KNX TinySerial 810

Communication Protocol

WEINZIERL ENGINEERING GmbH
Bahnhofstr. 6
DE-84558 Tyrlaching
GERMAY
Tel. +49 8623 / 987 98 - 03
Fax +49 8623 / 987 98 - 09
E-Mail: info@weinzierl.de
Web: www.weinzierl.de

Document history

Document status	Date	Editor
First draft	2011-10-20	K. Ströber
Review	2011-10-20	K. Ströber, F.Häusl
Review, Release	2011-10-24	J. Richards
Added telegram format and send/receive description	2011-11-18	Th. Weinzierl
Added kdrive implementation aspects	2011-11-30	J. Richards

Content

1. Application area

The KNX Tiny Serial Interface 810 offers a simple connection to the KNX Bus Twisted Pair including galvanic isolation. The connection is via UART (3-5V).

The communication is based on a modified TP-UART Protocol without real-time requirements to the host. The Tiny Serial Interface is ideally suited for connecting KNX Devices running Linux, Windows CE and Android, among others.

A Cross Platform SDK is also available to integrate Tiny Serial into your application.

2. Protocol

2.1. General

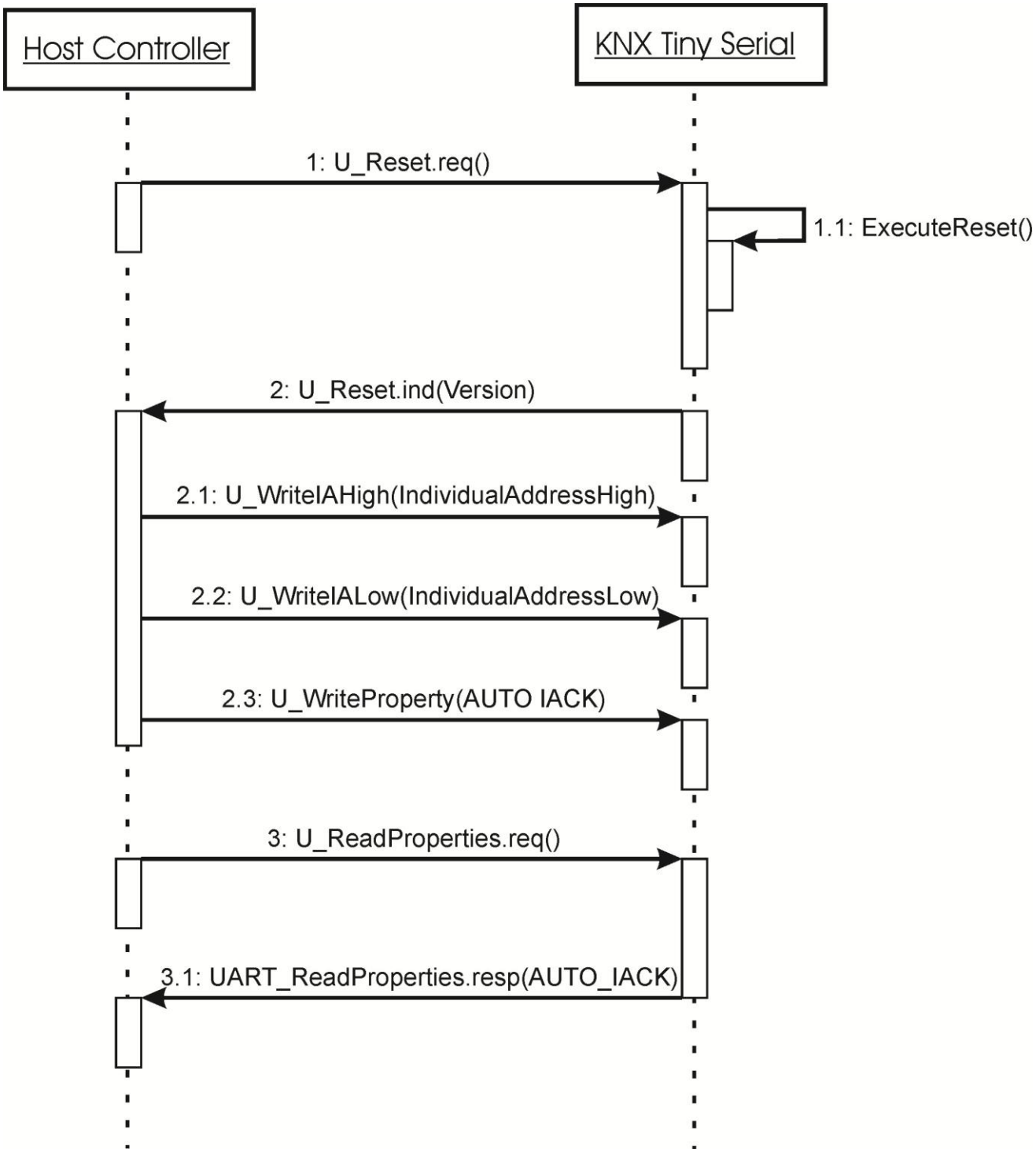
The KNX Tiny Serial, described in this document, is based on the TP-UART Protocol with the following modifications:

