
133	<a href="#">Status of the digital outputs</a>	USINT	•			
	<a href="#">StatusDigitalOutput03</a>	Bit 2				
	<a href="#">StatusDigitalOutput04</a>	Bit 3				

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
137	Status of the operating limits	USINT	●			
	StatusSupplyVoltage	Bit 0				
Flatstream <sup>1)</sup>						
196	IF1CfgMTU	UDINT				●
212	IF2CfgMTU	UDINT				●
204	IF1forwardDelay	UINT				●
220	IF2forwardDelay	UINT				●
0	IF1InputSequence	USINT	●			
64	IF2InputSequence	USINT	●			
N	IF1RxByteN (index N = 1 to 27)	USINT	●			
64 + N	IF2RxByteN (index N = 1 to 27)	USINT	●			
32	IF1OutputSequence	USINT			●	
96	IF2OutputSequence	USINT			●	
32 + N	IF1TxByteN (index N = 1 to 27)	USINT			●	
96 + N	IF2TxByteN (index N = 1 to 27)	USINT			●	

1) A separate Flatstream is available for each interface.

## 4.4 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.

### Information:

**It is not possible to change or expand the predefined configuration in this function model!**

Register	Offset <sup>1)</sup>	Name	Data type	Read		Write	
				Cyclic	Acyclic	Cyclic	Acyclic
Configuration - I/O and status							
1294	-	InputFilter	UINT				•
1281	-	OutputEnable	USINT				•
6273	-	CfO_ErrorID0007	USINT				•
Configuration - Interface							
260	-	IF1CfgPhy	UDINT				•
772	-	IF2CfgPhy	UDINT				•
268	-	IF1phyBaud	UDINT				•
780	-	IF2phyBaud	UDINT				•
Configuration - Handshake							
284	-	IF1hshCfg	UDINT				•
796	-	IF2hshCfg	UDINT				•
292	-	IF1hssXOnOff	UDINT				•
804	-	IF2hssXOnOff	UDINT				•
298	-	IF1hssPeriod	UINT				•
810	-	IF2hssPeriod	UINT				•
274	-	IF1hshInvTxF	UINT				•
786	-	IF2hshInvTxF	UINT				•
324	-	IF1rxlLockUnlock	UDINT				•
836	-	IF2rxlLockUnlock	UDINT				•
Configuration - Frame							
332	-	IF1rxCtoEomSize	UDINT				•
844	-	IF2rxCtoEomSize	UDINT				•
364	-	IF1txCtoEomSize	UDINT				•
876	-	IF2txCtoEomSize	UDINT				•
340	-	IF1rxEomChar01	UDINT				•
852	-	IF2rxEomChar01	UDINT				•
348	-	IF1rxEomChar23	UDINT				•
860	-	IF2rxEomChar23	UDINT				•
372	-	IF1txEomChar01	UDINT				•
884	-	IF2txEomChar01	UDINT				•
380	-	IF1txEomChar23	UDINT				•
892	-	IF2txEomChar23	UDINT				•
Communication							
135	18	Input state of digital inputs 1 to 4	USINT	•			
		DigitalInput01	Bit 0				
		...	...				
		DigitalInput04	Bit 3				
129	18	Output status of the digital outputs	USINT			•	
		DigitalOutput03	Bit 2				
		DigitalOutput04	Bit 3				

Register	Offset <sup>1)</sup>	Name	Data type	Read		Write	
				Cyclic	Acyclic	Cyclic	Acyclic
6145	16	Error message status bits	USINT	●			
		IF1StartBitError	Bit 0				
		IF1StopBitError	Bit 1				
		IF1ParityError	Bit 2				
		IF1RXoverrun	Bit 3				
		IF2StartBitError	Bit 4				
		IF2StopBitError	Bit 5				
		IF2ParityError	Bit 6				
		IF2RXoverrun	Bit 7				
6209	16	Acknowledging the status bits	USINT			●	
		IF1QuitStartBitError	Bit 0				
		IF1QuitStopBitError	Bit 1				
		IF1QuitParityError	Bit 2				
		IF1QuitRXoverrun	Bit 3				
		IF2QuitStartBitError	Bit 4				
		IF2QuitStopBitError	Bit 5				
		IF2QuitParityError	Bit 6				
		IF2QuitRXoverrun	Bit 7				
133	19	Status of the digital outputs	USINT	●			
		StatusDigitalOutput03	Bit 2				
		StatusDigitalOutput04	Bit 3				
137	-	Status of the operating limits	USINT	●			
		StatusSupplyVoltage	Bit 0				
Flatstream <sup>2)</sup>							
196	-	IF1CfgMTU	UDINT				●
212	-	IF2CfgMTU	UDINT				●
204	-	IF1forwardDelay	UINT				●
220	-	IF2forwardDelay	UINT				●
0	0	IF1InputSequence	USINT	●			
64	8	IF2InputSequence	USINT	●			
N	N	IF1RxByteN (index N = 1 to 7)	USINT	●			
64 + N	8 + N	IF2RxByteN (index N = 1 to 7)	USINT	●			
32	0	IF1OutputSequence	USINT			●	
96	8	IF2OutputSequence	USINT			●	
32 + N	N	IF1TxByteN (index N = 1 to 7)	USINT			●	
96 + N	8 + N	IF2TxByteN (index N = 1 to 7)	USINT			●	

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

2) A separate Flatstream is available for each interface.

#### 4.4.1 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 X67 user's manual (version 3.30 or later).

#### 4.4.2 CAN I/O bus controller

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

## 4.5 Configuration - I/O and status

### 4.5.1 Configuring the input filter

Name:

InputFilter

The filter value for all digital inputs can be configured in this register.

The filter value can be configured in steps of 100  $\mu$ s. It makes sense to enter values in steps of 2, however, since the input signals are sampled every 200  $\mu$ s.

Data type	Values	Filter
USINT	0	No software filter (bus controller default setting)
	2	0.2 ms
	...	...
	250	25 ms - Higher values are limited to this value.

### 4.5.2 Input/output configuration channels 3 and 4

Name:

OutputEnable

This register configures channels 3 and 4 as either an input or output.

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

Bit structure:

Bit	Name	Value	Information
0 - 1	Reserved	-	
2	Channel 03	0	Configured as input (bus controller default setting)
		1	Configured as output
3	Channel 04	0	Configured as input (bus controller default setting)
		1	Configured as output
4 - 7	Reserved	-	

### 4.5.3 Forward error to the application

Name:

CfO\_ErrorID0007

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

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

Bit structure:

Bit	Name	Value	Information
0	StartBitError - IF1	0	Ignore (bus controller default setting)
		1	Indicate faulty start bit
1	StopBitError - IF1	0	Ignore (bus controller default setting)
		1	Indicate faulty stop bit
2	ParityError - IF1	0	Ignore (bus controller default setting)
		1	Indicate faulty parity bit
3	RXoverflow - IF1	0	Ignore (bus controller default setting)
		1	Indicate overflow in the receive direction
4	StartBitError - IF2	0	Ignore (bus controller default setting)
		1	Indicate faulty start bit
5	StopBitError - IF2	0	Ignore (bus controller default setting)
		1	Indicate faulty stop bit
6	ParityError - IF2	0	Ignore (bus controller default setting)
		1	Indicate faulty parity bit
7	RXoverflow - IF2	0	Ignore (bus controller default setting)
		1	Indicate overflow in the receive direction

## 4.6 Configuration - Serial interface

### 4.6.1 Configuration - Interfaces

Name:

IF1CfgPhy to IF2CfgPhy

These registers are used to configure the interfaces. Only the corresponding interface values are permitted to be used for each register.

- IF1CfgPhy configures RS232 interface
- IF2CfgPhy configures RS422/485 interface

After all other configuration registers have been written, the last write command must enable the interface. If parameters need to be changed, the interface must first be disabled.

Data type	Values	Bus controller default setting
UDINT	See bit structure.	0x80245

Bit structure:

Bit	Description	Value	Information
0 - 7	Parity bit configuration <sup>1)</sup>	48	"0" - (low) bit is always 0
		49	"1" - (high) bit is always 1
		69	1 stop bit (bus controller default setting)
		78	"N" - (no) no bit
		79	"O" - (odd) odd parity
8 - 15	Number of stop bits	2	1 stop bit (bus controller default setting)
		4	2 stop bits
16 - 23	Number of data bits per character	7	7 data bits
		8	8 data bits (bus controller default setting)
24 - 31	Interface mode	0	Interface disabled (bus controller default setting)
		2	RS232 interface active
		4	RS422 interface active <sup>2)</sup>
		5	RS422 interface active as bus <sup>3)</sup>
		6	RS485 interface active with echo
		7	RS485 interface active without echo

1) ASCII-encoded decimal values

2) Connection between 2 stations

3) Connections between multiple stations possible. Transmit lines connected as with RS485 tri-state.

