Raoul RUBIEN, BSc

Daisy Chain Communication Protocol for Robotic Particles in Shape-Shifting Displays

MASTER'S THESIS
to achieve the university degree of
Diplom-Ingineur
Master's degree programme: Software Engineering and Management

submitted to:
Graz University of Technology

Institute for Technical Informatics
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe
Supervisor: Dott. Dott. mag. Matteo Lasagni

Graz, August 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz am, _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

This is a placeholder for the abstract. It summarizes the whole thesis to give a very short overview. Usually, this the abstract is written when the whole thesis text is finished.

write abstract

Contents

Contents

List of Figures

List of Tables

1. State of the Art

formulate SOTA

2. Introduction

Programmable matter are materials that ae able to change properties programmatically. Many different materials and approaches have been researched up to now. Among others, programmable matter can be natural materials, liquids, complexly manufactured materials, robotic entities et cetera. An exemplary, rather old programmable matter technology using liquids, is the well known liquid crystal display. The programmable matter is the liquid crystal which is aligned, and thus displaying shapes, programmatically. The field does not only consider nano-scaled programmable matter. Architectural robotics gives a good example for coarse-grained material. The Animated Work Environment [1], consisting of multiple work panels, provides a working environment for different requirements. The system interacts collaboratively with the user and switches panels according to the users need.

The extent of this work is located in the robotic field of programmable matter, where many preferably small scaled robotic entities are applied to achieve different properties. Our work applies many small scaled robotic parts to implement a two dimensional shape shifting surface. The surface is able to change its form. This can be used to approximate three dimensional shapes.

2.1. Background

A shape shifting surface, as shown in fig. 2.1 [2], consists of many parallel chains. The chains are aligned on a plain surface. Chains are capable of changing their shape programmatically. The chain links are so called particles, which interact with subsequent particles and prepare for folding. With that structure and many fine-grained shape-able chains, a high resolution shape shifting surface can be implemented. This technology allows to approximate a three dimensional envelope of any surface.

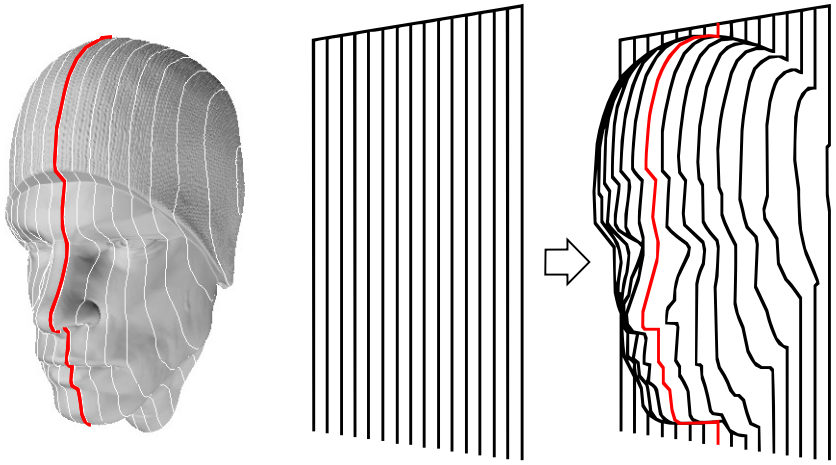


Fig. 2.1.: shape shifting surface principle - a grid of chains approximates a face structure

2.1.1. Mechanical Implementation

The shape shifting surface base of this work is proposed by Lasagni et. al [3]. In the proposal a particle consist of a very basic controller embedded in a mechanical body. Particles are connected by two joints to each consecutive neighbor. This joints reside per default in a locked state. The locked state can be changed programmatically. By collapsing joints particles can be folded.

Fig. 2.2 illustrates the mechanic principle of a folded chain. To fold chains, an external force must be applied. For that multiple chains are physically connected to a global mechanic thread winch, which applies the needed external force by pulling two threads per chain. Each of them is connected to the chains' end, while it slides through each other particle of the same chain. When applying the force, all chains are contracting. If chains contain unlocked joints, they will fold at their location to the unlocked joint's direction. Because of the mechanical implementation the contraction of all chains occurs synchronous. Thus any programmatically configured shape must be executed synchronous.

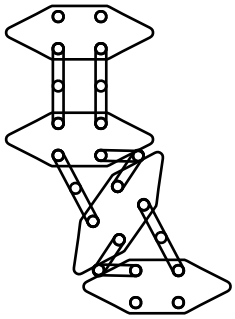


Fig. 2.2.: folded chain

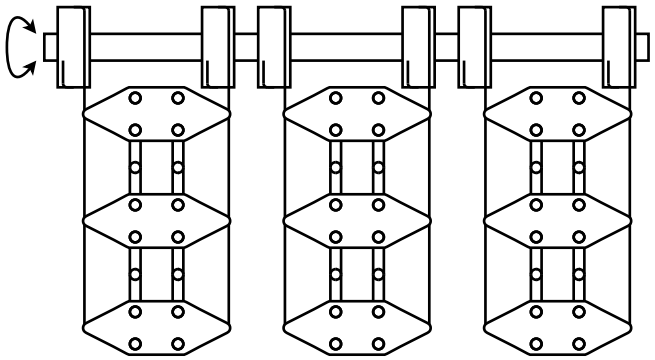


Fig. 2.3.: force mechanics

2.1.2. Electrical Implementation

The very basic controller located on particles provides a communication ability and is able to unlock joints. All combined particles can be viewed as a network of nodes. For the unlocking procedure a very simple actuator is used. It consists of a shape memory alloy (SMA) wire [4], [5], which contracts if the wire temperature rises. Two actuators connect consecutive particles. To actuate, the wire must be connected to power on each side to accordingly.

The power supply of each particle is obtained from a common power line, all chains are connected to. Each chains, again is is attached to the same power line. Within chains, particles are connected subsequently in parallel to the power supply.

With this power supply model, the power line is useable as communication bus system. Thus as communication method the parasitic 1-Wire® protocol is applied.

2.2. Limitations

In regular operation mode, particles receive commands and activate actuators in a repetitive manner. For this reason the power supply must be switched among levels of 0 +5, while communicating, and +12 Volts, for actuation.