- Configurable ‘IndividualAddress’
- Configurable ‘Acknowledge-Handling’ according to KNX specification

This document describes only the modifications to the TP-UART protocol.

2.2. Configuration procedure

The configuration procedure begins after the restart of the KNX Tiny Serial. When the KNX Tiny Serial is ready to operate, it sends the service ‘UART_Reset.ind’ to the host processor.
At this point the standard ‘Acknowledge handling’ is active.
Optionally the host processor can send its own "individual address" and switch on the non-selective "Acknowledge-Handling". This mode can be activated only if the "individual address" has previously been successfully set.
The current state can be requested by the host processor with service U_ReadProperty.req.


Host Controller
KNX Tiny Serial
1: U_Reset.req()
1.1: ExecuteReset()
2: U_Reset.ind(Version)
2.1: U_WriteIAHigh(IndividualAddressHigh)
2.2: U_WriteIALow(IndividualAddressLow)
2.3: U_WriteProperty(AUTO IACK)
3: U_ReadProperties.req()
3.1: UART_ReadProperties.resp(AUTO_IACK)

2.3. Property value

Bit#	Meaning
7	0
6	0
5	0
4	0
3	0
2	0
1	0
0	0 = non-selective IACK disabled 1 = non-selective IACK enabled

Figure 2: Description of Property-bit field

2.3.1. Non-selective IACK

- The non-selective-IACK flag is inactive after restart. It can be activated by the host processor.
- Before activating this flag of the KNX Tiny Serial, it must be ensured that the Individual address has been set.
- The host processor can read the current state of this flag .If the flag is set, the KNX Tiny Serial confirms all group telegrams, all broadcast messages and all individually addressed telegrams to its own address.

3. Communication parameters of the serial interface

The serial interface to the host controller is used with the following settings:

Parameter	Value	Comments
Baud Rate	19200 baud	
Start-Bits	1	active = low
Data-Bits	8 (LSB first)	active = low
Parity-Bit	1 (even parity)	active = low
Stop-Bits	1	
Idle-Level	High	

4. Communication mechanisms

Format of the KNX telegram

The following table shows the coding of KNX standard telegrams for group data:

Octet Number	Octet Name	Bit usage (MSB -> LSB)
Octet 0	Control Field	frame type std. = 0 0 Repeat flag, rep = 0 1 Priority, low: 11b Priority, low: 11b 0 0
Octet 1	Src. addr high	area address area address area address area address line address line address line address line address

Octet 2	Src. addr low	device address
Octet 3	Dest. addr high	group / ind. address
Octet 4	Dest. addr low	group / ind. address
Octet 5	DAF/LSDU/Length	DAF, group = 1 Hop count, std = 6d Hop count, std = 6d Hop count, std = 6d Length, Octet 7.. Length, Octet 7.. Length, Octet 7.. Length, Octet 7..
Octet 6	TPCI/APCI	TPCI TPCI TPCI TPCI TPCI TPCI APCI APCI
Octet 7	APCI/data	APCI APCI APCI or data APCI or data APCI or data APCI or data APCI or data APCI or data
Octet 8..21	data	
Octet 22	Check octet	

