

# X20DS1928

## 1 General information

The module is equipped with an EnDat encoder interface. The module automatically detects whether an encoder is connected with EnDat 2.1 or EnDat 2.2. This module can be used to evaluate encoders installed in B&R servo motors as well as encoders for external axes (encoders that scan any machine movement). The input signals are monitored. This makes it possible to detect open or shorted lines as well as encoder supply failures.

- EnDat 2.1 and EnDat 2.2 encoder interface
- Encoder input monitoring
- 5 VDC and GND for encoder supply
- NetTime function: Time stamp for position

### EnDat encoders

EnDat is a standard developed by Johannes Heidenhain GmbH ([www.heidenhain.de](http://www.heidenhain.de)) that incorporates the advantages of absolute and incremental position measurement and also offers a read/write parameter memory in the encoder. With absolute position measurement, the homing procedure is generally not required. Where necessary a multi-turn encoder should be installed. To save costs, a single-turn encoder and a reference switch can also be used. In this case, a homing procedure must be carried out.

### NetTime position timestamp

Highly dynamic positioning tasks require not only the position value, but also the exact time at which the position was determined. The module has a NetTime function for this, which adds a timestamp to the recorded position with microsecond accuracy.

The module provides the PLC with the position value and timestamp as absolute time value. The NetTime mechanisms ensure that the PLC NetTime clock and the local NetTime clock on the module have exactly the same absolute time at all times.

## 2 Order data


Model number	Short description	Figure
	<b>Digital signal processing and preparation</b>	
X20DS1928	X20 digital signal module, 1 EnDat 2.1/2.2 interface, NetTime function	
	<b>Required accessories</b>	
	<b>Bus modules</b>	
X20BM11	X20 bus module, 24 VDC keyed, internal I/O supply continuous	
X20BM15	X20 bus module, with node number switch, 24 VDC keyed, internal I/O supply continuous	
	<b>Terminal blocks</b>	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	

Table 1: X20DS1928 - Order data

### 3 Technical data

<b>Model number</b>	<b>X20DS1928</b>
<b>Short description</b>	
I/O module	1x EnDat interface
<b>General information</b>	
B&R ID code	0xA912
Status indicators	Counting direction, operating status, module status
Diagnostics	
Module run/error	Yes, using status LED and software
Counting direction	Yes, using status LED
Power consumption	
Bus	0.01 W
Internal I/O	1.3 W
Additional power dissipation caused by actuators (resistive) [W]	-
Type of signal lines	Shielded cables must be used for all signal lines
Certifications	
CE	Yes
KC	Yes
EAC	Yes
UL	cULus E115267 Industrial control equipment
HazLoc	cCSAus 244665 Process control equipment for hazardous locations Class I, Division 2, Groups ABCD, T5
ATEX	Zone 2, II 3G Ex nA nC IIA T5 Gc IP20, Ta (see X20 user's manual) FTZÜ 09 ATEX 0083X
<b>Encoder inputs</b>	
Type	EnDat 2.1/2.2
Angular position resolution	13-bit, with a 1 V <sub>SS</sub> signal
Encoder monitoring	Yes
Max. encoder cable length	10 m, with a line cross-section 4x 2x 0.14 mm <sup>2</sup> and 1x 2x 0.5 mm <sup>2</sup>
Sine/Cosine inputs	
Signal transmission	Differential signals, symmetrical
Signal frequency	DC up to 400 kHz
Differential voltage	1 V <sub>SS</sub>
Common-mode voltage	Max. ±10 V
Terminating resistor	120 Ω
<b>Encoder power supply</b>	
Output voltage	5 V (±5%)
Load capacity	300 mA
Protective measures	
Overload protection	Yes
Short-circuit proof	Yes
<b>Serial EnDat interface</b>	
Signal transmission	5 VDC differential signal, EIA RS-485 standard
Transmission status	See EnDat specification
<b>Electrical properties</b>	
Electrical isolation	Channel isolated from bus Channel not isolated from channel
<b>Operating conditions</b>	
Mounting orientation	
Horizontal	Yes
Vertical	Yes
Installation elevation above sea level	
0 to 2000 m	No limitations
>2000 m	Reduction of ambient temperature by 0.5°C per 100 m
Degree of protection per EN 60529	IP20
<b>Ambient conditions</b>	
Temperature	
Operation	
Horizontal mounting orientation	-25 to 60°C
Vertical mounting orientation	-25 to 50°C
Derating	See section "Derating"
Storage	-40 to 85°C
Transport	-40 to 85°C


Table 2: X20DS1928 - Technical data

Model number	X20DS1928
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order 1x X20TB12 terminal block separately Order 1x X20BM11 bus module separately
Spacing	12.5 <sup>+0.2</sup> mm

Table 2: X20DS1928 - Technical data

## 4 Status LEDs

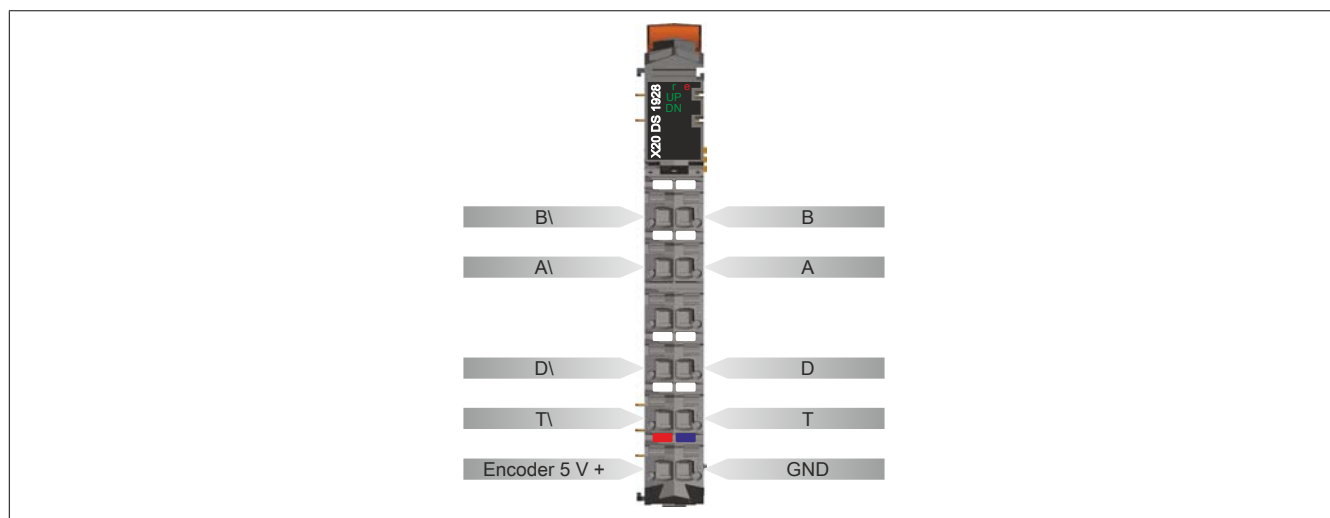
For a description of the various operating modes, see section "Additional information - Diagnostic LEDs" of the X20 system user's manual.

Image	LED	Color	Status	Description
	r	Green	Off	Module supply not connected
			Single flash	RESET mode
			Double flash	BOOT mode (during firmware update) <sup>1)</sup>
			Blinking	PREOPERATIONAL mode
			On	RUN mode
	e	Red	Off	Module supply not connected or everything is OK
			On	Error or reset state - Possible cause: • Encoder supply error
			Single flash	I/O error - Possible causes: • Sine/Cosine relative position error (open line) • Sine/Cosine absolute position error (reference)
			Double flash	System error - Possible causes: • EnDat communication error • EnDat position error • EnDat error defining parameters
			Triple flash	I/O error and system error
			Single flash, inverted	Error or reset state and I/O error
			Double flash, inverted	Error or reset state and system error
			Triple flash, inverted	Error or reset state, I/O error and system error
	UP	Green	On	The "UP/DN" LEDs are lit depending on the rotational direction and the speed of the connected encoder. The "UP" LED indicates when the encoder position changes in the positive direction.
	DN	Green	On	The "DN" LED indicates when the encoder position changes in the negative direction.

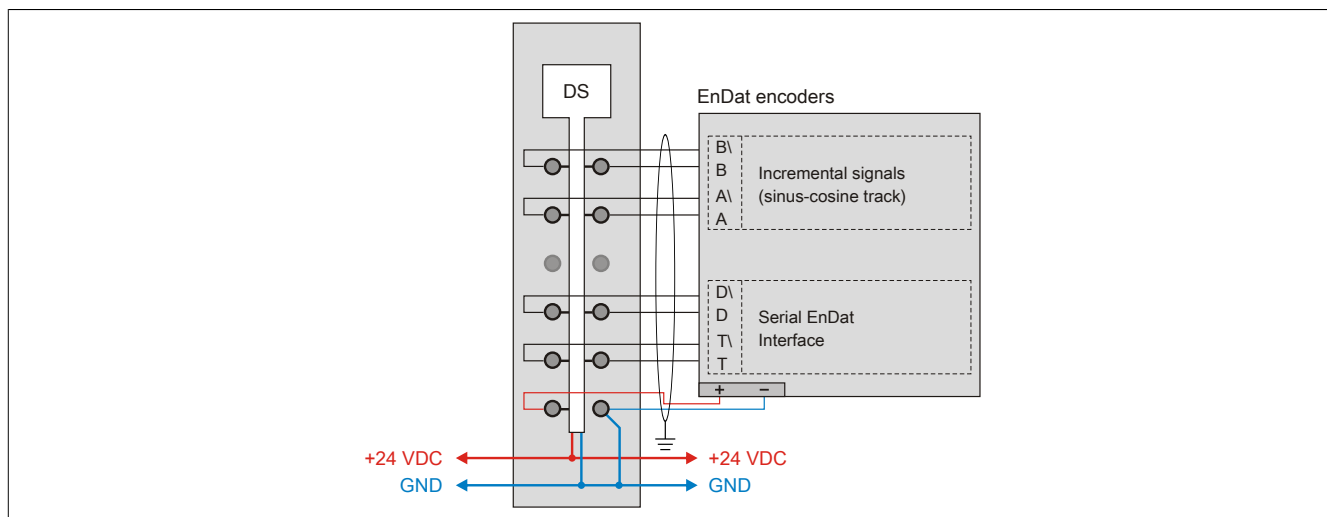
1) Depending on the configuration, a firmware update can take up to several minutes.

## 5 Pinout

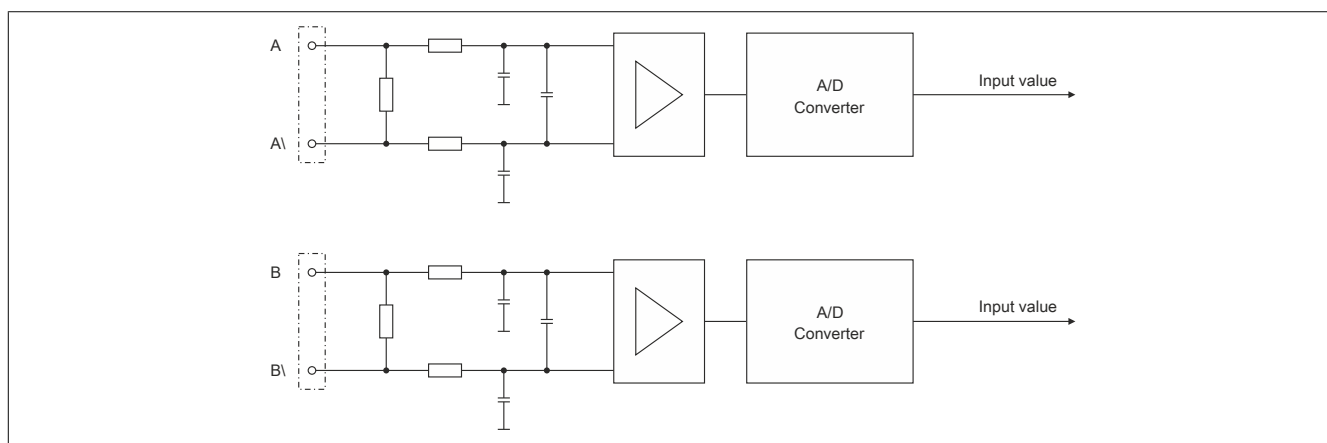
Shielded cables should be used for all signal lines.



