X20CM4800X

1 Order data

Model number	Short description	Figure
	Other functions	
X20CM4800X	X20 analog input module, vibration measurement, 4 IEPE analog inputs, 50 kHz sampling frequency, 24-bit converter resolution	
	Required accessories	
	Bus modules	
X20BM11	X20 bus module, 24 VDC keyed, internal I/O supply continuous	
	Terminal blocks	
X20TB12	X20 terminal block, 12-pin, 24 VDC keyed	
	Optional accessories	
	Sensor cable	
0ACC0020.01-1	Cable for accelerometer, length 2 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
0ACC0050.01-1	Cable for accelerometer, length 5 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
0ACC0100.01-1	Cable for accelerometer, length 10 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
0ACC0150.01-1	Cable for accelerometer, length 15 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
0ACC0200.01-1	Cable for accelerometer, length 20 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
0ACC0500.01-1	Cable for accelerometer, length 50 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
0ACC1000.01-1	Cable for accelerometer, length 100 m, 2x 0.34 mm², female M12 connector on the sensor side, can be used in cable drag chains, UL listed	
	Sensors	
0ACS100A.00-1	Accelerometer, nominal sensitivity 100 mV/g, top exit	
0ACS100A.90-1	Accelerometer, nominal sensitivity 100 mV/g, side exit	

Table 1: X20CM4800X - Order data

2 Module description

The module is intended for vibration measurement on machines and systems as well as for further evaluation of the data on the controller. The module has 4 input channels, whereby the selected sampling rate per input can be set between 200 Hz and 50 kHz.

Functions:

- Configuration of inputs
- Vibration measurement
- NetTime timestamp

If a measurement is started, the vibrations are measured with the set sampling and transferred with a configurable data resolution of 16, 24 or 32 bits.

A NetTime timestamp is generated when the measurement is started. This allows each recorded measured value to be assigned to a unique time. If the measurement is stopped, an additional NetTime timestamp is generated.

3 Technical data

Model number	**X20CM4800X**
Short description	
I/O module	X20 4-channel analog input module for vibration measurement of condition monitoring tasks
General information	
Isolation voltage between channel and bus	500 V_{eff}
Nominal voltage	24 VDC ±20%
B&R ID code	0xF1C5
Status indicators	Run, error, vibration inputs 1 to 4
Diagnostics	
Module run/error	Yes, using LED status indicator and software
Power consumption	
Bus	0.01 W
Internal I/O	1.5 W
Certifications	
CE	Yes
EAC	Yes
Analog inputs	
Quantity	4
Input type	IEPE sensor: Acceleration
Digital converter resolution	24-bit
Open-circuit detection	
Per channel	Less than 8 V or greater than 14 V for more than 4 s
Permissible input signal	±10 VAC
Conversion procedure	Sigma-delta
Type	Vibration input
Sampling frequency	Configurable from 200 Hz to 50 kHz
Input high pass cutoff frequency	34 mHz
Input low pass cutoff frequency	23 kHz
Sensor power supply	IEPE, 5 mA constant current source (4.9 - 5.5 mA), can be switched off for each channel
Electrical properties	
Electrical isolation	Channel isolated from bus Channel not isolated from channel
Operating conditions	
Mounting orientation	
Horizontal	Yes
Vertical	Yes
Installation elevation above sea level	
0 to 2000 m	No limitation
>2000 m	Reduction of ambient temperature by 0.5°C per 100 m
Degree of protection per EN 60529	IP20
Ambient conditions	
Temperature	
Operation	
Horizontal mounting orientation	-25 to 60°C
Vertical mounting orientation	-25 to 55°C
Derating	See section "Derating".
Storage	-40 to 85°C
Transport	-40 to 85°C
Relative humidity	
Operation	5 to 95%, non-condensing
Storage	5 to 95%, non-condensing
Transport	5 to 95%, non-condensing
Mechanical properties	
Note	Order 1x terminal block X20TB12 separately. Order 1x bus module X20BM11 separately.
Pitch	12.5 $^{+0.2}$ mm

Table 2: X20CM4800X - Technical data

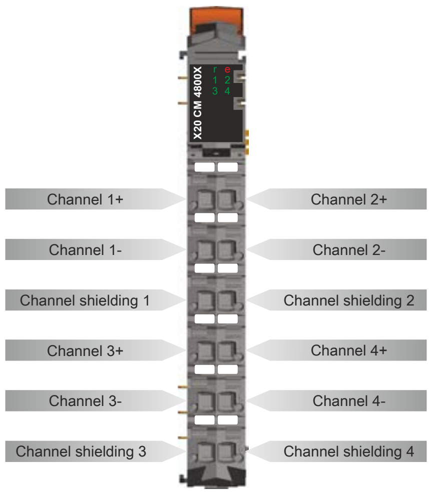
4 LED status indicators

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

Figure	LED	Color	Status	Description
	r	Green	Off	No power to module
			Single flash	Mode RESET
			Double flash	Mode BOOT (during firmware update)[1]
			Blinking	Mode PREOPERATIONAL
			On	Mode RUN
	e	Red	Off	Module not supplied with power or everything OK
			On	Warning, error or reset state
			Single flash	Open circuit on the active channel
			Double flash	Buffer overflow
	e + r	Red Green	On Single flash	Invalid firmware
	1 - 4	Green	Off	Channel inactive
			On	Channel active

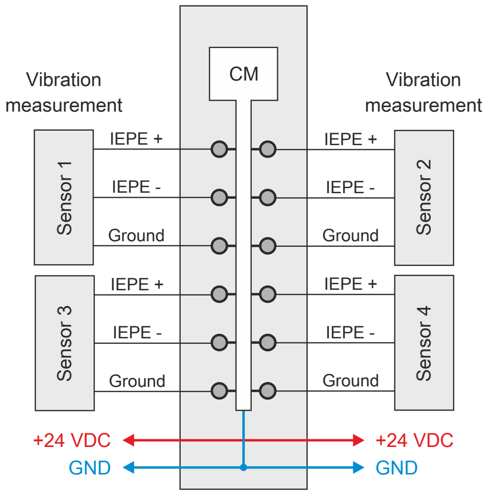
1) Depending on the configuration, a firmware update can take up to several minutes.

5 Pinout

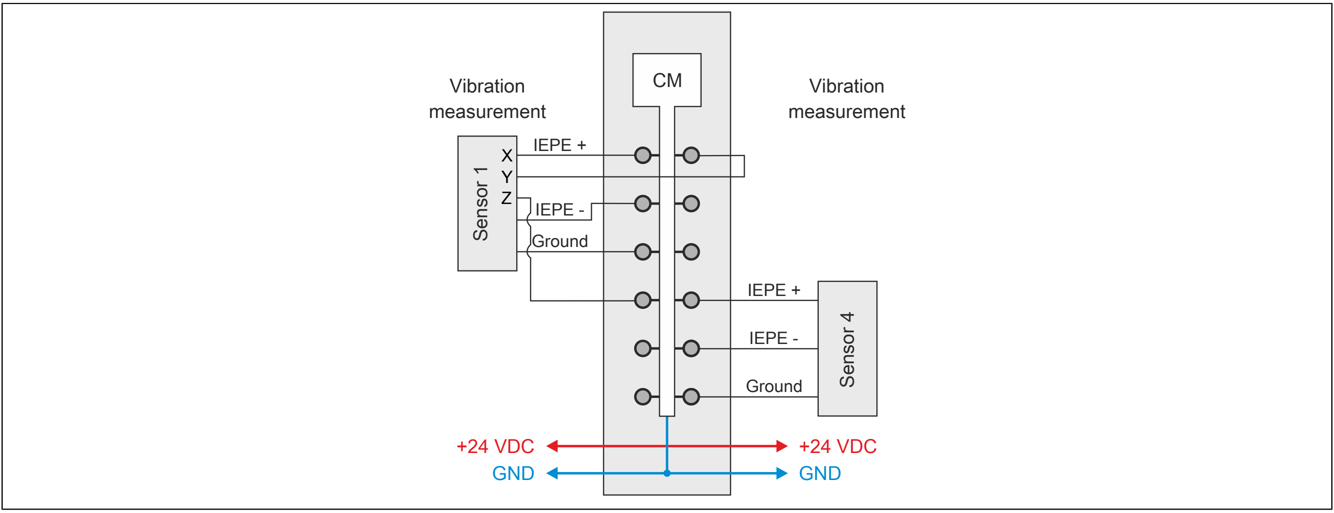


6 Connection example

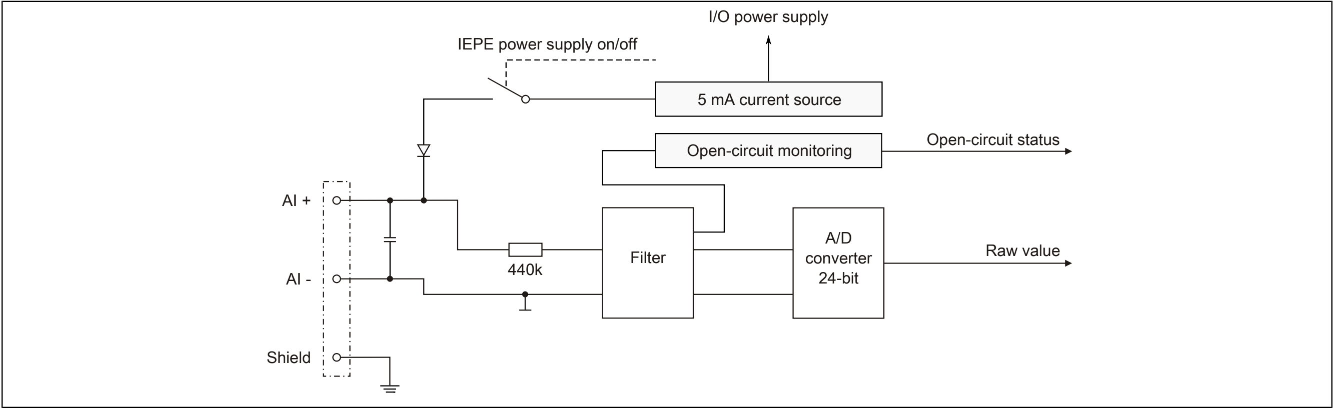
Connecting 1-axis sensors



Connecting 1-axis and 3-axis sensors

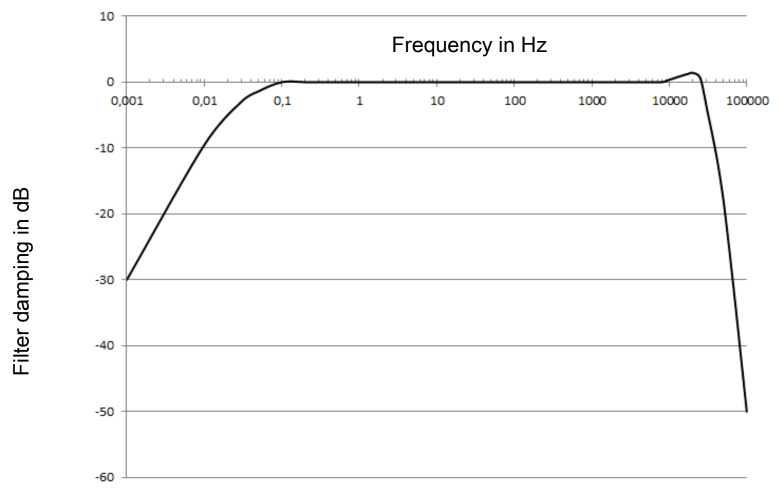


7 Input circuit diagram



8 Gain curve

The following diagram shows a typical gain curve for the module.



