Fraunhofer
ITWM

Project Hexabus

Bernd Lietzow
`lietzow@itwm.fhg.de`

February 10, 2012

1 Introduction

The following document describes the development of a software framework for the wireless home automation system *Hexabus*. The software was developed to run on microcontroller boards specifically manufactured for the project. The goal of the Hexabus project is to become an affordable, open, end extensible home automation platform. It is a part of the mySmartGrid project by Fraunhofer ITWM in Kaiserslautern. mySmartGrid aims to provide intelligent home device control [2]: Renewable energy sources, e.g. wind generators or solar panels, can not constantly produce electricity. Their power output varies with time of day and weather conditions and is often hard to predict. Since storing the produced energy is costly, ways for demand side management, controlling energy consumption by energy availability, are examined. In most households, there are appliances which can store energy in other forms, like heat pumps or freezers. These can run when a lot of electricity is available, and work off the stored energy when there is not enough electricity. Furthermore, household appliances which need little user interaction, like washing machines or dishwashers, can be automatically started when enough electricity is available[1]. Hexabus is being developed to support control of these devices within the mySmartGrid project is implemented.

Prior to this work, the hardware was available, but the software still lacked important functionality. The goal of this work was the development of a software framework that provides basic but essential functionality for the Hexabus platform.

1.1 Related Work

Several commercial building automation systems are already available. Popular examples are KNX and LON [12]. Those systems have been developed for the automation of large buildings. While they are extensible and support a variety of compatible devices, they are too expensive for most home use scenarios. They are intended to be installed, programmed, and maintained by trained professionals, making them even more expensive for non-professional home users. In addition most of them require separate wiring which makes them unsuitable for installation in rented apartments.

However, there are several systems aimed towards the home user market, like RWE SmartHome [5] and Plugwise [4]. While those systems are designed specifically for home users, they are still expensive. They are based on

[1] For example, a washing machine can be loaded by the user in the morning before they leave for work, and the home automation system starts the machine around noon, when the solar panels on the roof produce the most power.

proprietary or closed protocols, which limits their extensibility and the number of compatible devices.

For the mySmartGrid project, an extensible and adaptable system is needed, because it has to be compatible with the devices already used in the project, and it has to be adapted to the needs of the experiments.

1.2 What is Hexabus?

Hexabus aims to be an affordable, open, and extensible home automation system. The devices are built out of inexpensive hardware components and the protocol description as well as the software source code are publicly available [3]. The network protocol is based on IPv6, making software development and integration into existing home networks easy. The configuration will be possible over a simple point-and-click web interface so that the end user can set up and reconfigure their home automation system without the help of a professional. While there are going to be components available that are directly connected to the 230 V wiring of the house and therefore have to be installed by a trained electrician, most devices can be set up by the user themselves. This includes switches that are just plugged into the wall between the socket and the actual device, and appliances that come with Hexabus connectivity pre-installed.

1.3 Hexabus Devices

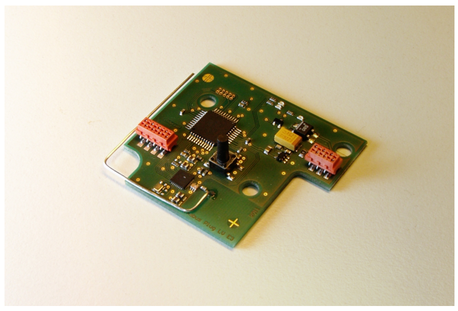
At the moment several Hexabus devices are available:

The *Hexabus USB Stick* (Figure 1a), which can be plugged into a PC or wireless router. It provides a 6LoWPAN [14] network interface that is used to communicate with other Hexabus devices. On a Linux PC, the network device can be used with the *cdc_ether* module, and there are also drivers available for Windows, MacOS, and the OpenWRT firmware package which is compatible with many wireless routers.

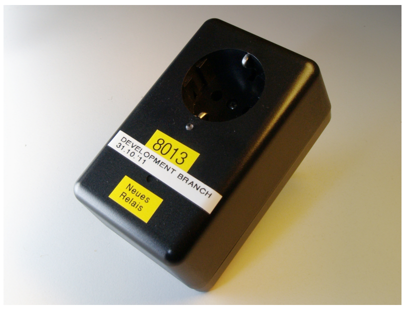
Hexabus Plugs (Figure 1c) are wall-wart style devices which can be plugged into a power socket. They have a socket themselves where other devices can be plugged in. The socket can be switched on or off by a relay. A microcontroller with a wireless interface can communicate with the Hexabus network and control the relay. There is also the *Hexabus Plug+* variant which additionally measures the electrical current drawn from the socket. The reading can be accessed from the software running on the microcontroller.

Most development is done on *Hexabus Prototyping Boards* (Figure 1b). They are identical to the electronics in the Hexabus Plugs except for the mains power

(a) Hexabus USB Stick



(b) Hexabus Prototyping Board



(c) Hexabus Plug

Figure 1: Hexabus Devices

part: They run off a safe 12 V power supply instead of the 230 V needed for the Hexabus Plugs, and they do not contain a relay. The prototyping boards are ideal for testing and development because software developed for them can be run on the Hexabus Plugs without any modification.