### 4.6.2 Setting the baud rate

Name:

IF1phyBaud to IF2phyBaud

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

## 4.7 Configuration - Handshake

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

### 4.7.1 RTS evaluation and frame detection

Name:

IF1hshCfg to IF2hshCfg

This register configures how the hardware RTS handshake line is controlled depending on the fill level of the receive buffer in addition to generally enabling frame detection on the hardware side.

The RTS line is enabled as long as data is being sent. This Tx-Framing mode can be used to control external interface converters.

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

Bit structure:

Bit	Description	Value	Information
0 - 7	Frame detection	0	RTS line freely available for other flow control methods (bus controller default setting)
		16	Tx frame detection switched on for RTS line
		80	Tx frame detection switched on for RTS line (without echo)
8 - 15	Flow control	0	RTS line freely available for other flow control methods (bus controller default setting)
		16	RTS line controlled by the fill level of the receive buffer
16 - 31	Reserved	0	

### 4.7.2 Software handshake control characters

Name:

IF1hssXOnOff to IF2hssXOnOff

These registers configure the XOn and XOff control characters, which are used for flow control via software handshake. Valid XOn/XOff control characters must be defined in order to ensure proper functionality.

The default values are XOn (17) and XOff (19), but other values can also be used.

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

Bit structure:

Bit	Description	Value	Information
0 - 15	XOff control character	19	Default XOff ASCII character
		65535	No software handshake (bus controller default setting)
16 - 31	XOn control character	17	Default XOn ASCII character
		65535	No software handshake (bus controller default setting)

### 4.7.3 Repeating the handshake

Name:

IF1hssPeriod to IF2hssPeriod

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

Data type	Value	Function
UINT	0	Automatic status repetition disabled
	500 to 10000	Repeat time in ms.
		Bus controller default setting: 5000

#### 4.7.4 Inverting RTS/CTS

Name:

IF1hshInvTxF to IF2hshInvTxF

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

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

Bit structure:

Bit	Name	Value	Information
0 - 7	Mask TX signal	0	No masking (bus controller default setting)
		1	Mask CTS signal
8 - 15	Invert signals	0	Inversion off (bus controller default setting)
		1	CTS signal inversion on
		16	RTS signal inversion on
		17	CTS and RTS signal inversion on

#### 4.7.5 Locking/unlocking the receive buffer

Name:

IF1rxILockUnlock to IF2rxILockUnlock

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	Values	Bus controller default setting
UDINT	See bit structure.	0x4000200

Bit structure:

Bit	Description	Value	Information
0 - 15	Lower limit of the receive buffer	0 to 4095	Bus controller default setting: 512
16 - 31	Upper limit of the receive buffer	0 to 4095	Bus controller default setting: 1024

## 4.8 Configuration - Frame

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

### 4.8.1 Configuring the receive frame

Name:

IF1rxCtoEomSize to IF2rxCtoEomSize

This register is used to configure the maximum number of bytes of the receive frame and the duration until a receive timeout is triggered.

Data type	Values	Bus controller default setting
UDINT	See bit structure.	0x40100

Bit structure:

Bit	Description	Value	Information
0 - 15	Maximum number of bytes of the receive frame	0	Function disabled
		1 to 4096	Configurable receive frame length in characters. Bus controller default setting: 256
16 - 31	Duration until a receive timeout is triggered.	0	Function disabled
		1 to 65535	Receive timeout in characters. Bus controller default setting: 4

#### Maximum number of bytes of 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.

#### Duration until a receive timeout is triggered.

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

### 4.8.2 Configuring the transmit frame

Name:

IF1txCtoEomSize to IF2txCtoEomSize

This register is used to configure the maximum byte number of the transmit frame and the duration until a transmit timeout is triggered.

Data type	Values	Bus controller default setting
UDINT	See bit structure.	0x51000

Bit structure:

Bit	Description	Value	Information
0 - 15	Maximum number of bytes of the transmit frame	0	Function disabled
		1 to 4096	Configurable transmit frame length in characters. Bus controller default setting: 256
16 - 31	Transmit timeout	0	Function disabled
		1 to 65535	Transmit timeout in characters. Bus controller default setting: 5

#### Maximum number of bytes of 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.

#### Transmit timeout

No characters are transmitted for the specified time. 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.



### 4.8.3 Define receive terminator

Name:

IF1rxEomChar01 to IF2rxEomChar01

IF1rxEomChar23 to IF2rxEomChar23

A possible receive termination character can be configured in each register.

All 4 characters are equal. The message is considered to be terminated as soon as one of the defined characters is transferred.

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

Bit structure:

Bit	Description	Value	Information
0 - 15	2. Character: IF1rxEomChar01 4. Character: IF1rxEomChar23	0 to 255	Frame terminator (ASCII code)
		65535	Function disabled (bus controller default setting)
16 - 31	1. Character: IF2rxEomChar01 3. Character: IF2rxEomChar23	0 to 255	Frame terminator (ASCII code)
		65535	Function disabled (bus controller default setting)

### 4.8.4 Define transmit terminator

Name:

IF1txEomChar01 to IF2txEomChar01

IF1txEomChar23 to IF2txEomChar23

A possible transmit termination character can be configured in each register.

All 4 characters are equal. The message is considered to be terminated as soon as one of the defined characters is transferred.

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

Bit structure:

Bit	Description	Value	Information
0 - 15	2. Character: IF1txEomChar01 4. Character: IF1txEomChar23	0 to 255	Frame terminator (ASCII code)
		65535	Function disabled (bus controller default setting)
16 - 31	1. Character: IF2txEomChar01 3. Character: IF2txEomChar23	0 to 255	Frame terminator (ASCII code)
		65535	Function disabled (bus controller default setting)

## 4.9 Configuration - Flatstream MTU

Name:

IF1CfgMTU to IF2CfgMTU

These registers configure the Maximum Transmission Unit settings. For a description, see the corresponding section in ["Flatstream communication" on page 21](#).

Data type	Values
UDINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0 - 7	<a href="#">Number of unacknowledged sequences</a>	1 to 7	Default = 1
8	<a href="#">Flatstream mode</a> (bit 0)	0	Multiple segments not permitted (default)
		1	Multiple segments permitted within MTU
9	<a href="#">Flatstream mode</a> (bit 1)	0	Maximum segment size is MTU size (default)
		1	Segment size can exceed MTU size
10 - 15	Reserved	-	
16 - 23	<a href="#">Number of enabled Tx and Rx bytes</a> (InputMTU size)	1 to 27	Default = 7 <sup>1)</sup>
24 - 31	<a href="#">Number of enabled Tx and Rx bytes</a> (OutputMTU size)	1 to 27	Default = 7 <sup>1)</sup>

1) The size cannot be altered in function model 254 - Bus controller. Fixed length = 7.

## 4.10 Communication

### 4.10.1 Digital inputs

#### Unfiltered

The input state is collected with a fixed offset to the network cycle and transferred in the same cycle.

#### Filtered

The filtered state is collected with a fixed offset to the network cycle and transferred in the same cycle. Filtering takes place asynchronously to the network in multiples of 200 µs with a network-related jitter of up to 50 µs.

#### 4.10.1.1 Input state of digital inputs 1 to 4

Name:

DigitalInput01 to DigitalInput04

This register contains the input state of digital inputs 1 to 4.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0	DigitalInput01	0 or 1	Input status of digital input 1
...	...	...	...
3	DigitalInput04	0 or 1	Input status of digital input 4

### 4.10.2 Digital outputs

The output state is transferred to the output channels with a fixed offset (<60 µs) based on the network cycle (SyncOut).