9 Derating

Depending on the maximum ambient temperature, additional dummy modules may need to be installed next to the module. If the modules are installed with a vertical mounting orientation, the dummy modules must already be used at a 5°C lower ambient temperature.

Maximum ambient temperature

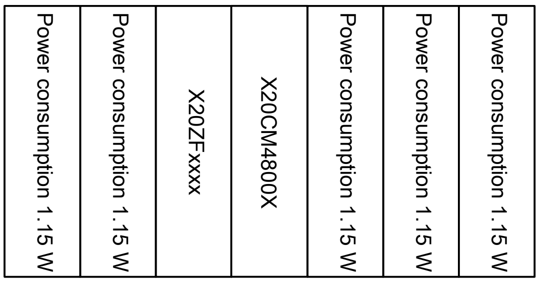
Up to: horizontal 45°C / vertical 40°C

- No precautions need be taken.

Up to: horizontal 50°C / vertical 45°C

- Depending on the supply voltage used, it may be necessary to install a dummy module.

- 24 V supply voltage: No dummy module necessary
- 28.8 V supply voltage: 1 dummy module required in front of it

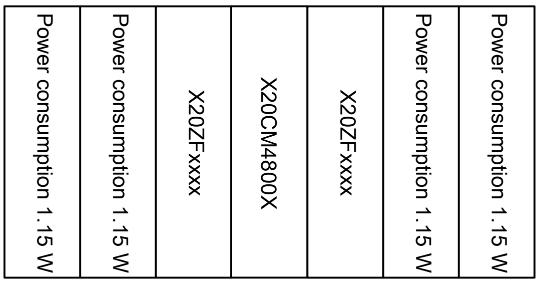


Up to: horizontal 55°C / vertical 50°C

- Always install 1 dummy module in front of it (see diagram above).

Up to: horizontal 60°C / vertical 55°C

- 2 dummy modules must always be installed.



10 Function description

10.1 Configuration of inputs

There is only a limited number of bytes available for transferring raw data to the X2X Link network that are distributed among all active channels. In order to make more bytes available on the channels used, unused channels can be switched off.

In addition, the IEPE sensor power supply can be enabled separately for each channel. If a sensor is connected to multiple channels, the power supply must be enabled for one channel only.

Information:

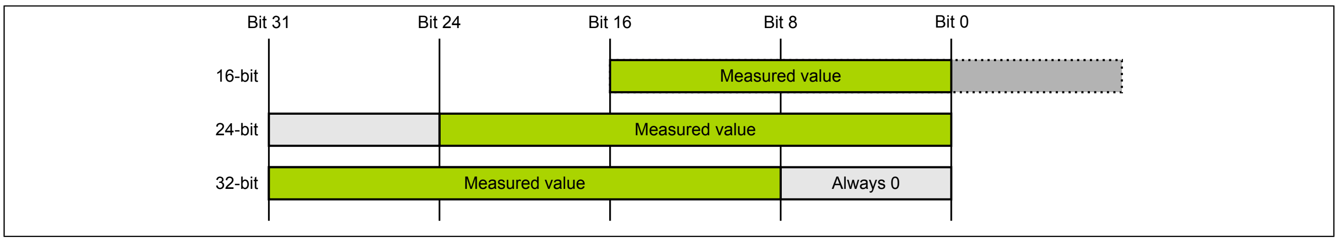
Registers are described in section "Channel configuration" on page 10.

10.1.1 Data resolution

The data is scaled by the module according to the set data resolution:

Mode	Data format	Maximum value +10 V	Minimum value -10 V	Scaling
16-bit	INT	32767 (0x7FFF)	-32767 (0x8000)	Shifted 8 bits to the right
24-bit	DINT	8388607 (0x7FFFFF)	-8388608 (0x800000)	Internal converter resolution of the module[1]
32-bit	DINT	2147483392 (0x7FFFFF00)	-2147483648 (0x80000000)	Shifted 8 bits to the left. Bits 0 to 7 are always 0.

1) The 3 bytes must be transferred from the application in a DINT. For negative values, bits 25 to 31 must be set to 1.



10.2 Vibration measurement

Up to 4 vibration sensors can be connected to the module. The raw data of the sensors is recorded with the set sampling rate and resolution.

However, processing and evaluation of the transferred data must be carried out in the application. For a description of how to convert raw data into a vibration value, see "Converting raw data to [mg]" on page 7.

Information:

There is no preprocessing of the data within the module.

10.2.1 Converting raw data to [mg]

The following formulas can be used to convert the raw value into a vibration value [mg]:

$$\text{Sensor resolution} = 0.0001 * \frac{V}{mg}$$

$$\text{Vibration value [mg]} = \text{Raw value} * \left(\frac{\text{MaximumResolution\_10V\_AnalogIn}}{\text{MaximumValue\_10V\_Digitalin}}\right) * \frac{1}{\text{Sensor resolution}}$$

Information:

The maximum resolution always depends on the sensor used. The module operates in the range of ±10 V. For a 100 mV/g sensor, this corresponds to a maximum value of ±100 g. With a 50 mV/g sensor, the maximum value is ±200 g.

Example

A 100 mV/g sensor is used; the module provides the value 4608 as a raw value and the data resolution is configured to 24 bits. This results in the following values for the calculation:

- Raw value = 4608
- MaximumValue_10V_AnalogIn= 10 V (module value, cannot be changed)
- MaximumValue_10V_Digitalin with 24 bits = 8388607 (see "Data resolution" on page 6)
- Sensor resolution = 100 mV/g = 0.1 V/g = 0.0001 V/mg

$$\text{Vibration value [mg]} = 4608 * \left(\frac{10\ V}{8388607}\right) * \frac{1}{0.0001\ V/mg} = 54.93\ mg$$

10.2.2 Flatstream

The data interface for the raw data is based on Flatstream communication. Operation takes place using library "AsFltGen".

Information:

Each channel has its own Flatstream.

For information about library "AsFltGen", see Automation Help.

The required bytes for transmitting the raw data at the Flatstream depend on the set sampling rate, the set data resolution and the bus cycle time used.

Information:

If the raw data is not transferred to the controller fast enough, it is buffered in the module per channel. As soon as the buffer is full, error message BufferOverflow occurs.

Calculation example for required number of bytes

The following settings are made for a measurement evaluation:

- Bus cycle time: 2 ms
- Sampling rate: 2 kHz (one value every 500 µs)
- Data resolution: 16 bits (2 bytes)

In the bus cycle time of 2 ms, this results in 4 values of 2 bytes each – a total of 8 bytes. The Flatstream data interface must be configured in accordance with this calculation.

If the bus cycle time is reduced from 2 ms to 1 ms, the number of bytes required for the raw data is reduced by half, i.e. 4 bytes. This means that other channels have more bytes available.

Setting the MTU size

The MTU size to be set for the Flatstream is equal to the calculated data bytes + 1 control byte. Therefore, the input MTU size must be configured to at least 8 + 1 = 9 bytes, or in the second case, to 4 + 1 = 5 bytes.

Since this is only the ideal value, reserve bytes must still be planned to compensate for any timing tolerances or communication errors. Otherwise, it could happen that not all data can be transferred in time, causing a buffer overflow at some point.

10.3 NetTime

If a measurement is started for a channel, a timestamp is automatically determined for the first raw value. The configured sampling rate can then be used to establish a unique time reference for each raw value.

In addition, a timestamp is automatically determined again even after the measurement has been completed.

Information:

The timestamps for the other raw values must be determined by the application. The module only provides the start or end timestamp.

Information:

Registers are described in section "NetTime" on page 12.

11 Register description

11.1 Function model 1 - Standard

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Configuration						
770	CfO_ChannelConfig01	UINT				●
	CfO_ChannelConfig01_rb			●		
774	CfO_ChannelConfig02	UINT				●
	CfO_ChannelConfig02_rb			●		
778	CfO_ChannelConfig03	UINT				●
	CfO_ChannelConfig03_rb			●		
782	CfO_ChannelConfig04	UINT				●
	CfO_ChannelConfig04_rb			●		
Communication						
1	Starting/stopping measurements and buffer overflow	USINT			●	
	Measurement01	Bit 0				
	...	..				
	Measurement04	Bit 3				
	BufferOverflowAck01	Bit 4				
	...	...				
	BufferOverflowAck04	Bit 7				
2	Status of the module	UINT	●			
	MeasurementState01	Bit 0				
	...	...				
	MeasurementState04	Bit 3				
	BufferOverflow01	Bit 4				
	...	...				
	BufferOverflow04	Bit 7				
	BrokenWire01	Bit 8				
	...	...				
	BrokenWire04	Bit 11				
NetTime						
788	Nettime_StartMeasCh01	DINT		●		
796	Nettime_StartMeasCh02	DINT		●		
804	Nettime_StartMeasCh03	DINT		●		
812	Nettime_StartMeasCh04	DINT		●		
820	Nettime_StopMeasCh01	DINT		●		
828	Nettime_StopMeasCh02	DINT		●		
836	Nettime_StopMeasCh03	DINT		●		
844	Nettime_StopMeasCh04	DINT		●		
Flatstream						
Channel 1						
1281	Ch1_CfO_OutputMTU	USINT				●
1283	Ch1_CfO_InputMTU	USINT				●
1285	Ch1_CfO_FlatstreamMode	USINT				●
1287	Ch1_CfO_Forward	USINT				●
1290	Ch1_CfO_ForwardDelay	UINT				●
1536	Ch1_CfO_InputSequence	USINT	●			
1536 + Index	Ch1_CfO_RxByteN (index N = 1 to 25)	USINT	●			
1536	Ch1_CfO_OutputSequence	USINT			●	
1536 + Index	Ch1_CfO_TxByteN (index N = 1 to 3)	USINT			●	
Channel 2						
1297	Ch2_CfO_OutputMTU	USINT				●
1299	Ch2_CfO_InputMTU	USINT				●
1301	Ch2_CfO_FlatstreamMode	USINT				●
1303	Ch2_CfO_Forward	USINT				●
1306	Ch2_CfO_ForwardDelay	UINT				●
1792	Ch2_CfO_InputSequence	USINT	●			
1792 + Index	Ch2_CfO_RxByteN (index N = 1 to 25)	USINT	●			
1792	Ch2_CfO_OutputSequence	USINT			●	
1792 + Index	Ch2_CfO_TxByteN (index N = 1 to 3)	USINT			●	
Channel 3						
1313	Ch3_CfO_OutputMTU	USINT				●
1315	Ch3_CfO_InputMTU	USINT				●
1317	Ch3_CfO_FlatstreamMode	USINT				●
1319	Ch3_CfO_Forward	USINT				●
1322	Ch3_CfO_ForwardDelay	UINT				●
2048	Ch3_CfO_InputSequence	USINT	●			
2048 + Index	Ch3_CfO_RxByteN (index N = 1 to 25)	USINT	●			
2048	Ch3_CfO_OutputSequence	USINT			●	
2048 + Index	Ch3_CfO_TxByteN (index N = 1 to 3)	USINT			●	

Register	Name	Data type	Read		Write	
			Cyclic	Acyclic	Cyclic	Acyclic
Channel 4						
1329	Ch4_CfO_OutputMTU	USINT				●
1331	Ch4_CfO_InputMTU	USINT				●
1333	Ch4_CfO_FlatstreamMode	USINT				●
1335	Ch4_CfO_Forward	USINT				●
1338	Ch4_CfO_ForwardDelay	UINT				●
2304	Ch4_CfO_InputSequence	USINT	●			
2304 + Index	Ch4_CfO_RxByteN (index N = 1 to 25)	USINT	●			
2304	Ch4_CfO_OutputSequence	USINT			●	
2304 + Index	Ch4_CfO_TxByteN (index N = 1 to 3)	USINT			●	

11.2 Configuration

11.2.1 Channel configuration

Name:
CfO_ChannelConfig01 to CfO_ChannelConfig04
CfO_ChannelConfig01_rb CfO_ChannelConfig04_rb

These registers can be used to configure the respective channels.