The prototyping boards and the Hexabus Plugs contain an ATMEL ATMega 1284 microcontroller. This is a microcontroller with an 8-bit RISC CPU that runs at 20 MHz and has 128 kiB of flash memory, 16 kiB of RAM, and 4 kiB EEPROM [7]. It runs the Contiki operating system. Wireless connectivity is handled by an ATRF212 chip which provides a wireless link on 868 MHz used for the 6LoWPAN IPv6 network connectivity. The microcontroller has several IO pins, some of which are used to control the relay and receive the power meter reading. Eight of the IO pins are connected to soldering points on the board allowing for addition of external electronics, for example a OneWire compatible temperature sensor, pushbuttons, or optocouplers controlling other electric or electronic devices. With this connectivity, the Hexabus prototyping boards are used to experimentally connect different appliances to a Hexabus network.

Contiki is a lightweight operating system for memory constrained systems [9]. It runs on a variety of 8-bit platforms, including the Texas Instruments MSP430 microcontroller series and the ATMEL ATMega microcontrollers used in the

Hexabus hardware.

Contiki provides the 6LoWPAN IPv6 stack used for the Hexabus network. It also provides cooperative multitasking by means of protothreads. As opposed to regular threads, protothreads do not have their own stack for local variables so the only way to preserve data over blocking calls (the only point where context switches can occur in cooperative multitasking) is through global variables. In exchange for this drawback, protothreads need very little memory because for each thread, only the instruction pointer has to be stored permanently in main memory [10].

1.4 Goals

The goal of this project was to develop the components needed to implement a home automation system, namely:

- The design of the network protocol
- The implementation of this protocol on embedded hardware
- An application for a PC that is able to interact with this protocol
- The program that controls the functions of each device

The idea of the Hexabus protocol is to offer most, if not all, functionality without a central server. All devices which generate input to the network, e.g. pushbuttons or temperature sensors, must broadcast their readings either when a certain events occur, for instance when a button is pressed, or periodically. The devices receiving these broadcasts must decide by themselves how to react on these broadcasts. For example, a light has to turn on when it receives the ”button pressed” broadcast from the pushbutton associated with it, and the room heater must switch off once the room temperature reading broadcast from a temperature sensor rises above a certain value.

2 The Hexabus Network Protocol

The Hexabus network protocol uses UDP as the transport layer protocol and IPv6 as the network layer protocol. Since memory and bandwidth of the embedded systems the Hexabus is intended to run on are limited, the 6LoWPAN

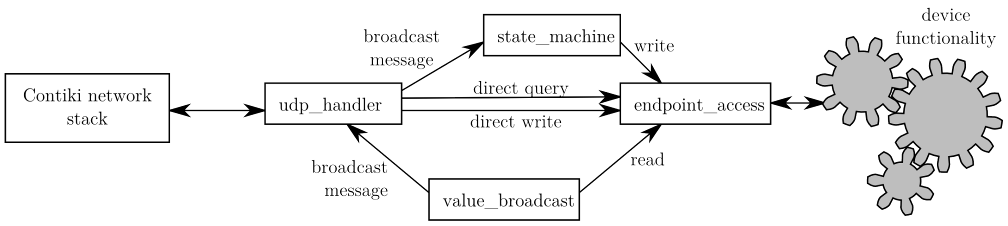


Figure 2: Interaction of Hexabus software components.

protocol is used. It is a set of standards which enables the use of *IPv6 over low-power wireless area networks* [14]. 6LoWPAN maintains compatibility to IPv6 while adapting the format to make network packets suitable for low power wireless network, for example by fragmentation and header compression [13]. 6LoWPAN typically uses the 802.15.4 wireless network standard [6], which is also used in ZigBee.

Mini-Glossary

Endpoint	A feature of a device. An endpoint is identified by it's endpoint ID and always has a value. It is typically accessible from the network.
Device descriptor	The value of the special endpoint with the EID 0. It contains information about the device's other endpoints.
Query	Requests a device to transmit the current value of a particular endpoint.
Write	Requests a device to set a particular endpoint to a certain value.
Info	A piece of information containing the current value of an endpoint.
Broadcast	An *info* being sent to the link-local multicast address, therefore being readable to all devices on the same Hexabus network[2].
Endpoint query	Requests a device to transmit an endpoint info.
Endpoint info	A piece of inforamtion about a particular endpoint: The datatype and a short text describing the functionality.

2.1 Hexabus Devices And Endpoints

A Hexabus device offers several functions called *endpoints*. Each endpoint has an identification number (the *endpoint ID* or *EID*) and a value. All capabilities of a particular device are represented by endpoints. The endpoint with the EID 0 is

[2]The notion of a *network* here refers to the PAN ID of the 6LoWPAN. The link-local multicasts therefore go to all devices on this PAN because the PAN is treated as a single IPv6 link.

a special endpoint containing the *device descriptor* that indicates which other endpoints exist on a device. Other endpoints may offer access to the device's functionality. For example, the endpoint 1 on a Hexabus Plug is the state of the relay. It can be switched on or off using *write* packets, or read with a *query* packet. Endpoint 2 is the power meter reading which can be queried, but not written.

2.1.1 The Device Descriptor

The endpoint 0 of each device contains the device descriptor. It is a 32 bit vector containing information about the device's endpoints. Each bit stands for an endpoint ID on the device. If the bit is set to 1, the endpoint exists, if it is set to 0, the endpoint does not exist. The least significant bit (bit number 0) of the device descriptor corresponds to endpoint ID 1, the next bit (bit number 1) corresponds to endpoint ID 2, and so on. This way, the first 32 endpoints of the device are described. Endpoint 32 contains another device descriptor if endpoints with IDs greater than 32 are present on the device (otherwise it is nonexistent). This device descriptor describes the endpoints 33 to 64. This pattern continues up to 255 endpoints.