During communication the microcontroller's power supply must be pre-buffered parasitic from the 1-Wire® bus. Buffering is not possible while power supply is switched to +12 Volts. Secondarily if the network's total current consumption exceeds the 1-Wire® maximum specification, parasitic powering is not feasible any more. A reliable 1-Wire® application with parasitic powering over a $1k\Omega$ to $5k\Omega$ pull-up resistor allows a maximum current drain of $\approx 2mA$ [6], [7, pp. 2], which does not scale for big particle networks.

ask ML if correct: $\approx 2mA$

The usage of 1-Wire® does not allow a scalable method to localize the position in the network. The first time a network is activated one must find out each microcontroller's 1-Wire® ID and map it to a position in network. To automate this task it is necessary to find out all 1-Wire® IDs and brute force strategy trying every particle to actuate. If two consecutive particles actuate the connecting actuators, the increasing current consumption can be measured. Unfortunately the complexity of this approach is $O(n^2)$.

In the network's use case, the 1-Wire® protocol allows communication to one particle only. The reason for that fact is the bus characteristic. This means while two communication end points are performing their transaction, no other communication can take place in the network.

For firmware development and activation a remote programming must be provided. We understand this procedure as programing the first particle, followed by autonomous replication of the firmware to the subsequent neighbors. The applied 1-Wire® protocol unfortunately does not provide an effective way to implement this feature.

2.3. Motivation

As a consequence of the current electrical implementation the system comprises severe limitations that must be overcome to be scalable for bigger networks. Our motivation is to develop a new daisy chain communication protocol. The protocol must be decoupled from the power supply. The protocol should not make use of supplementary wires because of several reasons. Additional wires introduce more sources of error and makes both, the electrical and mechanical system more complicated.

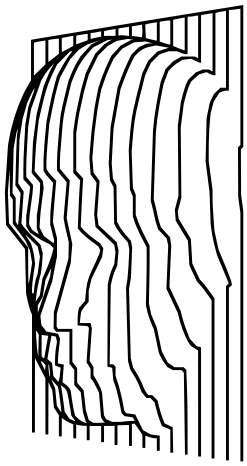


Fig. 2.4.: shape shifting surface - folded to approximate a face shape

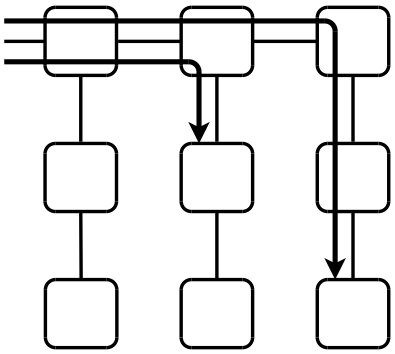


Fig. 2.5.: network structure - communication paths

The idea is to use the already available actuator wires as communication channel. The actuator wires, having $\approx 150\Omega/meter$, provide enough conductance to transmit signals. The protocol shall use one actuator wire as simplex channel. With two wires connected to each neighboring particle the protocol can communicate full duplex with subsequent neighbors. The protocol must ensure synchronous actuation.

For that reason we proposed in our preceding work (attached in appendix section A) a new particle hardware design and a corresponding network structure. The hardware design allows to exploit actuators for both, actuation and communication. In terms of communication, the proposed network structure consists of daisy chain alike connected particles, see fig. 2.5. Chains are connected at their first particle to their subsequent chain. This structure permits connecting multiple particles in a square lattice manner, which can be applied on a shape shifting surface as shown in fig. 2.4. The network allows an external communication with the topmost leftmost network particle. Commands passed to this particle, with different destination then the particle itself, are routed accordingly to the next neighboring particle. With this peer-to-peer (P2P) [8, pp. 120] communication technique particle require the knowledge whereto relay transmissions. For that reason the particle's awareness of position in network and connectivity to neighbors is necessary.

3. Protocol Design

In this chapter we elaborate the protocol design decisions with the focus on what are the aims be achieved rather than the software implementation design.

The protocol design is based on our preceding work, (appendix section A) a new network design to overcome the limitations of the current implementation has been proposed. The network design provides three connection ports for each particle. The connections are located on the top, right and bottom side which we term according to their hemispheric direction as north port, east port and south port. Between ports it applies a serial P2P [9, pp. 156] connection, as illustrated in fig. 3.5. The two wire connection of subsequent particles allows a lightweight full-duplex communication without the need of carrier sense multiple access (CSMA) [9, pp. 708]. The network topology is an unweighted rooted tree with the left most, topmost node as root [10, pp. 24]. Links are static which allows a simple global routing-algorithm. A global routing-algorithm calculates the shortest source to destination path with the knowledge of the connectivity status and costs of the whole network [8, pp. 281].

With this specification we are able to construct differently sized, fully connected networks by simply attaching multiple chains. The one and only communication entry point to the network is located at the origin node's north port. The proposal assumes the same hardware implementation for each connection port.

3.1. Requirements and Constraints

For the upcoming protocol development we have given requirements and constraints which we must consider during the whole process.

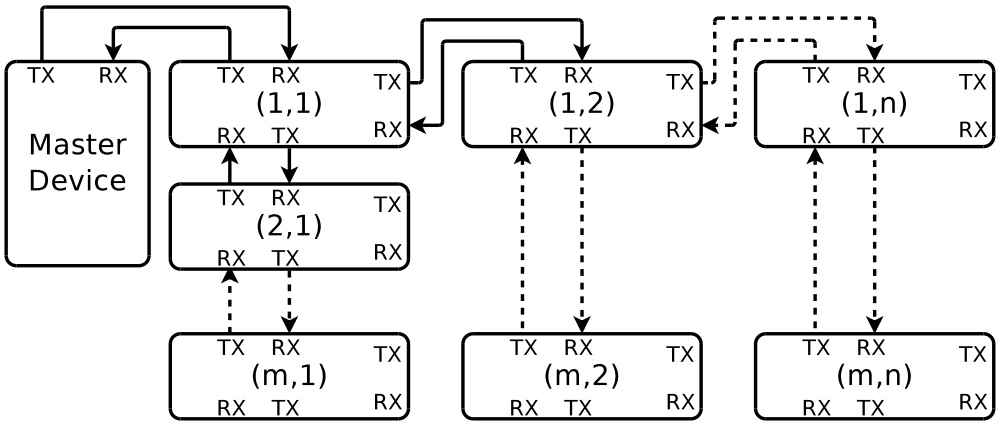


Fig. 3.1.: proposed network structure - each subsequent chain connected at the first particle to the previous chain

Scalability The protocol implementation must be scale-able for big networks. A maximum network size of 255^2 is acceptable.

