ElektorBus

Reference

By Michael Busser and Jens Nickel

CONTENT

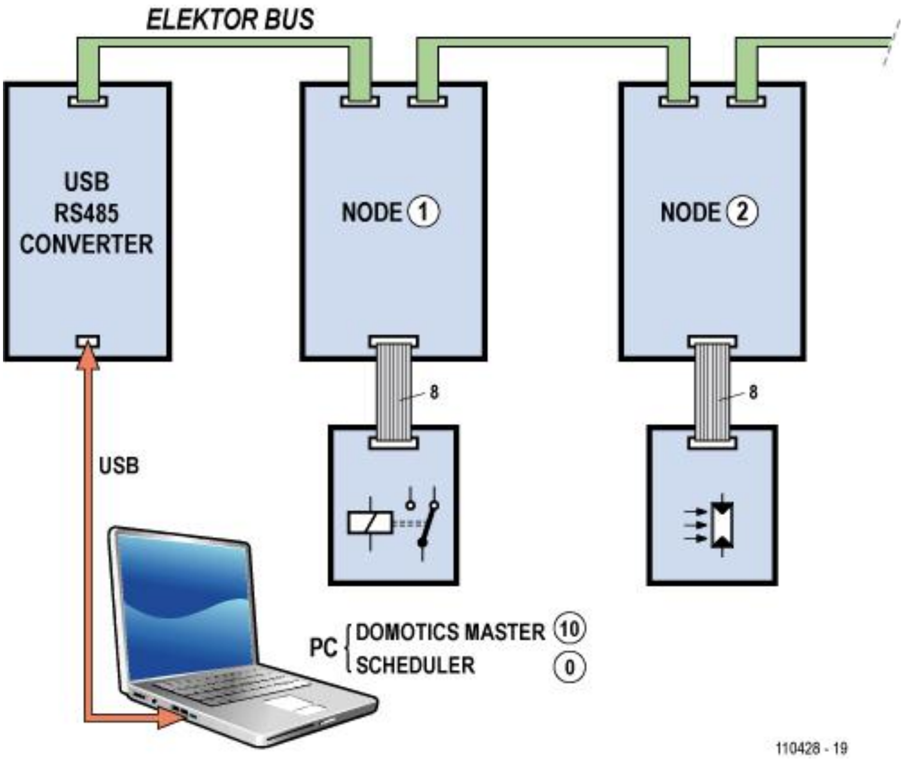
1. General

1.1. Basics / Bus System

Nodes are the participants of the communication.

Every node can talk directly to every other node.

There are 2 forms of communication:
* 1:1 (two nodes) and
* more nodes, all connected together on one bus.



The nodes have addresses, e.g. 0, 1, 2, 10.

One physical processor/board can combine the functions of more than one node, so it has more than one node-address.

Messages are byte-oriented, 1 byte = 8 bit

We use a protocol stack.
Protocols can be combined. For example, higher protocols can be used with different physical layers.

1.2. Physical layer

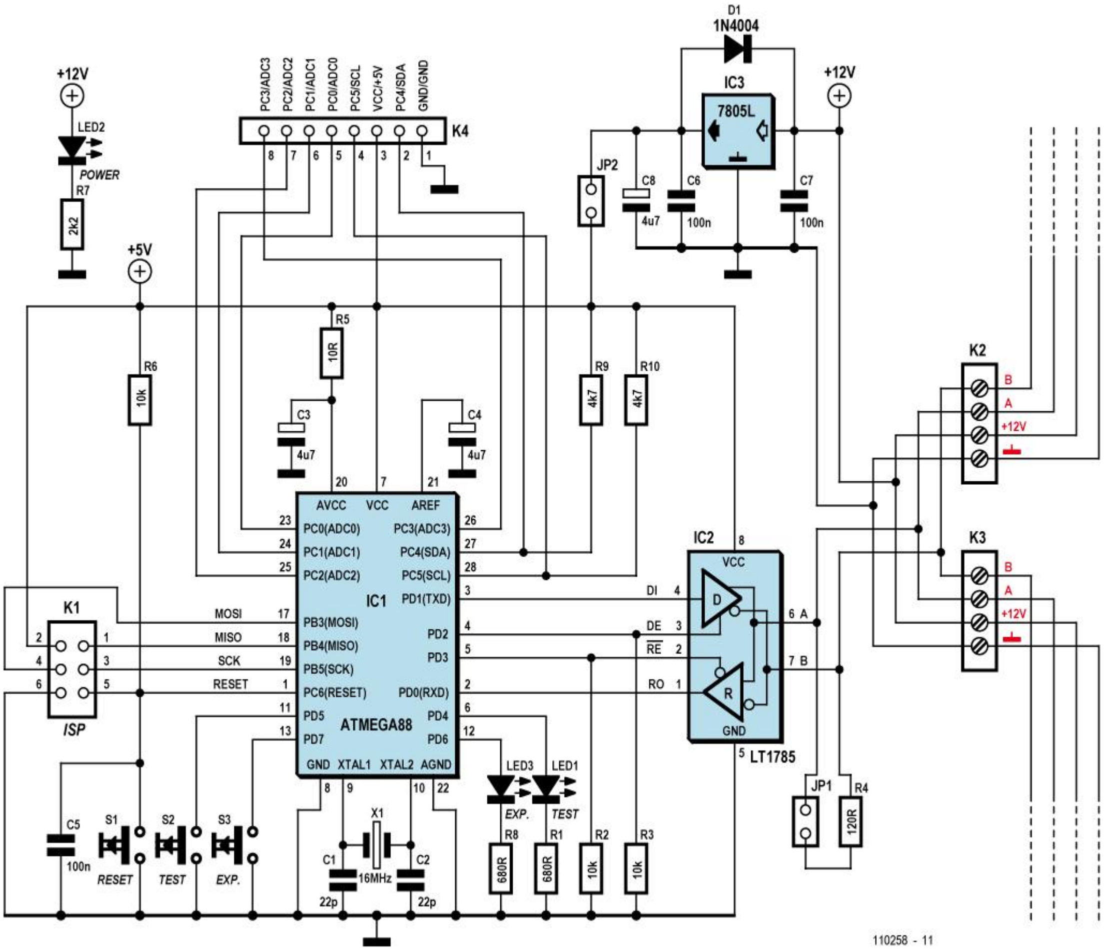
First implementation use RS485, UART-Protocol 8-N-1 and a power supply of the nodes over the bus. Others are possible.

Data-rate is 9600 Baud (= “1x”)
Higher data-rates possible, but not implemented yet.

Bus has 4 lines:

1. RS485-B
2. RS485-A
3. GND
4. 12 V

First hardware (ElektorBus Experimental node):



12-V and GND now swopped !!!!!