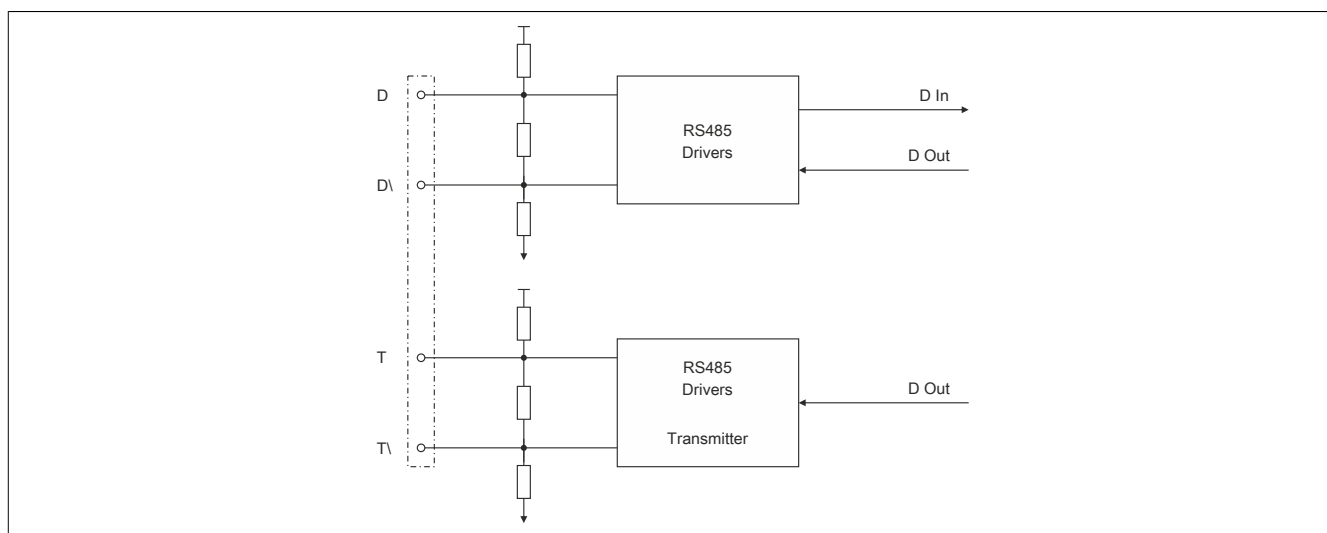
## 6 Connection example



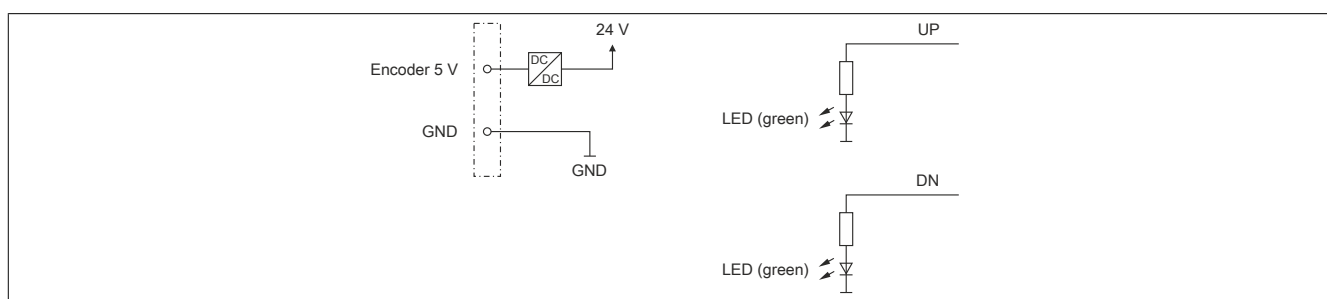
## 7 Input diagram for the incremental signals (sine-cosine track)



## 8 Input diagram for the serial EnDat interface



## 9 Encoder supply scheme and LEDs

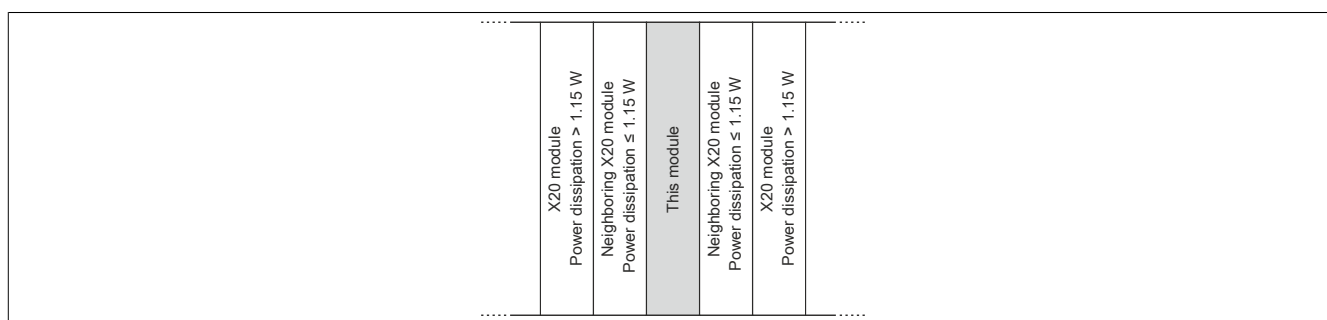


## 10 Derating

There is no derating when operated below 55°C.

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

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



## 11 Register description

### 11.1 General data points

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

General data points are described in section "Additional information - General data points" of the X20 system user's manual.

### 11.2 Register overview - Function model 0 (standard)

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Module configuration						
513	CfO_SlframeGenID	USINT				•
654	CfO_SystemCyclePrescaler	UINT				•
Basic functions						
683	SDCLifeCount	SINT	•			
4180	PositionHW	UDINT	•			
4188	PositionLW	UDINT	•			
	Position	DINT				
4172	PosTime (32-Bit)	DINT	•			
4174	PosTime (16-Bit)	INT	•			
4163	PosCycle	SINT	•			
Error management						
389	ErrorEnableID_1710	USINT				•
261	ErrorInfo	USINT	•			
	EncoderSupplyError	Bit 0				
	VssCheckError	Bit 2				
	SinCosPosError	Bit 3				
	EnDatComError	Bit 4				
	EnDatPosError	Bit 5				
	EnDatParSetError	Bit 6				
	EnDatRefWarning	Bit 7				
325	AckErrorInfo	USINT			•	
	AckEncoderSupplyError	Bit 0				
	AckVssCheckError	Bit 2				
	AckSinCosPosError	Bit 3				
	AckEnDatComError	Bit 4				
	AckEnDatPosError	Bit 5				
	AckEnDatParSetError	Bit 6				
	AckEnDatRefWarning	Bit 7				
4352	EnDatError	UINT	•			
4353	EnDatWarning	UINT	•			
4099	Acknowledging EnDat errors	USINT			•	
	EnDatAck	Bit 0				
Sin/Cos - Configuration						
1025	SinCosEnable	USINT				•
1027	SinCosRefSource	USINT				•
1034	SinCosVssMin	UINT				•
1038	SinCosVssMax	UINT				•
1044	SinCosQuitTime	UDINT				•
EnDat - Read ID						
4097	EnDatMode	USINT				•
4400 + N	OperatingParam_N (index N = 00 to 15)	UINT		•		
4352 + N	OperatingStatus_0N (index N = 0 to 3)	UINT		•		
4352 + N	ParamManuf_N (index N = 04 to 47)	UINT		•		
4416 + N	ParamManufEnDat22_N (index N = 01 to 63)	UINT		•		
EnDat - Read additional information						
4860 + N*8	EnDatInfoCmd0N (index N = 1 to 4)	UDINT				•
4935	Validity of info data	USINT	•			
	EnDatInfoOK01	Bit 0				
	...	...				
	EnDatInfoOK04	Bit 3				
4978 + N*16	EnDatInfo0N (index N = 1 to 4)	UINT	•			
		INT				
Flatstream mode						
4609	OutputMTU	USINT				•
4611	InputMTU	USINT				•
4613	FlatStreamMode	USINT				•
4615	Forward	USINT				•
4620	ForwardDelay	UINT				•

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
4672	InputSequence	USINT	•			
4672 + N	RxByteN (index N = 1 to 15)	USINT	•			
4704	OutputSequence	USINT			•	
4704 + N	TxByteN (index N = 1 to 15)	USINT			•	

### 11.3 Register overview - Function model 254 (bus controller)

Register	Offset <sup>1)</sup>	Name	Data type	Read		Write	
				Cyclic	Acyclic	Cyclic	Acyclic
Module configuration							
513	-	CfO_SlframeGenID	USINT				•
654	-	CfO_SystemCyclePrescaler	UINT				•
Basic functions							
4180	0	PositionHW	UDINT	•			
4188	4	PositionLW	UDINT	•			
4163	15	PosCycle	SINT	•			
Error management							
389	-	ErrorEnableID_1710	USINT	•			•
261	14	ErrorInfo	USINT				
		EncoderSupplyError	Bit 0				
		VssCheckError	Bit 2				
		SinCosPosError	Bit 3				
		EnDatComError	Bit 4				
		EnDatPosError	Bit 5				
		EnDatParSetError	Bit 6				
		EnDatRefWarning	Bit 7				
325	6	AckErrorInfo	USINT			•	
		AckEncoderSupplyError	Bit 0				
		AckVssCheckError	Bit 2				
		AckSinCosPosError	Bit 3				
		AckEnDatComError	Bit 4				
		AckEnDatPosError	Bit 5				
		AckEnDatParSetError	Bit 6				
		AckEnDatRefWarning	Bit 7				
4352	-	EnDatError	UINT		•		
4353	-	EnDatWarning	UINT		•		
4099	-	Acknowledging EnDat errors	USINT				•
		EnDatAck	Bit 0				
Sin/Cos - Configuration							
1025	-	SinCosEnable	USINT				•
1027	-	SinCosRefSource	USINT				•
1034	-	SinCosVssMin	UINT				•
1038	-	SinCosVssMax	UINT				•
1044	-	SinCosQuitTime	UDINT				•
EnDat - Read ID							
4097	-	EnDatMode	USINT				•
4400 + N	-	OperatingParam_N (index N = 00 to 15)	UINT		•		
4352 + N	-	OperatingStatus_0N (index N = 0 to 3)	UINT		•		
4352 + N	-	ParamManuf_N (index N = 04 to 47)	UINT		•		
4416 + N	-	ParamManufEnDat22_N (index N = 01 to 63)	UINT		•		
EnDat - Read additional information							
4860 + N*8	-	EnDatInfoCmd0N (index N = 1 to 4)	UDINT		•		•
4935	-	Validity of info data	USINT				
		EnDatInfoOK01	Bit 0				
		...	...				
		EnDatInfoOK04	Bit 3				
4978 + N*16	-	EnDatInfo0N (index N = 1 to 4)	UINT		•		
			INT				
Flatstream mode							
4609	-	OutputMTU	USINT				•
4611	-	InputMTU	USINT				•
4613	-	FlatStreamMode	USINT				•
4615	-	Forward	USINT				•
4620	-	ForwardDelay	UINT				•
4672	8	InputSequence	USINT	•			
4672 + N	9 - 13	RxByteN (index N = 1 to 5)	USINT	•			
4704	0	OutputSequence	USINT			•	
4704 + N	1 - 5	TxByteN (index N = 1 to 5)	USINT			•	

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

### 11.3.1 Using the module on the bus controller

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

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

### 11.3.2 CAN I/O bus controller

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

## 11.4 Module configuration

The following configuration registers can be used to define various module settings. They can be used, for example, to modify the module's behavior on an X2X Link network. The user can choose between 2 option registers.

### 11.4.1 Data query

Name:

CfO\_SlframeGenID

This register can be used to define when the synchronous/cyclic input data is generated. "X2X cycle optimized" should be set for jitter-free data acquisition. "Fast reaction" can be set for the best performance.

Data type	Value	Information
USINT	9	Fast reaction
	14	X2X cycle optimized; Bus controller default

### 11.4.2 Prescale factor

Name:

CfO\_SystemCyclePrescaler

In order for the module to communicate with the CPU as well as the EnDat encoder, the EnDat cycle time must be at least twice the module cycle time. The actual EnDat cycle time is a result of multiplying the module cycle time by the value in this register.

Data type	Value	Information
UINT	2	EnDat cycle: 200 to 400 µs (bus controller default setting)
	4	EnDat cycle: 400 to 800 µs
	8	EnDat cycle: 800 to 1,600 µs

## 11.5 Basic functions

This module can import a position when used together with an EnDat encoder. The received data is prepared in 2 different formats and given a [timestamp](#). 6 registers are available for further processing. This allows the user to choose which format is best suited for individual application.

### 11.5.1 SDC counter register

Name:

SDCLifeCount

The 8-bit counter register is needed for the SDC software package. It is incremented with the system clock to allow the SDC to check the validity of the data frame.

Data type	Value
SINT	-128 to 127



### 11.5.2 Absolute position values

Name:

PositionHW

PositionLW

The absolute position of the encoder is defined using 64-bit resolution. The position value is stored in the PositionHW and PositionLW registers. The upper 32 bits are stored in the PositionHW register, while the lower 32 bits are stored in the PositionLW register.

For SinCos signal evaluation, see ["Format of the SinCos signal" on page 15](#) for information regarding the data format.

Data type	Value
2x UDINT	0 to 4,294,967,295

### 11.5.3 SDC position value

Name:

Position

The SDC library requires a signed 32-bit position value. The position's low word can be accessed separately for this. The value can also be used as default position value, however.

For SinCos signal evaluation, see ["Format of the SinCos signal" on page 15](#) for information regarding the data format.