2.1.2 Querying and Writing Values

Each value can be directly set by sending a *write* packet to the device. The device then reacts to this either by executing the appropriate action to set the endpoint to the new value, or by sending an error packet if the endpoint is nonexistent or cannot be written.

Each endpoint has a data type associated with it, such as boolean, 8 bit integer, 32 bit integer or 32 bit floating point. The network protocol offers functions to find out which data type belongs to a particular endpoint. When writing to an endpoint, the correct data type has to be used.

On the Hexabus Plug, a process named *udp_handler* is used for accessing the network. If a *read* or *write* packet is received, the request is passed to a module named *endpoint_access* which provides access to the device's features themselves, translating generic *read* and *write* calls into the specific actions of the device (see Figure 2).

2.1.3 Broadcasting Values

The Hexabus concept relies on values being broadcast over the network. The process *value_broadcast* is responsible for periodically sending out these broadcasts. It has a timer that runs for 60 seconds. When this timer expires it is reset and another timer is started which runs for a (pseudo)random time between 0 and 60 seconds, before a broadcast packet is sent out. In every 60 second interval, a value is broadcast once, but the actual point of time when it is broadcast within the interval varies. This random timer is used in order to reduce network congestion which might occur if all devices try to broadcast at the same time repeatedly. If the interval was the same for all devices, the packets could collide every time the devices broadcast, but if a randomized timeout is used, these collisions rarely occur.

Of course there are also cases where one does not want to broadcast a value periodically, but immediately when a special event occurs. This is handled by the process where the event is handled. For example, if a broadcast has to be sent every time a button is pushed, the broadcasting function in value_broadcast is called from the Contiki process reading the pushbutton values.

2.1.4 The endpoint_access Module

The *endpoint_access* module converts the Hexabus protocol actions (reading and writing endpoints) to the concrete function calls for the processes and hardware components associated to the endpoints. endpoint_access provides the functions endpoint_read and endpoint_write, which call the more complex functions needed to actually execute the actions associated with the particular read or write calls. For example, when endpoint_access is instructed to *write* to endpoint 1, it calls the appropriate functions to switch the socket's relay on or off. It reports errors (e.g. when trying to write to a read-only endpoint, or when there is a datatype mismatch). There are also functions to find out the name and datatype of a particular endpoint.

2.1.5 Extending a Hexabus Device

Extending a Hexabus device can be done by adding a Contiki process for the desired functionality. This can be a simple software process receiving and broadcasting values to and from the network, but most commonly, the board's IO-port is used to attach different hardware components that will either be controlled via the Hexabus network, or provide data input to it. Then, the functionality has to be broken down into endpoints and values. Those endpoints have to be defined in the *endpoint_access* module, and the actions to read and write those endpoints have to be implemented. Additionally, if values from the endpoints should be broadcast, their EIDs have to be entered into the *value_broadcast* module configuration.

2.2 Modeling the Protocol

A model of this protocol was built using Promela [11]. Only the broadcast sending and receiving parts were modeled because the direct query/info functionality is only intended as a debugging mechanism which should only be used for manual setup and testing, not in continuous operation. The broadcasting and broadcast receiving mechanism of the protocol is stateless: A node can send out a broadcast at any time, and it can receive a broadcast at any time.

When only this portion of the protocol is taken into account, simulation of a single node is simple: It just has a single state, and can send *info* messages to a channel, or read *info* messages from a channel. But the nodes themselves also contain programs, and their internal state depends on the values they receive as broadcasts. Therefore the model was extended: There is a sending node, which sends out messages which have one of four values, they are called *info0*, *info1*, *info2*, and *info3*. There are N receiving nodes, which change their internal state when they receive a broadcast. The internal state of the receivers is modeled as an array localX$[N]$, where every receiver r_n can change the element localX$[n]$. The network is modeled by a process named *broadcastHub* which reads messages from one channel and copies them to several channels, one for each receiver. This allows for the inclusion of a lossy-demon to simulate a network link experiencing packet loss in each of the channels. The lossy-demon can either copy messages from its input to output channel, or "lose" them, meaning a message is read from the input channel without copying it to the output channel.

Consistency criteria were defined: We call the network *consistent* when *for every broadcasting endpoint e, every node whose internal state depends on the value of e, has received a broadcast of the* current *value of e at least once.*

In continuous use, a network can become nonconsistent when info broadcasts are lost. For normal operation, it is not necessary that a network is consistent all the time. A node which has not received the current value yet will just not react to it. As long as the network becomes consistent again before the values change[3], all the nodes will get the chance to react to the current value, albeit maybe not all at the same time.

[3]Or, since the state machines which compute the reaction on the broadcasts (see below) usually use greater-than or less-then as comparison operators, the network just has to become consistent again before the values change *too much*. So in practice the broadcasting interval should be chosen in a way so that several broadcasts occur before the value crosses a threshold which triggers a reaction of a node.

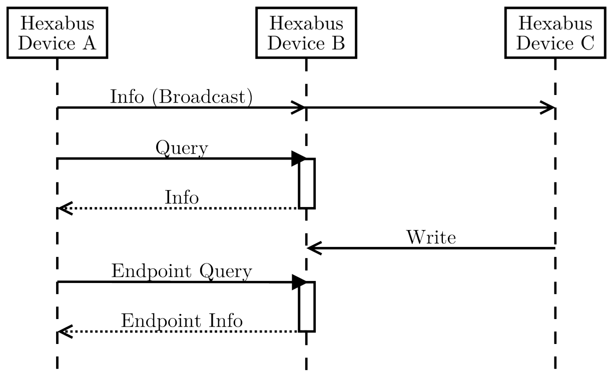


Figure 3: Example message exchange over the Hexabus network

To check whether a network will reach consistency one could use the following claims: (For all pairs of nodes n_a, n_b)

- $\Diamond\Box$localX$[a]$ $=$ localX$[b]$: At some time, the network becomes and stays consistent.
- $\Box\Diamond$localX$[a]$ $=$ localX$[b]$: The network will always be able to reach a consistent state.