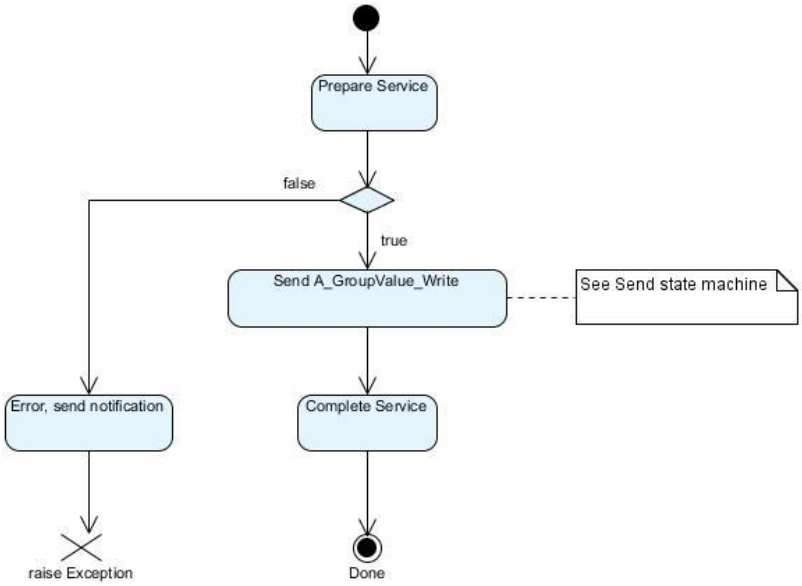
TPCI code for group data:
000000b GroupData

APCI codes for group data:
0000b GroupValue_Read
0001b GroupValue_Response
0010b GroupValue_Write

If the GroupData is less than or equal to 6 bits it is coupled with the APCI and the data field is empty.
The check octet is an bitwise exor over all preceding octets.

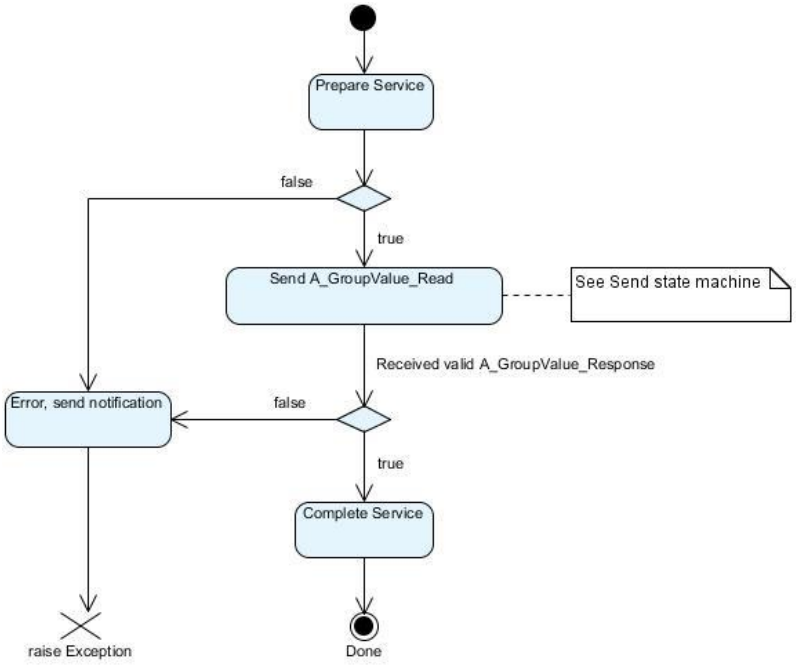
Group Value Write

Sends a GroupValue_Write telegram on the bus.