Data type	Values
UINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Enables/Disables channel	0	Channel disabled
		1	Channel enabled
1	Sensor power supply	0	Sensor power supply disabled
		1	Sensor power supply enabled
2 - 3	Data resolution[1]	0	32-bit
		1	24-bit
		2	16-bit
4 - 7	Sampling rate[2]	0	50000 Hz
		1	25000 Hz
		2	10000 Hz
		3	5000 Hz
		4	2500 Hz
		5	2000 Hz
		6	1000 Hz
		7	500 Hz
		8	200 Hz
8 - 15	Reserved	0	

1) The maximum or minimum value of the respective resolution corresponds to ±10 VAC.
2) The sampling rate of an analog signal with respect to 1 second. Specified in [Hz].
Examples:

- Sampling an analog signal once per second corresponds to a sampling rate of 1 Hz.
- Sampling an analog signal once per millisecond corresponds to a sampling rate of 1 kHz.

11.3 Communication

11.3.1 Starting/stopping measurements and buffer overflow

Name:
Measurement01 to Measurement04
BufferOverflowAck01 to BufferOverflowAck04

In this register, the measurements can be started or stopped. In addition, a potential buffer overflow can be acknowledged.

Data type	Values
USINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	Measurement01	0	Stop measurement of channel 1
		1	Start measurement of channel 1
...		...	
3	Measurement04	0	Stop measurement of channel 4
		1	Start measurement of channel 4
4	BufferOverflowAck01	0	Do not acknowledge channel 1 buffer overflow
		1	Acknowledge channel 1 buffer overflow
...		...	
7	BufferOverflowAck04	0	Do not acknowledge channel 4 buffer overflow
		1	Acknowledge channel 4 buffer overflow

11.3.2 Status of the module

Name:
MeasurementState01 to MeasurementState04
BufferOverflow01 to BufferOverflow04
BrokenWire01 to BrokenWire04

This register specifies the module status.

Data type	Values
UINT	See the bit structure.

Bit structure:

Bit	Description	Value	Information
0	MeasurementState01	0	Measurement of channel 1 stopped
		1	Measurement of channel 1 running
...		...	
3	MeasurementState04	0	Measurement of channel 4 stopped
		1	Measurement of channel 4 running
4	BufferOverflow01	0	No buffer overflow on channel 1
		1	Buffer overflow on channel 1
...		...	
7	BufferOverflow04	0	No buffer overflow on channel 4
		1	Buffer overflow on channel 4
8	BrokenWire01	0	No error on channel 1
		1	Open circuit - Channel 1
...		...	
11	BrokenWire04	0	No error on channel 4
		1	Open circuit - Channel 4
12 - 15	Reserved	-	

BufferOverflow

Each channel is equipped with an internal buffer of 50 kB. Depending on the set sampling rate, data resolution and MTU size, a buffer overflow occurs at the earliest after approx. 250 ms if the data is not transferred via Flatstream. The overflow must be acknowledged by the application using register BufferOverflowAck0x.

11.4 NetTime

11.4.1 Time of first valid scan

Name:
Nettime_StartMeasCh01 to Nettime_StartMeasCh04

The register writes the timestamp of the first valid sample after the measurements are started.

Data type	Values	Information
DINT	-2,147,483,648 to 2,147,483,647	Timestamp in µs

11.4.2 Time of last valid scan

Name:
Nettime_StopMeasCh01 to Nettime_StopMeasCh04

The register writes the timestamp of the last valid sample. The timestamp of the last valid sample of a measurement is available after measurement has ended.

Data type	Values	Information
DINT	-2,147,483,648 to 2,147,483,647	Timestamp in µs

11.4.3 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 moment that events occur to be determined system-wide with microsecond precision. Upcoming events can also be executed precisely at a specified moment.

NetTime TECHNOLOGY

11.4.3.1 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 microsecond resolution. The sign of the time information changes after 35 min, 47 s, 483 ms and 648 μs; an overflow occurs after 71 min, 34 s, 967 ms and 296 μs.

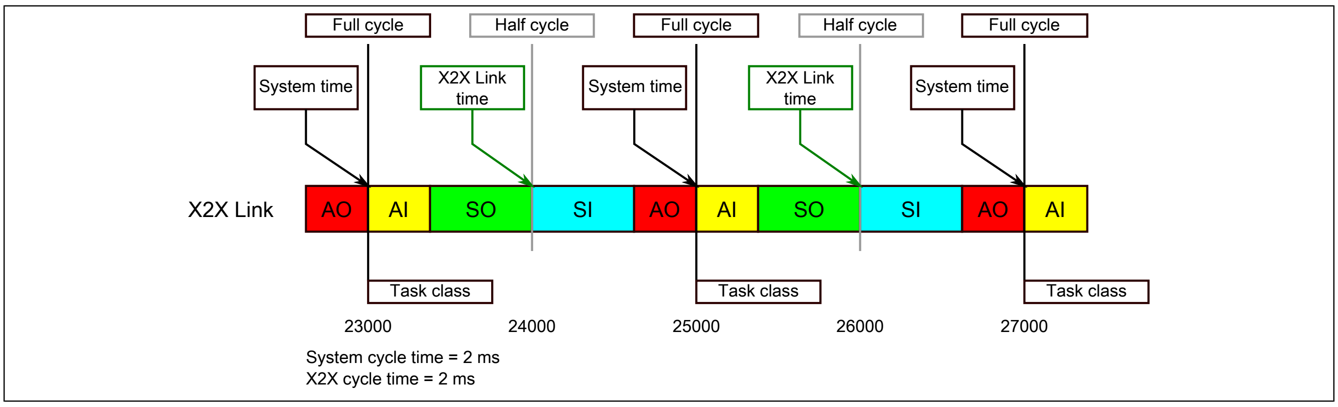
The initialization of the times is based on the system time during the startup of the X2X Link, the I/O modules or the POWERLINK interface.

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

11.4.3.1.1 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.

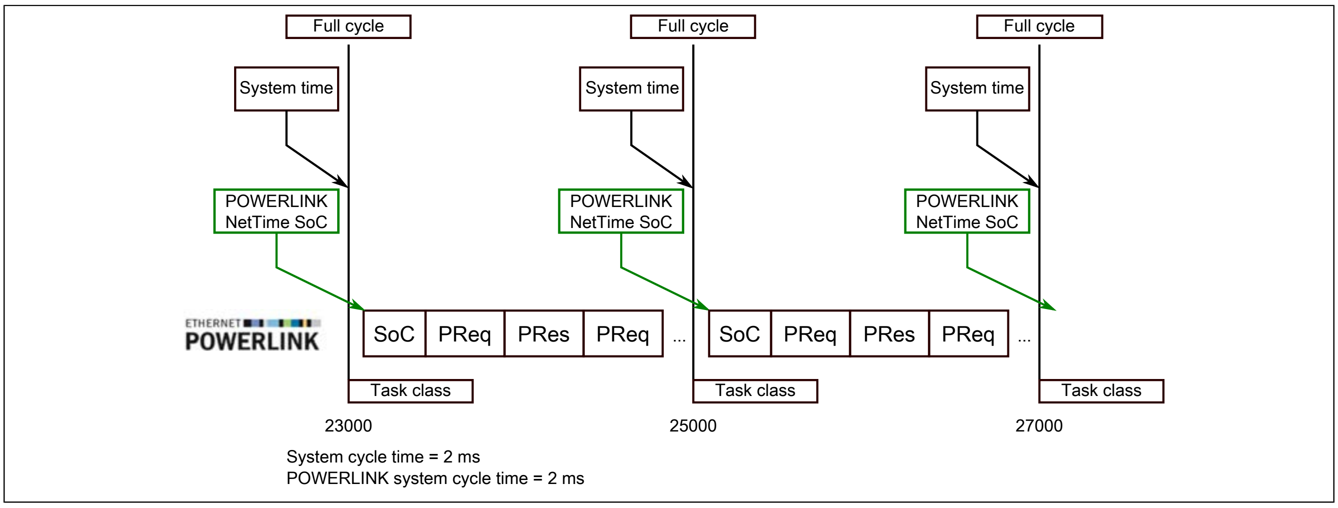
11.4.3.1.2 X2X Link reference moment



The reference moment on the X2X Link network is always calculated at the half cycle of the X2X Link cycle. This results in a difference between the system time and the X2X Link reference moment 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 moment are compared at time 25000 in the task, then the system time returns the value 25000 and the X2X Link reference moment returns the value 24000.

11.4.3.1.3 POWERLINK reference moment

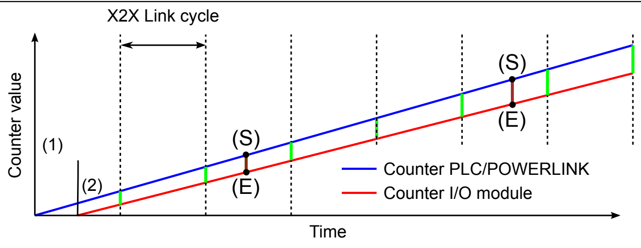


The reference moment on the POWERLINK network is always calculated at the start of cycle (SoC) of the POWERLINK network. The SoC starts 20 µs after the system tick. This results in the following difference between the system time and the POWERLINK reference time:

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 moment are compared at time 25000 in the task, then the system time returns the value 25000 and the POWERLINK reference moment returns the value 23020.

11.4.3.1.4 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 with microsecond resolution.

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 moment (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.

11.4.3.2 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 moment, the CPU can then evaluate the data using its own NetTime (or system time), if necessary.

11.4.3.2.1 Time-based inputs

NetTime Technology can be used to determine the exact moment 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 moment always lies in the past.

