OpenHome

An Open Source Home Automation System

Content

Design goals and considerations

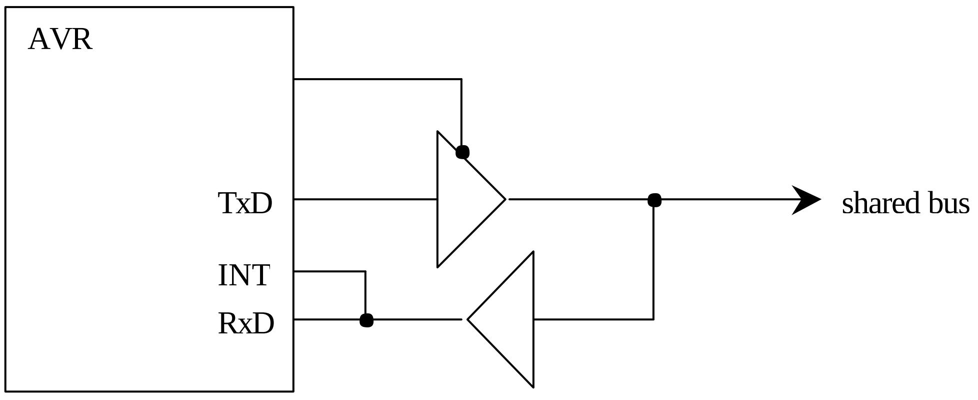
Me and a friend of mine started the OpenHome project as a project on remotely controlled lamps and similar things. During the design of hardware and software the idea to realise everything as extensible and configurable as possible to account on future changes was born. I searched the web for already existing projects but all I found were commercial systems like EIB, LON, X10 etc. After all, I decided to combine my own protocol with the best parts of already existing protocols. Therefore the following design considerations were made:

- Simple hardware
- Media independent
- Extensible (which can be used in many types of communications)
- Implementable on a at90s8515

The resulting protocol stack is mainly influenced by the LonTalk-Protocol which is very flexible and has proven its potential in millions of installations. The first idea was to implement the whole protocol on the avr-platform. Caused by hardware limitations and documentation availability this very soon turned out to be too difficult. Furthermore, it was somewhat oversized for a little self-made home automation system. Because of that, I designed my own protocol which is in fact a simplification of the LonTalk-Protocol.

Physical Layer

With the aim of designing the physical layer as simple as possible only the internal UART as well as an external bus driver are used.
See the following simplified schematic:



The software is independent of the medium the bus driver is attached to. Some media can provide some sort of collision detection but as for now this is neither planned nor implemented in the protocol stack.
The functionality is really simple. The avr-internal UART is attached to a single shared bus through a bus driver which is controlled by an arbitrary port pin. Moreover, the line feedback is connected to an external interrupt pin which can sense falling edges and which is used to discover transmissions of other devices on the bus immediately.
This layer is currently not specified in detail and is still under discussion.

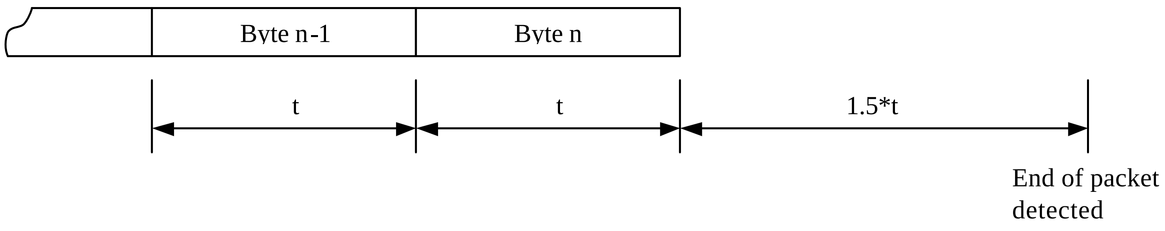
Link Layer

The link layer is primarily responsible for packet framing and error detection. It also contains the necessary logic for bus sharing and collision avoidance. Since no collision detection is performed this has to be done and corrected by upper layers, namely the transport layer.

Packet framing

Packet framing consists of detecting start and end of packets. The start of a packet is simply assumed if a byte is received. If a gap between subsequent bytes is detected this will be interpreted as the end of the packet. This implies that there must be no gap between the bytes in a packet. See the following sketch and definition for details.