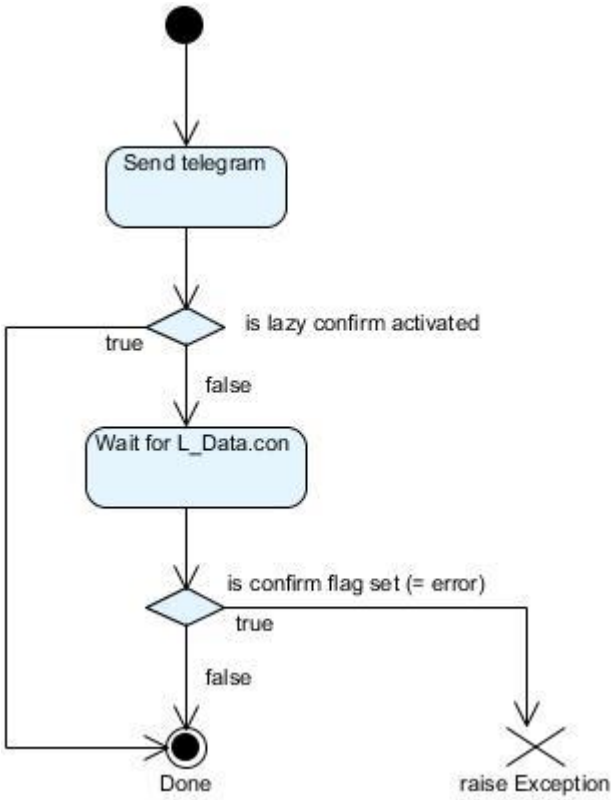
Group Value Read

Sends a GroupValue_Read telegram on the bus and waits for at least one GroupValue_Response telegrams.



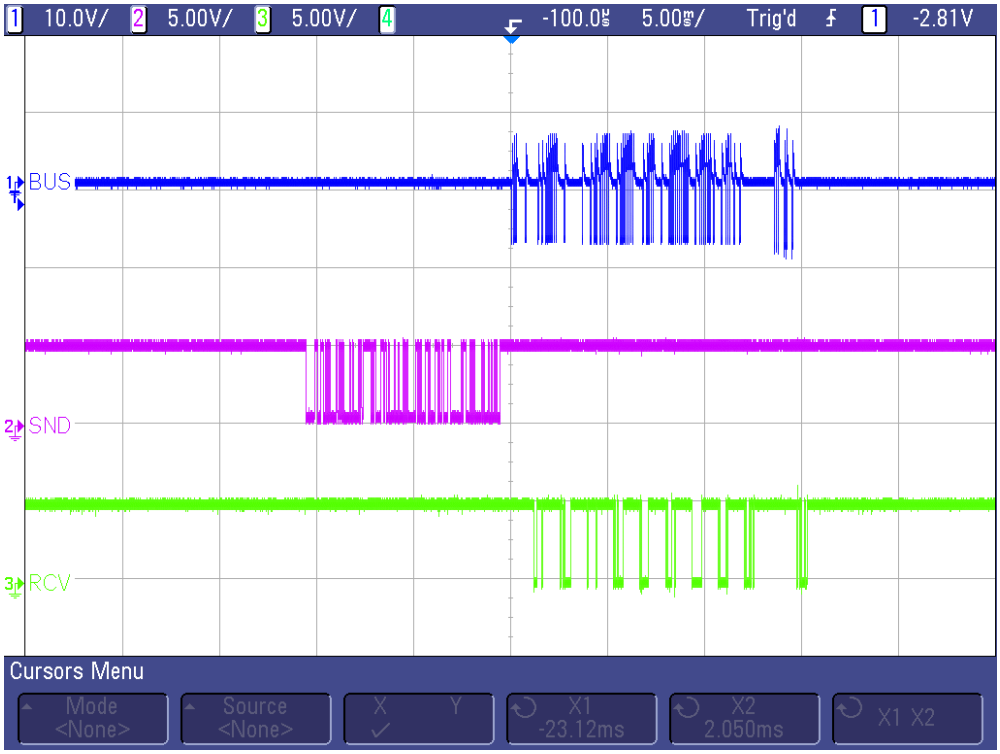
Send State Machine

The send state machine waits for a L_DATA confirm (LazyConfirm is a kdrive SDK property and should be ignored for the purposes of TinySerial).



Sending to KNX bus

In sending direction each octet to be sent on the bus is preceded by a control character U_L_Data with the corresponding index. The last octet is the checksum and is preceded by an U_L_DataEnd character with the right index. After the reception of the correct checksum the TinySerial starts to send the telegram on the bus as soon as the bus is free. During the send process each character on the bus is sent to the host. Repetitions will be sent automatically on the bus if required (Busy or NAK or no ACK).