11.4.3.2.2 Time-based outputs

NetTime Technology can be used to specify the exact moment of a rising edge on 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 moment.

11.4.3.2.3 Time-based measurements

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

11.5 Flatstream communication

11.5.1 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.

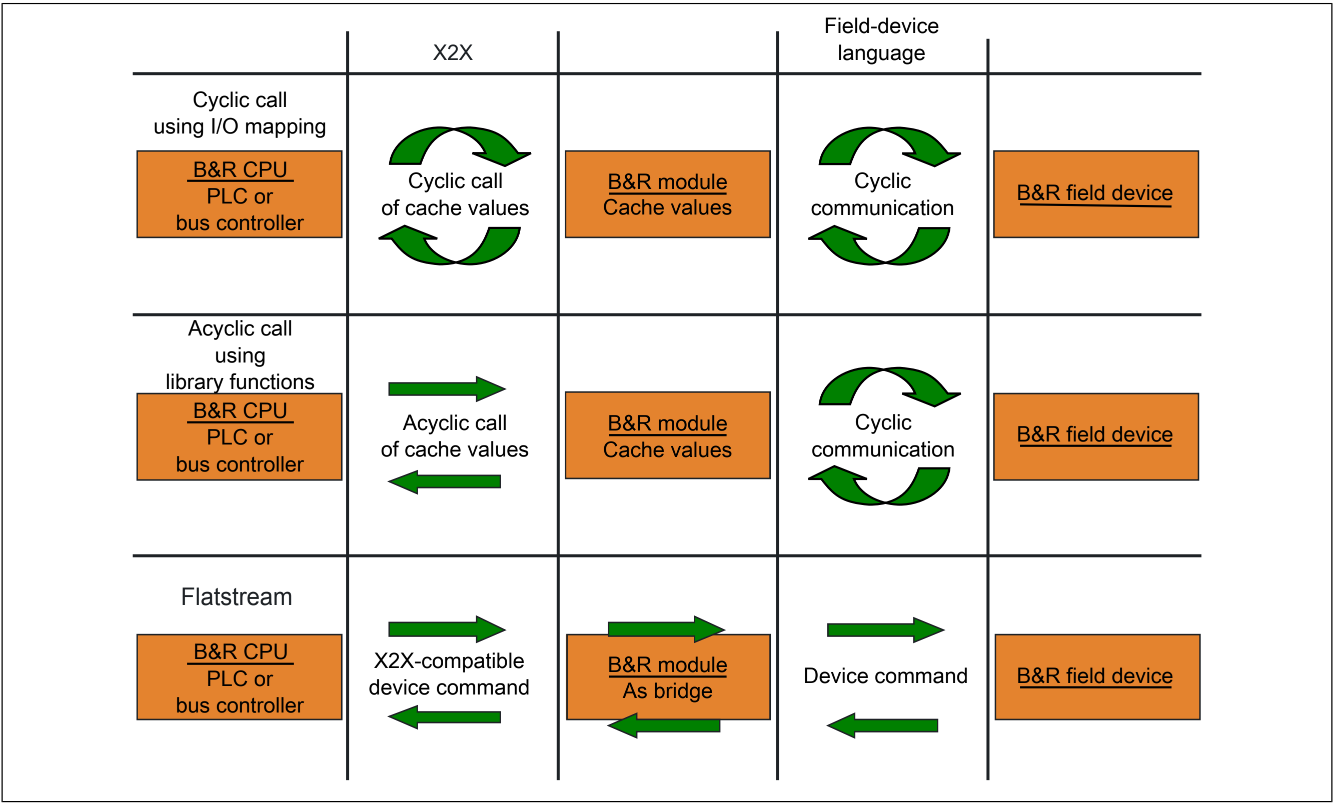

X2X
Field-device language
Cyclic call using I/O mapping
B&R CPU PLC or bus controller
Cyclic call of cache values
B&R module Cache values
Cyclic communication
B&R field device
Acyclic call using library functions
B&R CPU PLC or bus controller
Acyclic call of cache values
B&R module Cache values
Cyclic communication
B&R field device
Flatstream
B&R CPU PLC or bus controller
X2X-compatible device command
B&R module As bridge
Device command
B&R field device

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.

11.5.2 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.

11.5.3 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.

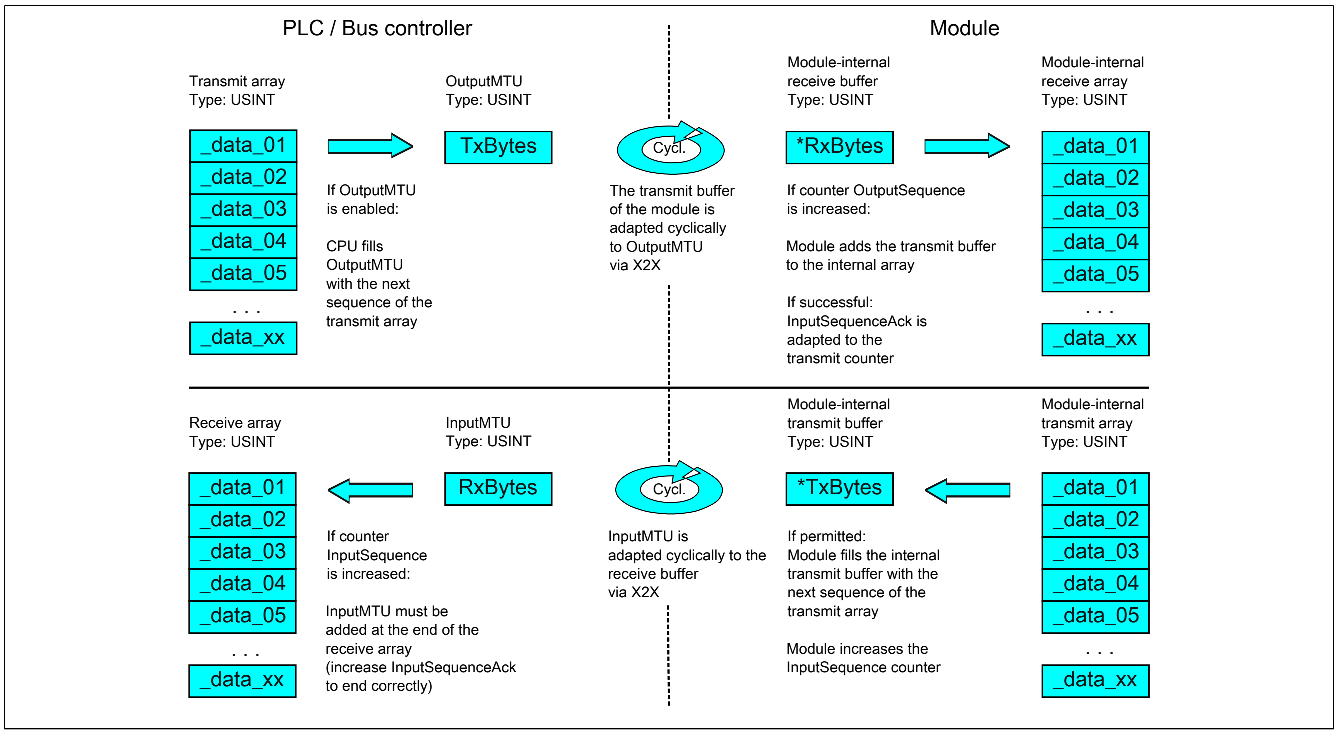


Figure 2: Flatstream communication

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.

11.5.4 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.

11.5.4.1 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.

11.5.4.1.1 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).

11.5.4.2 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.

11.5.4.2.1 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.

11.5.4.2.2 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

11.5.4.2.3 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

11.5.4.2.4 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.

11.5.4.2.5 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)

<u>InputSequenceCounter</u>

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).

<u>InputSyncBit</u>

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

<u>OutputSequenceAck</u>

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

<u>OutputSyncAck</u>

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

11.5.4.2.6 Relationship between OutputSequence and InputSequence

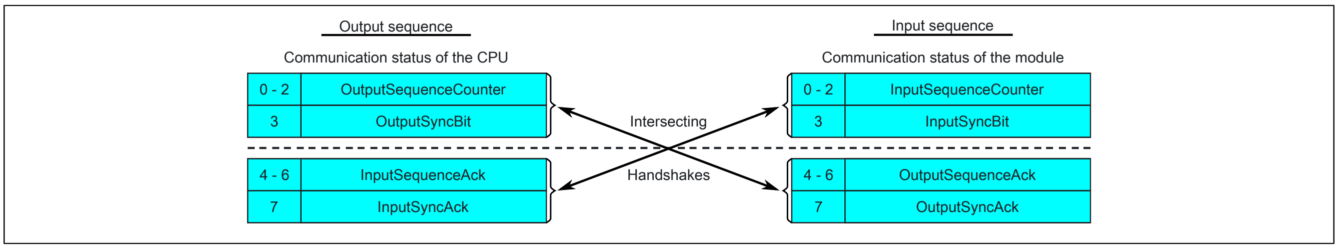


Figure 3: Relationship between OutputSequence and InputSequence

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.

11.5.4.3 Synchronization

During synchronization, a communication channel is opened. It is important to make sure that a module is present and that the current value of SequenceCounter is stored on the station receiving the message.
Flatstream can handle full-duplex communication. This means that both channels / communication directions can be handled separately. They must be synchronized independently so that simplex communication can theoretically be carried out as well.

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

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

Algorithm

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

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

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

Algorithm

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

11.5.4.4 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.