#### 4.10.2.1 Output status of the digital outputs

Name:

DigitalOutput03 to DigitalOutput04

This register is used to store the switching state of digital outputs 3 to 4.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Name	Value	Information
0 - 1	Reserved	-	
2	DigitalOutput03	0	Digital output 03 reset
		1	Digital output 03 set
3	DigitalOutput04	0	Digital output 04 reset
		1	Digital output 04 set
4 - 7	Reserved	-	

#### 4.10.2.2 Status of the digital outputs

Name:

StatusDigitalOutput03 to StatusDigitalOutput04

This register is used to indicate the status of digital outputs 3 and 4.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0 - 1	Reserved	-	
2	StatusDigitalOutput03	0	Channel 03: No error
		1	Channel 03: Short circuit or overload
3	StatusDigitalOutput04	0	Channel 04: No error
		1	Channel 04: Short circuit or overload
4 - 7	Reserved	-	

### 4.10.3 Error message status bits

Name:

IF1StartBitError to IF2StartBitError

IF1StopBitError to IF2StopBitError

IF1ParityError to IF2ParityError

IF1RXoverrun to IF2RXoverrun

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	IF1StartBitError	0	No error
		1	Start bit error occurred <sup>1)</sup>
1	IF1StopBitError	0	No error
		1	Stop bit error occurred <sup>1)</sup>
2	IF1ParityError	0	No error
		1	Parity bit error occurred <sup>1)</sup>
3	IF1RXoverrun	0	No error
		1	Receive buffer overflow occurred <sup>2)</sup>
4	IF2StartBitError	0	No error
		1	Start bit error occurred <sup>1)</sup>
5	IF2StopBitError	0	No error
		1	Stop bit error occurred <sup>1)</sup>
6	IF2ParityError	0	No error
		1	Parity bit error occurred <sup>1)</sup>
7	IF2RXoverrun	0	No error
		1	Receive buffer overflow occurred <sup>2)</sup>

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.

### 4.10.4 Acknowledging the status bits

Name:

IF1QuitStartBitError to IF2QuitStartBitError

IF1QuitStopBitError to IF2QuitStopBitError

IF1QuitParityError to IF2QuitParityError

IF1QuitRXoverrun to IF2QuitRXoverrun

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	IF1QuitStartBitError	0	No acknowledgment
		1	Acknowledge start bit error
1	IF1QuitStopBitError	0	No acknowledgment
		1	Acknowledge stop bit error
2	IF1QuitParityError	0	No acknowledgment
		1	Acknowledge parity bit error
3	IF1QuitRXoverrun	0	No acknowledgment
		1	Acknowledge receive buffer overflow error
4	IF2QuitStartBitError	0	No acknowledgment
		1	Acknowledge start bit error
5	IF2QuitStopBitError	0	No acknowledgment
		1	Acknowledge stop bit error
6	IF2QuitParityError	0	No acknowledgment
		1	Acknowledge parity bit error
7	IF2QuitRXoverrun	0	No acknowledgment
		1	Acknowledge receive buffer overflow error

#### 4.10.5 Status of the operating limits

Name:

StatusSupplyVoltage

This register can be used to read the status of the operating limits.

Data type	Value
USINT	See bit structure.

Bit structure:

Bit	Description	Value	Information
0	StatusSupplyVoltage	0	I/O power supply within the warning limits (18 to 30 V)
		1	I/O power supply outside the warning limits (<18 V or >30 V)
1 - 7	Reserved	0	

## 4.11 Flatstream communication

### 4.11.1 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.

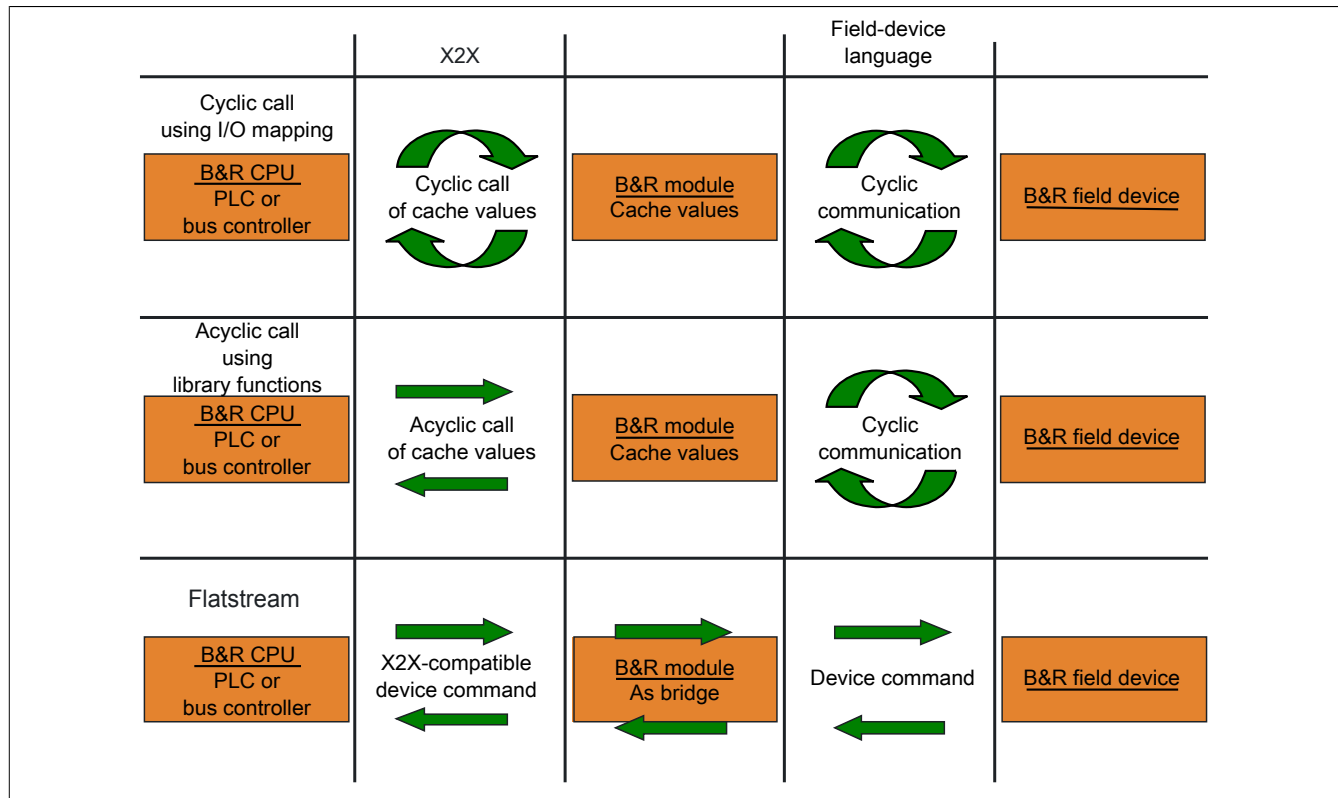


Figure 1: 3 types of communication

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.

#### 4.11.2 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.

### 4.11.3 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.

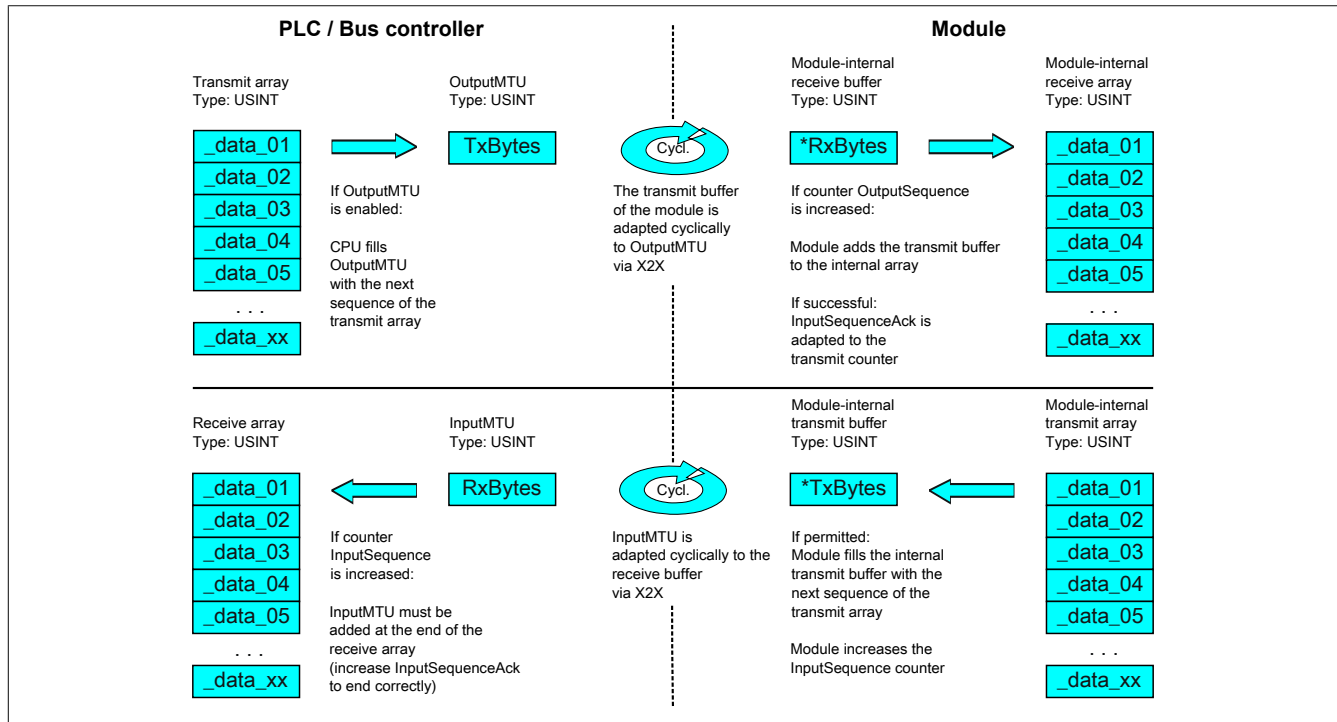


Figure 2: Flatstream communication

#### 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.

#### 4.11.4 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.

##### 4.11.4.1 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.

##### 4.11.4.1.1 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).



#### 4.11.4.2 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.

##### 4.11.4.2.1 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.

##### 4.11.4.2.2 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

##### 4.11.4.2.3 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 multi-segment MTUs, the next control byte is no longer expected in the first Rx byte of the subsequent MTU, but transferred directly after the current segment.