Data type	Value
DINT	-2,147,483,648 to 2,147,483,647

### 11.5.4 NetTime of the position values

Name:

PosTime

The current NetTime value is assigned to each determined position in this register. The NetTime is recorded with  $\mu$ s accuracy.

The SDC library requires a 16 bit value. The NetTime value is therefore also prepared in this format.

For more information about NetTime and timestamps, see ["NetTime technology" on page 54](#).

Data type	Value	Information
DINT	-2,147,483,648 to 2,147,483,647	NetTime in $\mu$ s
INT	-32,768 to 32,767	

### 11.5.5 Counter for position values

Name:

PosCycle

PosCycle is an integer counter that is incremented as soon as the module has saved a new valid position value.

Data type	Value
SINT	-128 to 127

## 11.6 Error management

This module can be used to diagnose error states. There are the following ways in which this is done:

- ["Module-based diagnostics" on page 10](#)
- ["EnDat-based diagnostics" on page 13](#)

### 11.6.1 Module-based diagnostics

The module diagnoses 7 different errors or warnings. Depending on the settings, the error bits can be called either individually or packed together.

#### 11.6.1.1 Configuring errors and warnings

Name:

ErrorEnableID\_1710

The implemented diagnostic algorithms can be enabled or disabled in this register.

Data type	Value	Bus controller default
USINT	See bit structure.	255

Bit structure:

Bit	Name	Value	Information
0	<a href="#">Encoder supply:</a>	0	Error detection disabled
		1	Error detection enabled (bus controller default setting)
1	Reserved	-	
2	<a href="#">Vss Sin/Cos:</a>	0	Error detection disabled
		1	Error detection enabled (bus controller default setting)
3	<a href="#">Position error:</a>	0	Error detection disabled
		1	Error detection enabled (bus controller default setting)
4	<a href="#">EnDat - Communication:</a>	0	Error detection disabled
		1	Error detection enabled (bus controller default setting)
5	<a href="#">EnDat - Position:</a>	0	Error detection disabled
		1	Error detection enabled (bus controller default setting)
6	<a href="#">EnDat - Parameters:</a>	0	Error detection disabled
		1	Error detection enabled (bus controller default setting)
7	<a href="#">EnDat - Reference warning:</a>	0	Warning disabled
		1	Warning enabled (bus controller default setting)

#### Encoder supply:

The encoder voltage supply is below the permitted limit.

#### Vss Sin/Cos:

The voltage value for the Sin/Cos track violates the configured limit values.

→ See register ["SinCosVssMin" on page 16](#) or ["SinCosVssMax" on page 16](#).

#### Position error:

The determined position value violates the requirements of the application.

#### EnDat - Communication:

Communication error on the EnDat interface (e.g. incorrect checksum)

#### EnDat - Position:

Encoder evaluates the determined position value as invalid.

#### EnDat - Parameters:

Inconsistent register values for encoder identification

→ Countermeasures: Check wiring or rescan (see ["EnDatAck" on page 14](#))

**EnDat - Reference warning:**

The digital interface provides an absolute position value that can be used to accurately describe the axis position. The position value is homed to this absolute value at the beginning of a measurement. The analog interface can be used to incrementally sample changes that occur very rapidly. This enables the module to continue sampling the position value at a high resolution. Both the analog and the digital signal are sampled cyclically. If the value determined incrementally deviates from the absolute value during operation then the referencing warning is displayed and the position must be referenced again.

**11.6.1.2 Status of errors and warnings**

Name:

ErrorInfo

EncoderSupplyError

VssCheckError

PositionError

EnDatComError

EnDatPosError

EnDatParSetError

EnDatRefWarning

This register indicates which error or warning is currently active. For a description of errors, see ["Configuring errors and warnings" on page 10](#).

Data type	Value
USINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	EncoderSupplyError	0	No error
		1	Encoder supply error
1	Reserved	-	
2	VssCheckError	0	No error
		1	Vss error on Sin/Cos track
3	PositionError	0	No error
		1	Position error
4	EnDatComError	0	No error
		1	EnDat communication error
5	EnDatPosError	0	Error detection disabled
		1	Error detection enabled
6	EnDatParSetError	0	Error detection disabled
		1	Error detection enabled
7	EnDatRefWarning	0	Warning disabled
		1	Warning enabled

### 11.6.1.3 Acknowledging errors and warnings

Name:

AckErrorInfo

AckEncoderSupplyError

AckVssCheckError

AckPositionError

AckEnDatComError

AckEnDatPosError

AckEnDatParSetError

AckEnDatRefWarning

This register is used to acknowledge an error message that occurred in the "Status of errors and warnings" on page 11 register. For a description of errors, see "Configuring errors and warnings" on page 10.

Data type	Value
USINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	AckEncoderSupplyError	0	No error acknowledgment
		1	Error acknowledgment
1	Reserved	-	
2	AckVssCheckError	0	No error acknowledgment
		1	Error acknowledgment
3	AckPositionError	0	No error acknowledgment
		1	Error acknowledgment
4	AckEnDatComError	0	No error acknowledgment
		1	Error acknowledgment
5	AckEnDatPosError	0	No error acknowledgment
		1	Error acknowledgment
6	AckEnDatParSetError	0	No error acknowledgment
		1	Error acknowledgment
7	AckEnDatRefWarning	0	No acknowledgment
		1	Acknowledgment

## 11.6.2 EnDat-based diagnostics

Memory areas are provided in the EnDat standard for error handling. Error management was tailored to utilize error detection according to the EnDat standard. Additional registers were implemented in the module which prepare these areas in the encoder memory.

The module allows access to all previously defined memory areas for error handling. The memory areas are mirrored in the module registers and can be interpreted by the user.

Detailed information regarding the errors that can be detected in this way can be found in the encoder's manual.

### 11.6.2.1 EnDat errors

Name:

EnDatError

This register is used to indicate critical conditions on the EnDat encoder. The system has generally ceased to work and requires service.

Data type	Value
UINT	See bit structure.

The bit structure described below is designed according to the general recommendations of the EnDat standard. The specification does not limit which trigger algorithms to use or which of the listed messages must be supported. Please refer to the encoder's manual for further details.

Bit structure:

Bit	Name	Value	Information
0	Illumination	0	OK
		1	Failed
1	Signal amplitude	0	OK
		1	Detected as having errors
2	Position value	0	OK
		1	Detected as having errors
3	Overvoltage	0	No
		1	Yes
4	Undervoltage	0	No
		1	Yes
5	Overcurrent	0	No
		1	Yes
6	Battery	0	OK
		1	Must be changed
7 - 15	Reserved	-	

### 11.6.2.2 EnDat warnings

Name:

EnDatWarning

This register is used to indicate critical conditions on the EnDat encoder. Encoder still functional, but must be checked immediately. This generally means that defined tolerances have been exceeded.

Data type	Value
UINT	See bit structure.

The bit structure described below is designed according to the general recommendations of the EnDat standard. The specification does not limit which trigger algorithms to use or which of the listed messages must be supported. Please refer to the encoder's manual for further details.

Bit structure:

Bit	Name	Value	Information
0	Frequency collision	0	No
		1	Yes
1	Temperature exceeded	0	No
		1	Yes
2	Control reserve - Lighting	0	Not required
		1	Required
3	Charge - Battery	0	OK
		1	Low
4	Reference point	0	Reached
		1	Not reached
5 - 15	Reserved	-	

### 11.6.2.3 Acknowledging EnDat errors

Name:

EnDatAck

"EnDatAck" acknowledges all errors and warnings from the ["EnDatError" on page 13](#) and ["EnDatWarning" on page 14](#) registers. It can also instruct the module to re-import the parameters for identification.

If one of the bits in this register is set, the system automatically resets it and the respective algorithm is run.

Data type	Value
USINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	EnDatAck	0	No acknowledgment
		1	Acknowledge
1	Rescan - Identification register	0	Imported parameters retained
		1	Reimport parameters
2 - 7	Reserved	-	

## 11.7 Sin/Cos - Analog interface configuration

In addition to the digital EnDat interface, this module is also equipped with an analog interface for sampling a differential sine-cosine signal. To increase the resolution, the EnDat standard specifies a cooperation between the analog and digital information. This enables a highly dynamic representation of the position while maintaining high resolution.

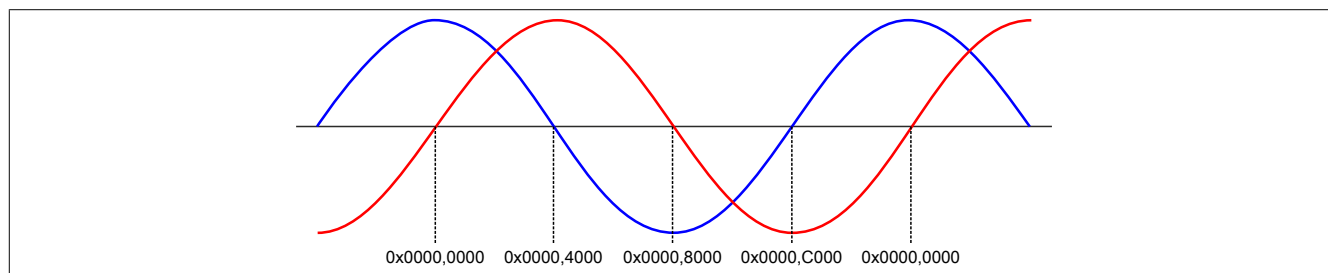
### 11.7.1 Format of the SinCos signal

The SinCos signal is represented as a position value in the "[Absolute position values](#)" on page 9 and "[SDC position value](#)" on page 9 registers. The following relationships apply:

- PositionLW and Position are identical in the function.
- PositionHW extends the integer range of PositionLW by adding multi-turn functionality.

64-bit register	PositionHW (unsigned)	PositionLW (unsigned)																
32-bit register	-	Position (signed)																
Format	Integer extension (to 48-bit)	Integer (16-bit)	Decimal places: (with 13-bit resolution)															
Information		A full sine wave corresponds to an increment of the integer.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0
			Important: The lower 3 bits always contain the value 0.															
Word/DWord	DWord	Word 1	Word 0															

Relationship between sine curve (red) and decimal places:



### 11.7.2 Enabling SinCos

Name:  
SinCosEnable

This register must always have the value 1 for configuration reasons.

Data type	Value	Information
USINT	1	Bus controller default: 1

### 11.7.3 Enabling SinCos reference source

Name:  
SinCosRefSource

This register must always have the value 1 for configuration reasons.

Data type	Value	Information
USINT	1	Bus controller default: 1

#### 11.7.4 Configuring the lower Vss value

Name:

SinCosVssMin

This register specifies the lower limit value for the peak-to-peak voltage of the sine/cosine track. The incoming signal is monitored in this way. If the incoming value falls below this specified limit, then the module reports the corresponding error.

Data type	Value	Information
UINT	0 to 1500	Values in mV Bus controller default: 800

#### 11.7.5 Configuring the upper Vss value

Name:

SinCosVssMax

This register specifies the upper limit value for the peak-to-peak voltage of the sine/cosine track. The incoming signal is monitored in this way. If the incoming value exceeds this specified limit, then the module reports the corresponding error.

Data type	Value	Information
UINT	0 to 1500	Values in mV Bus controller default: 1200

#### 11.7.6 Configuring the delay time after errors

Name:

SinCosQuitTime

If an error is detected on the analog interface, the last correctly read values remain valid. An interval can be defined in this register at which the module begins receiving correct values again after the error state without processing them further internally. Only then will newly sampled correct analog values be recognized as valid.

Data type	Value	Information
UDINT	0 to 20000000	Values in $\mu$ s Bus controller default: 100000



## 11.8 EnDat

### 11.8.1 EnDat - Digital interface configuration

The EnDat interface allows you to establish a point-to-point connection with exactly one EnDat encoder.

There are 2 ways to use the encoder data in the PLC program. One is to store the necessary encoder values temporarily in the module, where they can then be provided to the CPU. The other is to use the module's FlatStream mode, which supports the full range of commands defined in the EnDat specification.

Detailed information about the EnDat specification can be found in the document, "Technical Information – EnDat 2.2".

#### 11.8.1.1 Configuring EnDat module properties

Name:

EnDatMode

This register is used to define various module properties.

Data type	Value	Bus controller default
USINT	See bit structure.	0

Bit structure:

Bit	Name	Value	Information
0	EnDat interface	0	Disabled (bus controller default setting)
		1	Enabled
1	Format of imported position data	0	Unsigned (bus controller default setting)
		1	Signed
2	Fast EnDat cycle (6 MHz)	0	Enabled if encoder compatible (bus controller default setting)
		1	Disabled
3	Sin/Cos track	0	Enabled (bus controller default setting)
		1	Disabled
4 - 7	Reserved	-	

### 11.8.2 EnDat - Read ID

The EnDat interface does more than just help the user specify axis positions. It can also be used to readout certain data stored in the encoder memory.

The EnDat specification divides the encoder memory into logical groups. These include memory areas for the operating parameters, operating status, manufacturer parameters and manufacturer parameters according to EnDat 2.2.

The 4 most important memory areas are mirrored in the module registers. The information can be accessed in the application and used to identify a particular encoder.