Cheap microcontroller unit (MCU) The used MCU must be cheap to minimize costs of particles. They will be used in a high number.

Communication Throughput Using a low level MCU without dedicated hardware that implements the physical link layer we need to bit bang the communication. This means signal generating, transmission, reception and decoding of the physical layer (PHY) will be completely software driven. Thus we expect that a sophisticated protocol stack cannot be realized without harming the throughput.

Real Time Control The network behavior and command execution of single network participants must be predictable to ensure syncrhonicity between master device, chain contraction mechanics and network.

Time Synchronization Each particle will be clocked from its rather unstable internal RC circuit, instead of a stable common clock source. To assure synchronous chain interactions, each particle must be aware of both a global time and a compensating factor to compensate the local MCU clock drift.

Automatic Place in Network Detection The non scalable brute force method of the previous implementation must be replaced by a scalable neighbor detection method. The network must initialize completely autonomous

and subsequently be fully functional for interaction with a master device.

Addressing Mode Instead of fixed identifiers as used by the 1-Wire® protocol, particles must be accessible by a simple addressing scheme. As shown in fig. 3.1, particles are identified by their $(row \times column)$ placement in a matrix manner. When communicating to particles they must be addressable directly but also in a rectangular range manner. The range is defined by the upper left and lower right corners.

Remote Programming On firmware updates all particles must be re-written with the new program. For this task each particle must provide accessible serial programming interface (SPI) connectors, even if mounted in the mechanical body. Because of the particle's small size this is not possible. Also the price per particle would rise. A better approach is to program each particle once before mounting them to chains. Later firmware updates must be write-able using the proposed network structure. The replication process would be programming the origin node particle followed by transmission of the new firmware to the connected neighbors et cetera. This strategy parallelizes the programming process and skips the need of touching each single particle.

Debugging, Testing and Mainenance Future enhancements must be easily implementable and testable. Instead of programming followed by trial and error, an automated verification process is desired.

Self Syncrhonizing Line Code The preceding work in section A proposes an adequate line code. The coding combines both, clock and data into one signal. This transmission/reception (TX/RX) method is self synchronizing and has no need of additional clock wires.

3.2. Design

With respect to the requirements we decided to implement a lightweight bit-oriented daisy chain protocol that ensures synchronous execution of commands. The protocol process, see fig. 3.2, autonomously detects the particle's position in chain and self enumerates each network participant so that after an initialization phase, the network is fully functional and requires only slight attention with regards to synchronization. The proposed protocol design follows the Open System Interconnect (OSI) [11, pp. 384-386] standard but is to be

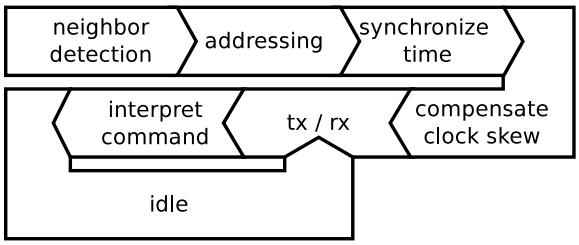


Fig. 3.2.: protocol process stages

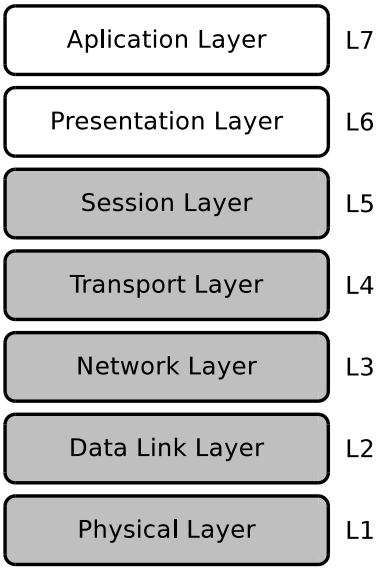
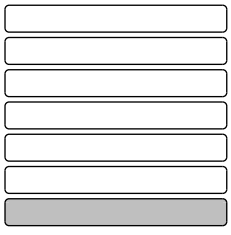


Fig. 3.3.: Open System Interconnect (OSI) layers

seen as just a sub set and does not implement all layers proposed by the OSI standard. The protocol stack [12, pp. 75] implements the physical link layer, data link layer, network layer, transport layer and session layer as highlighted in fig. 3.3. To keep track of the upcoming protocol description, highlighted margin notes indicate the belonging to the process or layer described in fig. 3.2 and fig. 3.3.

3.2.1. Physical Link Layer



Hardware

In physical link layer we specify the electrical circuits and channel coding. For transmitting data an unipolar [9, pp. 131] pulse-code moudlation (PCM) is

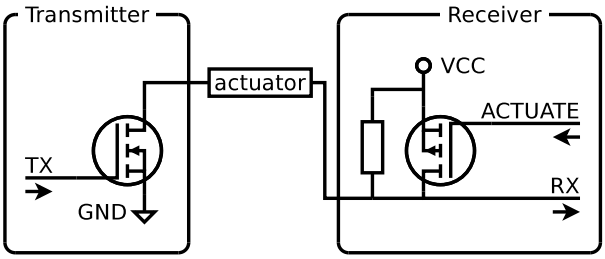


Fig. 3.4.: simplex peer-to-peer (P2P) communication link with complementary MOSFETs at both ends, connected via shape memory alloy (SMA) wire

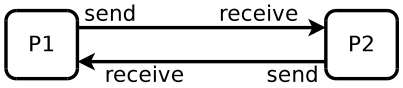


Fig. 3.5.: full-duplex serial peer-to-peer (P2P) connection

applied. The signal levels are switched among $0V$ and $5V$ which corresponds to ground (GND) and supply voltage (VCC).

The TX/RX hardware setup consists of an N-channel MOSFET on the transmission (TX) side and P-channel MOSFET on the reception (RX) side as shown in fig. 3.4. The receiver actively pulls up the voltage level on the link wire to VCC. For generating a falling edge on the link the transmitter pulls up the TX wire which activates the MOSFET and pulls down the the voltage level to GND. A rising edge is achieved by pulling the TX wire to GND. Generated edges are detected at the receiver's side on the RX wire. By implementing both, the TX/RX parts on each side we achieve a full-duplex communication link using two wires as shown in fig. 3.5.

Line Code

The physical link layer allows many degrees of freedom for PCM, however the MCU is limited in memory and speed. Thus, as line code a very simple PCM phase encoding (PE), the Manchester code, has been chosen. The advantages are the self synchronizing nature and simplicity with regards to the implementation. Furthermore the Manchester code code does not require additional clock wires nor a global clock.