### MessageEndBit

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

#### Information:

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

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

Flatstream formula for calculating message length:

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

#### 4.11.4.2.4 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.

#### 4.11.4.2.5 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.

#### 4.11.4.2.6 Relationship between OutputSequence and InputSequence

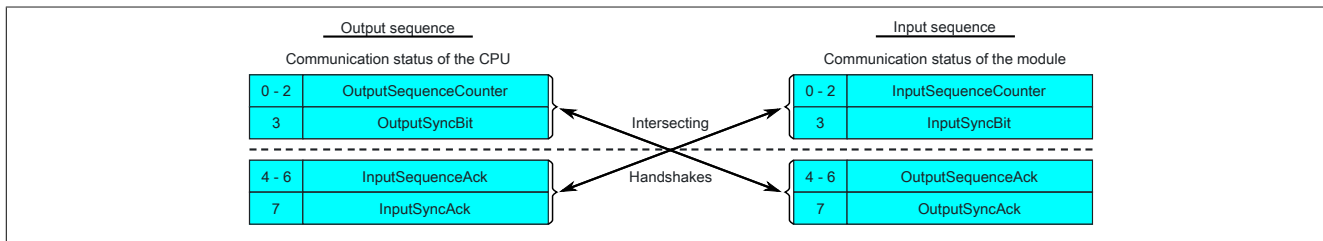


Figure 3: Relationship between OutputSequence and InputSequence

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.**

#### 4.11.4.3 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). <i>The module does not accept the current contents of InputMTU since the channel is not yet synchronized.</i> <i>The module matches OutputSequenceAck and OutputSyncAck to the values of OutputSequenceCounter and OutputSyncBit.</i>
2) If the 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). <i>The module does not accept the current contents of InputMTU since the channel is not yet synchronized.</i> <i>The module matches OutputSequenceAck and OutputSyncAck to the values of OutputSequenceCounter and OutputSyncBit.</i>
3) If the 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).
<b>Note:</b> Theoretically, data can be transferred from this point forward. However, it is still recommended to wait until the output direction is completely synchronized before transferring data.
<i>The module sets OutputSyncAck.</i>
The output direction is synchronized, and the 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

<i>The module writes 000 to InputSequenceCounter and resets InputSyncBit.</i> <i>The module monitors the high nibble of register "OutputSequence" and expects 000 in InputSequenceAck and 0 in InputSyncAck.</i>
1) The 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. <i>If the module registers the expected values in InputSequenceAck and InputSyncAck, it increments InputSequenceCounter.</i> <i>The module monitors the high nibble of register "OutputSequence" and expects 001 in InputSequenceAck and 0 in InputSyncAck.</i>
2) The 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. <i>If the module registers the expected values in InputSequenceAck and InputSyncAck, it sets InputSyncBit.</i> <i>The module monitors the high nibble of register "OutputSequence" and expects 1 in InputSyncAck.</i>
3) The CPU is permitted to set InputSyncAck.
<b>Note:</b> 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.

4.11.4.4 Transmitting and receiving

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

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

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

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.

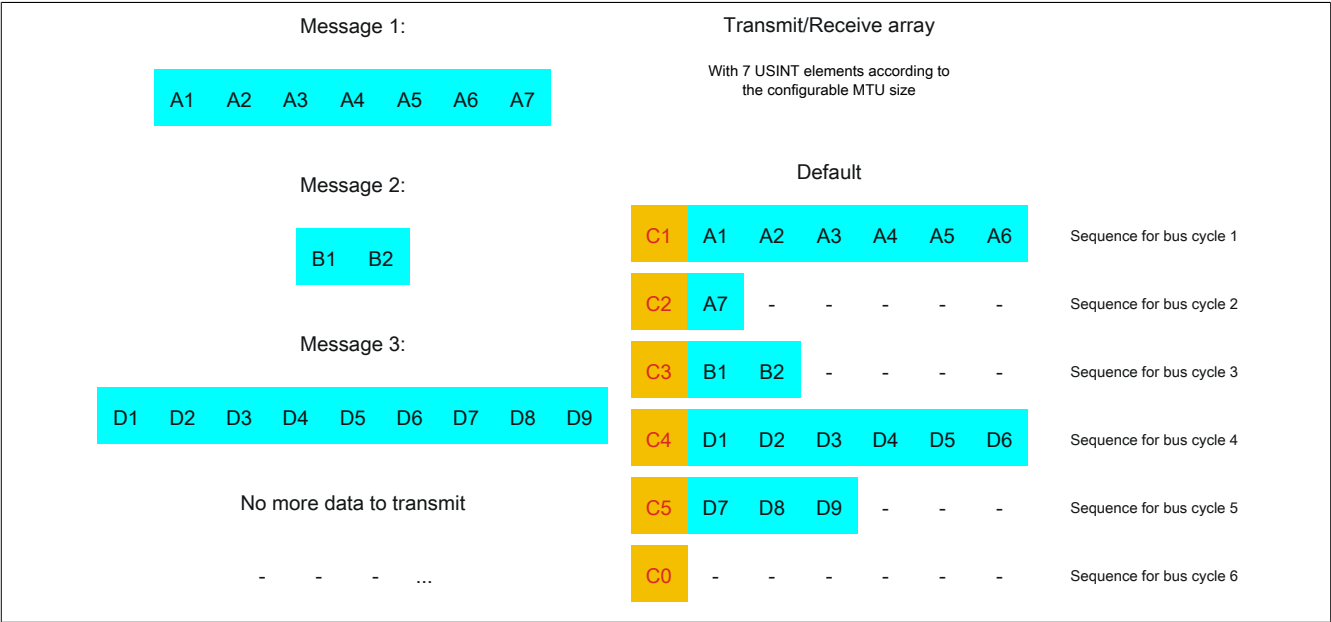


Figure 4: Transmit/Receive array (default)

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

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

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

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

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

Table 3: Flatstream determination of the control bytes for the default configuration example (part 1)

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

Table 4: Flatstream determination of the control bytes for the default configuration example (part 2)

#### 4.11.4.5 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.

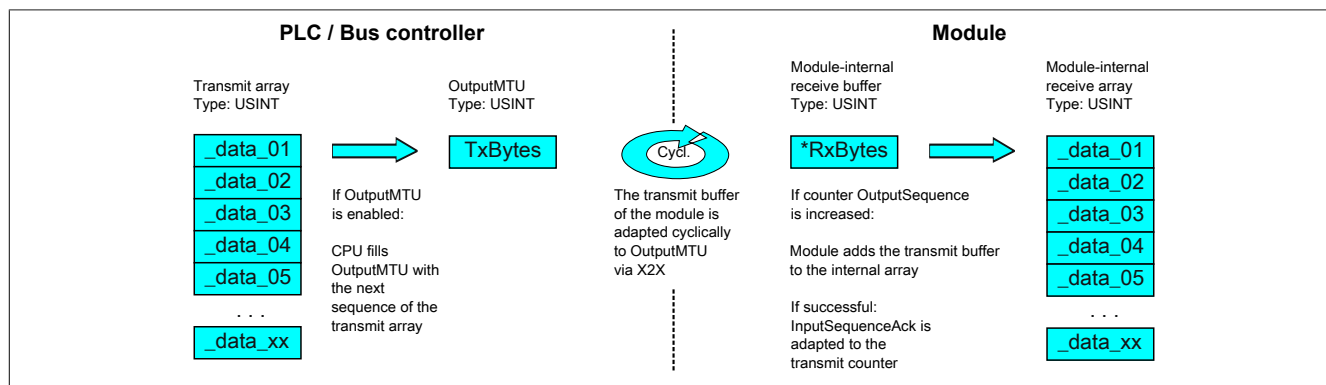


Figure 5: Flatstream communication (output)