#### Information:

**There are different types of EnDat. Please keep this in mind. EnDat has been continuously expanded to include new technical possibilities while maintaining backward compatibility. Several advancements have been made to the standard, which has resulted in a non-uniform structure.**

In general, data is queried from memory for identification purposes when the module is started. In addition, the data can be reimported using the "EnDatAck" on page 14 register. The module reads the data from the encoder, which is then mapped for the PLC.

#### 11.8.2.1 Operating Parameters

Name:

OperatingParam\_00 to OperatingParam\_15

These registers can be used to read out the current operating parameters. The data in these registers correspond exactly to the values on the encoder. More detailed information can be found in the encoder's manual or by referring to the latest EnDat specification.

Data type	Value
16x UINT	See encoder manual

### 11.8.2.2 Operating state

Name:

OperatingStatus\_00 to OperatingStatus\_03

This register can be used to read the encoder's current operating state. The first 2 registers from this group are identical to the "EnDatError" on page 13 and "EnDatWarning" on page 14 registers. A special setting is provided because they are update cyclically.

Information about write protection and other configuration settings is managed in registers 02 and 03. The data in these registers correspond exactly to the values on the encoder.

More detailed information can be found in the encoder's manual or by referring to the latest EnDat specification.

Data type	Value
4x UINT	See encoder manual

### 11.8.2.3 Manufacturer parameters

Name:

ParamManuf\_04 to ParamManuf\_47

These registers are used to prepare the manufacturer parameters according to the EnDat standard 2.1. The exact arrangement of information can be found in the documentation "Technical Information - EnDat 2.2".

Data type	Value
44x UINT	see "Technical Information - EnDat 2.2" or encoder manufacturer data

### 11.8.2.4 Additional manufacturer parameters according to EnDat 2.2

Name:

ParamManufEnDat22\_00 to ParamManufEnDat22\_63

These registers are used to prepare the manufacturer parameters according to the EnDat standard 2.2. The exact arrangement of information can be found in the documentation "Technical Information - EnDat 2.2".

Data type	Value
64x UINT	see "Technical Information - EnDat 2.2" or encoder manufacturer data

### 11.8.3 EnDat - Read additional information

In addition to the identification data, other information can also be accessed from the encoder. However, the following algorithm requires exact knowledge of the encoder's memory structure and the EnDat specification.

#### Configuration

There are 4 different channels that can be operated during a cycle. One register per channel each is used for configuration, (i.e. determines which data is read from the encoder and mirrored on the respective Info byte).

#### 11.8.3.1 Transmitting EnDat commands

Name:

EnDatInfoCmd01 to EnDatInfoCmd04

This register controls which data is processed on the corresponding Info byte for each channel. The register consists of up to 4 separate 8-bit values.

Data type	Value	Bus controller default
4x UDINT	See bit structure.	0

Bit structure:

Bit	Name	Information	
0 - 7	Command	Selects the response section	
8 - 15	Memory area codes	MRS code	
		<b>Parameters not in blocks</b>	<b>Parameters arranged in blocks</b>
16 - 23	Memory ID	Parameter numbers	Block number
24 - 31	Memory ID	-	Parameter numbers

There is a difference when querying data from an encoder using an EnDat 2.1 command or an EnDat 2.2 command. When querying encoder data with an EnDat 2.1 command (0x04 and 0x06), the parameter number and (optionally) the block number must be specified in addition to the MRS code.

When querying the memory with an EnDat 2.2 command, the parameter number and block number are not required. The module consecutively transmits all 4 words of the memory area, which was selected using the MRS code. The right command must be selected depending on which of the 4 response bytes is needed.

#### Memory area codes

The code to be defined is identical to the MRS code for the encoder memory. The EnDat specification has left a few of the encoders memory areas undefined and available for future developments. This is why a clear and reliable explanation cannot be provided here.

More detailed information can be found in the encoder's manual or by referring to the latest EnDat specification.

#### Parameter numbers

EnDat 2.1 requires the corresponding parameter number to be entered in order to specifically address the desired parameter in the encoder memory. Older EnDat versions did not divide the encoder memory into blocks. This is why there are memory areas that can be selected without specifying a block number. In this case, the parameter number must be entered on the third byte.

More detailed information can be found in the encoder's manual or by referring to the latest EnDat specification.

#### Block number

To expand the address range of the encoder memory, additional block numbers were added starting at the second section. If the desired parameter is located in this blocked area, then the block number must be specified on the third byte. In this case, the parameter number is entered on the fourth byte.

More detailed information can be found in the encoder's manual or by referring to the latest EnDat specification.

## Call

After being configured correctly, the position value is transmitted cyclically to the module. Each channel has 2 registers that serve as temporary storage. The module confirms successful receipt by setting an OK bit. The EnDat specification does not specify in which format the parameters must be received. Therefore, the module provides the information in 2 ways. Which of the two registers should be used for further processing depends on the parameters being read.

### 11.8.3.2 Validity of info data

Name:

EnDatInfoOK01 to EnDatInfoOK04

This register's bits provide information about the validity of the current info data in temporary storage.

Data type	Value
USINT	See bit structure.

Bit structure:

Bit	Name	Value	Information
0	EnDatInfoOK01	0	Value 01 invalid
		1	Value 01 valid
...	...	...	
3	EnDatInfoOK04	0	Value 04 invalid
		1	Value 04 valid
4 - 7	Reserved	-	

### 11.8.3.3 Reading EnDat information

Name:

EnDatInfo01 to EnDatInfo04

These registers provide the corresponding requested information as a signed or unsigned 2-byte value.

The EnDat specification does not specify the format of the received parameters. Which of the two data types should be used for further processing therefore depends on the parameter being read.

Data type	Value
UINT	0 to 65535
INT	-32,768 to 32,767

## 11.9 Flatstream communication

### 11.9.1 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.

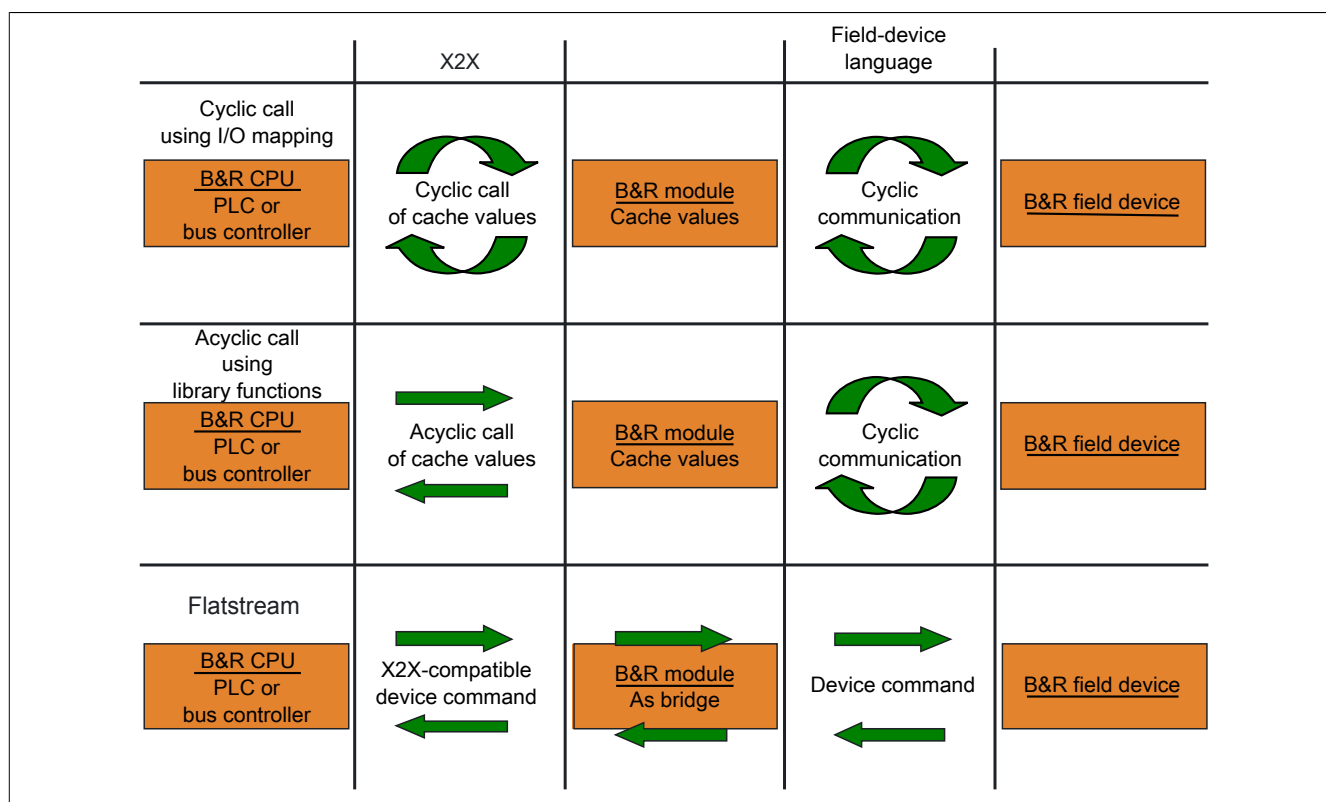


Figure 1: 3 types of communication

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.

### 11.9.2 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.

### 11.9.3 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.

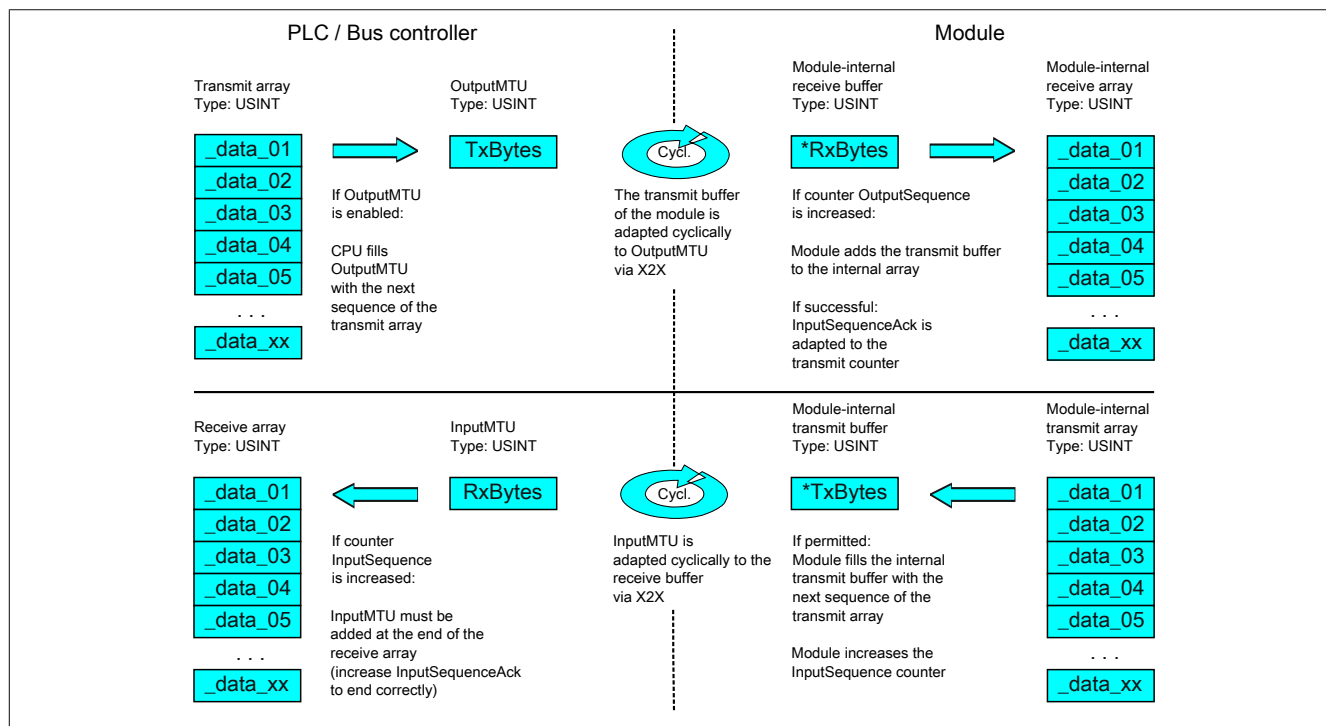


Figure 2: Flatstream communication

#### 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 once they have been completely transferred.

### 11.9.4 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.

#### 11.9.4.1 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.

##### 11.9.4.1.1 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).



### 11.9.4.2 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.

#### 11.9.4.2.1 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.

#### 11.9.4.2.2 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

#### 11.9.4.2.3 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	Description	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 of the 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 to be 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

**11.9.4.2.4 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	Description	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.

### 11.9.4.2.5 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	Description	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.