Channel 1: KNX bus signal (AC part)
Channel 2: Send signal (host to TinySerial)
Channel 3: Receive signal (TinySerial to host)

Example (Switch on a light)

Host to TinySerial (in hex)

```
80 BC 81 11 82 01 83 12 84 34 85 E1 86 00 87 81 48 15
```

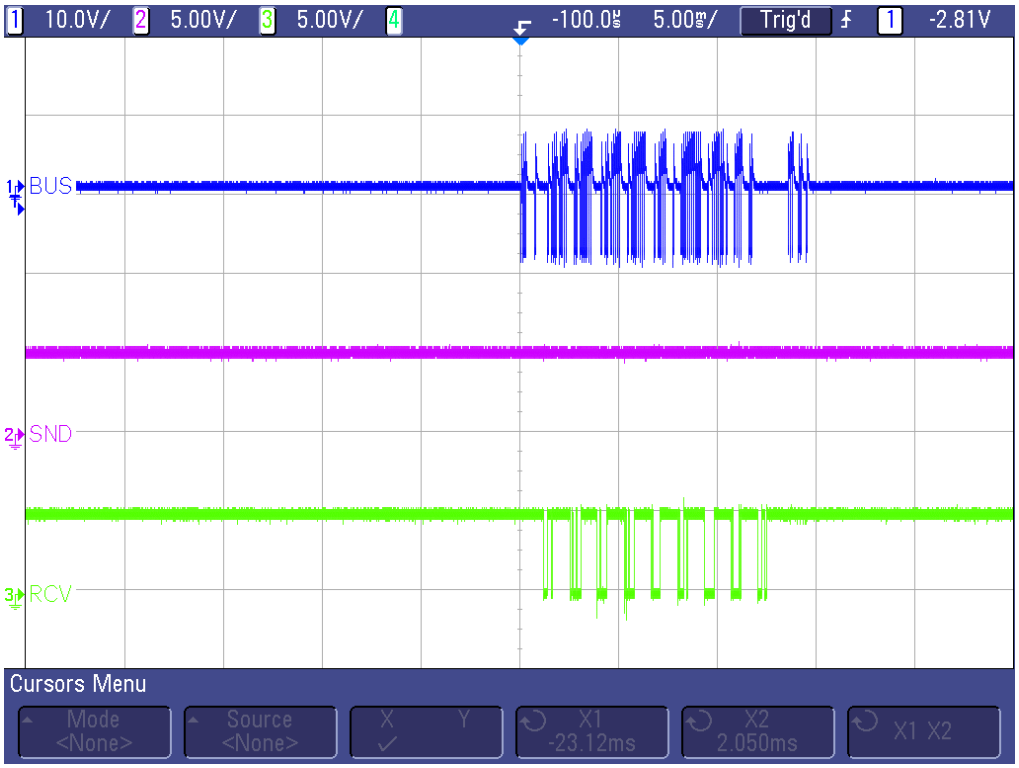
On Bus (in hex)

```
   BC    11    01    12    34    E1    00    81    15
```

Source address: 0x1101 (individual)
Destination address: 0x1234 (group)

Receiving from KNX bus

In receiving direction each octet received from the bus is transferred immediately to the host without additional control codes.



Channel 1: KNX bus signal (AC part)
Channel 2: Send signal (host to TinySerial)
Channel 3: Receive signal (TinySerial to host)

Coding of control characters

The table below shows all the services that are exchanged between the communication partners.
The middle column of the table contains the Control Field.

Services from UART			**Services To UART**		
		00			
		01	U_Reset.request		
		02	U_State.request		
UART_Reset.ind		03			
		04			
		05			
		06			
UART_State.ind	no error	07			
		08	U_Control.req	repeater not present	
		09	U_Control.req	repeater present	
		0A	U_Control.req	is repeater	