2. ElektorMessageProtocol

2.1. Basics

1. Messages have a fixed length of 16 Bytes.
2. The very first byte of every message is 0xAA which is used for synchronization purpose.
3. The second byte is a mode-byte determining the meaning of the following 14 bytes (and realizing an Ack-Mechanism).
4. ID follows (if ModeBit7 not 0).
   ID is always 1. Receiver-Address, 2. Sender-Address.
   A fragment-number is optional.
5. Application-Data follows. A higher protocol determines the details.
6. CRC or checksum is optional.
7. There is an acknowledge-mechanism at the Message-level.

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
0	1	0	1	0	1	0	1	0	=$AA (Start of Message)
1									Mode-Byte
2	Addressing and Fragmentation depending on Mode-Byte								ID-Byte 0
3									ID-Byte 1
4									ID-Byte 2
5									ID-Byte 3
6	Application data area								
7									
8									
9									
A									
B									
C									
D									
E	Might contain a CRC/Checksum depending on the Mode-Byte								Hi – CRC/Checksum
F									Lo – CRC/Checksum

2.2. Modebyte

8 bits of the Modebyte

7	6	5	4	3	2	1	0	
X								0 = ID-Bytes from Byte 2 1 = No ID-Bytes, payload from byte 2
	X							0 = 4 ID-Bytes (Byte 2..5) 1 = 2 ID-Bytes only (Byte 2..3)
		X						0 = with 16 bit CRC or Checksum 1 = without Checksum, can be used as additional data bytes
			X					0 = AAhex does not appear from byte 2 onwards 1 = advanced sync mechanism
				X				0 = all ID Bytes for addressing purposes 1 = last ID Byte is fragment nr
					X			0 = no segment address 1 = upper 6 bits representing the segment address
						X		0 = original message 1 = acknowledge message (see 2.3.)
							X	0 = no acknowledge message expected 1 = acknowledge message expected

Standard-Layout

(ModeBit7..ModeBit2 = 0)

	Bitposition								Meaning
Byte	7	6	5	4	3	2	1	0	
0	1	0	1	0	1	0	1	0	=$AA (Start of Message)
1	0	0	0	0	0	0	0	0	Mode-Byte = 0
2	Receiver address								Hi byte
3									Lo byte
4	Sender address								Hi byte
5									Lo byte
6	Application data area								
7									
8									
9									
A									
B									
C									
D									
E	Simple 14-bit-Checksum								7 bit High
F									7 bit Low

More Layout-Examples

Mode = 0x1C

	7	6	5	4	3	2	1	0	
0	1	0	1	0	1	0	1	0	=$AA Start
1	0	0	0	**1**	**1**	**1**	0	0	Mode-Byte = $1C
2	Segment Receiver						Node		ID-Byte 0
3	Receiver				Segment				ID-Byte 1
4	Sender		Node Sender						ID-Byte 2
5	Fragment-Number								ID-Byte 3
6									Data 0
7									Data 1
8									Data 2
9									Data 3
A									Data 4
B									Data 5
C									Data 6
D									Data 7
E	16-bit CRC								CRC
F									CRC

Mode = 0xA0

	7	6	5	4	3	2	1	0	
0	1	0	1	0	1	0	1	0	=$AA Start
1	1	0	1	0	0	0	0	0	Mode-Byte = $A0
2									Data 0
3									Data 1
4									Data 2
5									Data 3
6									Data 4
7									Data 5
8									Data 6
9									Data 7
A									Data 8
B									Data 9
C									Data 10
D									Data 11
E									Data 12
F									Data 13

2.3. Ack-Mechanism on Message-Level (MLevel)

Sender sends a message to Receiver.
If ModeBit1 = 1, the Receiver must send a (MLevel-)Ack-Message immediately back to the sender.

This Ack-Message contains exactly the same data-bytes as the Original-Message, receiver- and sender-address are swopped, ModeBit0 = 1, ModeBit1 = 0.

If ModeBit7..ModeBit2 = 0 (standard-layout), we have the following Mode-Bytes:

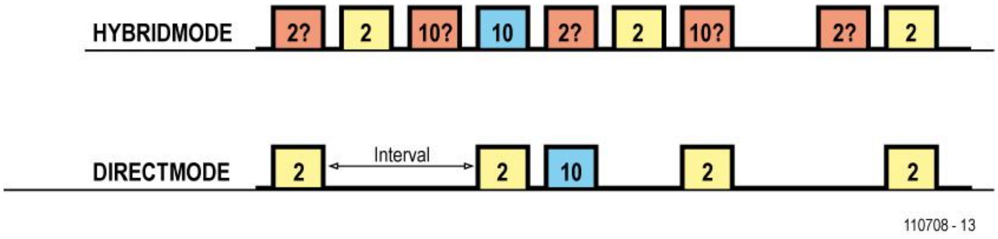
Mode = 2	Original-Message with request for Ack-Message.
Mode = 1	This is the Ack-Message.
Mode = 0	Original-Message, no Ack needed.

3. Collision Management

3.1. Basics

There is no hardware collision management, all is done in software.
The best Collision Management is avoiding collisions at all.
So every node must know when it is allowed to send a message.

There are two systems:
DirectMode is dedicated for 1:1 communication.
Hybrid Mode is dedicated for Bus-communication with more than two nodes.



3.2. Direct Mode

‘Direct mode’ is used when a bus participant (typically a sensor node) sends messages at predetermined time intervals (see figure). The other bus participant then uses these messages as a timebase. For example, if a controller wishes to send a message to the sensor, it can do so immediately after it sees the periodic message from that sensor.

Another possibility that is not yet implemented is the (more conventional) reverse of the above: the master generates the timebase and the slave replies. The master can send control commands as part of this exchange with the sensor node or can ask for particular readings.

3.3. Hybrid-Mode

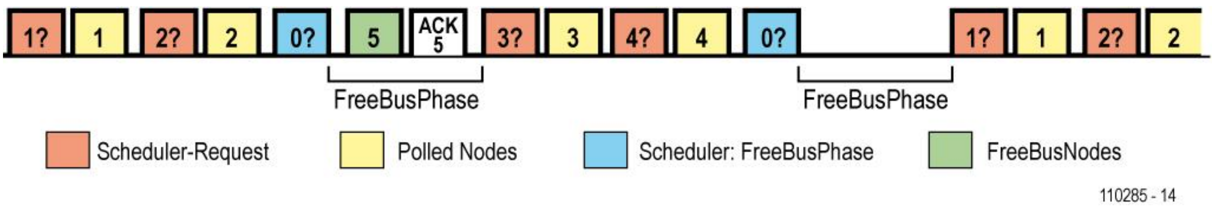
Scheduling

One node takes on the role of the scheduler. Its sender address is defined as 0, which makes it easy for the other nodes to recognise its messages. The scheduler maintains an array, with x elements, containing the addresses of the nodes that are to be scheduled cyclically. It is also possible, of course, to arrange for a particularly loquacious node to be interrogated more often than the others.