### 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

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



## 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

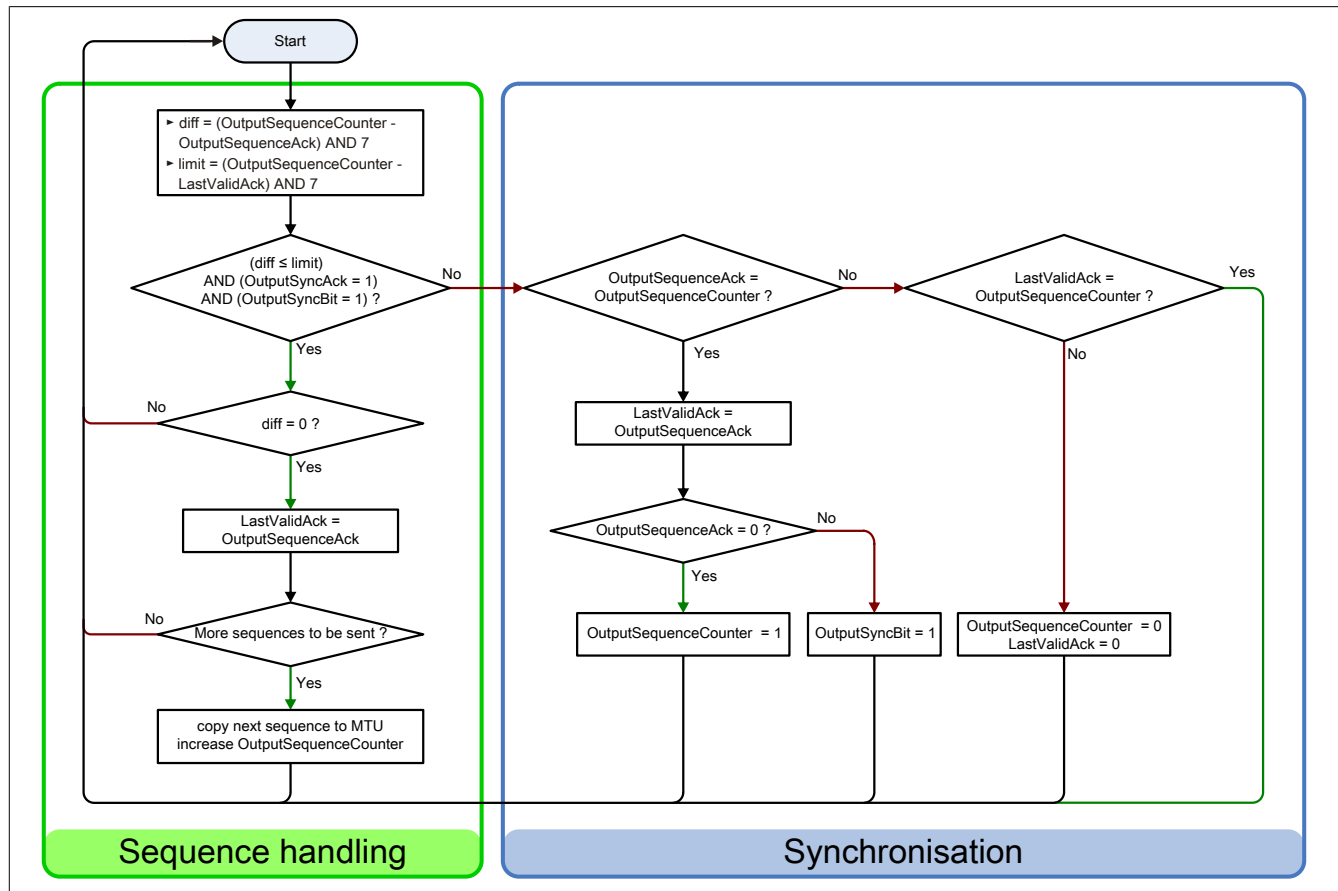


Figure 6: Flowchart for the output direction

#### 4.11.4.6 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.

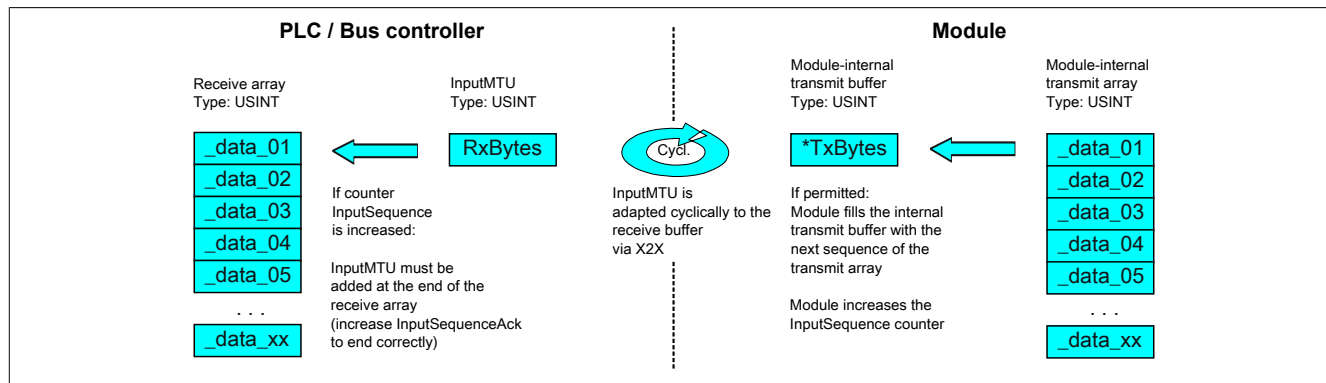


Figure 7: Flatstream communication (input)