L_DATA.conf	negative confirm	0B			
NACK	Busmonitor mode only	0C			
		0D			
		0E			
UART_State.ind		0F			
Services from UART			**Services To UART**		
L_LONG_DATA.request	system, repeated	10	U_AckInformation	not addressed	
		11	U_AckInformation	ack	
		12	U_AckInformation	invalid	
		13	U_AckInformation	busy	
L_LONG_DATA.request	high, repeated	14	U_AckInformation	invalid	
		15	U_AckInformation	n_ack	
		16	U_AckInformation	invalid	
UART_State.ind		17	U_AckInformation	invalid	
L_LONG_DATA.request	alarm, repeated	18			
		19			
UART_Properties.ind	data	1A	U_ReadProperties.req		
		1B			
L_LONG_DATA.request	normal, repeated	1C			
		1D			
		1E	U_Write.req	IndividualAddress Low	
UART_State.ind		1F	U_Write.req	IndividualAddress High	
		20			
		21			
		22	U_Write.req	SetProperties(xx)	
		23			
		24			
		25			
		26			
UART_State.ind		27			
		28			
		29			
Button_Pressed.ind		2A	LED_off.req		
		2B	LED_on.req		
		2C			
		2D			
		2E			
UART_State.ind		2F			
L_LONG_DATA.request	system, not repeated	30			
		31			
		32			

		33			
L_LONG_DATA.request	high, not repeated	34			
		35			
		36			
UART_State.ind		37			
L_LONG_DATA.request	alarm, not repeated	38			
Services from UART			**Services To UART**		
		39			
		3A			
		3B			
L_LONG_DATA.request	normal, not repeated	3C			
		3D			
		3E			
UART_State.ind		3F			
		40			
		41			
		42			
		43			
		44			
		45			
		46			
UART_State.ind		47	U_L_DataEnd	index	7
		48	U_L_DataEnd	index	8
		49	U_L_DataEnd	index	9
		4A	U_L_DataEnd	index	10
		4B	U_L_DataEnd	index	11
		4C	U_L_DataEnd	index	12
		4D	U_L_DataEnd	index	13
		4E	U_L_DataEnd	index	14
UART_State.ind		4F	U_L_DataEnd	index	15
		50	U_L_DataEnd	index	16
		51	U_L_DataEnd	index	17
		52	U_L_DataEnd	index	18
		53	U_L_DataEnd	index	19
		54	U_L_DataEnd	index	20
		55	U_L_DataEnd	index	21
		56	U_L_DataEnd	index	22
UART_State.ind		57	U_L_DataEnd	index	23
		58	U_L_DataEnd	index	24
		59	U_L_DataEnd	index	25
		5A	U_L_DataEnd	index	26
		5B	U_L_DataEnd	index	27

		5C	U_L_DataEnd	index	28
		5D	U_L_DataEnd	index	29
		5E	U_L_DataEnd	index	30
UART_State.ind		5F	U_L_DataEnd	index	31
		60	U_L_DataEnd	index	32
		61	U_L_DataEnd	index	33
Services from UART			**Services To UART**		
		62	U_L_DataEnd	index	34
		63	U_L_DataEnd	index	35
		64	U_L_DataEnd	index	36
		65	U_L_DataEnd	index	37
		66	U_L_DataEnd	index	38
UART_State.ind		67	U_L_DataEnd	index	39
		68	U_L_DataEnd	index	40
		69	U_L_DataEnd	index	41
		6A	U_L_DataEnd	index	42
		6B	U_L_DataEnd	index	43
		6C	U_L_DataEnd	index	44
		6D	U_L_DataEnd	index	45
		6E	U_L_DataEnd	index	46
UART_State.ind		6F	U_L_DataEnd	index	47
		70	U_L_DataEnd	index	48
		71	U_L_DataEnd	index	49
		72	U_L_DataEnd	index	50
		73	U_L_DataEnd	index	51
		74	U_L_DataEnd	index	52
		75	U_L_DataEnd	index	53
		76	U_L_DataEnd	index	54
UART_State.ind		77	U_L_DataEnd	index	55
		78	U_L_DataEnd	index	56
		79	U_L_DataEnd	index	57
		7A	U_L_DataEnd	index	58
		7B	U_L_DataEnd	index	59
		7C	U_L_DataEnd	index	60
		7D	U_L_DataEnd	index	61
		7E	U_L_DataEnd	index	62
UART_State.ind		7F	U_L_DataEnd	index	63
		80	U_L_Data	index	0
		81	U_L_Data	index	1
		82	U_L_Data	index	2
		83	U_L_Data	index	3
		84	U_L_Data	index	4