To schedule a node the scheduler sends out a special request message (SchedulerRequest), which includes the address of the polled node in the recipient address field. The scheduler then waits for a message with the same value in the transmitter address field (ResponseMessage), which can have any desired value in the recipient address field. The scheduler then turns to the next node in sequence and the process repeats. If a node fails to reply to a SchedulerRequest, the process would come grinding to a halt. For this reason a timer is started when the SchedulerRequest is sent out: if the timer expires without a reply being received, the scheduler stops waiting and moves on to the next polled node anyway.

FreeBusPhases

First, all the nodes that need to be interrogated periodically (such as temperature sensor nodes) are probed in turn. The scheduler then releases the bus for the unprompted transmission of messages. At this point any node that only occasionally has something to say (such as a light switch) is permitted to speak. The ‘free bus phase’ must of course only continue for a certain period of time, so that nothing is accidentally still being transmitted when the scheduling of the scheduled nodes resumes.



To start the FreeBusPhase the scheduler sends a special message, called *FreeBusMessage*. After this message was sent, the scheduler waits y milliseconds (70 to 100 ms).

FreeBusMessage
(send by the scheduler to inform the nodes about the upcoming phase of free bus access)
Receiver adress = 0
Sender adress = 0
Mode = 0

This message from one node of course can interfere with messages, send by other nodes at the same time in this phase.

Collision Detection in the FreeBusPhase

Due to to fact of possible collisions, the sending node may request an acknowledgement from the addressed receiver, typically (but not necessarily) the domotic master.
For this purpose we use the Ack-Mechanism of the MessageProtocol (see 2.3.).

A message in the FreebusPhase is formed by:

Receiver address = any address, typically, but not necessarily, the address of the domotic master
Sender address = the nodes address
Mode = 1 bit 0 = 1: acknowledge requested
bit 1 = 0: this is the original message

The addressed receiver must reply to such a message with a copy of the message except:

Receiver address = received sender address
Sender address = received receiver address = own address
Mode = 2 bit 0 = 0: acknowledge not requested
bit 1 = 1: this is an acknowledgement message

Collision Resolution and FreeBusPriority

If the sender of the original message doesn't receive an ack-message of the receiver, it sends the original message again. If 2 senders are sending at the same time, the collision must be resolved. So the 2 senders are waiting a different amount of FreeBusPhases. If the FreeBusPriority is 2, a sender will wait for 2 FreeBusPhases until it is allowed to send again.
Two senders which are allowed to send in the FreeBusPhase (=FreeBusNodes) must always have a different FreeBusPriority. One can take the Address as FreeBusPriority, or any other system to ensure that.

4. ElektorApplicationProtocol

4.1. Basics

The MessageProtocol does not define the layout and meaning of the data-bytes (payload).
So we need an application protocol mutually understood by the nodes on the bus (both sensors and actuators) and which will allow easy expansion to accommodate new hardware. So that we do not have to reinvent the protocol every few months, the ElektorApplicationProtocol is relatively simple and yet also flexible, fulfilling the following requirements as a minimum.

- Transmission of ten-bit values plus sign, either a reading from a sensor or, in the other direction, a control value to an actuator.
- The option to use twenty-bit values plus sign, for which we need a four-byte-per-channel mode.
- Setting of units and scaling factors for smart sensor nodes.
- Setting of measurement interval for sensor nodes.
- Setting of multiple thresholds.
- Notification of above- or below-threshold alarms.
- Configuration and calling-up of default presets for actuators (not implemented yet).
- acknowledge mechanism on Application level

Parts

We don’t need a whole message for each of the features above.
For example, a master can set a threshold and an interval on one sensor with only one message.
Another example: we can have more sensors at one node, to save costs. More than one sensor can send its value with only one message.
All these information-units (e.g. sensor-values, setting tresholds and so on) are called ***parts***. There are parts with 2 and 4 bytes. So with 8 data-bytes we can transport up to 4 parts in one message.

4.2. Channels and channel-addressing

At one node, we can address up to 8 actors and sensors. We address those “sub-nodes” with a channel-address 0..7.

Implicit addressing

If we want to send a part with 2 bytes, defining a 10-bit-Value, to or from a sensor/actor on Channel 0..3, we can use implicit addressing.
The part for the distinct channel is defined by its position.
So we can send or receive up to 4 of these 10-bit-values to/from a sensor-/actor-node within one message.

	Bitposition								Meaning
Byte	7	6	5	4	3	2	1	0	
0	1	0	1	0	1	0	1	0	=$AA (Start of Message)
1	0	0	0	0	0	0	0	0	Mode-Byte = 0
2	Receiver adress								Hi byte
3									Lo byte
4	Sender adress								Hi byte
5									Lo byte
6	Channel 0								Hi byte
7									Lo Byte
8	Channel 1								Hi byte
9									Lo Byte
A	Channel 2								Hi byte
B									Lo Byte
C	Channel 3								Hi byte
D									Lo Byte
E	Might contain a 16bit CRC value depending on the Mode-Byte, bit 5								
F									

Explicit addressing

For all the other parts we use an explicit addressing. We use the Bit2..Bit0 of the first byte of the part to define the channel-number.

4.3. 2-Byte-Part vs. 4-Byte-Part

The receiver decodes the data-bytes, beginning with the first data-byte.
There are parts with 2 Bytes and parts with 4 Bytes. When decoding the data-bytes, the receiver must know when a new part begins, so it must know how long the parts are.

We use Bit6 of the first data-byte for this purpose.

Hi.6 = 1 → this is a 2-byte-part

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
		1							Hi byte
									Lo byte

Hi.6 = 0 → this is a 4-byte-part

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
		0			**1**				Address
									Command / High
									First / Middle
									Second / Low

The decoder starts with the first data-byte 0. Then it decodes the first part. The next part begins at data-byte 0+x, x is 2 or 4.

If the first byte of a part is completely zero, this is a **void-part** with no information (2 bytes long). Because of that, the Bit3 of the first byte of a 4-byte-part must be 1.

4.4. Value-Parts vs. Command-Parts

To avoid a “AA” in the data for simple synchronization, the Bit7 of all the bytes of one part is always 0, when number-values are transported.
But we can use this Bit7, if we only have distinct byte-values. These distinct values can encode commands, e.g. D1hex or C1 hex, see below. A command part always begins with a first byte for 2-/4-byte-part-determination and channel-addressing. Then a second byte follows, encoding the distinct command. The Bit7 of the second byte is always 1.
So we can determine with Bit7 of the second byte of a part, if the part transports a numerical value or a command.