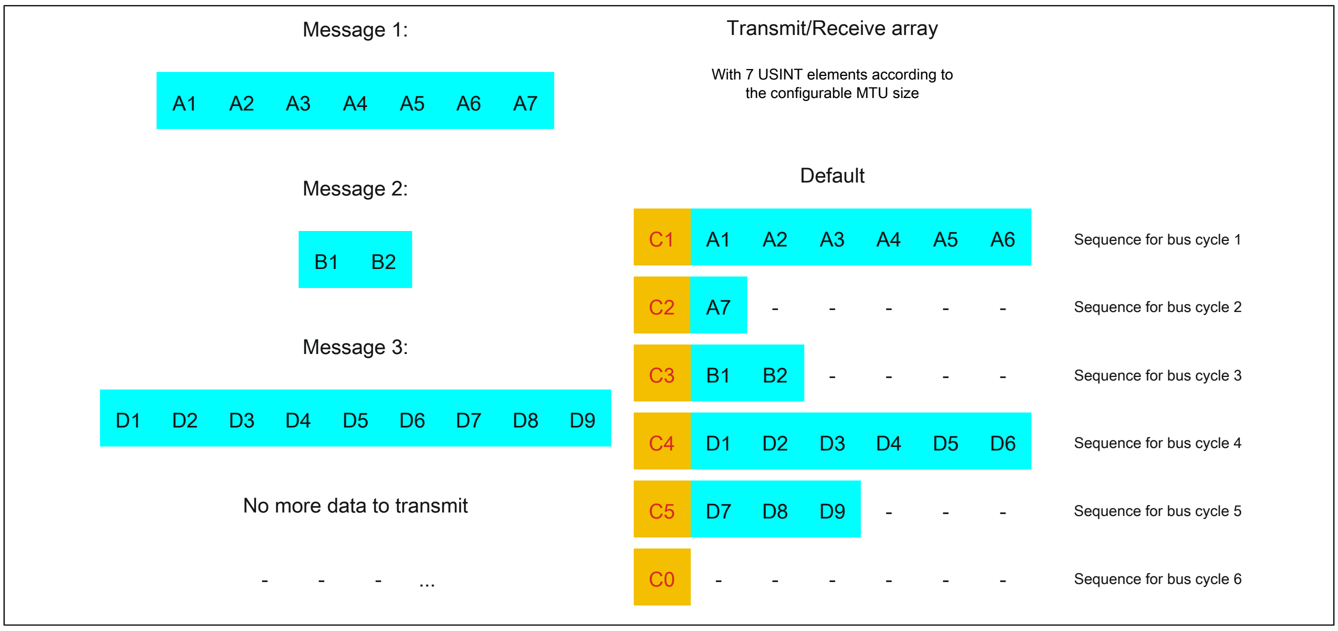


Figure 4: Transmit/Receive array (default)

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)

11.5.4.5 Transmitting data to a module (output)

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

Information:

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

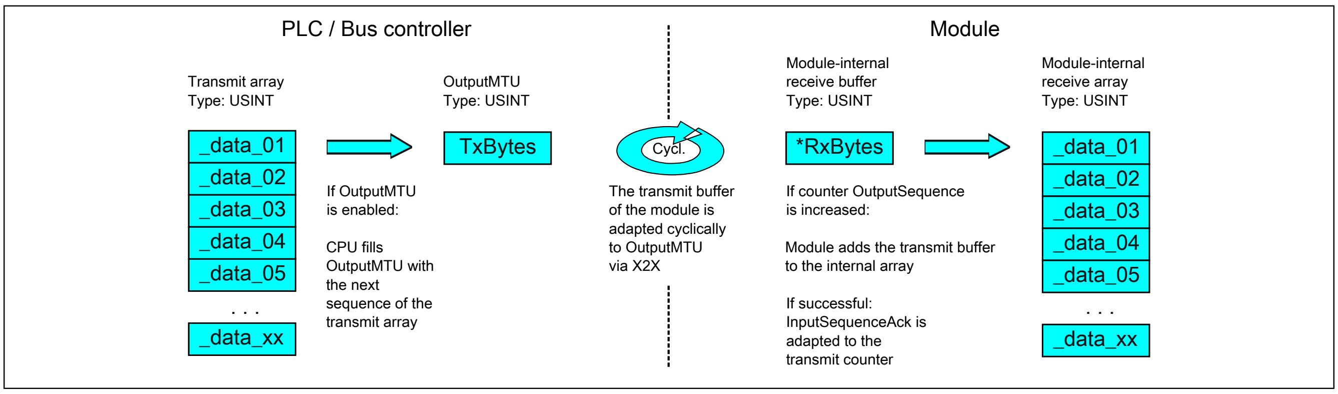


Figure 5: Flatstream communication (output)

Message smaller than OutputMTU

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

Algorithm

Cyclic status query: *- The module monitors OutputSequenceCounter.*
0) Cyclic checks: - The CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU must check whether OutputMTU is enabled. → If OutputSequenceCounter > InputSequenceAck: MTU is not enabled because the last sequence has not yet been acknowledged.
1) Preparation (create transmit array): - The CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU transfers the current element of the transmit array to OutputMTU. → The OutputMTU is transferred cyclically to the module's transmit buffer but not processed further. - The CPU must increase OutputSequenceCounter.
Reaction: *- The module accepts the bytes from the internal receive buffer and adds them to the internal receive array.* *- The module transmits acknowledgment and writes the value of OutputSequenceCounter to OutputSequenceAck.*
3) Completion: - The CPU must monitor OutputSequenceAck. → A sequence is only considered to have been transferred successfully if it has been acknowledged via OutputSequenceAck. In order to detect potential transfer errors in the last sequence as well, it is important to make sure that the length of the *Completion* phase is run through long enough. **Note:** To monitor communication times exactly, the task cycles that have passed since the last increase of OutputSequenceCounter should be counted. In this way, the number of previous bus cycles necessary for the transfer can be measured. If the monitoring counter exceeds a predefined threshold, then the sequence can be considered lost. (The relationship of bus to task cycle can be influenced by the user so that the threshold value must be determined individually.) - Subsequent sequences are only permitted to be transmitted in the next bus cycle after the completion check has been carried out successfully.

Message larger than OutputMTU

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

General flow chart

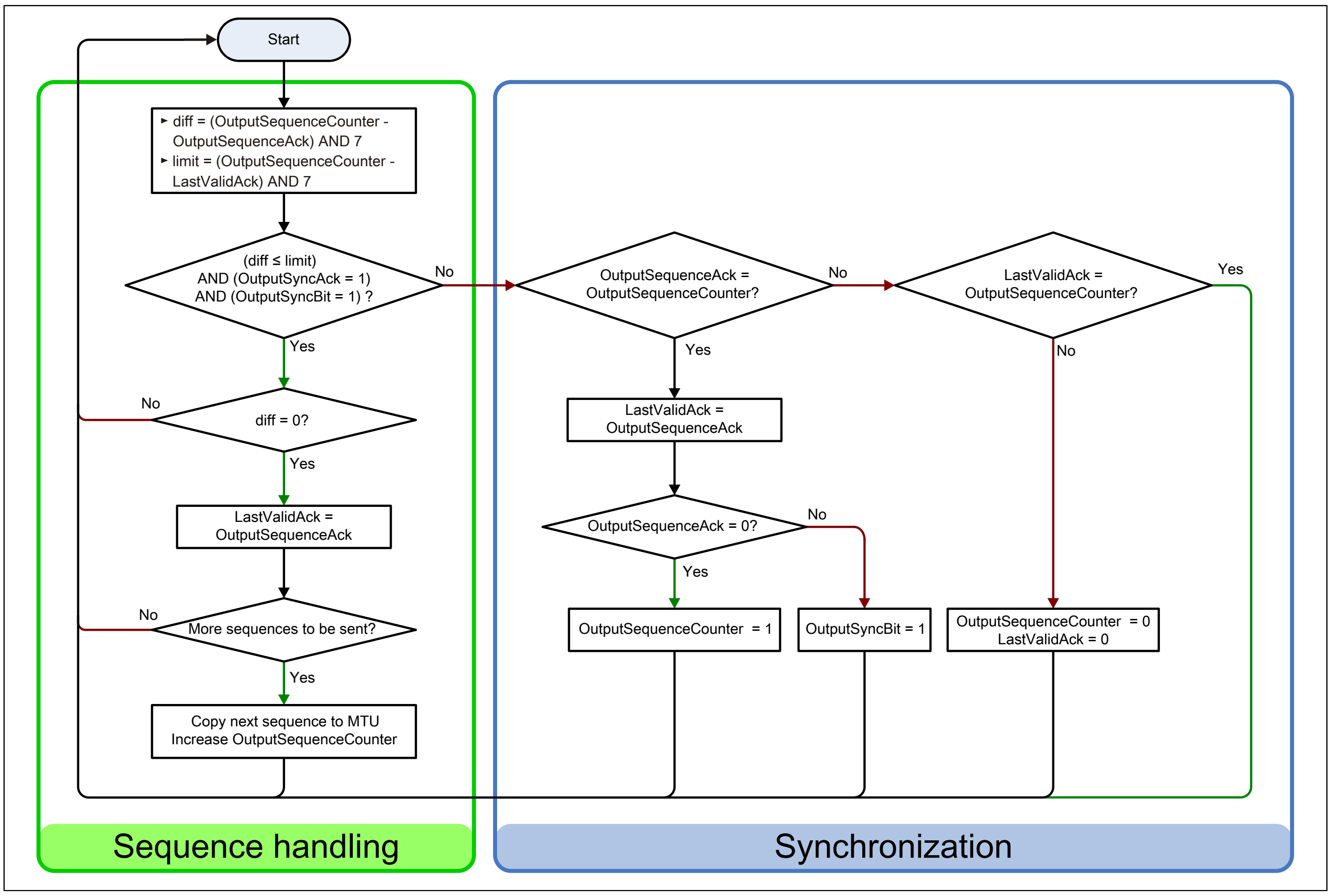

Start
diff = (OutputSequenceCounter - OutputSequenceAck) AND 7
limit = (OutputSequenceCounter - LastValidAck) AND 7
(diff ≤ limit) AND (OutputSyncAck = 1) AND (OutputSyncBit = 1) ?
No
Yes
diff = 0?
No
Yes
LastValidAck = OutputSequenceAck
More sequences to be sent?
No
Yes
Copy next sequence to MTU
Increase OutputSequenceCounter
OutputSequenceAck = OutputSequenceCounter?
No
Yes
LastValidAck = OutputSequenceAck
OutputSequenceAck = 0?
No
Yes
OutputSequenceCounter = 1
OutputSyncBit = 1
LastValidAck = OutputSequenceCounter?
Yes
No
OutputSequenceCounter = 0
LastValidAck = 0
Sequence handling
Synchronization

Figure 6: Flow chart for the output direction

11.5.4.6 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.

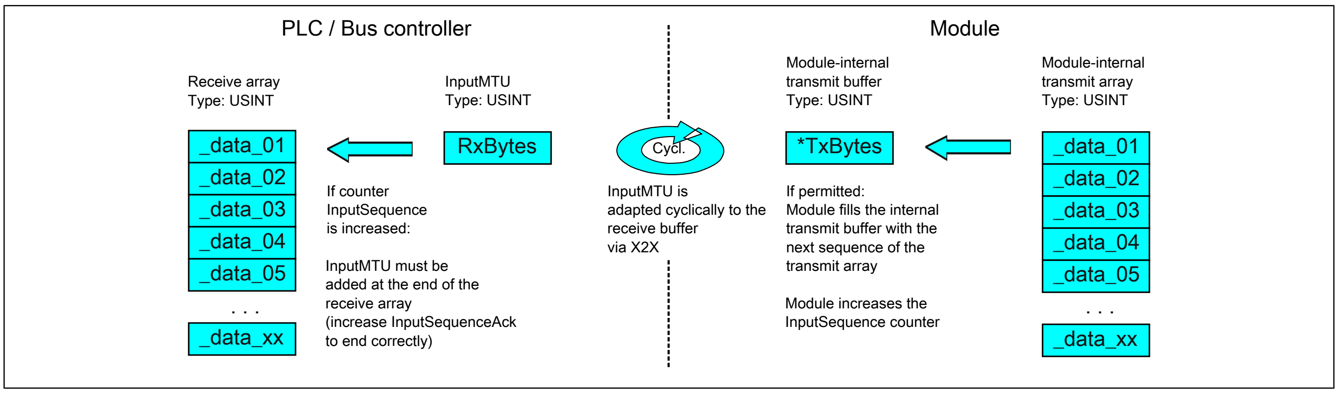

PLC / Bus controller
Module
Receive array
Type: USINT
_data_01
_data_02
_data_03
_data_04
_data_05
. . .
_data_xx
If counter InputSequence is increased:
InputMTU must be added at the end of the receive array (increase InputSequenceAck to end correctly)
InputMTU
Type: USINT
RxBytes
Cycl.
InputMTU is adapted cyclically to the receive buffer via X2X
Module-internal transmit buffer
Type: USINT
*TxBytes
If permitted: Module fills the internal transmit buffer with the next sequence of the transmit array
Module increases the InputSequence counter
Module-internal transmit array
Type: USINT
_data_01
_data_02
_data_03
_data_04
_data_05
. . .
_data_xx