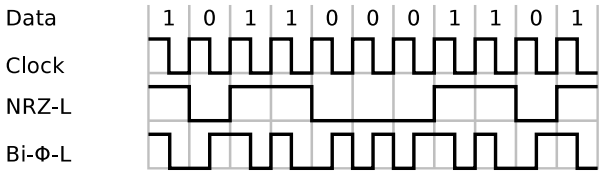


Fig. 3.6.: non return to zero (NRZ) vs. bi-phase-level (Bi-ϕ-L) code

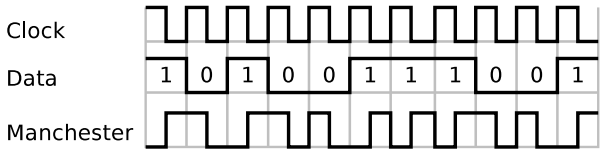


Fig. 3.7.: Manchester code

The code has some disadvantages. It does not porvide an error detection without introducing additional error detection bits into the payload [9, pp. 85-90]. Since the hardware allows unipolar signaling only, a transmission must start with the inverted line value. Otherwise a leading edge will be missed. To overcome this issue we assume each transmission starts with a constant start bit (STB).

The Manchester code belongs to the return to zero (RZ) code group. Thus it needs as wider transmission bandwidth. The rate the signal changes on the communication channel, baud rate, is dependent on the data and is up to two times higher [13, pp. 75-78]. The RZ group encodes signal levels which return to zero after each subsequent bit of same value. In contrast to RZ, non return to zero (NRZ) does do not return to zero on subsequent same valued bits. This leads to a lower transmission frequency. The comparison of NRZ and RZ using NRZ-Level (NRZ-L) and bi-phase-level (Bi-ϕ-L), better known as Manchester code, are lined up in fig. 3.6. The Manchester code provides a self-clocking by combining the transmission clock with the data. The code provides a clock transition in every bit interval. At the receiver's side this is used for reception clock synchronizing. The illustration in fig. 3.7 shows the principle of the Manchester code. Clock and data merging/separation is calculated with the very basic exclusive or ($\oplus$) operation as expressed in equation (3.1).

$$data \quad = \quad clock \oplus manchester \tag{3.1}$$

Implementation Investigation

Fortunately the Manchester code is simple to implement and can be processed relatively fast. For signal de-/coding we elaborated several bit bang approaches.

Manchester Decoding For both approaches an internal 16-bit timer/counter (TCNT) is used. The TCNT is set up to fast pulse width modulation (PWM) mode [14, pp. 80]. When received signals are processed we reference the current TCNT as a time stamp which is used for calculations. Both approaches are a time-memory tradeoff, differ in the baud rate but have the same throughput as illustrated in fig. 3.8.

On-the-fly Decoding In this approach the signal is decoded on-line in the interrupt service routine (ISR). Each edge change triggers the ISR where the signal is decoded according to the previous edge time stamp. This method does not need any buffering except of the previous timestamp value which is neglect-able. On the other hand the baud rate is limited by the duration the ISR takes to decode the signal edge.
Another aim is to exploit the Manchester code self synchronization property for global time synchronization. This method poorly sustains the global synchronization needs. From the last timestamp, without buffering, one can hardly infer timing adjustments. A better approach is to buffer and decode later.

Post Processing In this approach a signal edge is buffered as fast as possible in the correspoinding ISR and decoded later. The disadvantage of this method is the huge amount of buffer needed to store the signal.
The buffer size ($|\ buffer\ |$) per port depends on the amount of decoded data bytes the protocol should receive at once, the baud rate per bit (two signal changes per bit) and the time stamp resolution (16-bit TCNT). If the protocol must buffer 9 bytes before they are decoded the $|\ buffer\ |$ must be 288 bytes (see equation (3.2)). Additionally if if a full-duplex communication for all three ports should be achieved, three buffers must be allocated, which in total occupy 864 bytes. The calculation can be seen as an upper bound approximation. However the decoding process takes place partially parallel to the reception. Therefore the effectively needed

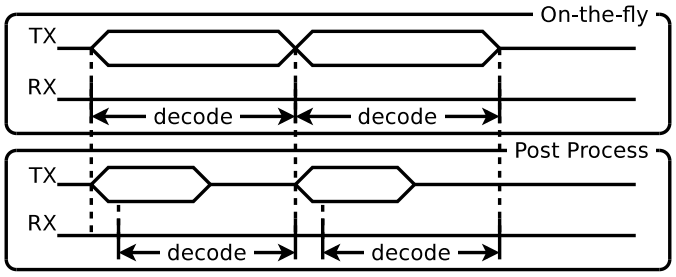


Fig. 3.8.: on-the-fly decoding vs. post processing

buffer can be reduced to fraction $\approx 20\%$ of the calculation above. The reduction is expected to be linear dependent on the PDU size ($\mid PDU \mid$). The mentioned fraction holds for a $\mid PDU \mid$ of ≈ 9 bytes.
With $\approx 1kB$ static RAM (SRAM) available on the MCU this approach is limited but a protocol data unit (PDU) of 9 bytes can be safely transferred at once, which is sufficient for the protocol implementation. Thus the maximum transfer unit (MTU) can be preliminary fixed at 9 byte.
With buffered data it is possible to infer much more timing adjustments which are needed for global synchronization. The baud rate limit is also higher as with the first approach.

$$\begin{aligned} \mid buffer \mid &= 8 * 2 * sizeof(uint16\_t) * bytes_{rx} \\ &= 32 * bytes_{rx} \end{aligned} \tag{3.2}$$

Manchester Coding For the signal generator the internal 16-bit TCNT compare match ISR is applied. Two approaches have been explored. The trivial approach uses two TCNT compare register where the advanced uses only one compare register, as illustrated in fig. 3.9. In both methods the TCNT value is never touched. Once transmission is enabled, each subsequent interrupt is scheduled by the ISR before.