2-Byte-Command-Part

C = Channel-Address-Bits

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
		1				C	C	C	Address
	1								Command

4-Byte-Command-Part

C = Channel-Address-Bits

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
		0			**1**	C	C	C	Address
	1								Command
									First Parameter-Byte
									Second Parameter-Byte

4.5. The Set-Bit and the Ack-Bit

Bit5 of the first byte of a part is the **Set-Bit**. It defines if we want to set a value on a channel (=1) or if we get a measurement value from that channel (=0).

Bit4 of the first Byte of a part is the **Ack-Bit**. With this bit we can determine if this is the original message or the acknowledge-message. So we can realize another acknowledge-mechanism.
Note: this is an acknowledge-mechanism on application level, it is independent from the acknowledge-mechanism on message-level, see 2.3..
Note: There is no acknowledge-requested-flag (like we have on message level, see 2.3.). The receiver must know that it is requested to send an ack-message.

4.6. Defined Value-Parts

Value2: 10bit incl. sign

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
	0	**1**	**SC**	**AO**	**S**	**D9**	**D8**	**D7**	Hi byte
	0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	Lo byte

D9..D0	10 databits representing the value
SC = 0	The value is a current value, e.g. a value from a sensor element
SC = 1	The value is set on the receiver of this part.
AO=0	Indication that this is the original message
AO=1	Acknowledge-message (at the application protocol level)
S=0	Sign, 0 → +
S=1	Sign, 1 → -

This part can be used by a sensor for example, who tells us its actual sensor value.

CALCULATION:
Representing values from –1023 to +1023
SIGN = 8 for negative values, 0 otherwise
LOW = lower seven bits of magnitude (in BASCOM: Low = Value And 127)
HIGH = upper three bits of magnitude (in BASCOM: Shift Value, Right, 7 : High = Value)

	Byte 1	Byte 2
Transmit reading	64 + SIGN + HIGH	LOW
Set value	96 + SIGN + HIGH	LOW
Switch on	96	1
Switch off	96	0

Acknowledgement from receiver: original byte 1 value plus 16.

Value4: 19bit incl. sign

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
	0	0	**SC**	**AO**	**1**	**C2**	**C1**	**C0**	Address
	0	0	**S**	**D 18**	**D 17**	**D 16**	**D 15**	**D 14**	High
	0	**D 13**	**D 12**	**D 11**	**D 10**	**D9**	**D8**	**D7**	Middle
	0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	Low

SC = 0	The value is a current value, e.g. a value from a sensor element
SC = 1	The value is set on the receiver of this part.
AO=0	Indication that this is the original message
AO=1	Achnowledge-message (at the application protocol level)
C2..C0	Identifies the channel belonging to this part (0..7)
S=0	Sign, 0 → +
S=1	Sign, 1 → -
D18..D0	19 databits representing the value

ValueFloat: transport a floating point value

	Bitposition								Meaning
Byte	7	6	5	4	3	2	1	0	
	0	0	**SC**	**AO**	**1**	**C2**	**C1**	**C0**	Address
	0	**1**	**S**	**MS**	**M3**	**M2**	**M1**	**M0**	High
	0	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**	**D7**	Middle
	0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	Low

SC = 0	The value is a current value, e.g. a value from a sensor element
SC = 1	The value is set to the receiver of this part.
AO=0	Indication that this is the original message
AO=1	Acknowledge-message (at the application protocol level)
C2..C0	Identifies the channel belonging to this part (0..7)
S=0	Sign of mantissa, 0 → +
S=1	Sign of mantissa, 1 → -
MS=0	Sign of exponent, 0 → +
MS=1	Sign of exponent, 1 → -
M3..M0	4 databits to encode the exponent
D13..D0	14 databits representing the mantissa

Even measurements of electrical quantities often require precision spanning a range of several orders of magnitude. For such cases we can use four bytes to represent a reading or setting. The figure shows how an individual sensor or actuator attached to a node is addressed using the channel bits C1 and C2 in the first byte. The bytes labelled ‘High’, ‘Middle’ and ‘Low’ carry the actual value. High.6 is set to indicate that the bytes represent a floating-point value; High.5 gives the sign of the mantissa. MS, M3, M2, M1 and M0 give the exponent (as a power of ten), and the remaining fourteen bits (D13 down to D0) give the magnitude of the mantissa. The largest number that can be represented is $+16383*10^{+15}$.

4.8. Defined Command-Parts

Limit: Set a threshold / Alarm

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
	0	**1**	**LA**	0	**1**	**C2**	**C1**	**C0**	Hi byte
	1	**1**	0	**1**	0	0	**L1**	**L0**	Lo byte

C2..C0	Identifies the channel belonging to this part (0..7)
LA=0	Alarming, channel C2..C0 exceeds upper or lower limit
LA=1	Use the actual value as a upper or lower limit at channel C2..C0
L1..L0	10 → upper Limit 01 → lower limit 00 → alarming 11 → undefined

This part is used to set the actual value of a sensor as a threshold.

CALCULATION:
CH = channel number

	Byte 1	Byte 2
Set lower threshold	104 + CH	209
Set upper threshold	104 + CH	210
Alarm: value below threshold	72 + CH	209
Alarm: value above threshold	72 + CH	210
Value between thresholds	72 + CH	208

Acknowledgement from receiver: original byte 1 value plus 16

Scale: set unit, scaling and physical quantity to a smart sensor

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
	0	0	**1**	0	**1**	**C2**	**C1**	**C0**	Address
	1	**1**	0	0	0	0	0	**1**	Command (=193 dec.)
	0	**Phys. Quantity**							First
	0	**unit**		**S**	**scale**				Second

C2..C0	Identifies the channel belonging to this part (0..7)
S=0	Sign of exponent, 0 → +
S=1	Sign of exponent, 1 → -
Phys. Quantity	01_{hex} = 1 = raw ADC-Value 10_{hex} = 16 = Voltage 11_{hex} = 17 = Current 12_{hex} = 18 = Resistance 14_{hex} = 20 = Power 21_{hex} = 33 = Temperature 22_{hex} = 34 = Humidity 24_{hex} = 35 = Pressure
Unit	00 = SI-Units
S=0	Sign of exponent, 0 → +
S=1	Sign of exponent, 1 → -
scale	0..15 Provides the exponent (base=10)

Example:

Phys. Quantity = Current,	S=1,	scale=3	→ mA
Phys. Quantity = Resistance,	S=0,	scale=3	→ kΩ
Phys. Quantity = Resistance,	S=0,	scale=6	→ MΩ

CALCULATION:
CH = channel number
POT = exponent (‘power of ten’) absolute value
PSIGN = 16 for negative exponent, 0 otherwise

	Byte 1	*Byte 2*	*Byte 3*	*Byte 4*
Set	40 + CH	193	see above	PSIGN + POT
Voltage in V	40 + CH	193	16	0
Voltage in mV	40 + CH	193	16	19
Current in mA	40 + CH	193	17	19