		85	U_L_Data	index	5
		86	U_L_Data	index	6
UART_State.ind		87	U_L_Data	index	7
		88	U_L_Data	index	8
		89	U_L_Data	index	9
		8A	U_L_Data	index	10
Services from UART			**Services To UART**		
L_DATA.conf	positiv confirm	8B	U_L_Data	index	11
		8C	U_L_Data	index	12
		8D	U_L_Data	index	13
		8E	U_L_Data	index	14
UART_State.ind		8F	U_L_Data	index	15
L_DATA.request	system, repeated	90	U_L_Data	index	16
		91	U_L_Data	index	17
		92	U_L_Data	index	18
		93	U_L_Data	index	19
L_DATA.request	high, repeated	94	U_L_Data	index	20
		95	U_L_Data	index	21
		96	U_L_Data	index	22
UART_State.ind		97	U_L_Data	index	23
L_DATA.request	alarm, repeated	98	U_L_Data	index	24
		99	U_L_Data	index	25
		9A	U_L_Data	index	26
		9B	U_L_Data	index	27
L_DATA.request	normal, repeated	9C	U_L_Data	index	28
		9D	U_L_Data	index	29
		9E	U_L_Data	index	30
UART_State.ind		9F	U_L_Data	index	31
		A0	U_L_Data	index	32
		A1	U_L_Data	index	33
		A2	U_L_Data	index	34
		A3	U_L_Data	index	35
		A4	U_L_Data	index	36
		A5	U_L_Data	index	37
		A6	U_L_Data	index	38
UART_State.ind		A7	U_L_Data	index	39
		A8	U_L_Data	index	40
		A9	U_L_Data	index	41
		AA	U_L_Data	index	42
		AB	U_L_Data	index	43
		AC	U_L_Data	index	44
		AD	U_L_Data	index	45

		AE	U_L_Data	index	46
UART_State.ind		AF	U_L_Data	index	47
L_DATA.request	system, not repeated	B0	U_L_Data	index	48
		B1	U_L_Data	index	49
		B2	U_L_Data	index	50
		B3	U_L_Data	index	51
Services from UART			**Services To UART**		
L_DATA.request	high, not repeated	B4	U_L_Data	index	52
		B5	U_L_Data	index	53
		B6	U_L_Data	index	54
UART_State.ind		B7	U_L_Data	index	55
L_DATA.request	alarm, not repeated	B8	U_L_Data	index	56
		B9	U_L_Data	index	57
		BA	U_L_Data	index	58
		BB	U_L_Data	index	59
L_DATA.request	normal, not repeated	BC	U_L_Data	index	60
		BD	U_L_Data	index	61
		BE	U_L_Data	index	62
UART_State.ind		BF			
BUSY	Busmonitor mode only	C0			
		C1			
		C2			
		C3			
		C4			
		C5			
		C6			
UART_State.ind		C7			
		C8			
		C9			
		CA			
		CB			
ACK	Busmonitor mode only	CC			
		CD			
		CE			
UART_State.ind		CF			
		D0			
		D1			
		D2			
		D3			
		D4			

		D5			
		D6			
UART_State.ind		D7			
		D8			
		D9			
		DA			
		DB			
		DC			
Services from UART			**Services To UART**		
		DD			
		DE			
UART_State.ind	SC, RE, PE, TW	DF			
		E0			
		E1			
		E2			
		E3			
		E4			
		E5			
		E6			
UART_State.ind	SC, RE, TE	E7			
		E8			
		E9			
		EA			
		EB			
		EC			
		ED			
		EE			
UART_State.ind	SC, RE, TE, TW	EF			
L_POLL_DATA.req	system	F0			
		F1			
		F2			
		F3			
		F4			
		F5			
		F6			
UART_State.ind	SC, RE, TE, PE	F7			
		F8			
		F9			
		FA			
		FB			
		FC			
		FD			

		FE			
UART_State.ind	SC, RE, TE, PE, TW	FF			

5. Software support

To use the KNX TinySerial Interface together with an operating system like Linux or Windows we recommend the kdriveExpress SDK. It is a cross-platform implementation to access the KNX system via different hardware interfaces.

See www.weinzierl.de for more information.

kdrive TinySerial Implementation

The kdrive TinySerial class is a C++ implementation of the TP-UART protocol described above. It communicates with the TinySerial module and offers an application interface via Common EMI (External Messaging Interface). The following outlines the operation and design of the TinySerialPort class.