If one assumes that packet loss over the wireless link is purely random, a scenario where the network can never become consistent can be constructed. But under the assumption that of every n packets, at least some fraction m of the packets reach their destination, we can see that the network can always become consistent after at most $n - m + 1$ repeated broadcasts of the same value. To simulate this situation, the node *periodicLossy* was implemented. It either consumes or forwards messages, but it guarantees that a certain fraction of messages are forwarded.

The source code of the model can be found in Appendix C.

The model does not take into account the time constraints such a system has: There are scenarios with direct user interaction, e.g. the user pushing a button to turn on a lamp, where the reaction has to be carried out within a short amount of time. It is however not desirable to just send off multiple broadcasts in quick succession because this might congest the network. This problem still has to be investigated.

2.3 The Hexabus Network Packet Format

Hexabus network packets are transmitted using 6LoWPAN. This is fully compatible to IPv6, so the packets can be routed to different networks. Using a router[4], the Hexabus devices can communicate with computers or other devices

[4]The test networks use an OpenWRT equipped wireless router with a Hexabus USB stick plugged into it

connected to the wireless or wired local area network. It is also possible to route the packets over the Internet to remote machines.

Hexabus devices listen on port 61616 for incoming UDP packets. This can be broadcasts or packets directed to the device itself. Port 61616 was chosen because for port numbers 61616 through 61631, the 6LoWPAN header compression can compress the port number to four bits [14].

Byte	0-3	4	5	6.. (variable length)	$n-1, n$
Field	`"HX0B"`	Packet type	Flags	Payload	CRC

Hexabus packets are wrapped in UDP packets. The UDP data of a Hexabus packet starts with the bytes `0x48 0x58 0x30 0x42` (`"HX0B"`) to identify it as a Hexabus packet. Then follow one byte denoting the packet type and one byte reserved for flags that may be present in future versions of the protocol[5], but are not yet implemented. The actual payload starts at byte 6. Its length is variable and depends on the packet type and data type being transmitted. The last two bytes contain the checksum[6].

2.3.1 Error Packets

An *error* packet indicates something went wrong.

Byte	0-3	4	5	6	7, 8
Field	Header	Packet type	Flags	Error Code	CRC
Content	`"HX0B"`	`0x00`		*see below*	

The packet type 0 marks the packet as an error packet. The error code field gives more specific information:

Error code	**Error**	**Description**
`0x01`	Unknown EID	A query or write packet was received, but the EID matches none of the endpoints on the receiving device.
`0x02`	Write Read-Only	A set value-packet was received, but the EID corresponds to a read-only endpoint.
`0x03`	CRC Failed	The CRC check of a packet failed.
`0x04`	Data type mismatch	A write packet was received, but the data type does not match that of the endpoint.

[5]e.g. a request for acknowledgement to make sure a packet has reached its destination

[6]The checksum is generated via the CRC16-Kermit method: Generator Polynomial: 0x1021, Initial value: 0x0000, Reflect Input: true, Reflect Output: true, XOR Output: 0x0000 [1]

2.3.2 Info Packets

An *info* packet contains information about an endpoint's value. This can be the reply to a query of that endpoint or an autonomous broadcast of a value. If it is a reply to a query it is sent to the source address of the query packet. If it is a broadcast it is sent to the link-local multicast address.

Byte	0-3	4	5	6	7	8..	$n-1, n$
Field	Header	Type	Flags	EID	Data type	Value	CRC
Content	`"HX0B"`	`0x01`					

An info packet has the packet type 1. The EID field contains the endpoint ID of the endpoint on the sending device whose value is transmitted. The length of the value field varies depending on the data type. The following data types are implemented:

Number	**Data type**	**Length (bytes)**
`0x00`	Reserved (used to denote "no data at all" internally)	
`0x01`	Boolean (represented by values `0x00` (false) and `0x01` (true))	1
`0x02`	8 bit unsigned integer	1
`0x03`	32 bit unsigned integer	4
`0x04`	Date and time data structure	8
`0x05`	32 bit floating point value	4
`0x06`	Character string (fixed length)	128
`0x07`	Timestamp: Seconds since device was booted (32 bit unsigned integer)	4

2.3.3 Write Packets

A write packet is used to directly set an endpoint's value on a specific device to a specific value.

Byte	0-3	4	5	6	7	8..	$n-1, n$
Field	Header	Type	Flags	EID	Data type	Value	CRC
Content	`"HX0B"`	`0x04`					

The structure of a *write* packet is basically the same as an *info* packet. It is marked by packet type 4. The difference to the info packet is that in an info packet, the endpoint ID refers to an endpoint on the sending device, but in a write packet, the endpoint ID refers to an endpoint on the receiving device.

If a device receives a write packet for an endpoint which can not be written to (e.g. power metering on a Hexabus Plug) it will reply with an error packet containing the error code *write read-only*. If a device receives a write packet for an endpoint ID not present on this device, it will reply with an error packet containing the error code *unknown EID*. Upon receiving a write packet whose data type does not match that of the endpoint to be written to, a device replies with an error packet with the error code *data type mismatch*.