Trigger transmission of preset quantity and units from sensor: byte 1 = 8 + CH

Interval: set interval to a smart sensor

Byte	Bitposition								Meaning
	7	6	5	4	3	2	1	0	
	0	0	**1**	0	**1**	**C2**	**C1**	**C0**	Address
	1	**1**	**1**	0	0	0	0	0	Command (E0 hex.=224 dec.)
	0	**Interval Value**							First
	0	**Interval Scale**							Second

C2..C0 Identifies the channel belonging to this part (0..7)

Interval value 7 bit Value 0..127

Interval Scale

Hex	Dec	Interval
04	4	1 µs
05	5	10 µs
06	6	100 µs
07	7	1 ms
08	8	10 ms
09	9	100 ms
0A	10	1 s
0B	11	10 s
0C	12	100 s
10	16	1 minute
11	17	10 minutes
12	18	100 minutes
18	24	1 hour
19	25	10 hours
20	32	1 day
21	33	10 days
22	34	100 days
28	40	1 month
30	48	1 year
31	49	10 years

Requesting readings

Note: this is not implemented yet in the Javascript Library JSBus, see 5.

It is possible to use the application protocol to set a target value on a node from a controller. Sensor nodes can also report current readings. Until now it has however not been possible to prompt a particular sensor or actuator node to send these values: the scheduler does divide up the transmit time slots, but does not carry out polling in the strict sense of the word.

CALCULATION:

Reading request

	Byte 1	Byte 2
Request reading	104 + CH	240 ($F0_{hex}$)
Request lower threshold	104 + CH	241 ($F1_{hex}$)
Request upper threshold	104 + CH	242 ($F2_{hex}$)

In the above, ‘CH’ represents the channel number from 0 to 7.

Absolute Treshold

Note: this is not implemented yet in the Javascript Library JSBus, see 5.

This is a format to transmit absolute threshold values, as so far we have only been able to use the current reading as the setting for an upper or lower threshold.

CALCULATION:

Set absolute threshold

	Byte 1	Byte 2	subsequent bytes
Set lower threshold	104 + CH	217 ($D9_{hex}$)	value (2 or 4 bytes)
Set upper threshold	104 + CH	218 (DA_{hex})	value (2 or 4 bytes)

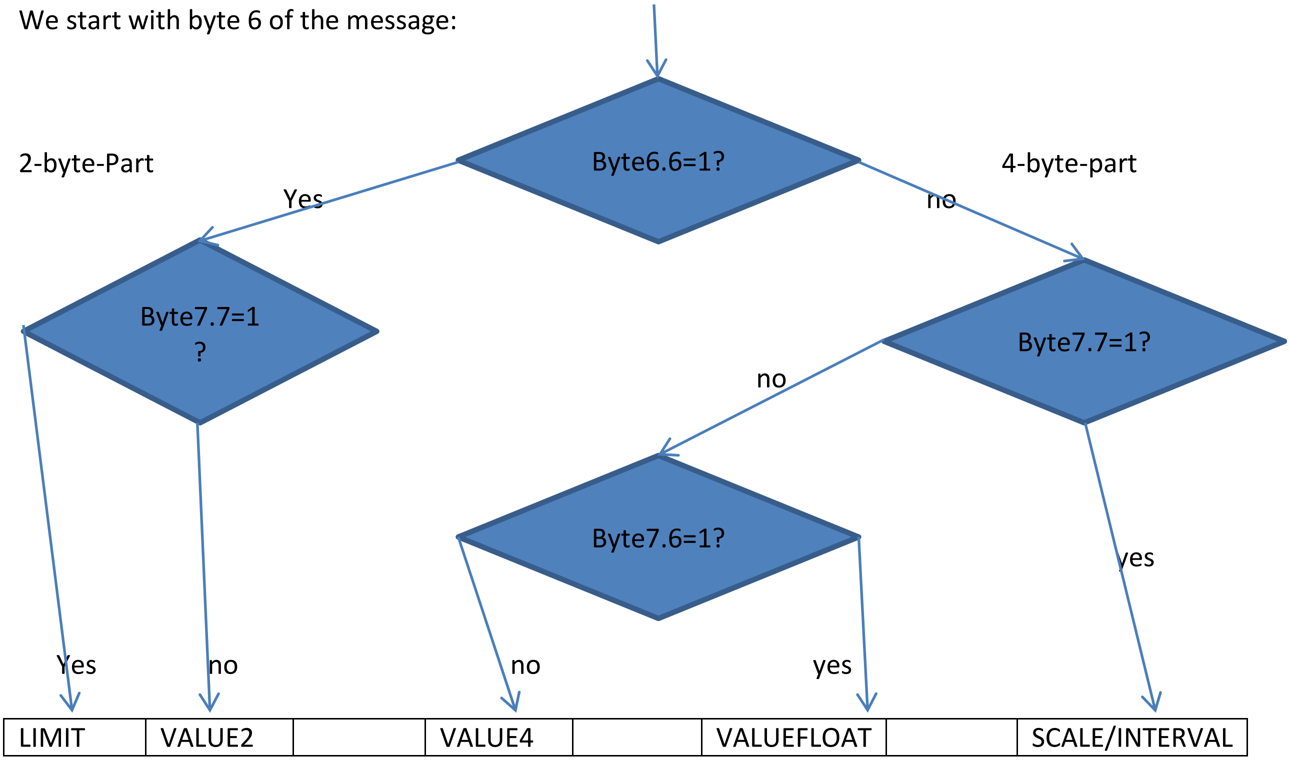
Report absolute threshold value from sensor: byte 1 = 72 + CH

In the above, ‘CH’ represents the channel number from 0 to 7.

4.8. Evaluate part type at a received message

The interpretation of the data in the application data area (byte 6..13 of the message) depends on the type of the part sended. So the first goal is to figure out the type of the part. There are some Symbols defined for the different type of parts.

We start with byte 6 of the message:



Now we know, if the part has 2 or 4 byte, we proceed with byte 8 in case of a 2-byte-part or with byte 10 in case of a 4-byte-part.

If we received a part type VALUE2, we also have to derive the channel number from the position of the part itself inside the message (see 4.3.).