*There must be no gap between subsequent bytes which are sent on the bus. If t is the time needed to transmit one byte the end of the current packet will be assumed if no byte is received for more than 1.5*t. The idle time to signal packet end is called BETA_1.*



After sending a packet over the bus every node has to wait a specific time (called BETA_2) before it's trying to transmit another packet. This time is used to enable the receiving node to process the packet and should be adjusted according to bus- and CPU-loads. If a node does not finish processing in time the next packet will be ignored by this node because there is not enough avr-internal memory left to buffer more than one packet.

Packet structure and error detection

The overall structure of all transmitted packets is as follows:

Header	LPDU	CRC

Header:

Length								Backlog							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

- Length: length of LPDU (without header and CRC)
- Backlog: expected traffic (=number of packets) following this packet

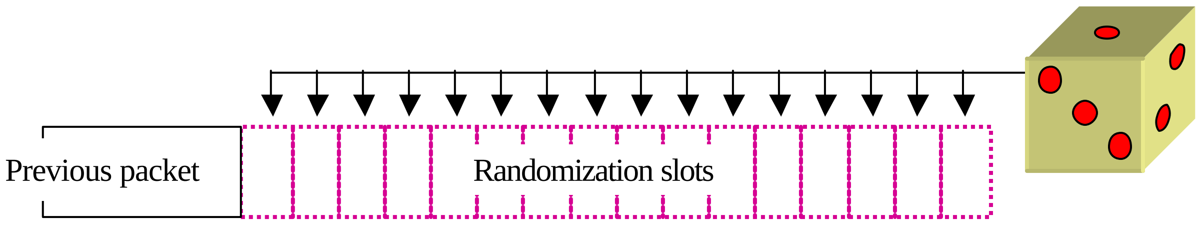
The specific meaning of the backlog field is discussed in the next paragraph.

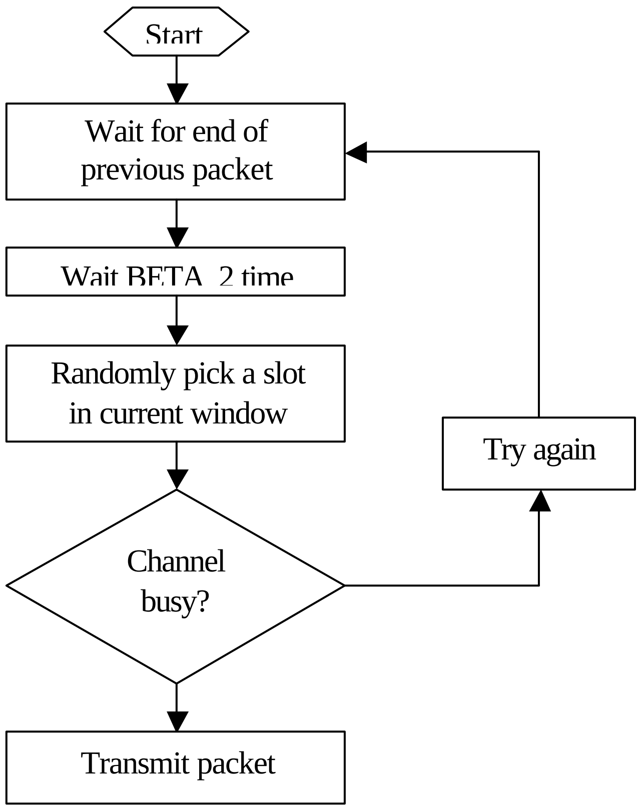
CRC: 16 bit checksum of Header and LPDU

Collision avoidance and slot timing

Collisions are a fundamental problem on shared busses. If two or more nodes on the bus start to send simultaneously a collision happens. The most common algorithm to prevent collisions is called CSMA (exactly: CSMA/CD which means *Carrier Sense Multiple Access with Collision Detection*) and used by Ethernet. If a node wants to start sending it listens the bus if another transmission is already in progress. But since line, drivers and other things have propagation delays this does not solve the problem. If a collision happens all transmitting nodes immediately stop sending and wait for a random period. Statistics show that the bandwidth is used efficiently if the traffic is low but CSMA performs very poor at high offered traffic. This is solved through CSMA/CA which adds some sort of *collision avoidance* to CSMA. The one used in this case is a p-Persistent CSMA.

Exactly after the last byte of each packet a so called slot timer starts running. A time slot is a fixed time which is – compared to the transmission time of one byte - relatively small. The width of these slots is called *ALPHA*. If a node wants to send a packet it has to wait a random number of time slots before it tries to transmit. During that time the node listens the bus and if another node starts to transmit the node backs off until next cycle. The following charts are used for illustration.