If the endpoint is present on the device, is writable, and the endpoint ID matches, the data is *written* to the endpoint as described above.

2.3.4 Query Packets

The *query* packets are used to request an immediate and direct (not broadcast) transmission of an *info* packet containing the value of a specific endpoint.

Byte	0-3	4	5	6	7, 8
Field	Header	Packet type	Flags	endpoint ID	CRC
Content	`"HX0B"`	`0x03`			

The packet type marking a packet as a *query* packet is packet type number 3. A device receiving a *query* packet either responds with an info packet containing the endpoint ID of the queried endpoint and its value, or—if the endpoint is not present on the device—with an error packet carrying the error code *unknown EID*.

2.3.5 Endpoint Query and Info Packets

To find out an endpoint's properties, the *endpoint query* packet can be used. It uses the same layout as the *query* packet, but has packet type `0x0A` (10). When a device receives an endpoint query for an existing endpoint, it responds with an *endpoint info* packet. The endpoint info packet contains the datatype of the endpoint and a character string containing a textual description of the endpoint. The endpoint info packet uses the layout from the character string info packet, but with datatype `0x09`. The datatype and endpoint ID of an endpoint info packet are set to the datatype and endpoint ID of the endpoint it describes[7]. If a device receives an endpoint query for a nonexisting endpoint, it responds with the "unknown EID" error message. When the endpoint info for endpoint ID 0 (device descriptor endpoint) is queried, the response contains the name of the device itself.

[7] In contrast to the *info* packets, the datatype-field is not needed to find out the length of the payload, since endpoint info packets always contain a 128 byte character string.

2.4 Implementation of the Network Protocol

The network protocol described here was implemented on the Hexabus prototyping boards under the Contiki operating system. It also runs on the Hexabus Plugs. The Linux command line application *hexaswitch* was written. It can be used to send all packet types and values, and to receive info packets as direct messages and broadcasts and print them out on the console. The network part and protocol implementation of this application were later split off into a library named *libhexabus* which can be used to implement other Hexabus compatible applications.

3 Rule-based Device Control Using State Machines

The main part of the Hexabus software concept is the device control logic. Each device has to react to the values broadcast over the network and execute the appropriate actions.

The programs needed to control a device can depend on a multitude of value broadcasts and also have temporal dependencies. For example, a rule formulated in natural language could read: "Between 5pm and 7am, when the motion detector is triggered, turn on the lights for 5 minutes."

To represent these rules, the concept of state machines is used. Each device has a state machine which can control its endpoints.

These state machines consist of
- a set of states S with an initial state $S_0 \in S$
- an input alphabet Σ consisting of triples (d, e_r, v_r)
- an output alphabet Λ consisting of pairs (e_l, v_l)
- a transition function $T : S \times \Sigma \to \Lambda \times S \times S$

Σ contains triples (d, e_r, v_r) of a device IP address d, an endpoint ID e_r, and a value v_r. These are called remote endpoints, because the endpoints from which the values originate typically reside on remote devices. Λ contains pairs (e_l, v_l) of endpoint IDs e_l and values v_l. Since these endpoints are on the same device as the state machine, we call them local endpoints.

The transition function T differs from normal state machines: It is defined as $T : S \times \Sigma \to \Lambda \times S \times S$. The transition function maps to triples (a, s_g, s_b) and is

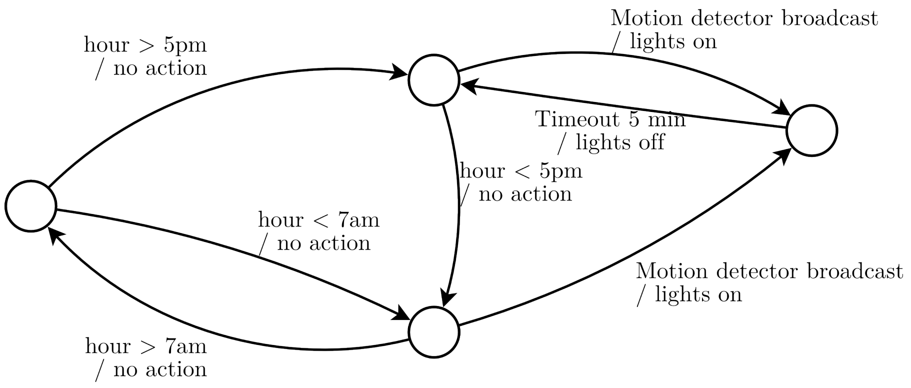

hour > 5pm / no action
Motion detector broadcast / lights on
Timeout 5 min / lights off
hour < 5pm / no action
hour < 7am / no action
Motion detector broadcast / lights on
hour > 7am / no action

Figure 4: State machine representing the control program: “Between 5pm and 7am, when the motion detector is triggered, turn on the lights for 5 minutes.”

read as follows: If the machine is in state $s \in S$ and receives a broadcast $b \in \Sigma$, and $T(s, b) = (a, s_g, s_b)$, try running the action $a = (e_l, v_l) \in \Lambda$, i.e. set the endpoint e_l to the value v_l. If the action succeeds, go to the *good state* s_g, if it fails, go to the *bad state* s_b.

The above example could be expressed with the state machine shown in Figure 4.

3.1 Implementation of the State Machine

The state machines implemented in the Hexabus devices have two kinds of transitions: Value-dependent transitions and date/time dependent transitions. To store those transitions, three tables are used: The conditions table, the table of value-dependent transitions, and the table of date/time-dependent transitions.