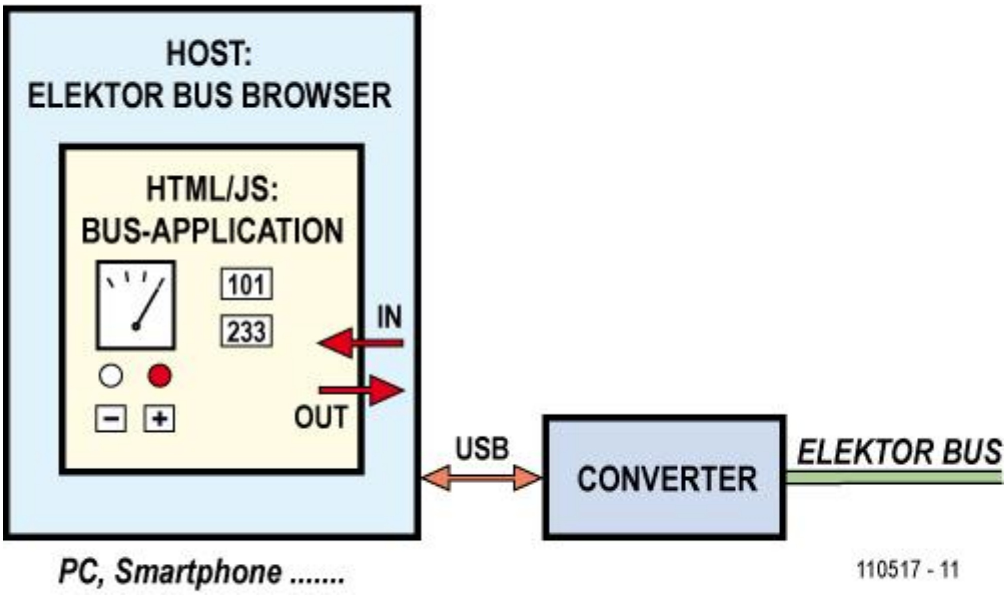
5. Rapid Application Development

5.1 Basics

On the PC/Master-side. A C-library for the controller side will follow.

We would ideally like to have a library which
- implements the ElektorBus protocol, freeing the developer to concentrate on the application proper;
- provides a clear separation between the application code and protocol code;
- makes it easy for an electronics engineer to design and program a user interface; and
- is platform-independent, so that the same application can run equally well on a PC and on a smartphone.

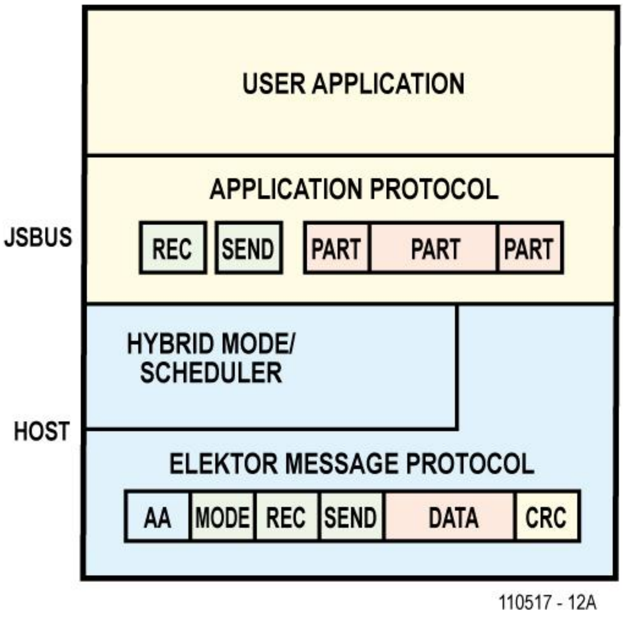
HTML-Approach:



We use a kind of a browser, which can display our tailor-made Bus User Interface, realized with HTML. The HTML and Javascript code form the core of the application, wrapped within the browser which itself is written in a more conventional programming language such as Visual Basic .NET or Java. We can think of the ElektorBus browser the ‘host’ in our system.

5.2. Implementation of the protocol stack

In principle it would be possible to implement all three bus protocols (the ‘Elektor Message Protocol’, ‘Hybrid Mode’ (which is optional) and the ‘Application Protocol’) within the host. On the other hand, it would be possible to make the host transparent, passing the 16 raw bytes in a received message packet directly through to the Javascript code, where the details of the protocol could be implemented. We choose a middle road: the simple Elektor Message Protocol and the rather timing-sensitive Hybrid Mode and scheduler are implemented within the host, while the Application Protocol, which requires rather more code and which some readers will perhaps want to extend, is implemented with the help of a small Javascript library JSBus.



5.3. The In-/Out-Command

The host receives the sixteen bytes of the message sent over the bus using the start byte synchronisation system. The message is ‘unpacked’ into a data structure that contains (among other things) the transmitter address, the receiver address and the eight payload bytes. These parts are then encoded into a string (called ‘InCommand’) and passed in to the Javascript code. The InCommand string is formatted as plain ASCII (see the text box) which ensures that it will be treated compatibly across different platforms.

The ElektorBus application, written in Javascript, and the ElektorBus browser, written (for example) in Visual Basic .NET, communicate with one another using these simple text strings. The JSON syntax is used to encode the necessary information in a data structure within the string to be passed outwards from the Javascript application to the host or inwards from host to Javascript application. The data structures for InCommand and OutCommand are very similar.

OutCommand

Command	command type (‘Send’ or ‘Url’ or ‘Scheduler’ or ‘SMS’)
Url	‘Url’: file name for HTML page to be loaded. ‘SMS’: SMS-Number or ‘1’ for sending an SMS to default SMS-number
Options	‘SMS’: SMS-Text
Mode	mode byte for the message to be sent (needed as part of the acknowledge mechanism)
Receiver	receiver address
Sender	transmitter address
Data	‘Send’: array of eight data bytes. ‘Scheduler’: addresses of up to eight scheduled nodes

InCommand

Command	command type (‘Rec’ or ‘Status’ or ‘SetAddress’)
Mode	mode byte of the received message (Status 2 = OK; –1 = error)
Valid	checksum OK? (not yet implemented)
Receiver	receiver address
Sender	transmitter address
Data	‘Rec’: array of eight data bytes. ‘SetAddress’: First data byte is the Address of the node.

JSON Syntax

In JSON syntax an InCommand appears as in following example:

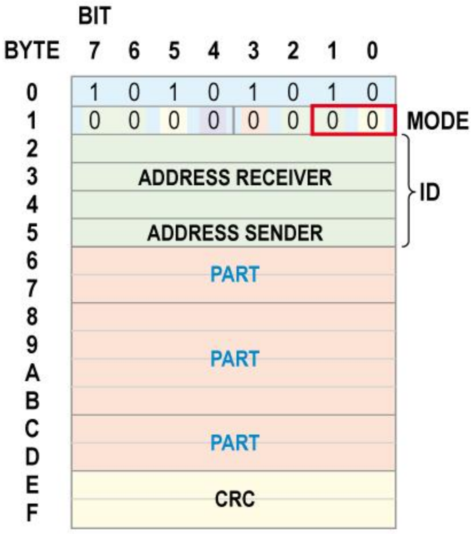
```
{"Command":"Rec","Mode":0,"Valid":0,"Sender":2,"Receiver":10,"Data":[0,0,64,1,0,0,0,0]}
```

5.4. Messages and Parts

The Javascript library works internally with two data structures to describe messages and parts (items of payload information such as two-byte values, alarm reports, quantity settings and so on) that are being transmitted and received.