Figure 7: Flatstream communication (input)

Algorithm

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

General flow chart

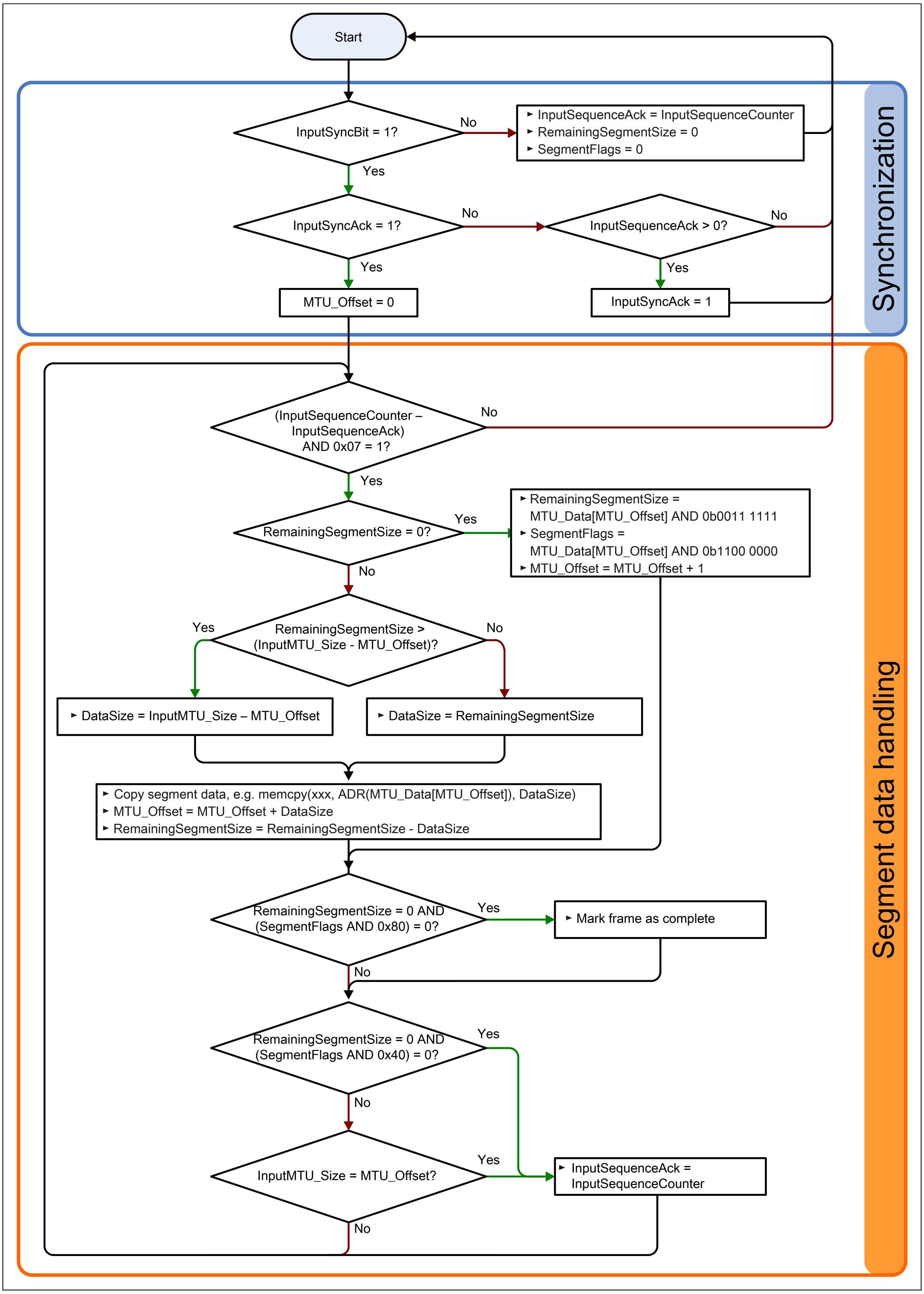

Start
InputSyncBit = 1?
No
Yes
▸ InputSequenceAck = InputSequenceCounter
▸ RemainingSegmentSize = 0
▸ SegmentFlags = 0
InputSyncAck = 1?
No
Yes
InputSequenceAck > 0?
No
Yes
MTU_Offset = 0
InputSyncAck = 1
Synchronization
(InputSequenceCounter – InputSequenceAck) AND 0x07 = 1?
No
Yes
RemainingSegmentSize = 0?
Yes
No
▸ RemainingSegmentSize = MTU_Data[MTU_Offset] AND 0b0011 1111
▸ SegmentFlags = MTU_Data[MTU_Offset] AND 0b1100 0000
▸ MTU_Offset = MTU_Offset + 1
RemainingSegmentSize > (InputMTU_Size - MTU_Offset)?
Yes
No
▸ DataSize = InputMTU_Size – MTU_Offset
▸ DataSize = RemainingSegmentSize
▸ Copy segment data, e.g. memcpy(xxx, ADR(MTU_Data[MTU_Offset]), DataSize)
▸ MTU_Offset = MTU_Offset + DataSize
▸ RemainingSegmentSize = RemainingSegmentSize - DataSize
RemainingSegmentSize = 0 AND (SegmentFlags AND 0x80) = 0?
Yes
No
▸ Mark frame as complete
RemainingSegmentSize = 0 AND (SegmentFlags AND 0x40) = 0?
Yes
No
InputMTU_Size = MTU_Offset?
Yes
No
▸ InputSequenceAck = InputSequenceCounter
Segment data handling

Figure 8: Flow chart for the input direction

11.5.4.7 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.

11.5.4.8 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.

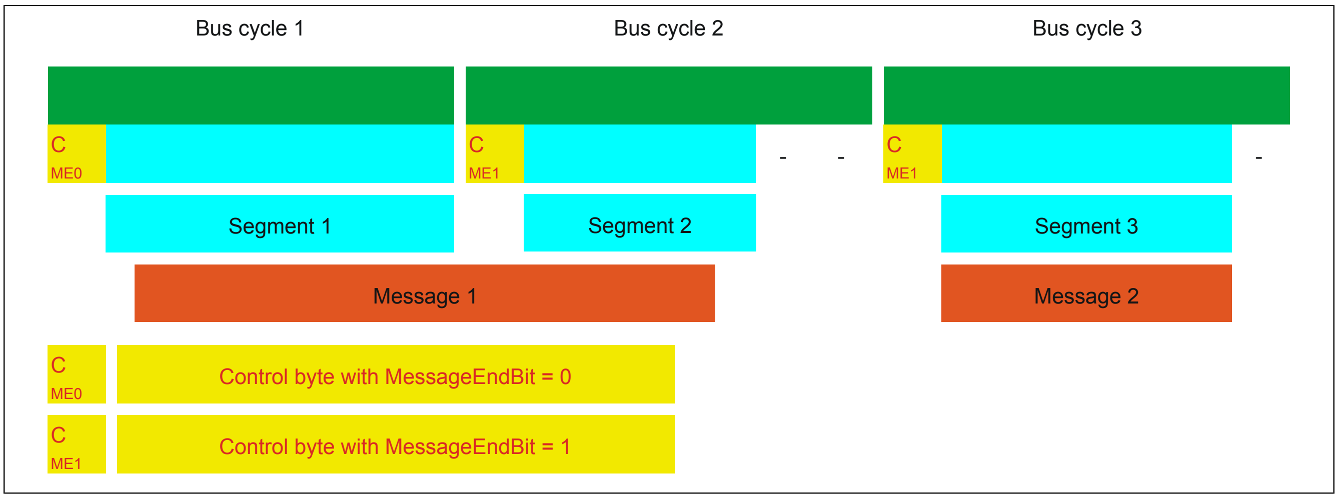


Figure 9: Message arrangement in the MTU (default)

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.

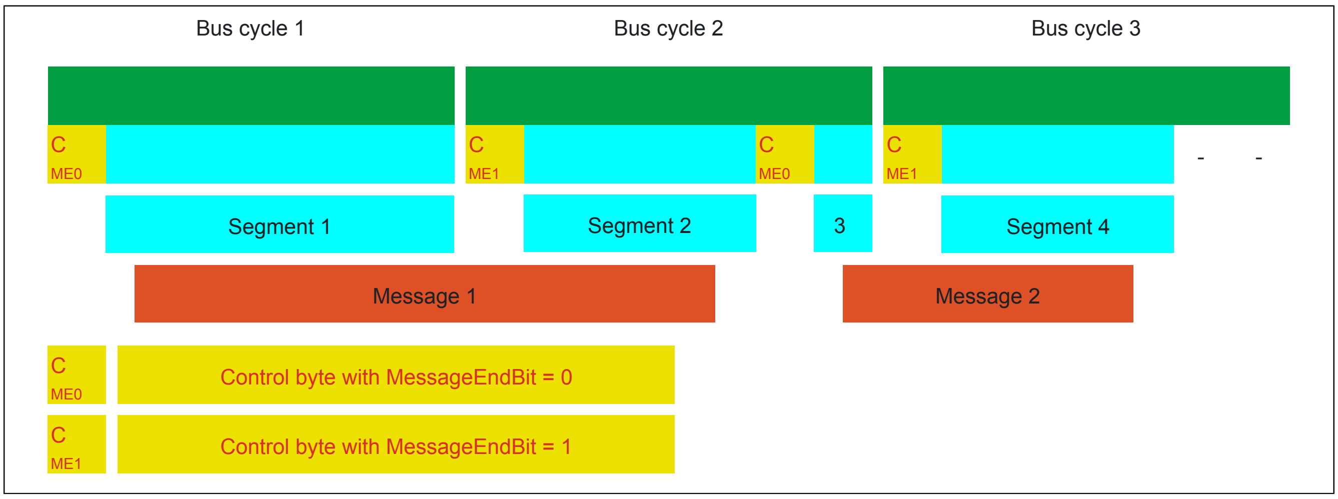


Figure 10: Arrangement of messages in the MTU (MultiSegmentMTUs)

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.