#### 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

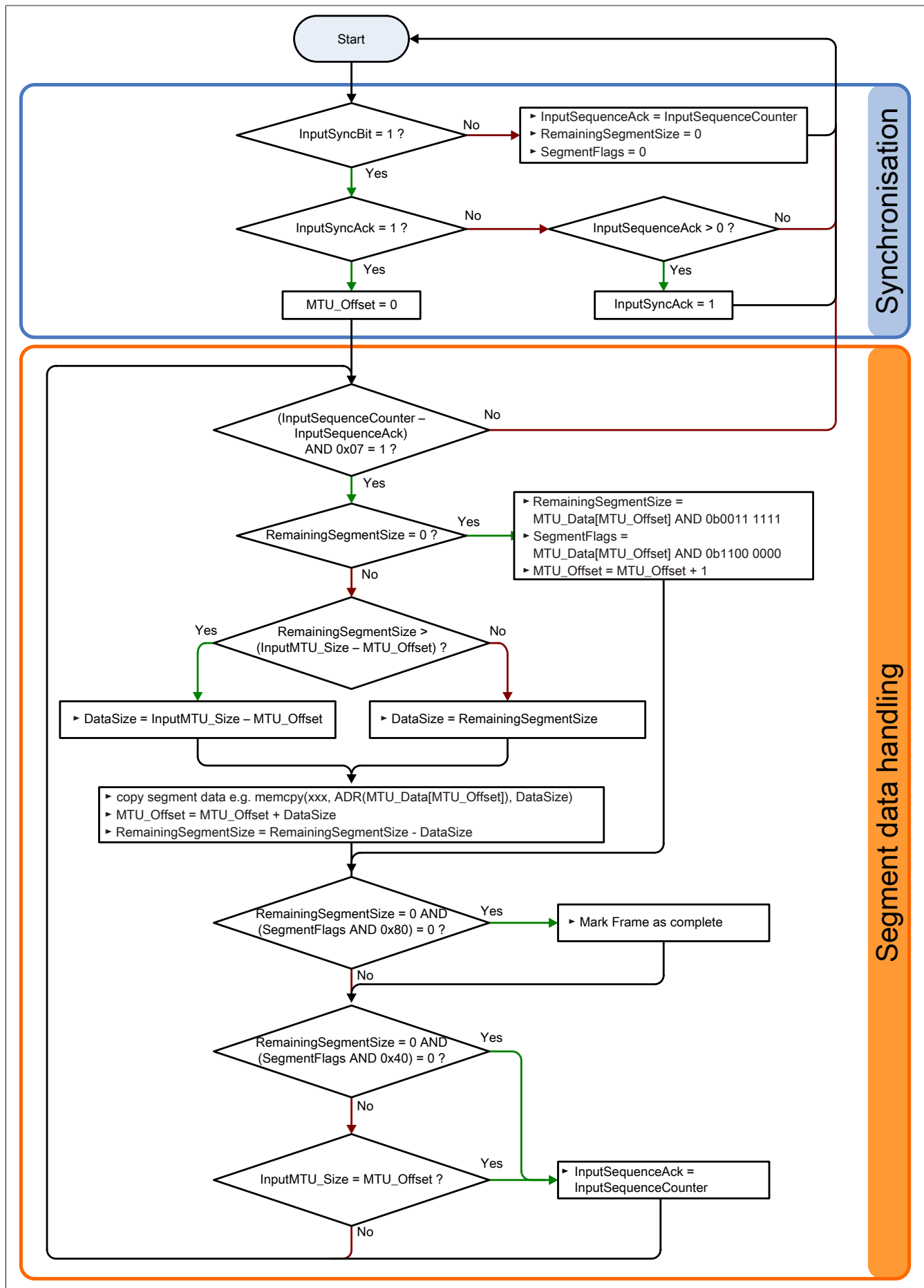


Figure 8: Flowchart for the input direction

#### 4.11.4.7 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.

#### 4.11.4.8 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.

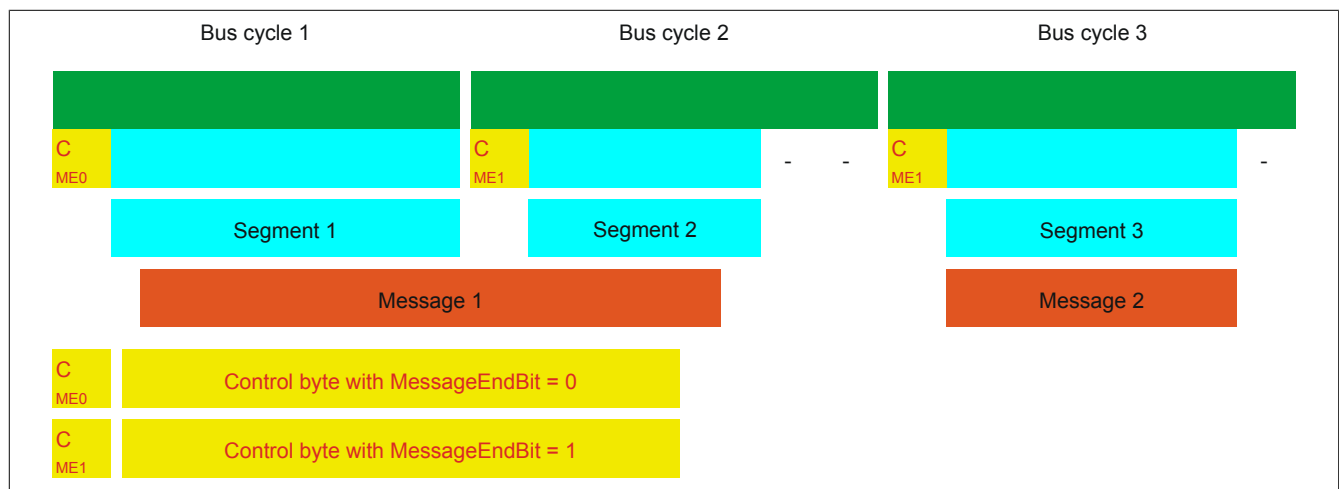


Figure 9: Message arrangement in the MTU (default)

### 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.

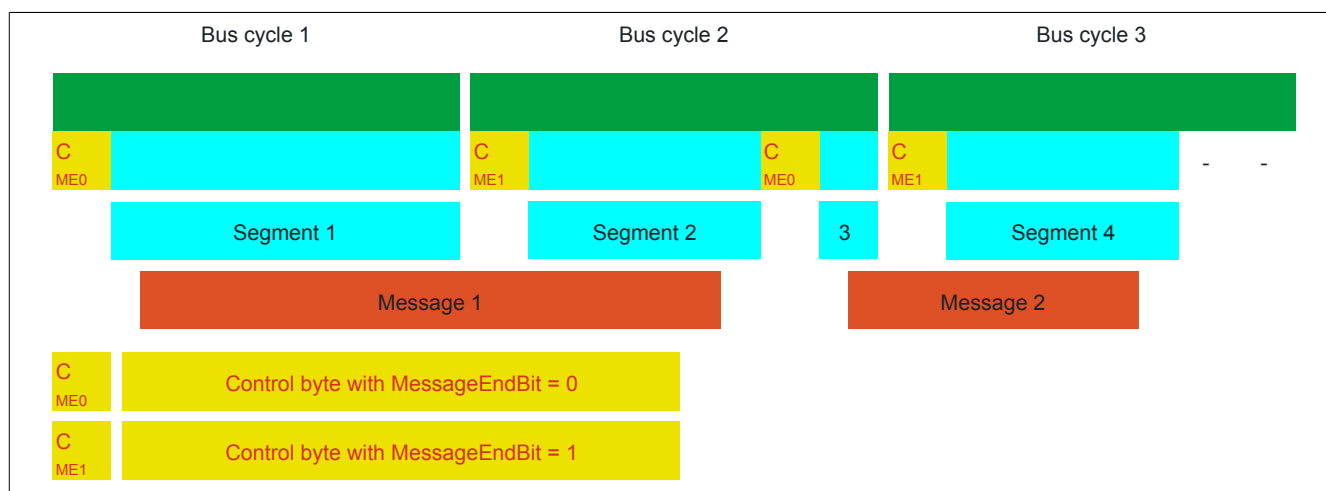


Figure 10: Arrangement of messages in the MTU (MultiSegmentMTUs)

### 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.

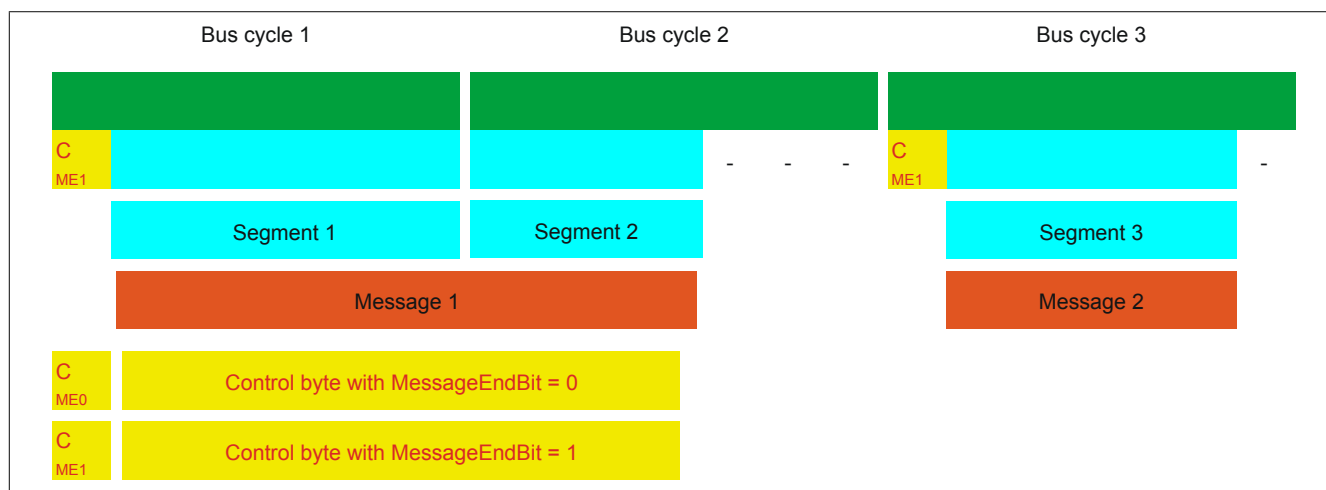


Figure 11: Arrangement of messages in the MTU (large segments)

## Using both options

Using both options at the same time is also permitted.