3.1.1 Transitions

Each transition has a condition index which gives the position of its condition entry in the conditions table. This is intended to save some memory since transitions sharing the same conditions can point to the same entry in the conditions table. Transitions also have a number of state indices: fromState, the state the machine has to be in in order for the transition to be applicable, goodState and badState. There is no state table since states are entirely defined by the entries in the transition table. Furthermore it stores an endpoint ID and a value, which constitute the action described above.

The C-Code defining an entry in the transition table can be found in Appendix A.

To execute the action, the endpoint_access module is used (see Figure 2). The endpoint_write function returns 0 if the action succeeded, and an error code if it fails. According to this return value the state machine changes into either goodState or badState.

3.1.2 Conditions

Conditions fall into two categories for the two kinds of transitions. They are stored in the condition table, as a data structure which holds a source IP address, a source endpoint ID, a comparison operator, and a constant value to compare with.

For all transitions going out of the current state, the value-dependent transitions are checked each time a value broadcast is received. The first transition whose condition holds is executed. For the value-dependent transition, the comparison operator can be $=$, $\leq$, $\geq$, $<$, $>$, or $\neq$ and is stored encoded as an 8 bit value in the condition data structure. To execute a transition, its associated condition has to hold: First the source IP is checked. The broadcast has to originate from the host specified in the condition. Two special source IPs are defined: The unspecified IP address consisting entirely of zeros, which is interpreted as "any host", and the localhost IP address ::1. If the IP matches, the value of the endpoint is compared to the condition's constant according to the comparison operator, and the transition executed if the condition holds.

The implementation of the data structure used for an entry in the condition table is given in Appendix B.

The date/time-dependent transitions are checked periodically, every five seconds. The conditions for them are handled a bit differently. Source IP and endpoint ID do not matter since the conditions are checked against the internal clock of the device, and the 8 bit value for the comparison operator is used in a different way. The bits $0..6$ are used to denote the different fields of the date/time data structure. When a bit is set to 1, the corresponding field is checked: Bit number 0 stands for the hour field, bit number 1 for the minutes, bit number 2 for the seconds, and so on[8]. Bit number 7 is the comparison operator: The selected field of the date/time stored in the condition is compared to the current date and time. If bit number 7 is set the condition evaluates to true if the corresponding field in the current date/time is greater or equal to the condition's constant. When bit number 7 is not set it evaluates to true if it is less than the condition's constant.

[8]Bit no. 3: Day of month; bit no. 4: Month; bit no. 5: Year; bit no. 6: Day of week.

It is also possible to execute a transition if the state machine has remained in a certain state for a set amount of time. Those transitions have *timestamp conditions*: Their constant holds a 32 bit value indicating the number of seconds the machine can remain in the fromState until the transition is executed. To facilitate this the device counts the seconds since it was booted. Each time the state machine executes a transition, this timestamp is copied into a local variable called inStateSince. When checking a timestamp condition, this variable is subtracted from the current timestamp and the transition executed if the difference is higher than the condition's constant.

3.1.3 Storing the Tables

The tables which contain the transitions and conditions are stored in the device's EEPROM. In order to reduce memory usage, a transition or condition data structure is only copied into RAM when it is currently being checked. With the current configuration, 512 bytes in the EEPROM are reserved for each table. This is enough for 18 conditions and 2×36 transitions.

When work on the state machines was first started, the transition and condition tables had to be hard-coded so that they were written into the device's EEPROM at boot-up. Later, a student worker developed a web interface which runs on the devices' embedded HTTP server allowing the user to write and change the state machine's condition and transition tables while the device is running.

4 Course of the Project

To evaluate the fitness of the 6LoWPAN protocol and the ATMega microcontrollers, at the beginning of the project experiments were done on ATMEL AVR Raven microcontroller kits. These kits contain a USB stick with a 2.4GHz transceiver as well as an experimenting board with two microcontrollers and an LCD on it. One microcontroller is an ATMega 3290PV, which drives the LCD. The other is an ATMega 1284PV which is very similar to the microcontrollers used in the current Hexabus devices. The boards are capable of running Contiki and can communicate via 6LoWPAN. The microcontrollers and radio transceiver components used for the actual Hexabus electronics were chosen because of their similarity with the components of the AVR Raven products.

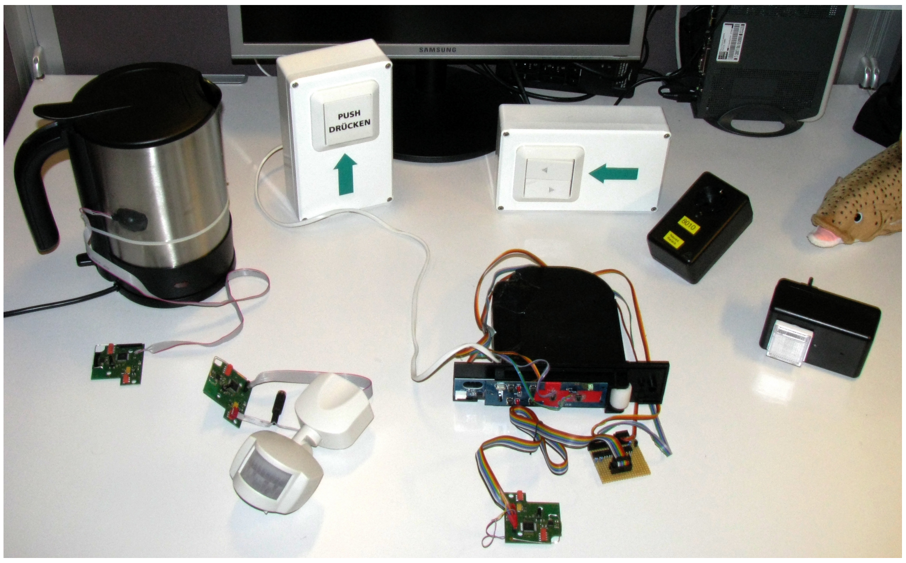


Figure 5: Several experimental Hexabus-enabled devices (left to right): Temperature controlled water boiler[9], infrared motion detector, pushbuttons, electric window blind motor, a Hexabus Plug with a temperature sensor, and a Hexabus Plug with a light used as visual output