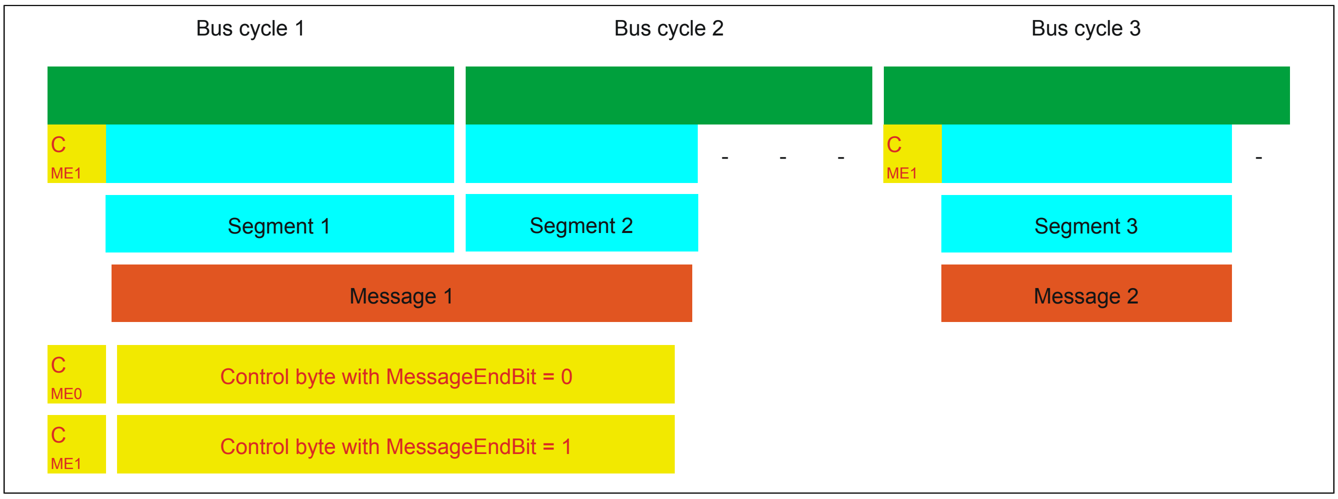


Figure 11: Arrangement of messages in the MTU (large segments)

Using both options

Using both options at the same time is also permitted.

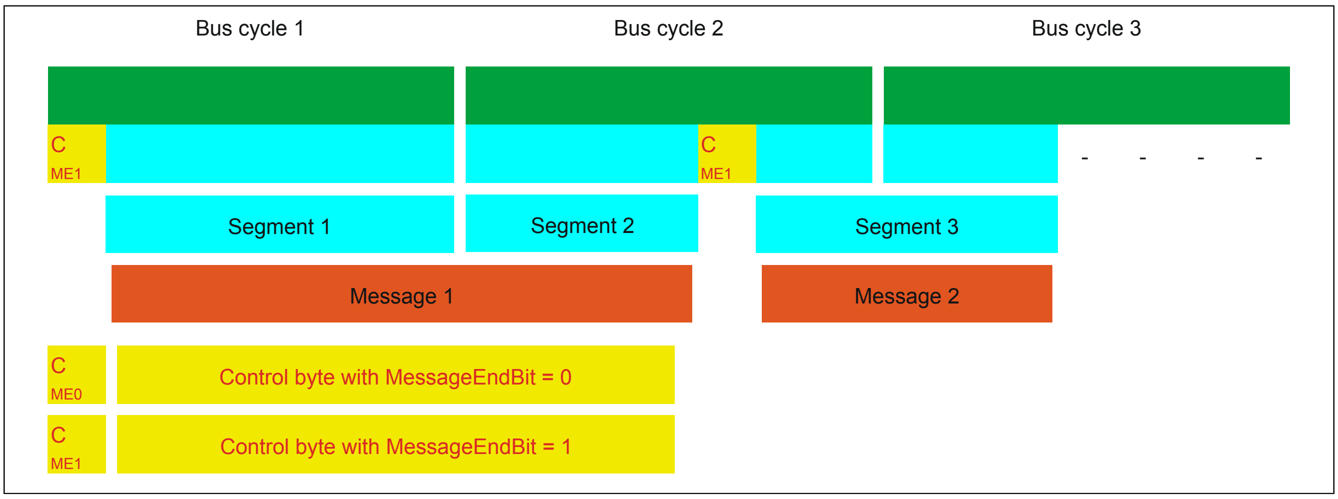


Figure 12: Arrangement of messages in the MTU (large segments and MultiSegmentMTUs)

11.5.4.9 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.

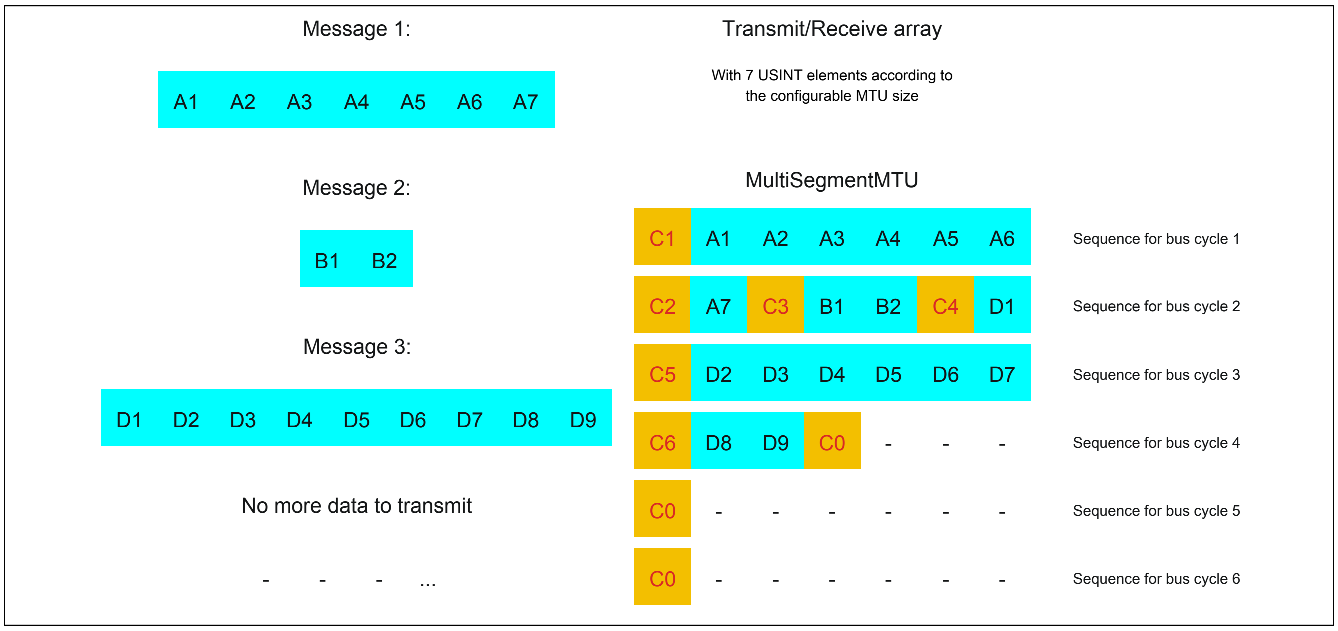


Figure 13: Transmit/receive array (MultiSegmentMTUs)

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.

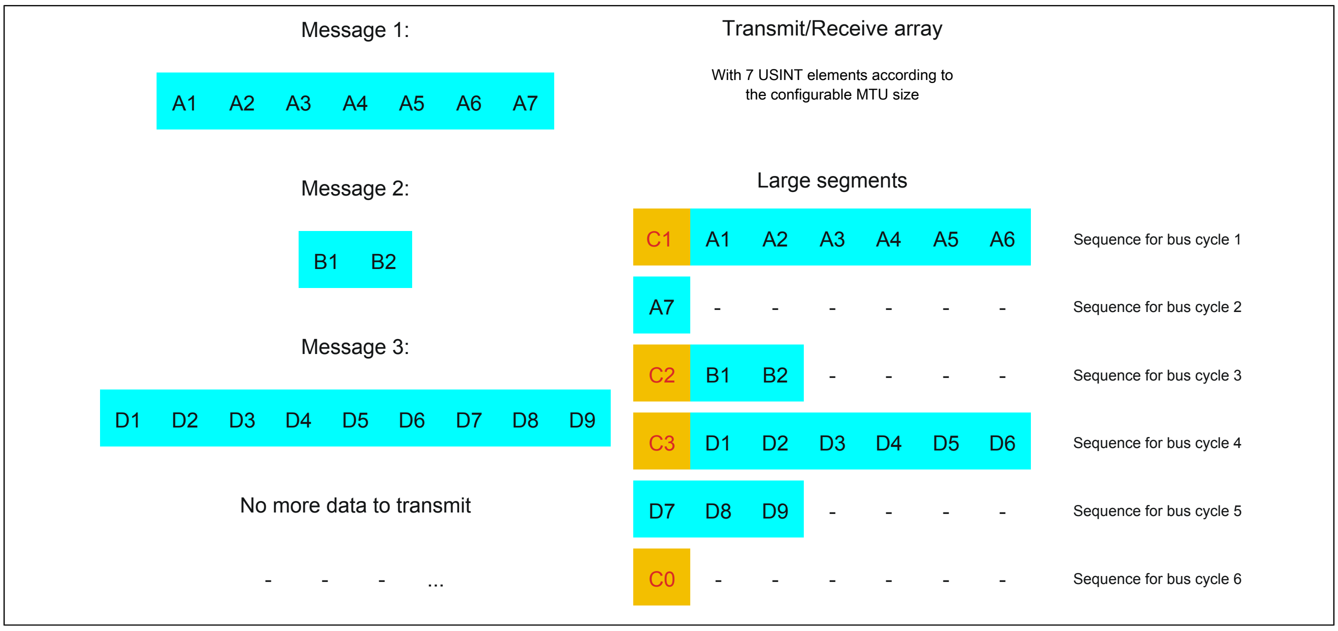


Figure 14: Transmit/receive array (large segments)

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.