Two Compare Register Approach The method applies two TCNT compare registers, thus two ISRs. Each interrupt is set up to occur once at each TX clock phase. Both have a phase shift of π. With this setup the interrupt occurring in the center of a clock, generates the Manchester code by

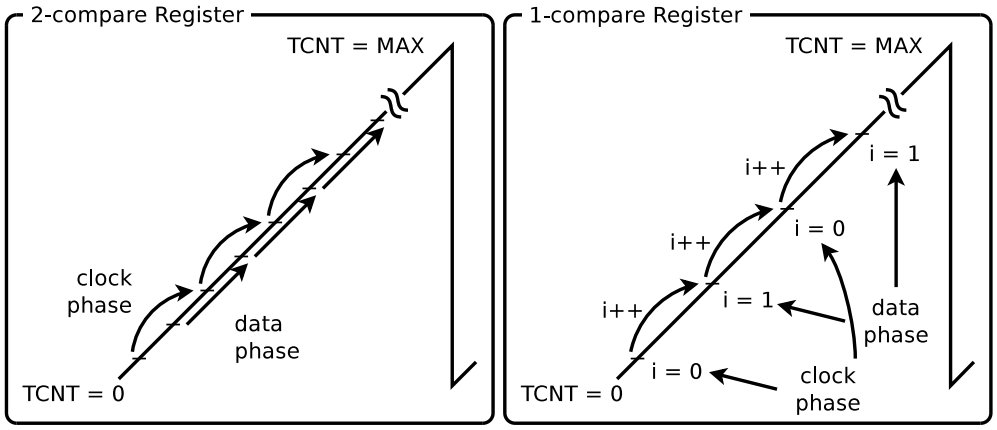


Fig. 3.9.: signal generator scheduling - two compare register vs. one compare register approach

terms of equation (3.1). The second interrupt rectifies the signal according to the next data to be transmitted (see fig. 3.7). To save one compare register and advanced approach has been tried out.

One Compare Register Approach This approach applies one interrupt for generating signals in both phases, clock and data. For phase tracking we use a 1-bit counter which is incremented by one each time the interrupt occurs. When the counter equals zero the interrupt occurs at the beginning of a phase, whereas if the counter is equals one the interrupt occurs at the half of a phase. With this information we can apply the same signal generator strategy as described in the first approach by using just one TCNT compare register.

Design Decision

We have chosen to use a post processing decoding method, which consumes plenty of SRAM [15, pp. 315]. The approach has a higher baud rate but the same throughput (see fig. 3.8). With this approach we have retrospection into the reception buffer which allows an global time synchronization strategy to evaluate reception timings out of the buffered reception.

The signal coding approach reduces the number of necessary TCNT compare register to a minimum, which is vital for the protocol implementation. The

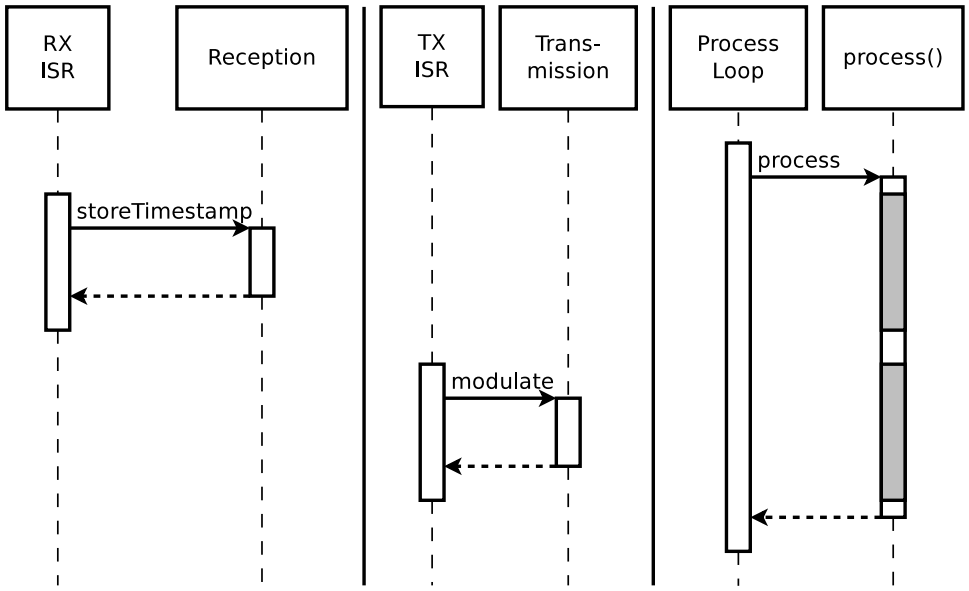


Fig. 3.10.: reception and decoding sequence diagram - gray highlighted areas are interrupted intervals

second compare register is needed for a different purpose. It also does not touch the TCNT value which is continuously used for decoding and other tasks.

The basic design approach of coding/decoding and processing is illustrated in fig. 3.10. Gray highlighted areas are interrupted intervals due to ISR calls. TX/RX ISRs calls do never overlap.

3.2.2. Data Link Layer

The data link layer is responsible for collecting streams of received bits [13, pp. 27] and vice versa creating streams of bits from packages. It provides connectivity of subsequent network nodes only. The package frame introduced in this layer consists of one package header (HDR) field, containing several data link layer control bits as shown in fig. 3.11. An extensive package listing

can be found in section B. The HDR field contains a STB, parity bit (PRT) and a broadcast bit (BCT). The remaining bit is not used.

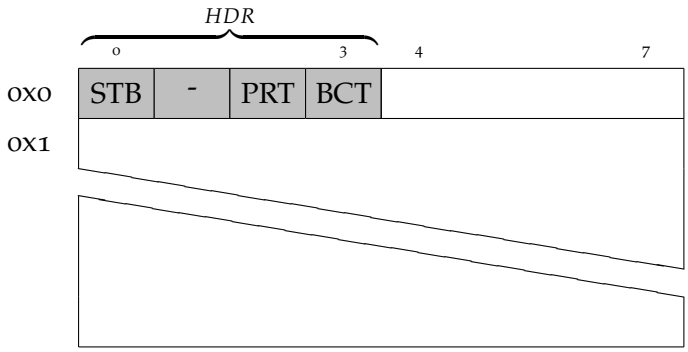


Fig. 3.11.: header field

STB The STB is constantly set to one and necessary for the Manchester code. Without STB the RX of PDU's starting with a zero bit will fail, since the first signal edge cannot be detected on the line as described in section 3.2.1.

PRT The PRT is used for error detection of up to 1-bit. To assure an even amount of total ones the PDU's PRT it is set accordingly (1-bit even parity).
Different error corrections/detections or automatic repeat requests (ARQs) are planned for future work since they are non vital features for the network protocol.