Start
Wait for end of previous packet
Wait BETA_2 time
Randomly pick a slot in current window
Channel busy?
Try again
Transmit packet

Initially 16 randomisation slots are used which is a compromise between collision avoidance and channel response time. Although this is sufficient if traffic is low it will not perform well as traffic increases. To keep collisions low, even at high offered traffic, the number of randomisation slots is increased as traffic increases. Therefore, some knowledge about the expected traffic on the bus is needed which is provided through the link layer backlog field.

The *backlog field* informs other nodes on the bus about the expected number of packets caused by the transmission of this packet. Its value is…

- zero for unacknowledged messages
- one when sending acknowledged or request messages
- the group size for multicast messages
- the number of nodes in destination subnet for broadcast messages

Every node on the bus records how many packets are pending using a backlog counter. If a packet is received the backlog counter will decrease by one and increase by the value of the backlog field in the packet. Finally the number of randomisation slots is calculated according to the following formula:

randomisation_slots = 16 * (backlog_counter + 1)

The problem is that all nodes must have a consistent view of slot timing which is affected through clock accuracy and node response time jitters. Therefor the slot time ALPHA must be chosen long enough to ensure consistent timing and short enough to preserve as much bandwidth as possible.

Network layer

The network layer handles addressing and routing of packets. To support a variety of applications three addressing schemes are used:

- Unicast: addressing a single node
- Multicast: addressing a group of nodes
- Broadcast: addressing all nodes in the subnet

Furthermore, there exist two classes of unicast addresses: logical and physical addresses. Altogether this leads to the following four address classes:

- Logical address
- Physical address
- Group address
- Broadcast

Normally logical addresses are used. Every device on the bus has a free configurable *logical address*. For configuring the logical address and for diagnostics a *physical address* is used that is unique to every device.
For sending a message to a group of nodes (multicast) group addresses are used. A device may be member of up to eight groups. A full qualified group node address consists of a group number and a member number. The group number identifies the whole group. The member number identifies the node within a group. A group destination address (=group address) consists only of the group number while a group source address (=group node address) consists of group and member number.
Broadcasts only contain the source address.

Packet structure

Header	Destination addr.	Source addr.	NPDU

Header:

RES		NPDU		SAF		DAF	
7	6	5	4	3	2	1	0

DAF: destination address format

Code	Description	Width (bit)
00	Logical address	16
01	Group address (group number)	8
10	Physical address	32
11	Broadcast	0

SAF: source address format

Code	Description	Width (bit)
00	Logical address	16
01	Group address (group + member number)	16
10	Physical address	32
11	-- not valid --	-

NPDU: enclosed network layer PDU format

Code	Description
00	Single message
01	Transaction
10	Session
11	-- not valid --

See transport layer for detailed discussion of the above PDU formats.

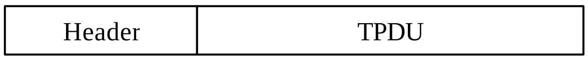
RES: reserved for future use, set to zero

Transport layer

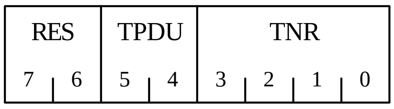
The transport layer provides a service for reliable packet delivery. In this implementation it also contains a request/response (session) service. Strictly speaking, the session service is located in the session layer but as it can easily be implemented in the transport layer it is located there. The exact behaviour of this layer depends on the NPDU passed by the network layer. All in all, that layer provides the following services:

- Unacknowledged (non- repeated) service
- Unacknowledged repeated service
- Acknowledged service
- Session service

Packet structure



Header:



TNR: transaction number

TPDU: enclosed tranport layer PDU format

Code	Description
00	Acknowledged message
01	Unacknowledged/Repeated message
10	Acknowledgement
11	Reminder

RES: reserved for future use, set to zero

Unacknowledged service

This service is very simple because the transport layer takes no action and all application data is passed directly to the network layer as NPDU=00 (single message). Since no header is added the **above packet structure does not apply**.
This one is the fastest protocol service and is useful when the application can tolerate occasional loss of packets.

Transaction sub-layer

The transaction service is used to preserve packet ordering as well as to detect/reject duplicates. It functions the following:
A transmitter completes one transaction before issuing next. With every new transaction the transaction number (TNR) is incremented by one. A receiver starts checking the incoming transaction number and the source address for duplicates. If the node has already received a packet with the same source address and the same transaction number then a duplicate has been detected and the packet will be dropped. Otherwise the node will store the new transaction number and source address in a cache. After all the application layer receives only one message. It may be possible that a node receives more than one transaction at the same time therefor it is recording at least the last four transactions.