#### 11.9.4.2.6 Relationship between OutputSequence and InputSequence

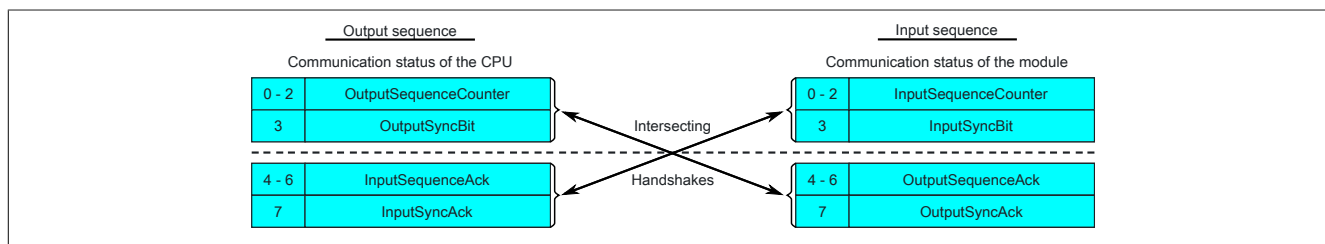


Figure 3: Relationship between OutputSequence and InputSequence

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.**

### 11.9.4.3 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.

11.9.4.4 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.

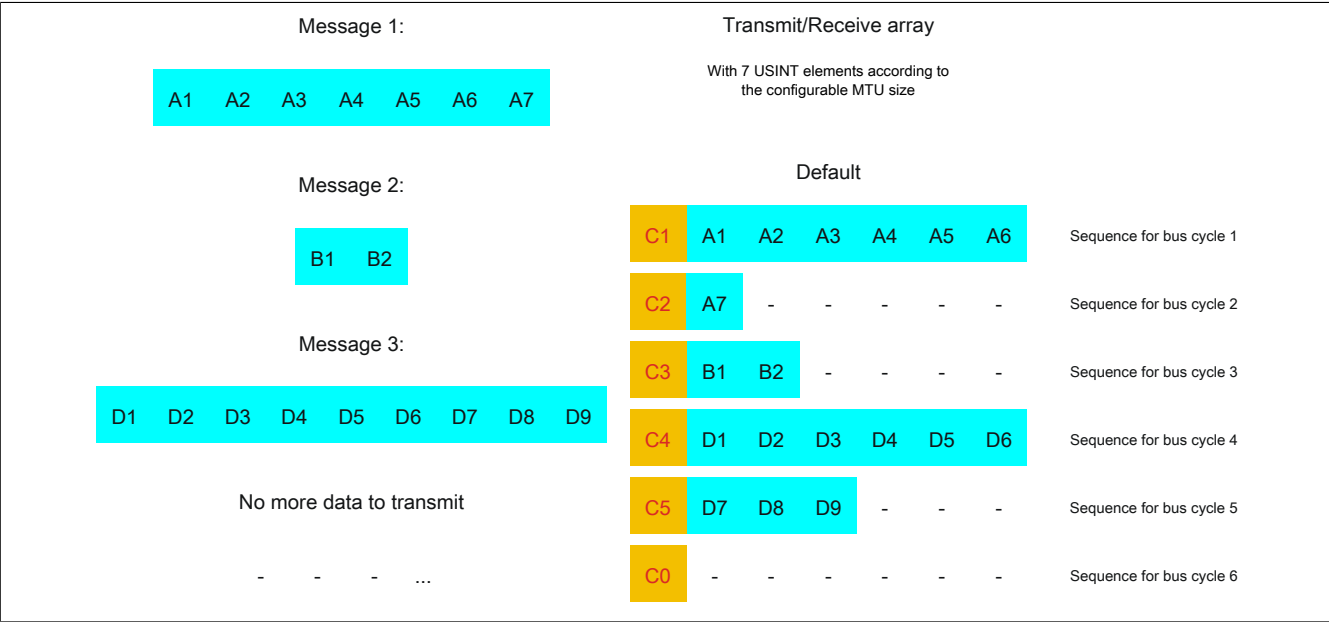


Figure 4: Transmit/Receive array (default)

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)

### 11.9.4.5 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.

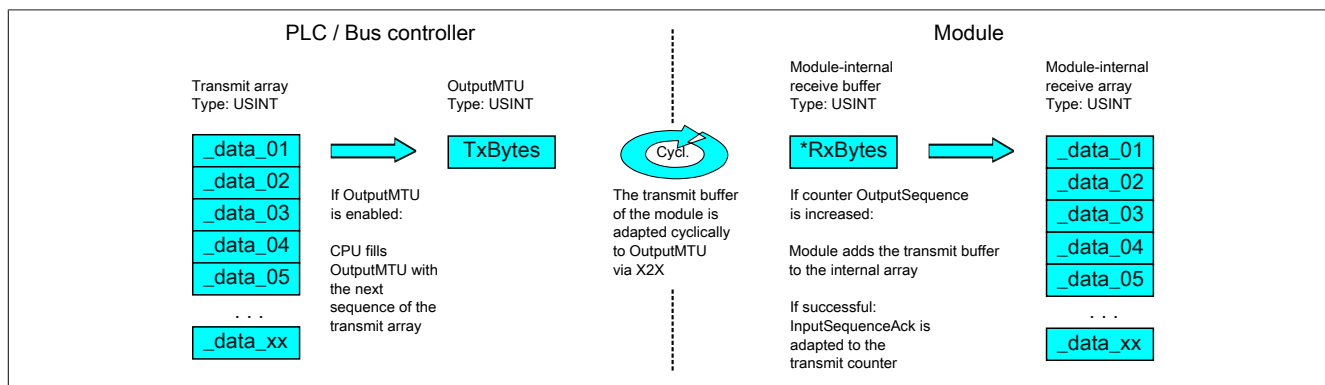


Figure 5: Flatstream communication (output)

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>→ The 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 flow chart

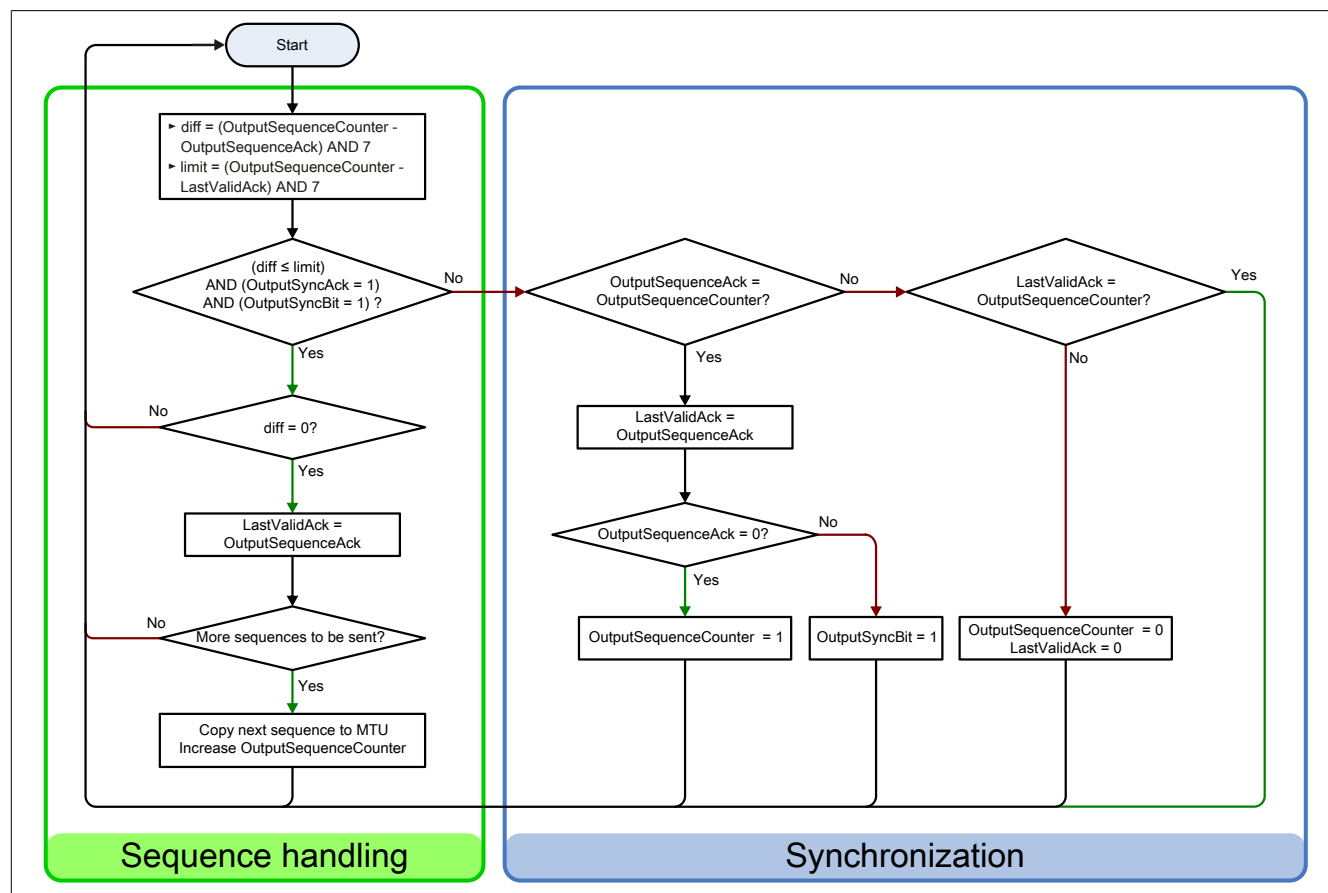


Figure 6: Flow chart for the output direction

### 11.9.4.6 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 does not change in this regard.

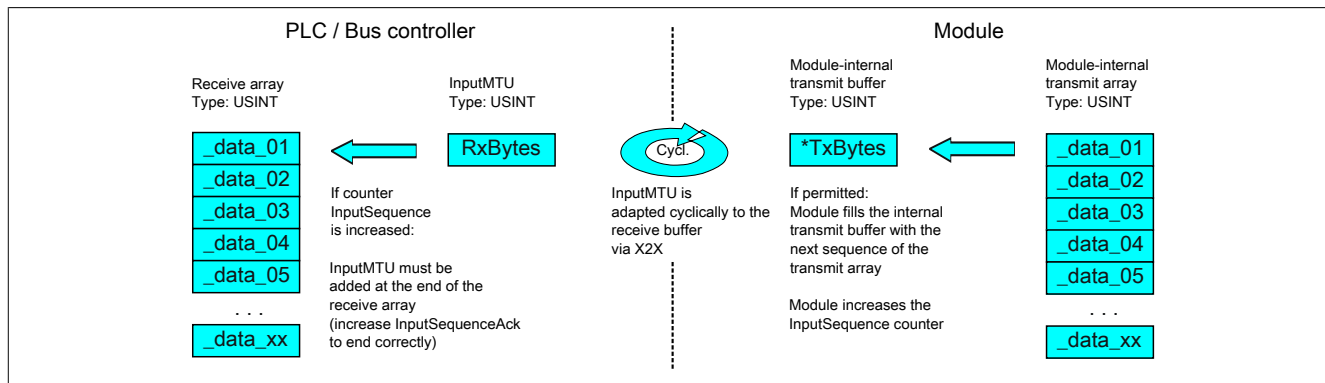


Figure 7: Flatstream communication (input)

#### 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:*

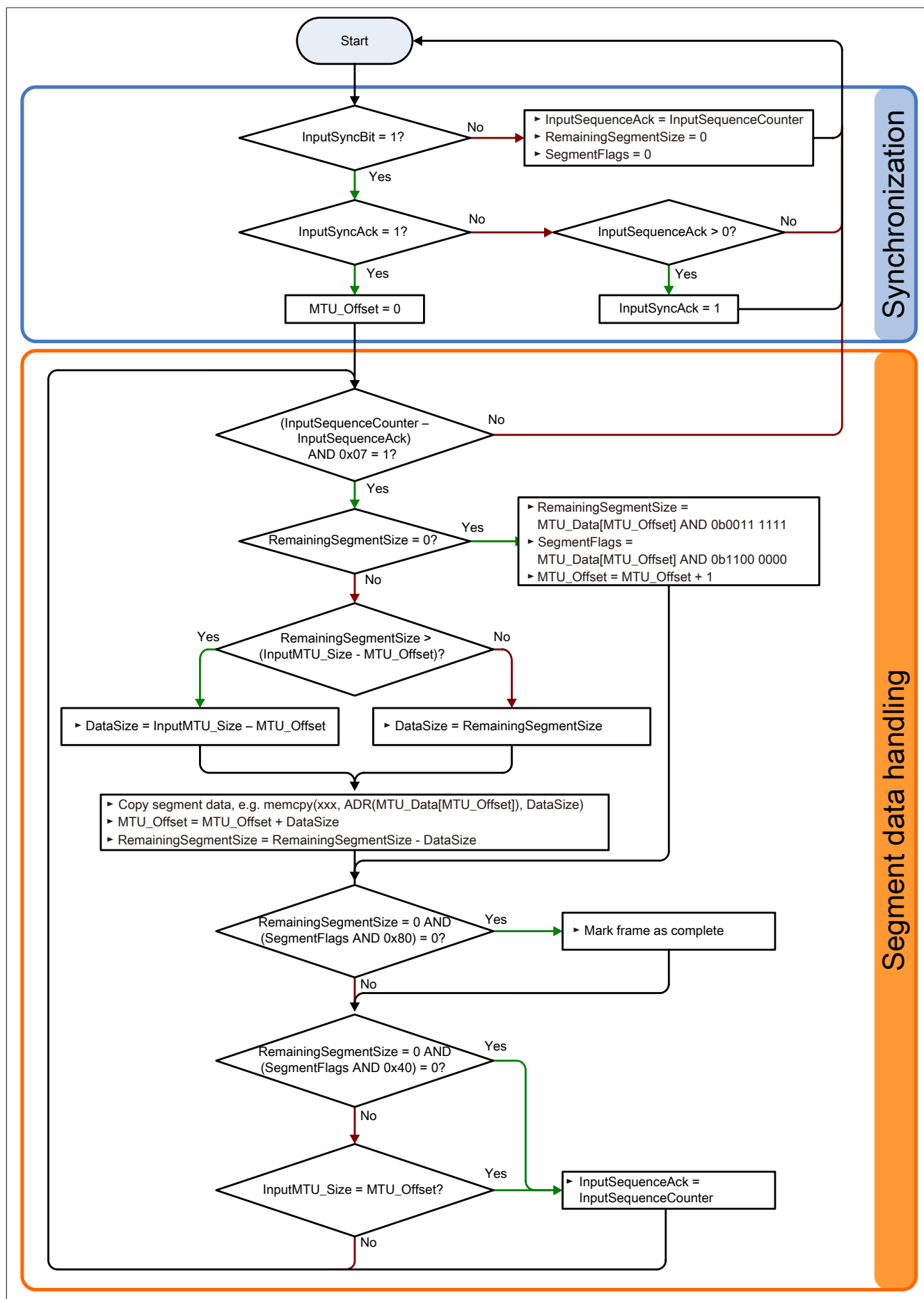
- The module transfers the current element of the internal transmit array to the internal transmit buffer.
- The module increases InputSequenceCounter.