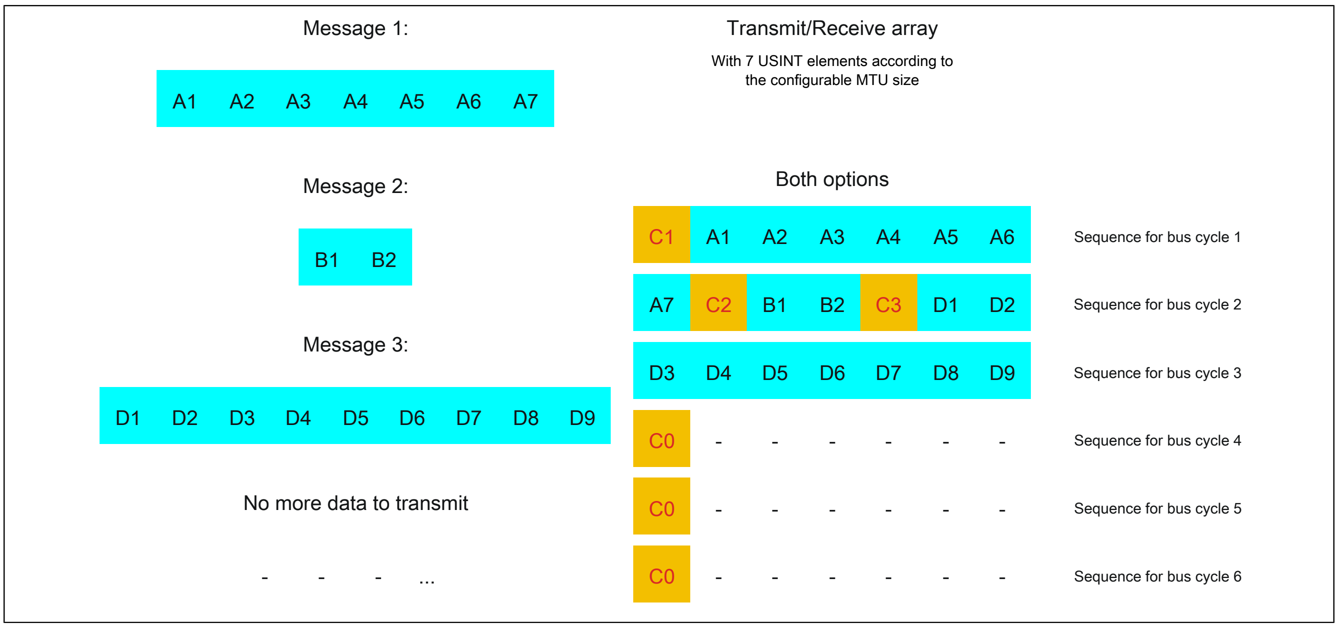


Figure 15: Transmit/receive array (large segments and MultiSegmentMTUs)

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

11.5.5 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.

11.5.5.1 Function principle

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

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

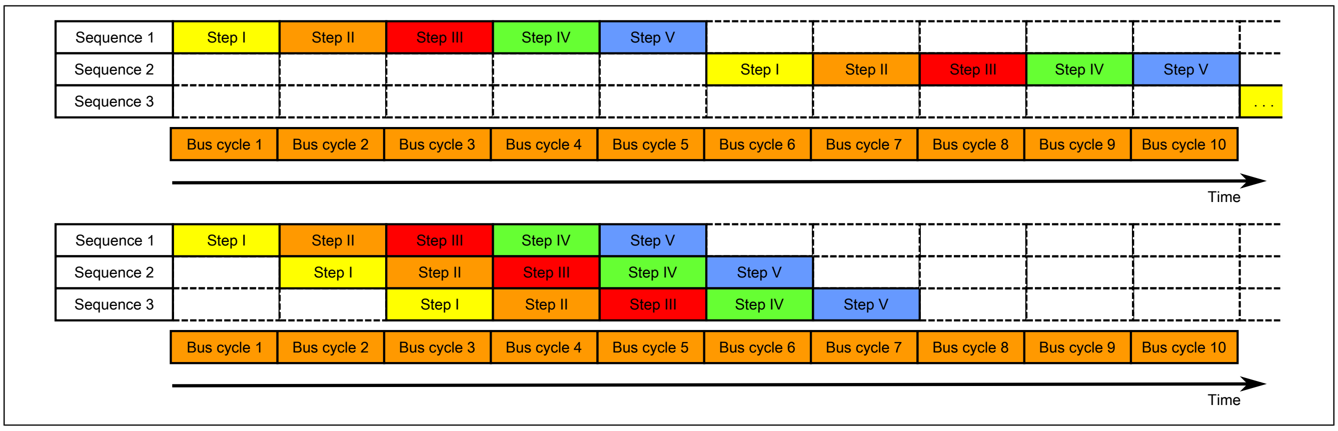


Figure 16: Comparison of transfer without/with Forward

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.

11.5.5.2 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.

11.5.5.2.1 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

11.5.5.2.2 Delay time

Name:
ForwardDelay

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

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

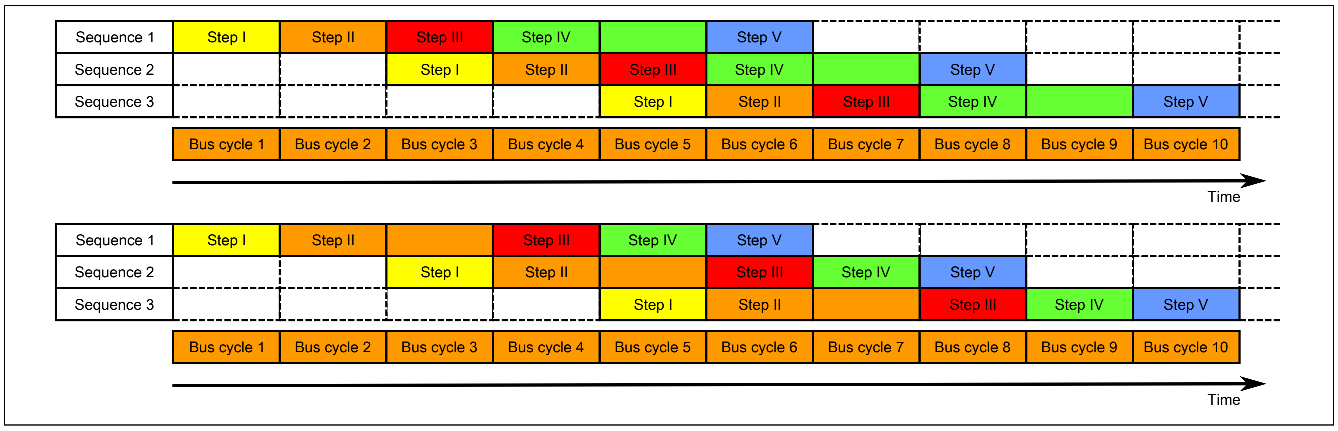


Figure 17: Effect of ForwardDelay when using Flatstream communication with the Forward function

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

11.5.5.3 Transmitting and receiving with Forward

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

Algorithm for transmitting

Cyclic status query: *- The module monitors OutputSequenceCounter.*
0) Cyclic checks: - The CPU must check OutputSyncAck. → If OutputSyncAck = 0: Reset OutputSyncBit and resynchronize the channel. - The CPU must check whether OutputMTU is enabled. → If OutputSequenceCounter > OutputSequenceAck + 7, then it is not enabled because the last sequence has not yet been acknowledged.
1) Preparation (create transmit array): - The CPU must split up the message into valid segments and create the necessary control bytes. - The CPU must add the segments and control bytes to the transmit array.
2) Transmit: - The CPU must transfer the current part of the transmit array to OutputMTU. - The CPU must increase OutputSequenceCounter for the sequence to be accepted by the module. - The CPU is then permitted to *transmit* in the next bus cycle if the MTU has been enabled.
The module responds since OutputSequenceCounter > OutputSequenceAck: *- The module accepts data from the internal receive buffer and appends it to the end of the internal receive array.* *- The module is acknowledged and the currently received value of OutputSequenceCounter is transferred to OutputSequenceAck.* *- The module queries the status cyclically again.*
3) Completion (acknowledgment): - The CPU must check OutputSequenceAck cyclically. → A sequence is only considered to have been transferred successfully if it has been acknowledged via OutputSequenceAck. In order to detect potential transfer errors in the last sequence as well, it is important to make sure that the algorithm is run through long enough. **Note:** To monitor communication times exactly, the task cycles that have passed since the last increase of OutputSequenceCounter should be counted. In this way, the number of previous bus cycles necessary for the transfer can be measured. If the monitoring counter exceeds a predefined threshold, then the sequence can be considered lost (the relationship of bus to task cycle can be influenced by the user so that the threshold value must be determined individually).

Algorithm for receiving

0) Cyclic status query: - The CPU must monitor InputSequenceCounter.
Cyclic checks: *- The module checks InputSyncAck.* *- The module checks if InputMTU for enabling.* *→ Enabling criteria: InputSequenceCounter > InputSequenceAck + Forward*
Preparation: *- The module forms the control bytes / segments and creates the transmit array.*
Action: *- The module transfers the current part of the transmit array to the receive buffer.* *- The module increases InputSequenceCounter.* *- The module waits for a new bus cycle after time from ForwardDelay has expired.* *- The module repeats the action if InputMTU is enabled.*
1) Receiving (InputSequenceCounter > InputSequenceAck): - The CPU must apply data from InputMTU and append it to the end of the receive array. - The CPU must match InputSequenceAck to InputSequenceCounter of the sequence currently being processed.
Completion: *- The module monitors InputSequenceAck.* *→ A sequence is only considered to have been transferred successfully if it has been acknowledged via InputSequenceAck.*

Details/Background

1. Illegal SequenceCounter size (counter offset)
   Error situation: MTU not enabled
   If the difference between SequenceCounter and SequenceAck during transmission is larger than permitted, a transfer error occurs. In this case, all unacknowledged sequences must be repeated with the old 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.

11.5.5.4 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).

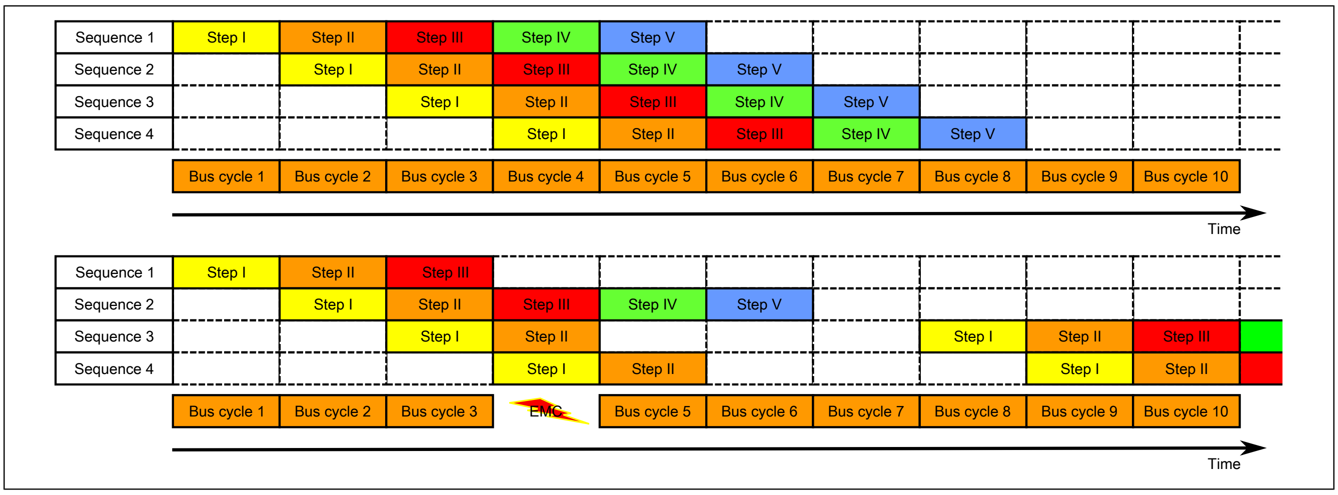


Figure 18: Effect of a lost bus cycle

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.

11.6 Required cycle time

The cycle time depends on the sampling rate, data resolution and MTU size. It must be selected such that no buffer overflow occurs when the measured values are transmitted via the Flatstream.