BCT The BCT bit changes the reception mode. A received package with BCT set affects the next reception only. On the subsequent reception, signal edges are replicated to each other port as they occur. Thus the signal is broadcasted with a minimum latency. This state remains until the next package is decoded and interpreted. To retain the broadcast state longer, subsequent packages must have the BCT flag set continuously. The broadcast implemented in this layer differs from the multicast routing concept in the network layer. This layer permits instant simultaneous broadcasting with a minimum of latency, whereas the network layer must completely receive a package until it is able to apply the routing algorithm.

For simplicity, as byte order we use the MCU's internal endianness without Marshalling/Unmarshalling tier [16, pp. 35]. Any data package is constructed as C-Union/Struct [17, pp. 27] which, for transmission, is iterated bit-wise beginning at struct's base address. A Little-Endian byte order example starting with the struct's least significant bit or byte (LSB) is illustrated in fig. 3.13. The corresponding bit stream is illustrated in fig. 3.14.

```
typedef struct Data {
  uint8_t  x;
  uint8_t  y;
  uint16_t z;
} Data;

Data data = {
  .x= 0xCA,
  .y= 0xFE,
  .z= 0xBEEF,
};
```

Fig. 3.12.: data structure example

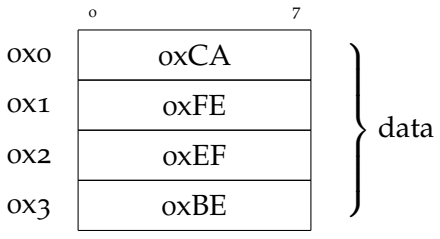
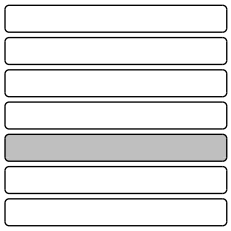


Fig. 3.13.: Little-Endian data structure on microcontroller unit (MCU)

3.2.3. Network Layer



The network layer is responsible for addressing, routing, flow-control and extends the P2P communication boundaries to node-to-node communication. This requires an addressing and routing scheme. The introduced frame consists of optional address fields as shown in fig. 3.15 and fig. 3.16. The number of address fields varies depending on the use case: unicast (one address), multicast (two addresses) or TX to neighbor (no address).

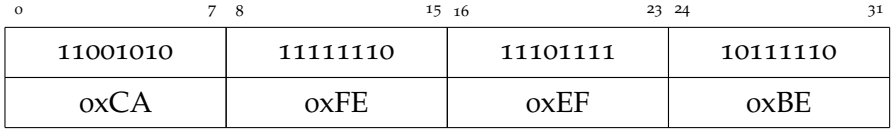


Fig. 3.14.: transmission bit stream example containing the the structure data as explained in fig. 3.12

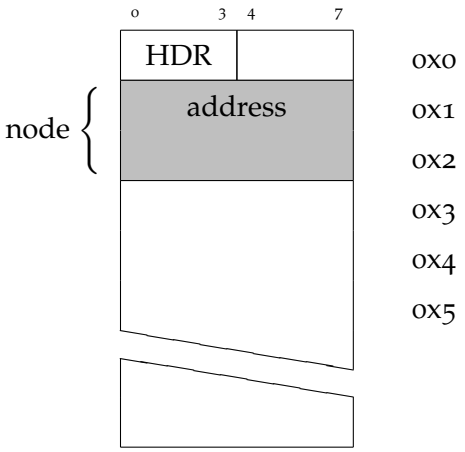


Fig. 3.15.: unicast package

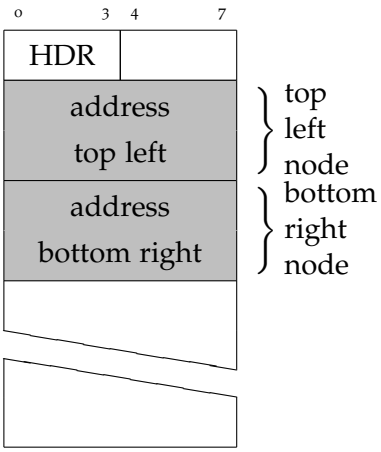
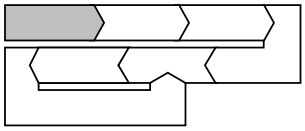


Fig. 3.16.: multicast package

Discovery

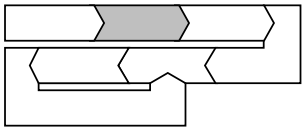


When the network is booted, particles need to sense their position in network before any addressing can be started. The discovery phase starts immediately after nodes are powered. In the discovery phase nodes generate pulses on the transmission wires, which are sensed by neighboring nodes. For proper sensing the origin node assumes a connected master device must not send any pulses within this phase. The sensing phase consists of three parts as shown in fig. 3.17: the counting-phase, classification-phase and post-classification-phase. During the whole counting-phase, incoming pulses are counted. If a port counter exceeds a specific threshold, the affected port is marked as connected. In the classification-phase a node tries to classify itself according to the ports connectivity. If a classification cannot be done within this time window, the protocol assumes no neighbors are connected. The post-classification-phase keeps pulsing for a short safety period. The design decision was to provide more pulses than necessary, where a fraction of pulses are enough to assure a valid classification. When the discovery phase is finished each node is classified as one of the listed types. A classification matrix can be found in fig. 3.18.

Origin node The top left node, connected at east port and south port.
Inter node A node connected at north port and south port.

Inter head A node connected within two head nodes at north port, east port and south port.
Tail node A node connected at north port only.
Orphan node A node without any connected ports.

Addressing



As addressing scheme we orient on a matrix manner numbering which uses a $(row \times column)$ coordinate for each matrix entry. The first network address starts at $(1,1)$ which equals the origin node and resides on the top left in the lattice. The subsequent node connected at the south port has the incremented row address $(2,1)$. The subsequent east node is addressed $(1,2)$. Although we orient on a matrix numbering which naturally is rectangular, the addressing scheme makes no assumption about the real network size. The scheme may also be used to address networks having irregular chain lengths. For simplicity we consider addressing rectangular shaped networks only. Addressing methods are unicast, multicast and broadcast.

Unicast A direct addressing mode where the address consists of address row (ROW) and address column (COL). The unicast PDU is illustrated in fig. 3.15.
Multicast A range addressing mode where the address consists of top left address row (ROW1), top left address column (COL1) and bottom right address row (ROW2), bottom right address column (COL2). The multicast PDU is illustrated in fig. 3.16.
Broadcast This addressing mode requires no additional address information.