Open

The following functions are performed on open:

- Open the Serial Port (19200, 8E1, No flow control)
- Reset the controller (5s timeout) see **Reset**.
- If reset is successful, sets the controller properties. See **Set Properties**.
- If reset is not successful, throws an Exception (open failed).
- Starts the receive thread.

Reset

To reset the TinySerial module we send U_Reset.request and wait for the reset indication U_Reset.ind. If we send the reset request and receive anything other than the reset indication we immediately resend the reset request again. That is, if there is traffic on the bus we continue to hit the TinySerial module with a reset

request until we directly receive a reset indication. In this way we can be somewhat certain that the response is a valid response. If we accept other bytes before we find the response we can't be certain that it is a response or part of a telegram sequence.

Set Properties

In set properties we simply write the individual address. To do this, we first disable non-selective IACK, then write the individual address, then enable non-selective IACK.

- Send U_Write.req 0x22 plus data 0x00 (disable IACK)
- Send U_Write.req 0x1F plus high byte of individual address
- Send U_Write.req 0x1E plus low byte of individual address
- Send U_Write.req 0x22 plus data 0x01 (enable IACK)

Rx Thread

The receive thread waits for telegram frames from the TinySerial module. See **Wait Telegram**. Once a valid KNX telegram is received it is added to a receive queue (where it is then routed to a packet notification mechanism). The rx thread also implements a receive timeout. If the timeout elapses it sends a connection state request to get the state of the bus, see **Get Connection State**.

Get Connection State

To determine the connection state, that is, whether the TP_UART is connected to the KNX bus, we send the U_State.request. If the response is not zero we set the connection state to connected, otherwise disconnected.

Wait Telegram

Wait telegram waits for a single character from the TinySerial module. If it does not receive a character within the timeout period (500ms) it simply returns. Once it receives a byte it checks to see if it is one of the following:

- Reset Indication UART_Reset.ind : set controller properties, see **Set Properties**
- Long Data L_LONG_DATA.request There are System, Alarm, High and Normal priorities and repeated or not-repeated which means there are 8 different long data bytes that we have to check. We differentiate between repeated and not-repeated. If it is repeated we receive it but discard it. We still receive it to ensure that our framing (state machine) is correct, that is, once we have received the repeated frame we know that we expect that start of the next frame. To receive a long data frame we use the following algorithm: receive 7 bytes. These are: `extended control frame (1 byte), src address (2 bytes), dest (2 bytes) length (1 byte) and apci (1 byte)`. From these bytes we can determine the expected length of the remainder of the frame by reading the length. We the read length bytes. Once we have these bytes we the read the single checksum byte, build the complete frame, see **Build Frame**, and calculate the checksum, see **Calculate Checksum**. If any of the above steps fail, we disregard the frame and return.
- Data L_DATA.request. Like the long data, there are 8 different types with 4 priorities and repeated/not-repeated combinations. The procedure is the same however the initial encoding format that we expect is as follows: `// src address (2 bytes), dest (2 bytes) address type/length (1 byte) and apci (1 byte)`. Once we receive the 6 bytes we can determine the length of the remaining data. This is determined from the address type / length byte (the lower 4 bits), i.e. addressTypeLength & 0x0F. We follow the same procedure as the long frames, receive data (i.e. calculated length), receive checksum byte, build the complete frame see **Build Frame**, and calculate checksum, see **Calculate Checksum**.
- Confirm L_DATA.conf. Indicates we have received a positive confirm from the transmission of a previous telegram. When we send a telegram we wait

for this confirm byte. That is, we use thread synchronization (an event) to wait on the confirm byte in the rx thread.

Build Frame

To build the complete frame we start with a L_DATA_Ind (0x29). We then add the control byte that we received from the TinySerial in WaitTelegram. This is one of the long data or data bytes, with the priority and repeated states. We then append the header frame (used to get the remaining length) and then the remaining data bytes (length). The checksum is not appended to the frame.

Calculate Checksum

```
unsigned char cs = 0xFF;
std::vector<unsigned char>::const_iterator iter = frame.begin() + 1;
std::vector<unsigned char>::const_iterator end = frame.end();
for (; iter != end; ++iter)
{
	cs ^= *iter;
}
cs ^= checksum;
return !cs ? true : false;
```