The ***Message*** object basically consists of the familiar components of an ElektorBus message.

Mode	mode byte
Receiver	receiver address
Sender	transmitter address
Data	array of eight data bytes
Valid	checksum OK? (not yet implemented)



110517 - 13

Within the eight data bytes we can convey up to four parts in accordance with the Application Protocol. Each ***Part*** is characterised by the following properties.

Valid	check sum OK? (not yet implemented)
Sender	transmitter address
Receiver	receiver address
Channel	channel number
Setflag	desired setting or current value?
Ackflag	acknowledge message or original message (application-level flag)
Mode	message's mode byte (with message-level acknowledge flags)
Parttype	type of part, with the following constant values defined: PARTTYPE_VALUE2, PARTTYPE_VALUE4, PARTTYPE_VALUEFLOAT, PARTTYPE_LIMIT, PARTTYPE_SCALE, PARTTYPE_INTERVAL
Numvalue	numerical data value (for example from –1023 to 1023 in the case of PARTTYPE_VALUE2)
Limit	0 = value between thresholds; 1 = below lower threshold; 2 = above upper threshold
Quantity	physical quantity (from 0 to 127, see 4.8.)
Unit	unit of measurement (from 0 to 3, see 4.8.)
Scale	power of ten scaling (from –15 to +15)
Interval	Interval unit/scale (from 0..127, see 4.8.)
Preset	reserved
Options	reserved

5.5. The Javascript Library JSBus

Main variables/functions in the JSBus Javascript library:

ownAddress

To allow dynamic address selection the Javascript library defines a variable `ownAddress`. We can switch the Address in the host, the address to a new value it is passed on to the Javascript (by an InCommand with Type ‘SetAddress’) and the variable `ownAddress` is suitably modified. The variable can then be used in the node code. For example, a node would send the status of its test LED using the following code:

```
var parts = InitParts();
parts = TransmitValue(parts, ownAddress, 10, 1, 0, LedStatus);
SendParts(parts, true);
```

Parttypes

var PARTTYPE_VALUE2 = 2;
var PARTTYPE_VALUE4 = 4;
var PARTTYPE_VALUEFLOAT = 12;
var PARTTYPE_LIMIT = 32;
var PARTTYPE_SCALE = 48;
var PARTTYPE_INTERVAL = 64;

Encoding and sending Parts

```
function InitParts()
```

Returns an empty array of parts. Called as follows: var parts = Initparts();.

```
function SetLimit(parts, sender, receiver, channel, mode, limit, numvalue)
function SetScale(parts, sender, receiver, channel, mode, quantity, unit, scale)
function SetValue(parts, sender, receiver, channel, mode, setvalue)
```

These functions append a new part to an existing array parts, respectively representing a threshold, a quantity, unit and scaling value, and a set-point for a given sensor or actuator. The return value is the extended array.

```
function TransmitValue (parts, sender, receiver, channel, mode, value)
```

This function is comparable in operation to `SetValue`, except that here the master does not send a value: instead a node sends a value to the master.

Quantity-constants:

var RAWVALUE = 1;
var VOLTAGE = 16;
var CURRENT = 17;
var RESISTANCE = 18;
var POWER = 20;
var TEMPERATURE = 33;
var HUMIDITY = 34;
var PRESSURE = 36;

Example:

```
var parts = InitParts();
parts = SetScale(parts, 10, 2, 0, 0, TEMPERATURE, 0, -4);
SendParts(parts, true);
```

`SetIntervalValue(parts, sender, receiver, channel, mode, interval, numvalue)`
Like SetScale, but to set an interval on smart sensor

Interval-constants:

```
var INTERVAL_MILLISECONDS = 7;
var INTERVAL_CENTISECONDS = 8;
var INTERVAL_DECISECONDS = 9;
var INTERVAL_SECONDS = 10;
```

`function SendParts(parts, overrideQueue)`
Encodes and sends all parts in the array in one or more messages.

`function PartText(part)`
Returns a textual representation of a part, for example for debugging purposes.

User Interface Control-Element Functions

id = ID of the HTML-Control-Element

`function RadioButtonSetvalue(id, setvalue)`
Sets or resets a radio button (setvalue = 0 or 1).

```
function TextboxSetvalue(id, setvalue)
function TextSetvalue(id, setvalue)
function TextboxSetvalueScaled(id, setvalue, scale)
```

Sets the text in a text box or text element.
Scale is the exponent of a floating point value.

Functions controlling the Host

`function GotoUrl(url)`
Causes the host to load a new HTML page (url = file name with HTML-code without trailing '.htm' extension).

`function SetScheduler(status, schedulednode1, … , schedulednode8)`
Switches the scheduler in the host on or off (status = SCHEDULER_ON or SCHEDULER_OFF or SCHEDULER_DIRECTMODE) and provides the scheduler with a new list of nodes that should be regularly requested to send a message. A zero value terminates the list.

```
var SCHEDULER_OFF = 1;
var SCHEDULER_ON = 2;
var SCHEDULER_DIRECTMODE = 3;
```

`function SendSMS(number,text)`
Number can be an SMS-number or '1'. '1' means that the Default-SMS-number (to be set in the Host-application) shall be used.

Functions to be called by the library JSBus

```
function ProcessPart (part)
```

The library will call that function for every part of received message. It is absolutely necessary that you implement this function in every HTML-Page of your User-Interface.

Normally, the host processes the scheduler messages, as they are part of the collision management system HybridMode. But one can configure the host that the scheduler Messages are also given to Javascript. JSBus also calls `ProcessPart` in that case, with a part = null. You must check for this null-value in your function code. See 5.6..

Functions to process received parts / Automatic Ack-mechanism

```
function ProcessReceivedParts(parts)

        var ackparts = InitParts();

        for (var p = 0; p < parts.length; p = p + 1)
        {
                ProcessPart(parts[p])

                if (parts[p].Mode == 1)
                {
                        ackparts = AddAutoAckPart(ackparts,parts[p]);
                }
        }
        SendParts(ackparts, false);
}
```

This function is called if one message is received and decoded to parts. `Parts` is the array of parts. This function performs an automatic Ack-mechanism on message level (see 2.2.). All parts of a message with Mode == 1 are automatically encoded back and the Ack-message is sent out with Mode==2.
For each part, the function `ProcessPart` is called (implemented in the HTML-User-Interface-Pages)

Important Built-in-Javascript-functions

```
var sendinterval = setInterval("SendValues()", 500);
```

The first parameter expected by the function `setInterval` is the name of another function, which is to be called regularly. The second parameter is the interval between these calls in milliseconds. The return value is a variable that uniquely identifies this repeating action: the value can be reused later to stop the repeating action by calling
`clearInterval(sendinterval).`