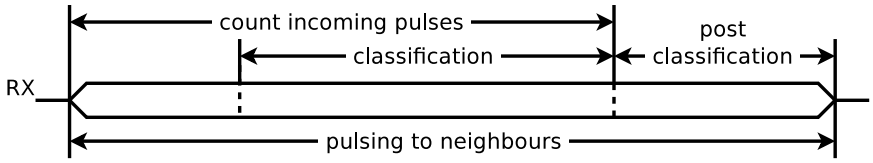


Fig. 3.17.: discovery phase

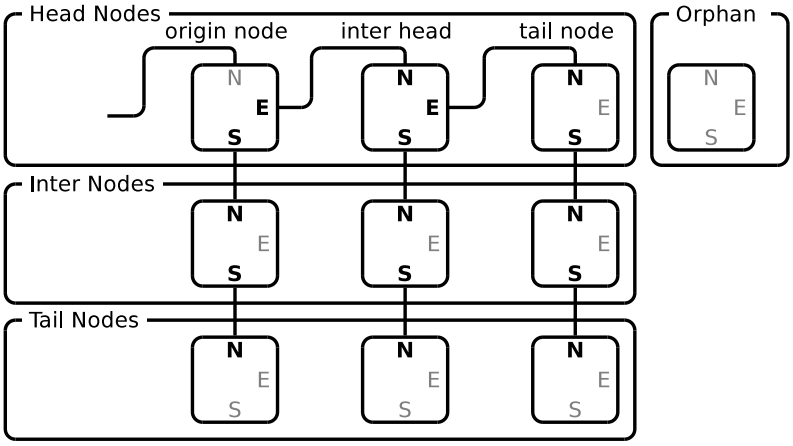


Fig. 3.18.: node classification matrix highlighting the possible node types: origin node, inter head, inter node, tail node and orphan node

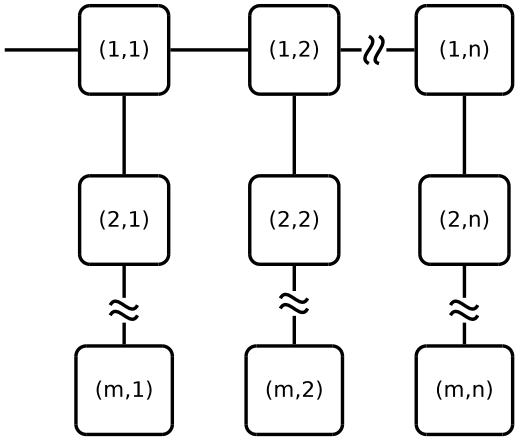


Fig. 3.19.: $(rows \times columns)$ network addressing schema

Routing

Since we have a rather simple, static and well defined network structure we opted for a global routing-algorithm. The algorithm needs no further information except the network topology, which is fixed as unweighted rooted tree with the origin node and the node's address. With this definition and

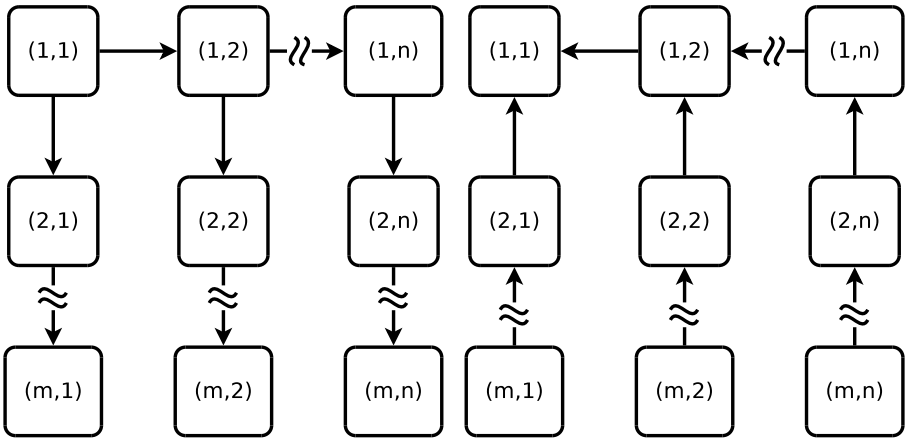


Fig. 3.20.: directed out-tree

Fig. 3.21.: directed in-tree

the addressing scheme (fig. 3.19) we are able to define the global routing-algorithm by a simple set of rules. According to the package direction we split the undirected network into two directed spanning trees. The directed out-tree in fig. 3.20 shows all possible paths from the origin node. The directed in-tree in fig. 3.21 shows the flow back to the origin node. The routing rules may result into none, one or two ports. At this point we assume unconnected ports are excluded.

Unicast Routing In this use case packages carry one address field. A intuitive stet of directed out-tree and directed in-tree routing rules is described by $R_{out}(...)$ in equation (3.3) and $R_{in}(...)$ in equation (3.4). If the BCT bit is set, no routing decisions will be met at this layer. The routing algorithm applies $R_{out}(...)$ for forward routing. In case $R_{out}(...)$ does not provide any result backward routing $R_{in}(...)$ is applied. The unicast PDU is illustrated in fig. 3.22.

$$R_{out}(BCT,\ local,\ dest) \mapsto r \subset \{east, south\}$$

$$R_{out}(...) = \begin{cases} \{\} & if \quad (BCT = 1) \\ \{east\} & elif\ (local.COL < dest.COL) \\ \{south\} & elif\ (local.COL = dest.COL)\ and \\ & \quad (local.ROW < dest.ROW) \\ \{\} & otherwise \end{cases} \tag{3.3}$$

$$R_{in}(local,\ dest) \mapsto r \subset \{north\}$$

$$R_{in}(...) = \begin{cases} \{north\} & if\ (dest.COL < local.COL) \\ \{\} & otherwise \end{cases} \tag{3.4}$$

Multicast Routing In this case the protocol assumes multicast transmissions are issued only by the origin node. Packages can only traverse the directed out-tree which simplifies the routing rules (equation (3.5)). The multicast range is defined by a rectangular shape having two address coordinates, the top-left and bottom-right node. In this use case packages always carry two address fields. The routing rules forward a package to the east until the bottom-right node's column is reached. If the top-left node's column is reached, the package is duplicated and routed also to the south. The multicast PDU is illustrated in fig. 3.23.

$$R_{out}(local,\ dest1,\ dest2) \mapsto r \subset \{east, south\}$$

$$R_{out}(...) = \begin{cases} \{\} & if \quad (dest2.ROW < local.ROW) \\ \{east\} & elif\ (local.COL < dest1.COL) \\ \{east,\ south\} & elif\ (dest1.COL \leq local.COL)\ and \\ & \quad (local.COL \leq dest2.COL) \end{cases} \tag{3.5}$$

Broadcast Routing A Broadcast routing is already implemented in the data link layer. If the BCT flag is set a routing algorithm at a higher level is excluded. In this use case no address fields are carried at all.