Unacknowledged repeated service

Using this service the message is sent multiple (currently four) times as NPDU=01 (Transaction) and TPDU=01 (Unacknowledged/Repeated message). The receiver's transaction sub-layer ensures that the application receives the message at most one time. This service does not guarantee that the receiver gets the message but chances are good that at least one gets through. It is especially useful for large groups because the size of the group does not influence the number of messages sent on the bus.

Acknowledged service

This service is useful if the transmitter must know if a message got through. Though it may be possible that a transaction fails it is very unlikely and must be handled by upper layers. If used for large groups the bandwidth is used less efficiently because sending an acknowledged message to a group of N nodes causes N+1 packets on the network. You may consider using the unacknowledged repeated service in this case. Because there are slight differences in functionality between unicast and multicast acknowledged service they will be discussed separately.

Unicast acknowledged service

The message is sent as NPDU=01 (Transaction) and TPDU = 00 (Acknowledged message) using a logical destination address. The receiver of such a message sends back an acknowledgement (TPDU=10) with the same transaction number and his logical address. The transaction succeeded if the transmitter receives the acknowledgement. If no acknowledgement is received within a reasonable time the transmitter will retry the transaction. The transaction failed if no acknowledgement is received after three retries.

Multicast acknowledged service

As exactly as the unicast service the message is sent as NPDU=01 (Transaction) and TPDU=00 (Acknowledged message) but to a group destination address. All receivers of this message are required to send back an acknowledgement using their group node address as source address which means that the acknowledge also contains the group member number. The transaction succeeded if the transmitter received acknowledgements from all receivers. If some ACK’s are missing the transmitter starts retrying the transaction by firstly sending a reminder (TPDU=11) and then the original message again. The reminder contains a bitmap of group members who have already acknowledged. Only those receivers whose acknowledge did not get through will send another one.

Session service

This service provides a request/reply protocol. As noted above it is located in the transport layer because of its easy implementation. If you have a closer look at the acknowledged service you may notice that this already is a request/reply protocol. The only difference is that the acknowledgements do not contain any reply data. Therefore the same logic and data structures apply except the following modifications:

- "Acknowledged message" is called "Request"
- "Acknowledgement" is called "Reply"
- The acknowledgement contains data (the reply) provided by upper layers
- "Unacknowledged/Repeated message" makes no sense for this service and is not allowed

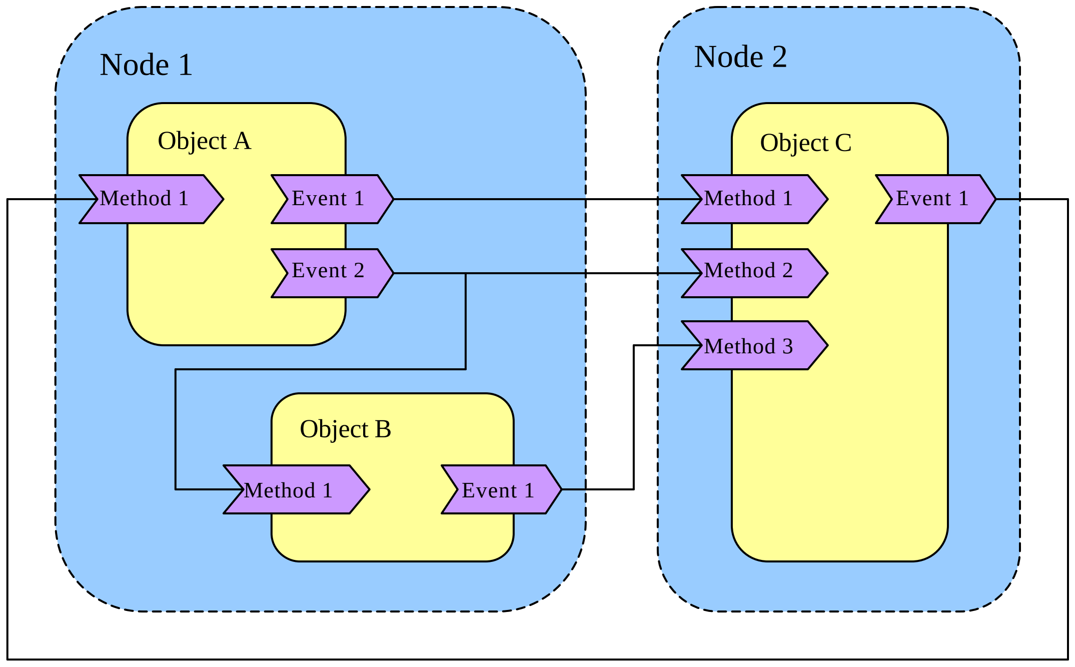
Note that the receiver’s application may receive duplicate requests if the transmitter has to retry.