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.



#### 11.9.4.7 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 once the channel has been resynchronized.

### 11.9.4.8 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	Description	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.

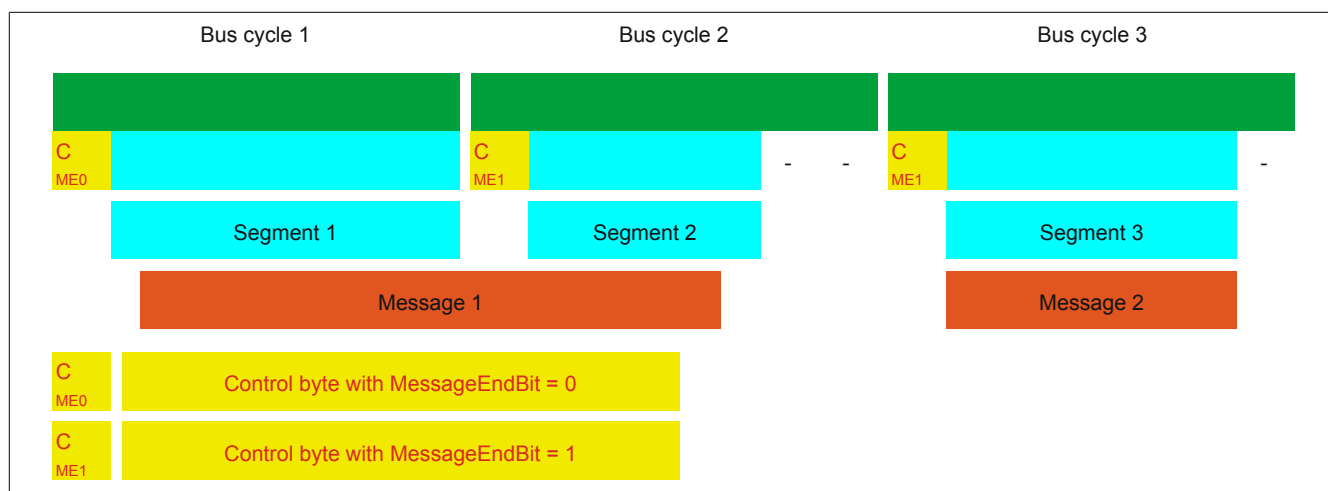


Figure 9: Message arrangement in the MTU (default)

### 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.

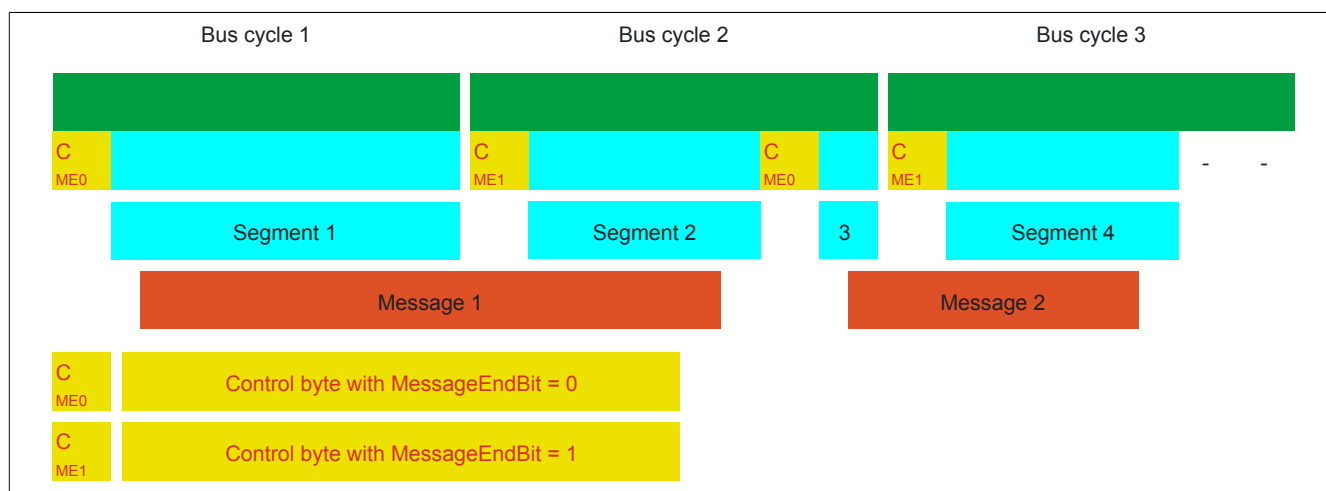


Figure 10: Arrangement of messages in the MTU (MultiSegmentMTUs)

### 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.

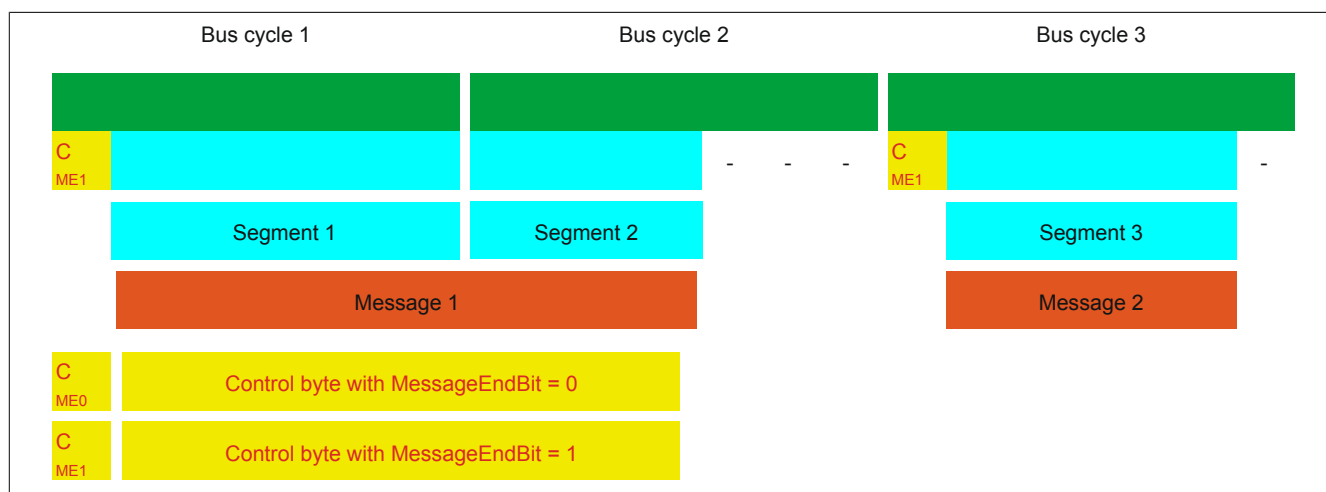


Figure 11: Arrangement of messages in the MTU (large segments)

## Using both options

Using both options at the same time is also permitted.

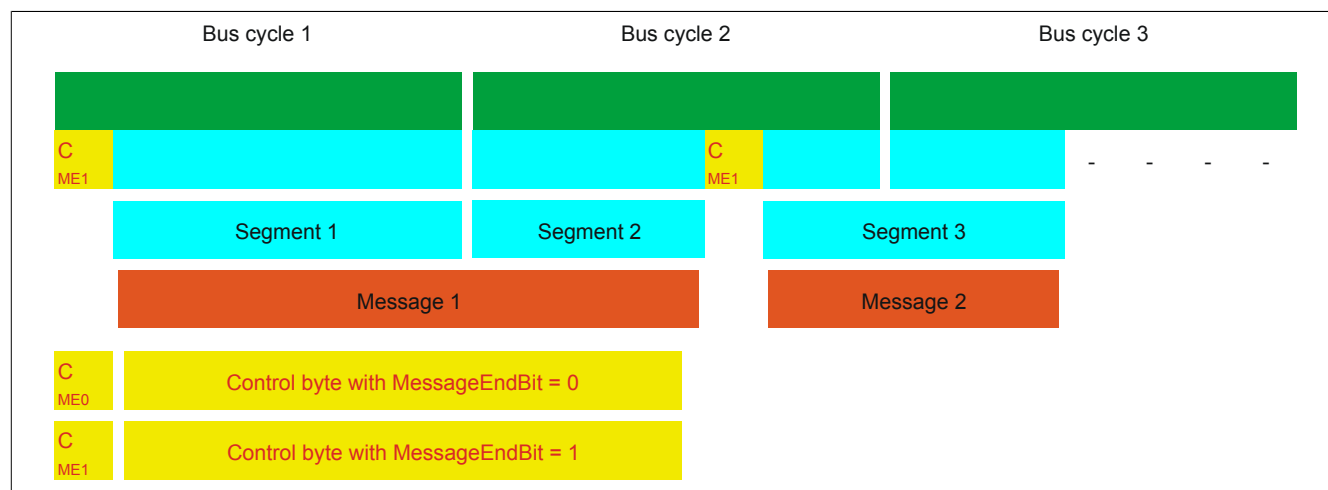


Figure 12: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

11.9.4.9 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.

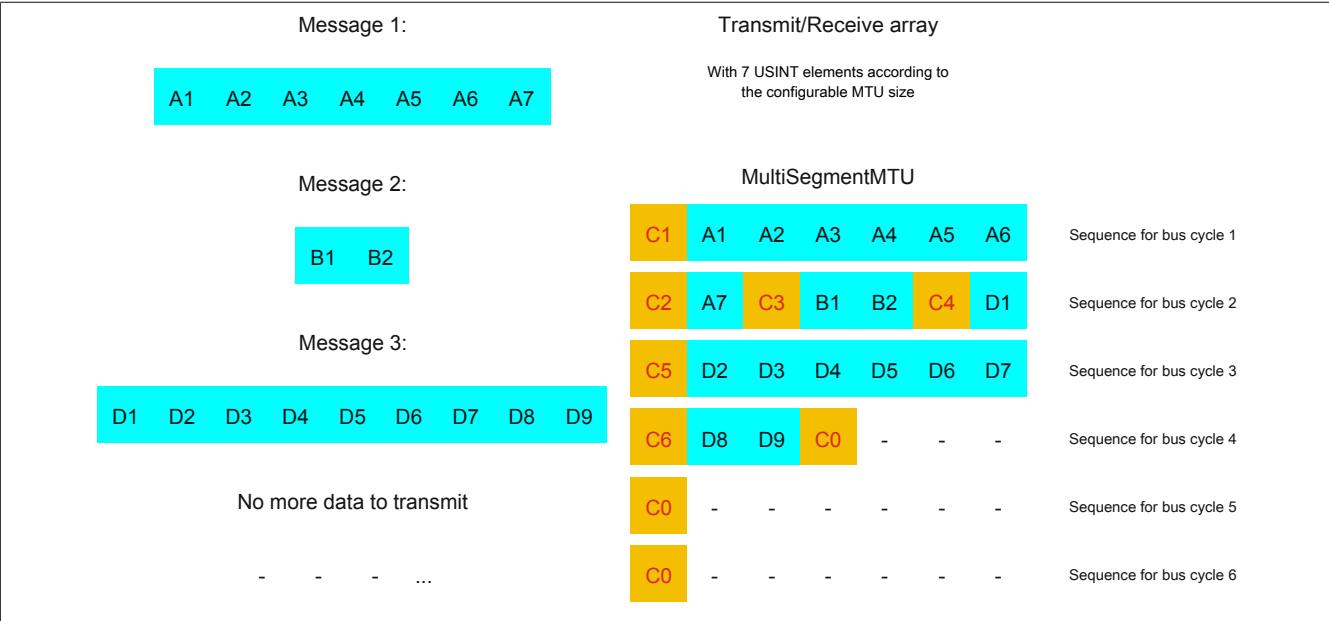


Figure 13: Transmit/receive array (MultiSegmentMTUs)



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.