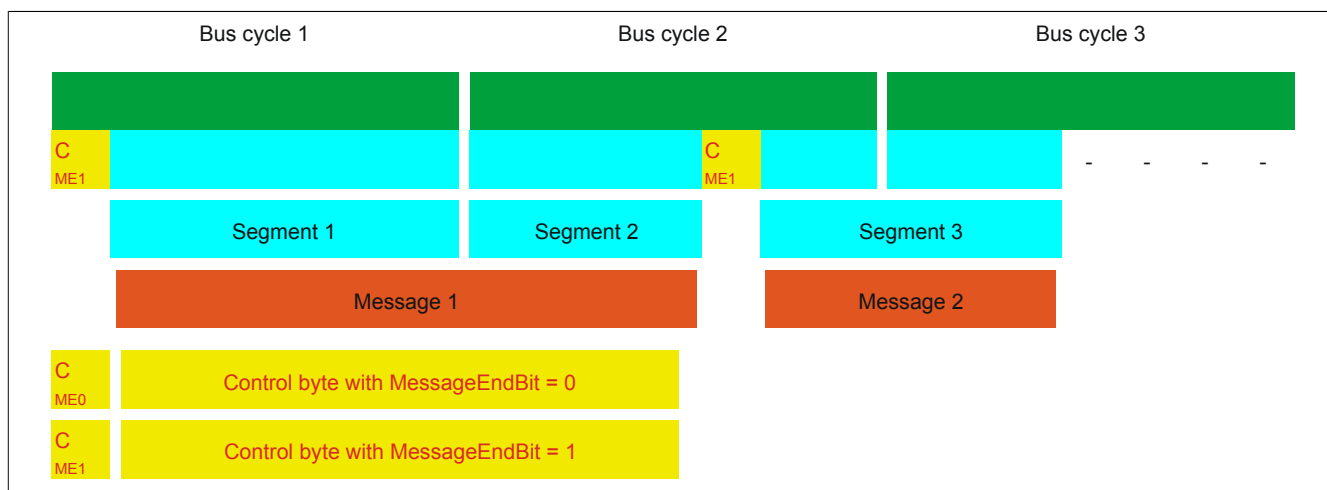


Figure 12: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

4.11.4.9 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.

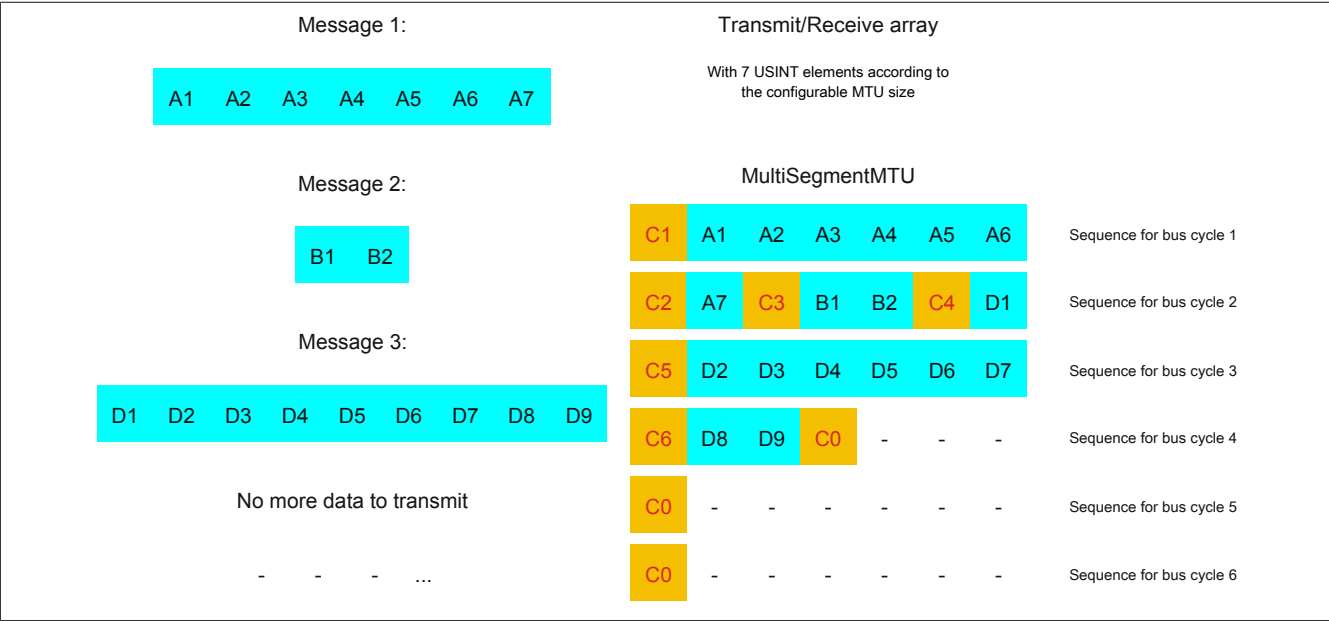


Figure 13: Transmit/receive array (MultiSegmentMTUs)



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.

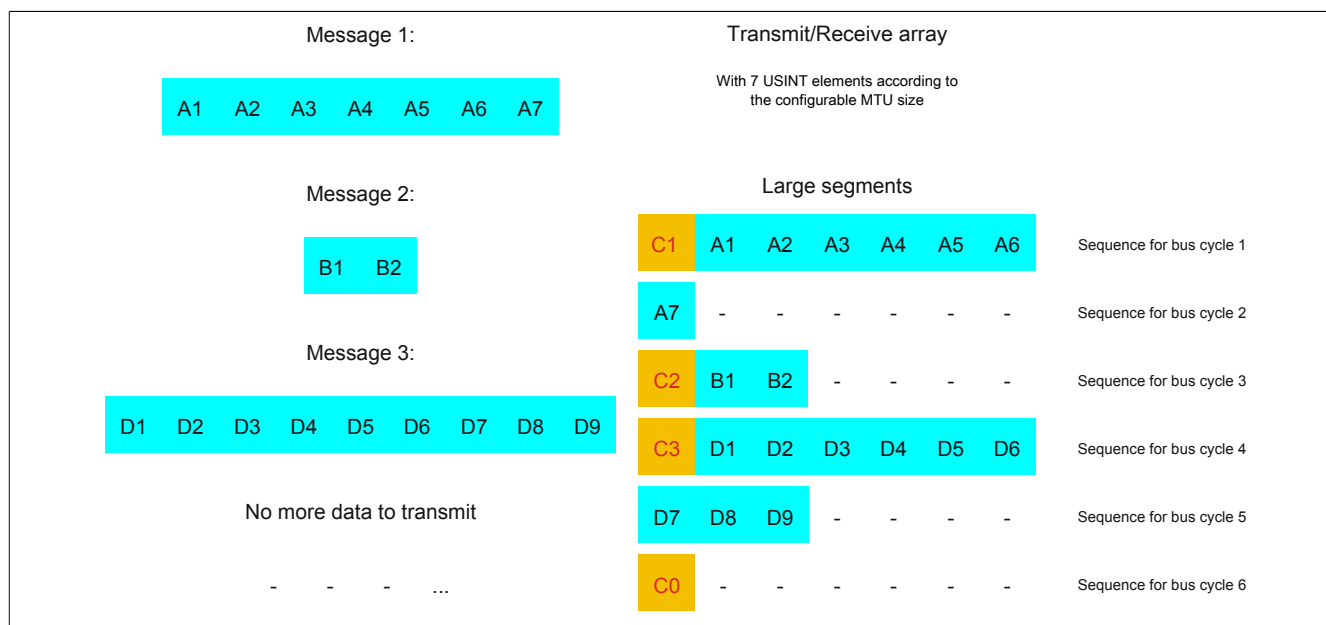


Figure 14: Transmit/receive array (large segments)

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.

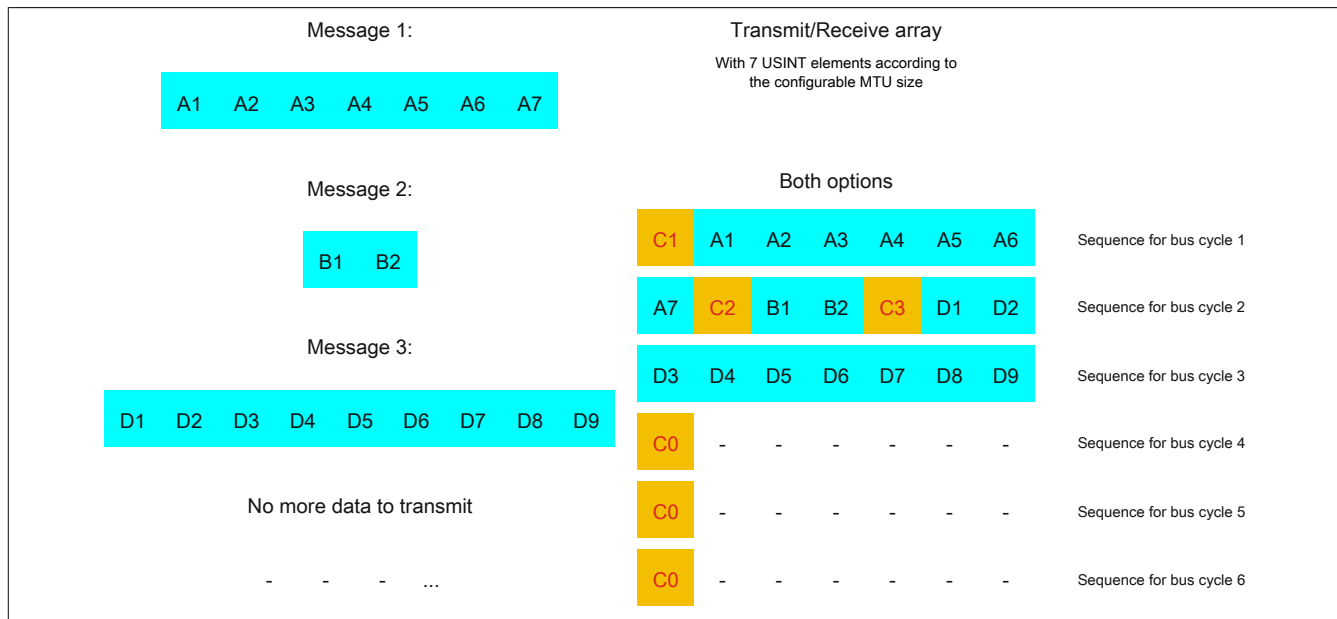


Figure 15: Transmit/receive array (large segments and MultiSegmentMTUs)