5.6. Structure of an HTML-UI-Page

```
<SCRIPT src='JSBus.txt' Language='javascript' ></SCRIPT>
<SCRIPT Language='javascript' >

function ProcessPart(part)
{
      if (part==null)
      {
            CODE TO PROCESS THE SCHEDULER MESSAGES
      }
      else
      {
            CODE TO PROCESS THE RECEIVED PARTS
      }
}


OTHER USERDEFINED FUNCTIONS

</SCRIPT>

<FORM Name='Bus'>
<STYLE type='text/css'>

      CSS STYLE DEFINITIONS

</STYLE>

HTML CONTROLS CODE

</FORM>
```

Example

```
<SCRIPT src='JSBus.txt' Language='javascript' ></SCRIPT>
<SCRIPT Language='javascript' >
function ProcessPart(part)
{
        if (part==null)
        {

        }
        else
        {
                if (((part.Sender == 1)||(part.Sender == 2)) && (part.Parttype ==
                PARTTYPE_VALUE2))
                {
                        if (part.Channel == 1)
                        {RadioButtonSetvalue('LED'+part.Sender,part.Numvalue);};
                }

                if ((part.Sender == 2) && (part.Parttype == PARTTYPE_VALUE2))
                {
                        if (part.Channel == 0) {TextboxSetvalue('ADC', part.Numvalue);};
                }
        }

}

function SetSensorScale(quantity)
{
        var parts = InitParts();
        parts = SetScale(parts, 10, 2, 0, 0, quantity, 0, 0);
        SendParts(parts, true);

        if (quantity==RESISTANCE) {TextSetvalue('unit','Ohm');};
        if (quantity==RAWVALUE) {TextSetvalue('unit','ADC-Value');};
}
</SCRIPT>

<FORM Name='Bus'>

<STYLE type='text/css'>
#head {font-size:20}
</STYLE>

<DIV ID='head' >ElektorBusBrowser </DIV> <br/>

Scheduler

<BUTTON Type='button' onclick='javascript:SetScheduler(SCHEDULER_ON,2,10,0,0,0,0,0,0)' >
on</BUTTON>

<BUTTON Type='button' onclick='javascript:SetScheduler(SCHEDULER_OFF,2,10,0,0,0,0,0,0)' >
off</BUTTON>

<br/><br/><br/>

LED Node 1
<INPUT Type='radio' ID='LED1' Name='LED1' Value='LED1' />

LED Node 2
<INPUT Type='radio' ID='LED2' Name='LED2' Value='LED2' /> <br/><br/>

<INPUT Type='text' ID='ADC' Value='' /> <SPAN ID='unit' >ADC-Value</SPAN> <br/>

<BUTTON Type='button' onclick='javascript:SetSensorScale(RESISTANCE)'>Ohm</BUTTON>
<BUTTON Type='button' onclick='javascript:SetSensorScale(RAWVALUE)'>Adc raw</BUTTON>

<br/><br/>

<BUTTON Type='button' onclick='javascript:GotoUrl(“Limit“)'>Set-Limit-Page</BUTTON> <br/><br/>

</FORM>
```

6. Appendix

6.1. AVR-Democode: Meaning of the Bytes in AVR-EEPROM

Byte		
00		
01		
02	OwnAddress	
03		
04	Scheduling 01 = Scheduled 00 = FreeBusNode	
05	FreeBusPriority 0x = we must wait x FreeBusPhases in case of collisions	
06	Type of Device	
07		
08		
09		

6.2. Message Examples

```
→   AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FreeBusMsg
→   AA 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 Query to 02
←   AA 00 00 0A 00 02 47 7F 40 00 00 00 00 00 BB CC Reply from 02 to 0A
→   AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
→   AA 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00
←   AA 00 00 0A 00 02 47 7F 40 00 00 00 00 00 BB CC
→   AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
→   AA 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00
←   AA 00 00 0A 00 02 47 7F 40 00 00 00 00 00 BB CC
→   AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
→   AA 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00
←   AA 00 00 0A 00 02 47 7F 40 00 00 00 00 00 BB CC
```

→ SchedulerMessage
← Message from scheduled node

S: AA 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
R: AA 00 00 0A 00 01 00 EB 18 1C 1D 02 03 44 00 00

S: AA 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
R: AA 04 00 00 00 01 00 EB 18 1C 1D 42 03 43 00 00

6.3. HTML and Javascript basics

1. An HTML element starts with an opening tag <TAG... > and ends with a closing tag </TAG>. Between the tags, the element can contain text and more HTML elements. If an element does not have any content it can be closed within the start tag: <TAG... />.
2. So-called 'attributes' are used to qualify a tag further. Each takes the form AttributeName = 'AttributeValue'. Attribute values can be enclosed in either single or double quotation marks. If the attribute value must itself contain quotation marks (for example when it includes a call to a Javascript function) the nested quotation marks should be of the other sort than the enclosing quotation marks.
3. HTML is not case sensitive, and so upper and lower case letters can be freely mixed. The official recommendation is to write all tags and attribute names in lower case. However, when calling Javascript functions, for example when setting the 'onclick' attribute of a button, strict attention must be paid to the correct capitalisation of function names and variables.
4. It is advisable to give each element a meaningful and unique ID attribute. Plain text can be enclosed within a <SPAN> element or, if it is to have a paragraph to itself, within a <DIV> element.
5. The <DIV> tag produces a new paragraph. An ordinary line break can be produced using the <BR/> tag.
6. Javascript code within an HTML file must be enclosed between <SCRIPT> tags as shown in the example listing. Within the code identifiers are strictly case sensitive.
7. In Javascript, basic blocks are enclosed within curly brackets. Unlike C, Javascript does not require a semicolon after each statement, but it is recommended.
8. Comments are introduced by a pair of slash characters.
9. A function call always requires a pair of brackets after the function name, even if there are no parameters. A function returns a value using the keyword `return`. A subroutine that has no return value is still considered a function, and its definition is still introduced by the keyword `function`.
10. Conditional statements can be introduced by the `if` keyword (in lower case!) with the condition itself always enclosed in brackets. The sequence '`||`' corresponds to 'OR' in Basic and '`&&`' to 'AND'. And it cannot be stressed enough that equality comparisons require a doubled equals sign. The expression '`!=`' means 'is not equal to'. The boolean values '`true`' and '`false`' are written thus, in lower case like most Javascript keywords.
11. Array indices are enclosed in square brackets and always count from zero. Arrays must be properly declared: see the function `InitParts()` in the JSBus library.
12. Simple variables and constants are declared using the `var` keyword. Javascript distinguishes two types of simple variable: strings and integers. Numbers and strings can be mixed in expressions without having to specify the type conversions explicitly. For example the expression 'TEXT'+1 evaluates to 'TEXT1'.

Various HTML and Javascript tutorials can be found on the internet.