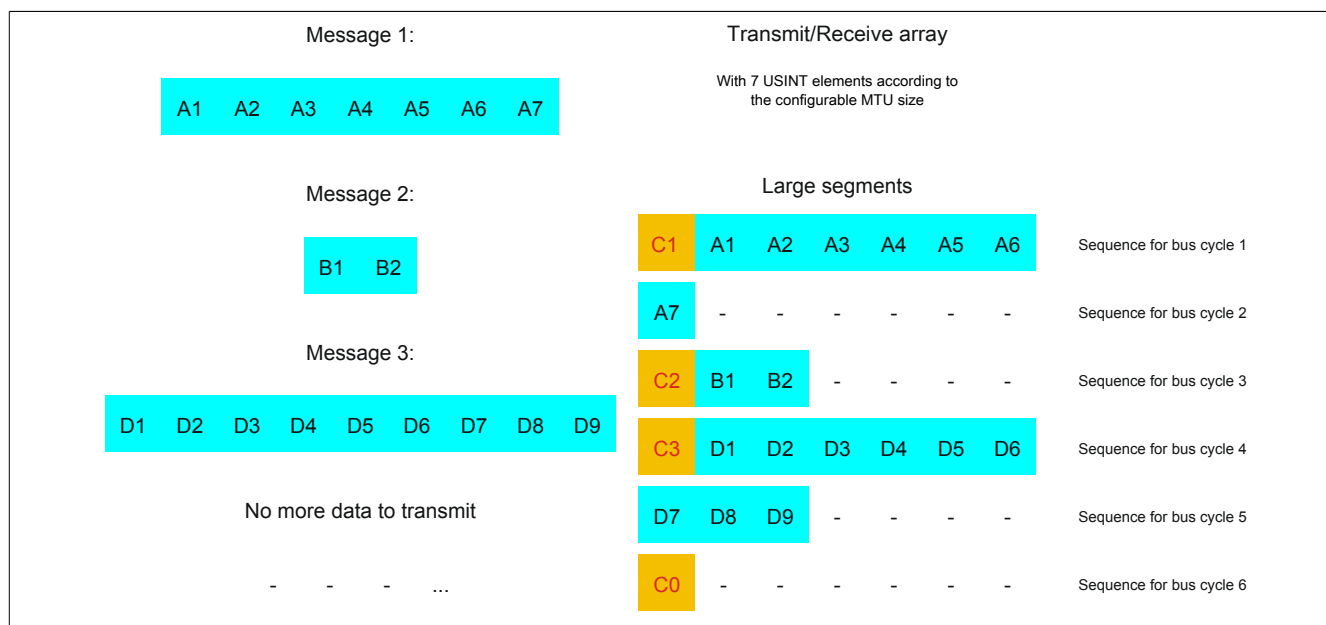


Figure 14: Transmit/receive array (large segments)

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.

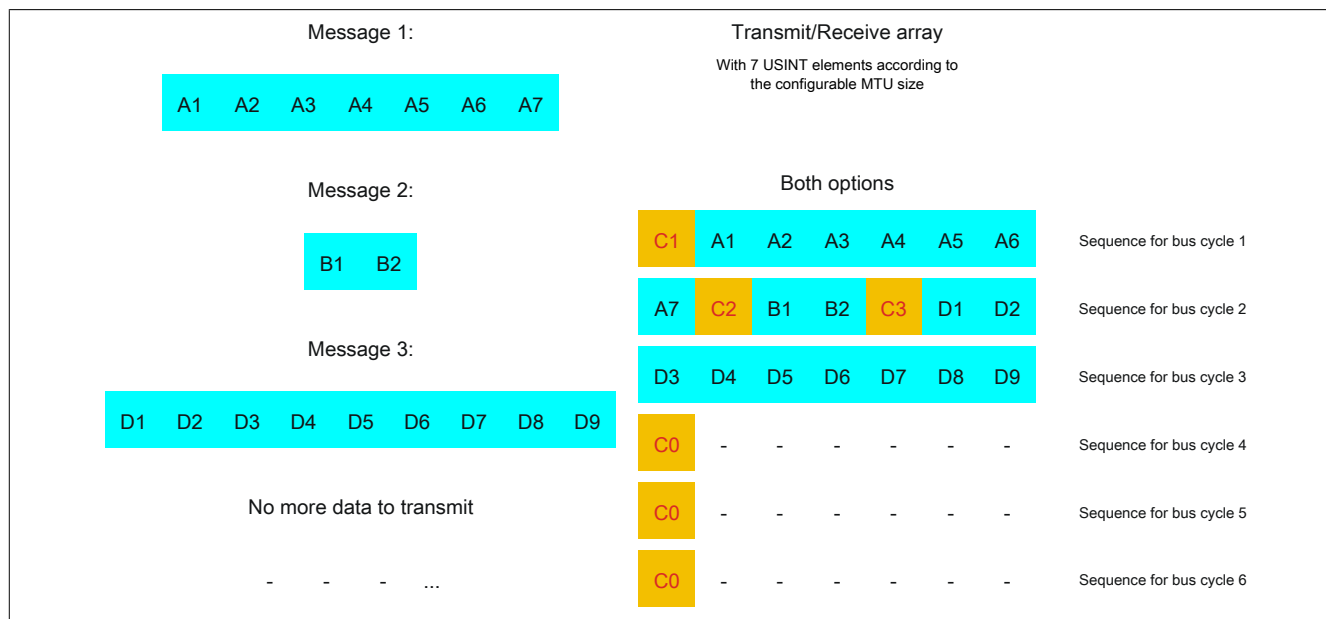


Figure 15: Transmit/receive array (large segments and MultiSegmentMTUs)

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

### 11.9.5 Example of Forward functionality on X2X Link

Forward functionality 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.

#### 11.9.5.1 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 matching of MTU and module buffer	Check SequenceAck
<b>Resource</b>	Sender (task to transmit)	Bus system (direction 1)	Recipient (task to receive)	Bus system (direction 2)	Sender (task for Ack checking)

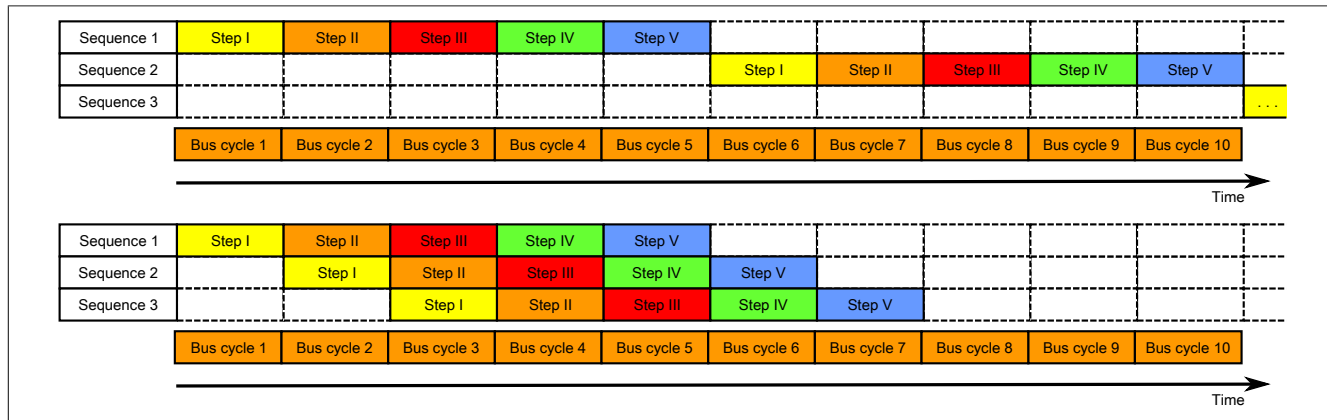


Figure 16: Comparison of transfer without/with Forward

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.

### 11.9.5.2 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.

#### 11.9.5.2.1 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

#### 11.9.5.2.2 Delay time

Name:  
ForwardDelay

Register "ForwardDelay" is used to specify the delay time in  $\mu\text{s}$ . 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 [ $\mu\text{s}$ ] Default: 0

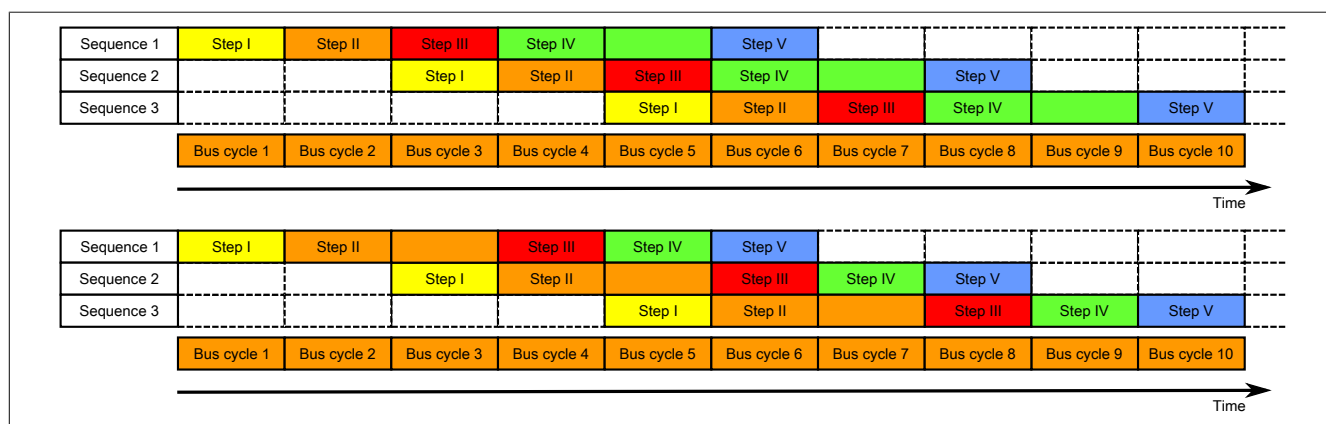


Figure 17: Effect of ForwardDelay when using Flatstream communication with the Forward function

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.

### 11.9.5.3 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> 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.

#### 11.9.5.4 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).

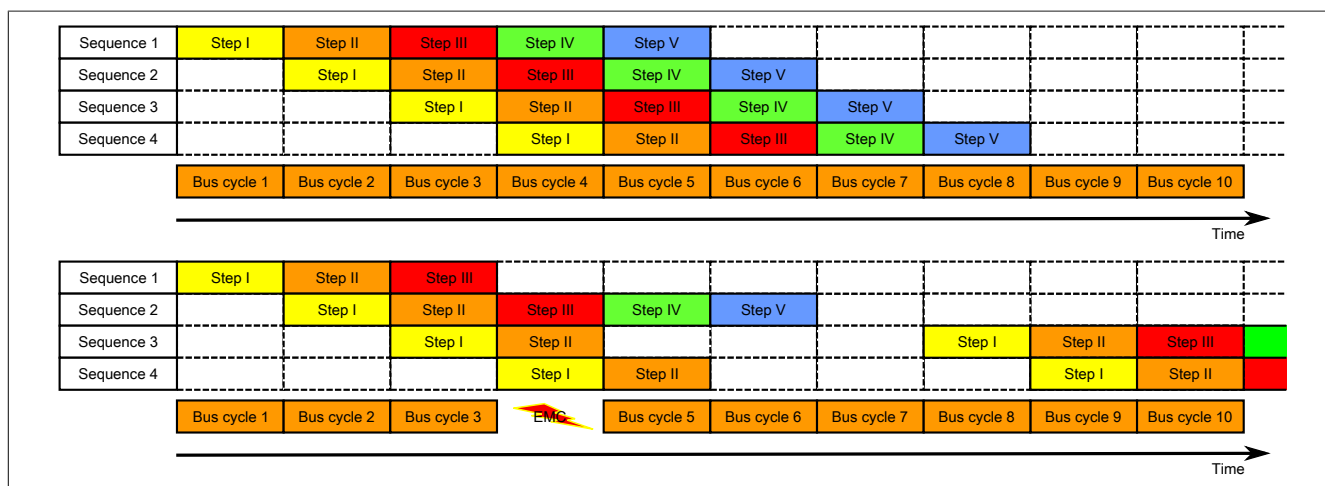


Figure 18: Effect of a lost bus cycle

##### 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.



## 11.10 EnDat with FlatStream

EnDat is a synchronous interface capable of half-duplex communication. Various features have been included to ensure that signals are transmitted without errors.