The Hardware for the Hexabus Plugs and USB sticks was developed by embedded brains GmbH in Puchheim. Basic functionality (adaption of Contiki, IPv6 connectivity, wirelessly switching a socket on and off and reading the power measurement) was implemented at Fraunhofer ESK in Munich.

Once the Hexabus prototyping boards and USB sticks were delivered, development and experiments were switched over to them. Now the 868 MHz frequency band was used. Since now also Hexabus Plugs were available, the power meters contained in them could be used for testing value broadcasting and reacting to broadcasts.

After the idea of the protocol was specified, a model was built using Promela and simulated using SPIN [11]. This model can be used in the future to analyze the behaviour of the network if not all value broadcasts reach their destination: Several broadcasting and receiving nodes can be connected with a lossy-demon that drops packets randomly or with a pre-set pattern. This can be used to find out how packet loss impairs the operation of a Hexabus network.

As soon as the basic functionality of the network protocol was implemented, the software was being used by other students and student workers for their own projects:

Several Hexabus boards were equipped with temperature sensors and distributed in the hallway and offices. A data logging application was implemented using libhexabus, and a temperature controlled water boiler was built using Hexabus components. These serve as experiments for a diploma thesis conducted at FH Kaiserslautern (development of a distributed heating control system). Other experiments include a Hexabus controlled window blind motor, an infrared motion detector, a door sensor, a sunlight sensor, and several boxes with pushbuttons serving as input devices for the system. Some of these prototypes can be seen in Figure 5.

These experiments serve as a demonstration of the Hexabus system's capabilities as well as a means of finding out what is needed to make a functional home automation system. Having more people work with the software also provided valuable input in the form of feature requests and bug reports, as newly implemented features were not only tested for a simple test application or specified test cases, but put to a thorough test being used in several applications simultaneously.

To coordinate the software development between the different people working on and with the software the *git* distributed version control system [8] was used. A central repository was hosted on github.com which also provides a Wiki for documentation and an issue tracking system for bug reports and feature requests.

5 Conclusion

This document described the development of various software components for the wireless home automation system *Hexabus*. The concept of the system is that all devices offering input to the network have to broadcast their readings, and all actor devices decide using control rules implemented as a state machine how to react to the broadcasted readings. The network protocol used for broadcasting the readings as well as for directly accessing the devices is described. The implementation of the state machines executing the control rules is also described.

Several test devices for the system have been implemented, and the system is currently being tested by several people in different environments.

[9]The image only shows the boiler with the temperature sensor. The heating coil of the boiler is controlled by a solid state relay not pictured here.

5.1 Future Work

The current version of the protocol does not contain a mechanism to deal with lost packets. This is no problem when a packet of a periodic value broadcast is lost, since the receiver will just wait for the next one. But when pushbuttons or other devices for user input are used, a lost packet can cause the system to not react to the user's input in the desired way. Imagine a system where one pushbutton controls two lamps. A single push of the button turns both of the lamps on or off. Since each lamp only knows its own state, it has to toggle when the "button pushed" broadcast is received. Now when the user pushes the button, it could happen that one of the lamps receives the broadcast and the other does not. From that point on one of the lamps is on while the other is off. A way to work around this problems should be investigated, for example using acknowledgement packets or sequence numbers in certain types of broadcasts.

From the current state of development, several improvements to the state machine implementation can be made. A considerable amount of space in the EEPROM can be saved by using a separate table for the source IP addresses. An IPv6 address is 16 bytes long, but it is often the case that many conditions share the same source IP address, so several 16 byte "source address" entries could be compressed to 1 byte indices pointing to a single 16 byte address table entry. Another approach to fit more entries into the EEPROM could be making the tables' start addresses dynamic, so that a larger condition table can be used if the transition tables have less entries and vice-versa. The tables could also be made smaller by introducing transitions that have multiple conditions which are connected by boolean operators.

The number of transitions can be reduced further by introducing broadcast groups: A common usage scenario is several light switches controlling a lamp. With the current state machine implementation, a condition is needed for each light switch. This could be simplified by introducing *group IDs*. All the light switches would share the same group ID. This group ID would be included in their broadcasts, and instead of reacting to each broadcasting light switch IP separately, the lamp could just have one transition reacting to that group ID.

It is still not clear whether the state machine concept is the best solution for rule-based control in a home automation system. Different ways of programming the devices, e.g. using a simple interpreted programming language like FORTH, should be examined.

Furthermore, the distributed state machines are not an easy concept for an end user to understand. If the system should be programmable by users themselves without prior training, there has to be another layer of abstraction. A user interface which is easier to understand should be designed and implemented. The user could interact with a simple point-and-click interface providing a view of their network with devices or parts of the network by graphical objects. They could then draw lines between the objects to define control rules. This control program could then be compiled, sliced into the programs for the individual devices and uploaded automatically.

References

[1] Catalogue of parametrised CRC algorithms. Website: `http://regregex.bbcmicro.net/crc-catalogue.htm`.