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

### 4.11.5 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.

#### 4.11.5.1 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
<b>Actions</b>	Transfer sequence from transmit array, increase SequenceCounter	Cyclic matching of MTU and module buffer	Append sequence to receive array, adjust SequenceAck	Cyclic synchronization MTU and module buffer	Check SequenceAck
<b>Resource</b>	Transmitter (task to transmit)	Bus system (direction 1)	Receiver (task to receive)	Bus system (direction 2)	Transmitter (task for Ack checking)

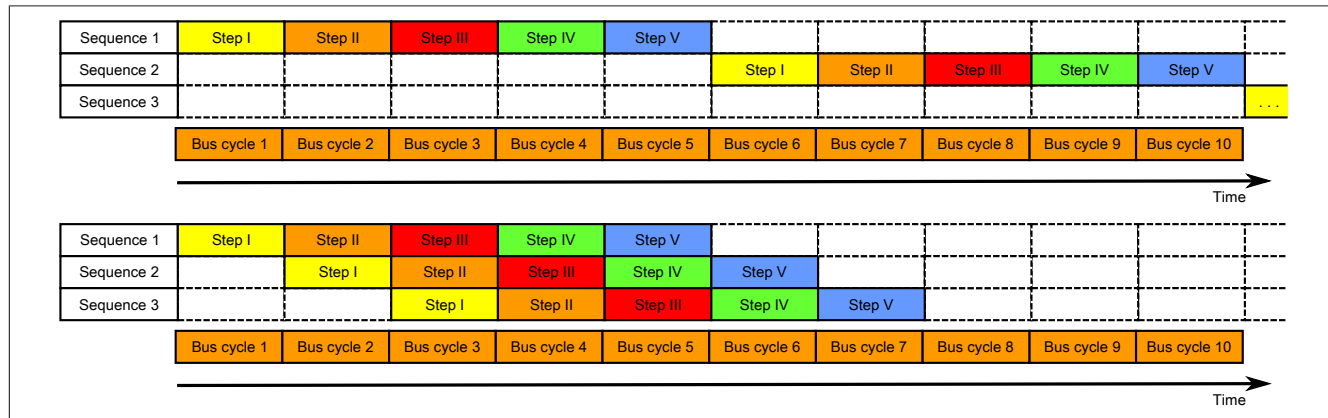


Figure 16: Comparison of transfer without/with Forward

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

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

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

### 4.11.5.2 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.

#### 4.11.5.2.1 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

#### 4.11.5.2.2 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

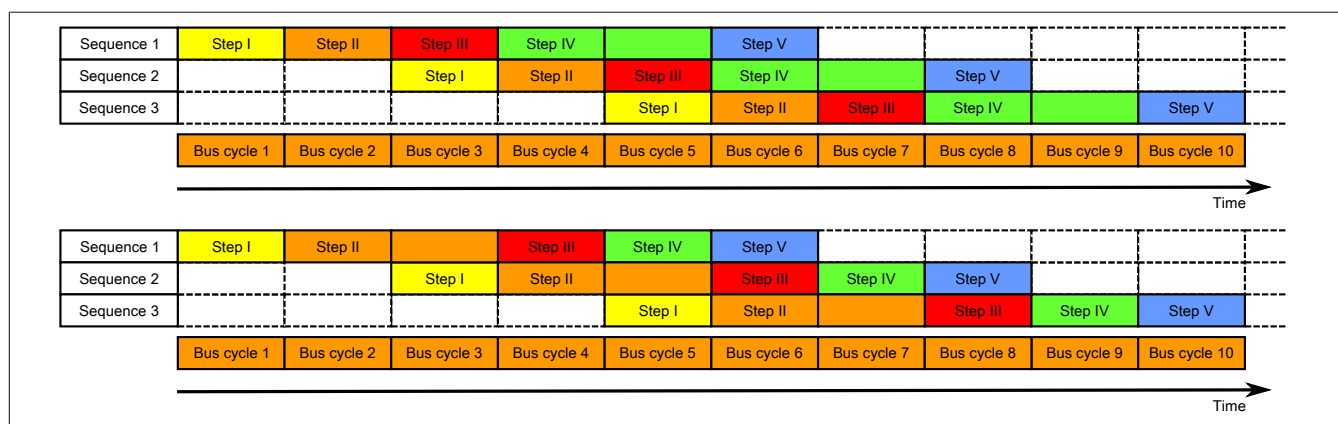


Figure 17: Effect of ForwardDelay when using Flatstream communication with the Forward function

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.

#### 4.11.5.3 Transmitting and receiving with Forward

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

##### Algorithm for transmitting

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

##### Algorithm for receiving

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

## 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 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.

#### 4.11.5.4 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 X2X Link transfers if this type of interference occurs. For example, if an invalid checksum occurs, the I/O system will ignore the data from this bus cycle and the receiver receives the last valid data once more. With conventional (cyclic) data points, this error can often be ignored. In the following cycle, the same data point is again retrieved, adjusted and transferred.

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

##### Loss of acknowledgment (SequenceAck)

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

##### Loss of transmission (SequenceCounter, MTU):

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

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

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

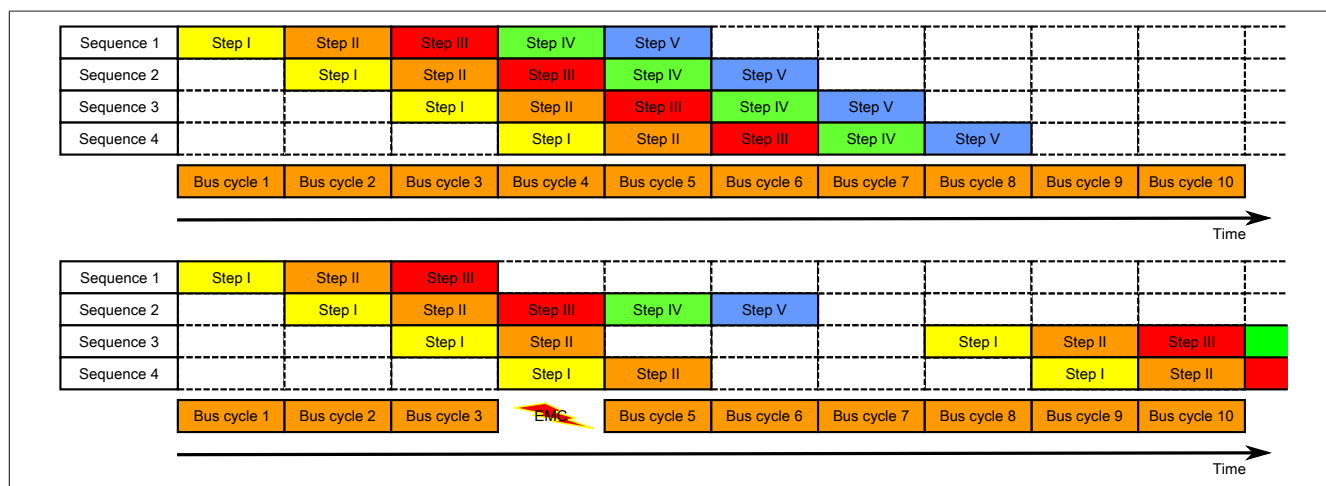


Figure 18: Effect of a lost bus cycle

##### Loss of acknowledgment

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

##### Loss of transmission

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

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

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



## 4.12 Serial with FlatStream

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

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

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

### Operation

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

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

## 4.13 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

## 4.14 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

## 4.15 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