- An automatically generated checksum is sent together with a signal and evaluated by the recipient.
- The command which the encoder is responding to is repeated at the start of a response.

In Flatstream mode, the module acts as a bridge between the CPU and the EnDat slave. EnDat-specific algorithms were implemented to monitor timeouts and handle checksums. During normal operation, the user does not have access to these details.

More detailed information can be found in the documentation "Technical Information - EnDat 2.2" and the encoder's manufacturer data.

### 11.10.1 Overview of conventional EnDat commands for the Flatstream mode

Command byte [hex]	Command	EnDat 2.2 only
0x00	<a href="#">Reset</a>	
0x01	<a href="#">Acknowledge error</a>	
0x04	<a href="#">Read parameter</a>	
0x05	<a href="#">Write parameter</a>	
0x06	<a href="#">Read parameter from memory block</a>	•
0x07	<a href="#">Write parameter to memory block</a>	•
0x08	<a href="#">Read word 1 from additional information</a>	•
0x09	<a href="#">Read word 2 from additional information</a>	•
0x0A	<a href="#">Read word 3 from additional information</a>	•
0x0B	<a href="#">Read word 4 from additional information</a>	•

### 11.10.2 Reset (0x00)

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x00	Command (Reset)
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x00	Repetition (safety)
Master		

### 11.10.3 Acknowledge error (0x01)

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x01	Command (Acknowledge error)
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x01	Repetition (safety)
Master		

### 11.10.4 Read parameter (0x04)

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x04	Command (read parameter)
2	MRS code	Memory area to read
3	Parameter no.	
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x04	Repetition (safety)
2	MRS code	
3	Parameter no.	
4	Value_L	Value read
5	Value_H	
Master		

### 11.10.5 Write parameter (0x05)

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x05	Command (write parameter)
2	MRS code	Memory area to write to
3	Parameter no.	
4	Value_L	Value to be written
5	Value_H	
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x05	Repetition (safety)
2	MRS code	
3	Parameter no.	
Master		

### 11.10.6 Read parameter from memory block (0x06)

#### Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x06	Command (read parameter from memory block)
2	MRS code	Memory area to read
3	Block no.	
4	Parameter no.	
Slave		

#### Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x06	Repetition (safety)
2	MRS code	
3	Block no.	
4	Parameter no.	
5	Value_L	Value read
6	Value_H	
Master		

### 11.10.7 Write parameter in memory block (0x07)

#### Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x07	Command (write parameter in memory block)
2	MRS code	Memory area to write to
3	Block no.	
4	Parameter no.	
5	Value_L	Value to be written
6	Value_H	
Slave		

#### Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x07	Repetition (safety)
2	MRS code	
3	Block no.	
4	Parameter no.	
Master		

**11.10.8 Read word 1 from additional information (0x08)**

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x08	Command (read word 1 from additional information)
2	MRS code	Memory area to read
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x08	Repetition (safety)
2	MRS code	
3	Value_L	Word 1 from additional information
4	Value_H	
Master		

**11.10.9 Read word 2 from additional information (0x09)**

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x09	Command (read word 2 from additional information)
2	MRS code	Memory area to read
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x09	Repetition (safety)
2	MRS code	
3	Value_L	Read word 1 from additional information (overhead)
4	Value_H	
5	Value_L	Word 2 from additional information
6	Value_H	
Master		

**11.10.10 Read word 3 from additional information (0x0A)**

Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x0A	Command (read word 3 from additional information)
2	MRS code	Memory area to read
Slave		

Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x0A	Repetition (safety)
2	MRS code	
3	Value_L	Read word 1 from additional information (overhead)
4	Value_H	
5	Value_L	Read word 2 from additional information (overhead)
6	Value_H	
7	Value_L	Word 3 from additional information
8	Value_H	
Master		

### 11.10.11 Read word 4 from additional information (0x0B)

#### Master command

Protocol bytes		Information
No.	Name	
Master		
1	0x0B	Command (read word 4 from additional information)
2	MRS code	Memory area to read
Slave		

#### Slave response

Protocol bytes		Information
No.	Name	
Slave		
1	0x0B	Repetition (safety)
2	MRS code	
3	Value_L	Read word 1 from additional information (overhead)
4	Value_H	
5	Value_L	Read word 2 from additional information (overhead)
6	Value_H	
7	Value_L	Read word 3 from additional information (overhead)
8	Value_H	
9	Value_L	Word 4 from additional information
10	Value_H	
Master		

## 11.11 NetTime technology

NetTime refers to the ability to precisely synchronize and transfer system times between individual components of the controller or network (CPU, I/O modules, X2X Link, POWERLINK, etc.).

This allows the time that events occur to be determined system-wide with microsecond precision. Upcoming events can also be executed precisely at a given time.



### 11.11.1 Time information

Various time information is available in the controller or on the network:

- System time (on the PLC, Automation PC, etc.)
- X2X Link time (for each X2X Link network)
- POWERLINK time (for each POWERLINK network)
- Time data points of I/O modules

The NetTime is based on 32-bit counters, which are increased with  $\mu\text{s}$  timing. The sign of the time information changes after 35 min, 47 s, 483 ms and 648  $\mu\text{s}$ ; an overflow occurs after 71 min, 34 s, 967 ms and 296  $\mu\text{s}$ .

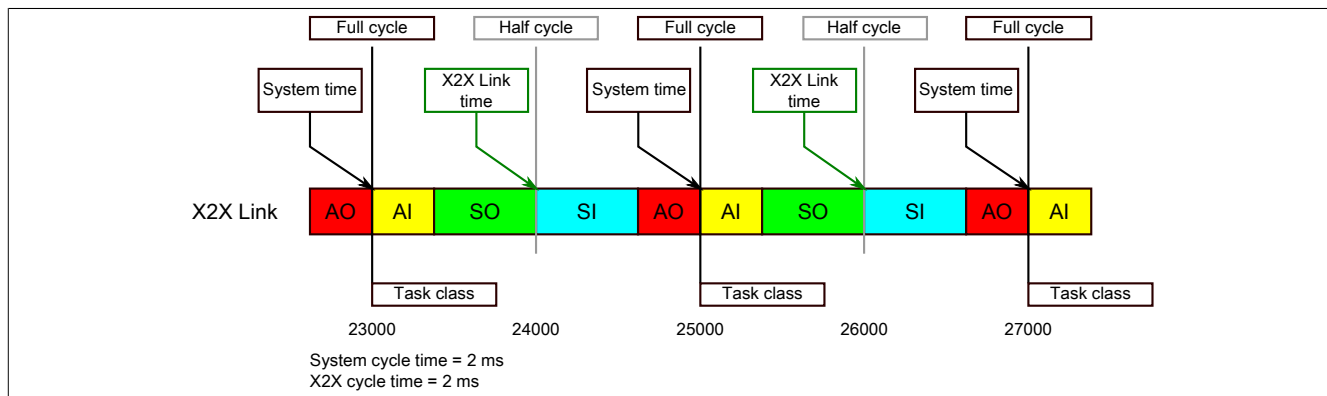
The initialization of the times is based on the system time during the startup of the X2X Link, the I/O modules or the POWERLINK interface.

Current time information in the application can also be determined via library AslOTime.

#### 11.11.1.1 PLC/Controller data points

The NetTime I/O data points of the PLC or the controller are latched to each system clock and made available.

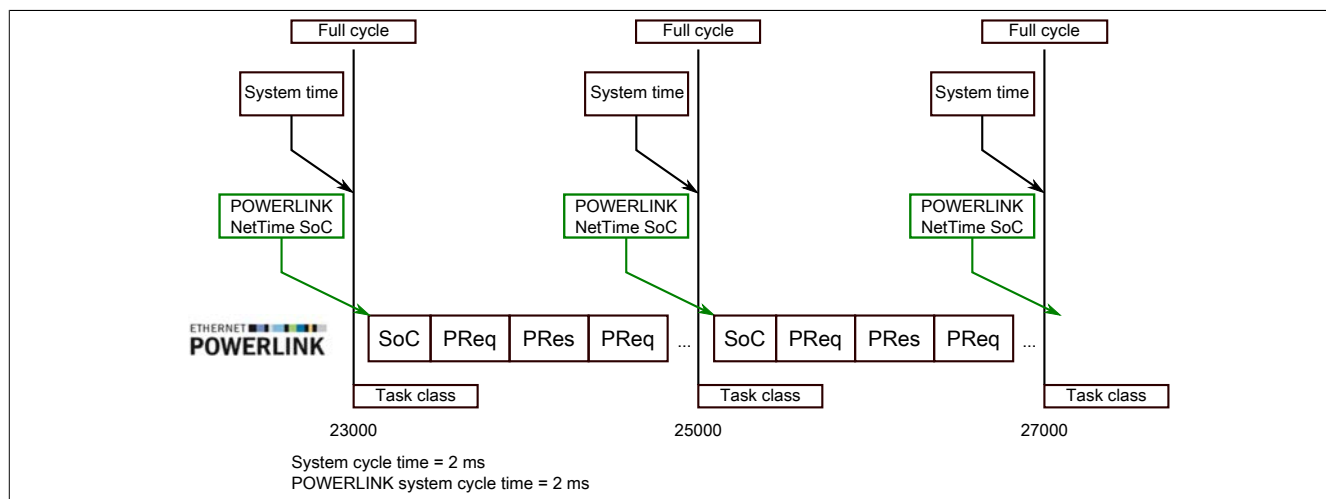
#### 11.11.1.2 X2X Link reference time



The reference time on the X2X Link network is always formed at the half cycle of the X2X Link cycle. This results in a difference between the system time and the X2X Link reference time when the reference time is read out.

In the example above, this results in a difference of 1 ms, i.e. if the system time and X2X Link reference time are compared at time 25000 in the task, then the system time returns the value 25000 and the X2X Link reference time returns the value 24000.

### 11.11.1.3 POWERLINK reference time

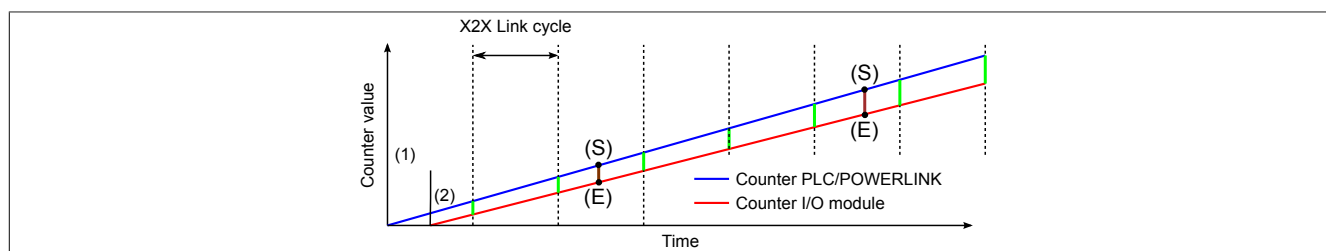


The reference time at POWERLINK is always formed at the SoC (Start of Cycle) of the POWERLINK network. The SoC starts 20 μs after the system tick. This results in the following difference between the system time and the POWERLINK reference time:

POWERLINK reference time = System time - POWERLINK cycle time + 20 μs.

In the example above, this means a difference of 1980 μs, i.e. if the system time and POWERLINK reference time are compared at time 25000 in the task, then the system time returns the value 25000 and the POWERLINK reference time returns the value 23020.

### 11.11.1.4 Synchronization of system time/POWERLINK time and I/O module



At startup, the internal counters for the PLC/POWERLINK (1) and the I/O module (2) start at different times and increase the values at μs intervals.

At the beginning of each X2X Link cycle, the PLC or the POWERLINK network sends time information to the I/O module. The I/O module compares this time information with the module's internal time and forms a difference (green line) between the two times and stores it.

When a NetTime event (E) occurs, the internal module time is read out and corrected with the stored difference value (brown line). This means that the exact system time (S) of an event can always be determined, even if the counters are not absolutely synchronous.

#### Note

The deviation from the clock signal is strongly exaggerated in the picture as a red line.

### 11.11.2 Timestamp functions

NetTime-capable modules provide various timestamp functions depending on the scope of functions. If a timestamp event occurs, the module immediately saves the current NetTime. After the respective data is transferred to the CPU, including this precise time, the CPU can then evaluate the data using its own NetTime (or system time), if necessary.

#### 11.11.2.1 Time-based inputs

NetTime Technology can be used to determine the exact time of a rising edge at an input. The rising and falling edges can also be detected and the duration between 2 events can be determined.

#### **Information:**

**The determined time always lies in the past.**

#### 11.11.2.2 Time-based outputs

NetTime Technology can be used to specify the exact time of a rising edge at an output. The rising and falling edges can also be specified and a pulse pattern generated from them.

#### **Information:**

**The specified time must always be in the future and the set X2X Link cycle time must be taken into account for the definition of the time.**

#### 11.11.2.3 Time-based measurements

NetTime Technology can be used to determine the exact time of a measurement that has taken place. Both the start and the end time of the measurement can be transmitted.

### 11.12 Minimum I/O update time

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

Minimum I/O update time
100 µs

### 11.13 Minimum cycle time

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

Minimum cycle time
100 µs