[2] MySmartGrid. Website: `http://www.mysmartgrid.de/`.

[3] mysmartgrid/hexabus on GitHub. Website: `http://github.com/mysmartgrid/hexabus`.

[4] Plugwise. Website: `http://www.plugwise.com`.

[5] RWE SmartHome. Website: `www.rwe-smarthome.de`.

[6] IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, 5 2011.

[7] Atmel Corporation, San Jose, CA 95131. *ATMega 1284P Datasheet*.

[8] Scott Chacon. *Pro Git (Expert's Voice in Software Development)*. Apress, August 2009.

[9] Adam Dunkels, Björn Grönvall, and Timo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conferece on Local Computer Networks, 2004*, pages 455–462, November 2004.

[10] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

[11] Gerard J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.

[12] Herrmann Merz, Thomas Hansemann, and Christof Hübner. *Gebäudeautomation*. Carl Hanser Verlag München, 2nd edition, 2010.

[13] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, pages 78–82, New York, NY, USA, 2007. ACM.

[14] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. John Wiley & Sons Ltd, 2009.

A C data structure for a state machine transition

```c
struct transition {
  uint8_t fromState;      // current state
  uint8_t cond;           // index of condition that must be matched
  uint8_t eid;            // id of endpoint which should do something
  uint8_t goodState;      // new state if everything went fine
  uint8_t badState;       // new state if something went wrong
  struct hxb_value value; // Data for the endpoint
} __attribute__ ((packed));
```

B C data structure for a state machine condition

hxb_value is a wrapper structure for the different data types supported as constants.

```c
struct condition {
  uint8_t sourceIP[16];      // source IP address
  uint8_t sourceEID;         // EID we expect data from
  uint8_t op;                // comparison operator
  struct hxb_value value;    // the constant to compare with
} __attribute__ ((packed));
```

C Promela Model for a simple Hexabus network

```
// Number of messages that can be in a channel simultaneously
#define chanSize 1

#define N 3 // number of receiving nodes

```

```
chan broadcastChannel[N] = [chanSize] of {mtype};

// Messages
mtype = { info0, info1, info2, info3 }

byte sensorX = 0;

// this node sends out broadcasts
proctype sendingNode(chan output) {
  StInitial:
    goto StNetworkJoined;
  StNetworkJoined:
    if
      :: (sensorX == 0) -> output!info0; sensorX++; goto StNetworkJoined;
      :: (sensorX == 1) -> output!info1; sensorX++; goto StNetworkJoined;
      :: (sensorX == 2) -> output!info2; sensorX++; goto StNetworkJoined;
      :: (sensorX == 3) -> output!info3; sensorX = 3; goto StNetworkJoined; //
           keep broadcasting "3"
    fi;
}

proctype broadcastHub(chan input) {
  int i;
  StInitial:
    if
      :: input?info0 -> goto StSendZero;
      :: input?info1 -> goto StSendOne;
      :: input?info2 -> goto StSendTwo;
      :: input?info3 -> goto StSendThree;
    fi;
  StSendZero:
    i = 0;
    do
      :: (i < N) -> broadcastChannel[i]!info0; i++;
      :: else goto StInitial;
    od;
  StSendOne:
    i = 0;
    do
      :: (i < N) -> broadcastChannel[i]!info1; i++;
      :: else goto StInitial;
    od;
  StSendTwo:
    i = 0;
    do
      :: (i < N) -> broadcastChannel[i]!info2; i++;
      :: else goto StInitial;
    od;
  StSendThree:
    i = 0;
    do
      :: (i < N) -> broadcastChannel[i]!info3; i++;
      :: else goto StInitial;
    od;
}

int localX[N];

proctype receivingNode(chan input; int ID) {
  StInitial:
    goto StNetworkJoined;
  StNetworkJoined:
    if
      :: input?info0 -> localX[ID] = 0; goto StNetworkJoined;
      :: input?info1 -> localX[ID] = 1; goto StNetworkJoined;
      :: input?info2 -> localX[ID] = 2; goto StNetworkJoined;
      :: input?info3 -> localX[ID] = 3; goto StNetworkJoined;
    fi;
}

proctype periodicLossy(chan input, output; byte keep, lose) {
  byte kept; byte lost;
  StInitial:
    kept = 0; lost = 0; goto StRunning;
  StRunning:
    if
      :: (kept < keep) -> kept++; goto StForward;
      :: (lost < lose) -> lost++; goto StEat;
      :: (kept == keep && lost == lose) -> kept = 0; lost = 0; goto StRunning;
    fi;
  StForward:
    if
      :: input?info0 -> output!info0; goto StRunning;
      :: input?info1 -> output!info1; goto StRunning;
      :: input?info2 -> output!info2; goto StRunning;
      :: input?info3 -> output!info3; goto StRunning;
    fi;
  StEat:
    if
      :: input?info0 -> goto StRunning;
      :: input?info1 -> goto StRunning;
      :: input?info2 -> goto StRunning;
      :: input?info3 -> goto StRunning;
    fi;
}

init {
  int receivers = 0;
  chan send2hub = [chanSize] of {mtype};
  chan lossyChannel = [chanSize] of {mtype};

  run sendingNode(send2hub);
  run broadcastHub(send2hub);

  do
    :: (receivers < N-1) -> run receivingNode(broadcastChannel[receivers],
        receivers); receivers++;
    :: (receivers == N-1) -> run periodicLossy(broadcastChannel[receivers],
        lossyChannel, 1, 9); run receivingNode(lossyChannel, receivers);
        receivers++;
    :: else break;
  od;
}
```