Session layer

This layer is eventually empty since it's functionality is realised in the transport layer.

Presentation layer

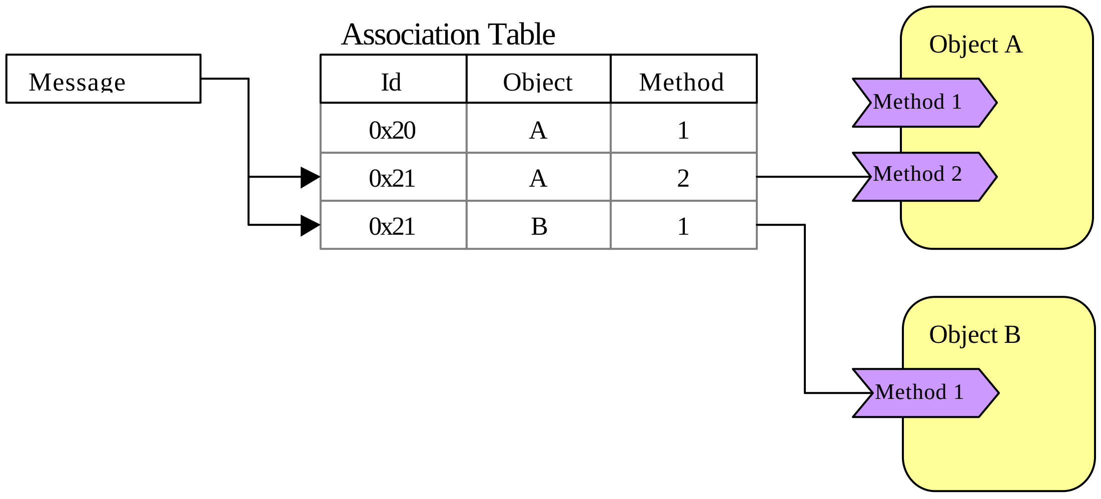
All application functionality is abstracted through an object oriented paradigm. Each node may contain one or more systems or application defined objects. These objects can contain methods, properties and events but the approach is method-centric. As in JAVA properties are abstracted through a pair of get-/set-methods. This leaves only methods and events to be implemented, properties are a matter of definitions. Events can be understood as method callers which can be connected to matching methods of other objects. These connections can be configured independent from the application layer as part of the network management. See the following sketch for illustration.



Though some standard types are defined, the actual interpretation of the method arguments and return values is left up to the application.

Associations

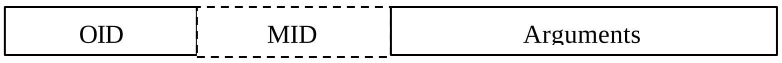
Some difficulties arise if a event (or the application) wants to call more than one method. The problem is that the object id may vary between different nodes making explicit addressing impossible. It should also be possible to call multiple methods within a node efficiently. This is solved through a 1 to N association table which each node contains.
Normally an incoming message references one or more entries in the association table. These entries refer to an object-method within the node. The following figure should illustrate this.



Id	Object	Method
0x20	A	1
0x21	A	2
0x21	B	1

The last entry of the association table must contain 0xff as id to indicate the end of the table.

Packet structure



OID: Object- / Association-Id

- 0..31: Object reference
- 32..255: Association reference

MID: Method- / Property- / Event-Id

Depending on the actual OID-value the MID field *may be omitted*. If OID references an object (0..31) the MID field specifies the destination method within this object. If OID references an association the MID field is left out.

System object

Object 0 is the system object that is implemented by all nodes and contains all necessary network management functions. See the following table for a list of all methods.

Id	Function	Arguments	Return value
0	Set logic address	uint16_t addr	void
1	Get logic address	Void	uint16_t
2	Set group address	uint8_t group, uint16_t addr	void
3	Get group address	uint8_t group	uint16_t
4	Set segment size	uint8_t size	void
5	Get segment size	Void	uint8_t
6	Set group size	uint8_t group, uint8_t size	void
7	Get group size	uint8_t group	uint8_t
8	Set association	uint8_t index, struct assoc_t assoc	void
9	Get association	uint8_t index	struct assoc_t
10	Get error statistics	Void	struct err_t

Event configuration

Events are not configured through the system object but within the objects itself. Accessing events is an implicit method call. Simply add 0x80 to the event-id and pass the event configuration as argument. The following structure is expected:

```
struct event {
        net_addr_t addr;              // peer network address
        tsp_service_e service;        // transport layer service type
        uint8_t assoc;                // association
};
```