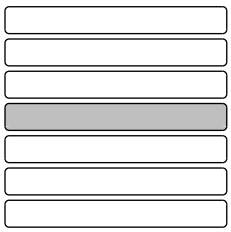
When a PDU is passed over the last link it may be transmitted without the destination address, since this field is obsolete. The protocol design intends

this method of communication to neighboring nodes but this part is intended for future port, thus not implemented but. The software design is prepared for this kind of extension. For simplicity packages pass the last link unmodified.

Flow Control

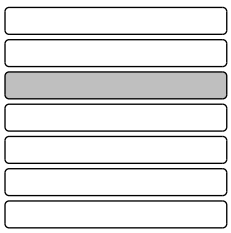
Despite the OSI layer design the protocol does not provide flow-control at this level. Due to the lightweight and real time nature of the protocol the flow-control is seen to be rather a part of the real time protocol, thus handled beyond the transport layer as proposed in [18, pp. 501].

3.2.4. Transport Layer



The transport layer introduces a command id (CMD) an optional variable sized payload field. The command id (CMD) field describes the action to be executed by the receiving system. Depending on the CMD the package may contain a payload field, whereas the payload size depends on the CMD's specification. The non continuous payload field's characteristic is based on a implementation decision. The aim was not to patch a TX buffer, but instead use a C-Union/Struct, which for transmission aims is iterated bit-wise from the struct's base until the end. This decision requires a proper field alignment within the structure which lead to placing the CMD field into the unused 4 bits in the HDR, while appending the remaining payload as illustrated in fig. 3.22 and fig. 3.23. Commands received by this layer are passed to an interpreter which is discussed in the implementation part, section 4.

3.2.5. Session Layer



Besides global time synchronization and actuator scheduling the session layer implements the real time protocol's flow-control. Due to the rather low baud rate we assume transmitted PDU's do not require retransmissions, thus no acknowledgement (ACK) packages except of the special case are transmitted: the very first transmission following the discovery process, the enumeration phase.

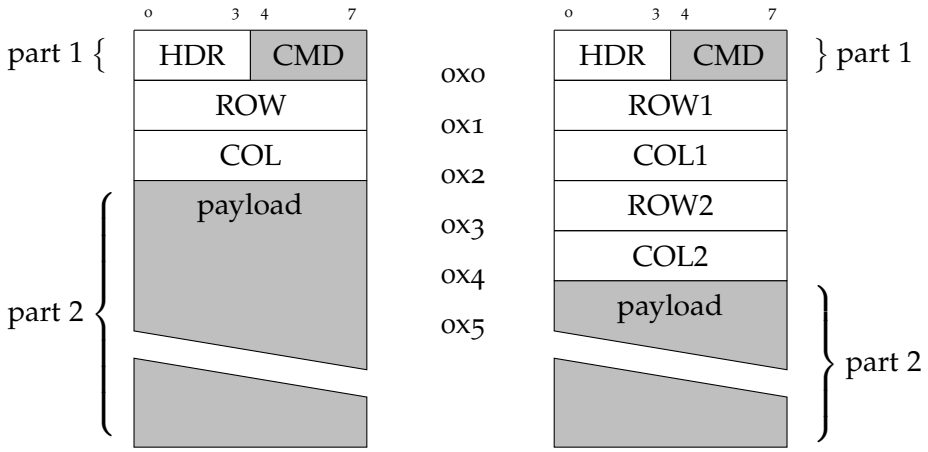


Fig. 3.22.: unicast protocol data unit (PDU)

Fig. 3.23.: multicast protocol data unit (PDU)

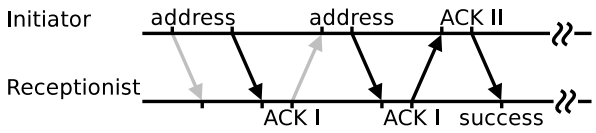


Fig. 3.24.: flow-control in addressing phase, highlighted arrows represent protocol data units (PDUs) followed by timed out reception (RX)

Flow Control

A correlation between the node indegree and the discovery process duration has been observed. On tail nodes the discovery phase ends $\approx 15\%$ earlier than on other nodes. This is due to the overlapping ISR's triggered by incoming signals and pulse generating ISR. The overlap leads to a non relevant pulse jitter which can be ignored. In detail this means some nodes may be already listening to incoming data while others are still pulsing the discovery signal.

To disambiguate discovery from data signals this the protocol's flow-control implements a simple stop-and-wait-protocol strategy for the first PDU transmissions after the discovery phase. This applies to the addressing phase only. However, if necessary the flow-control may be used also for different packages.

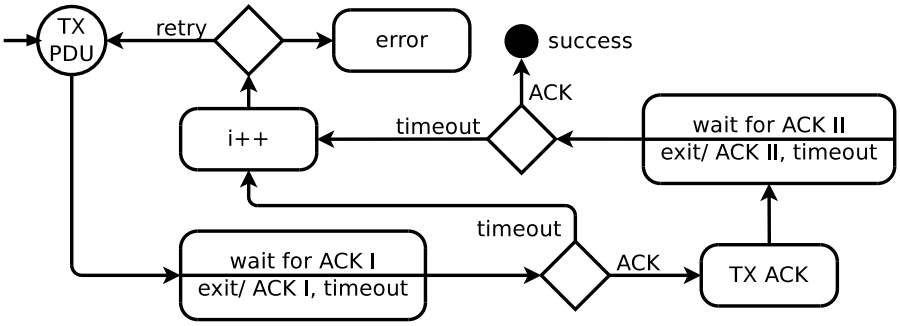


Fig. 3.25.: initiator transmission (TX) flow-control

The stop-and-wait-protocol implementation distinguishes into the node initiating a communication (initiator) and the node waiting for reception (receptionist). In the addressing phase the initiator sends a package containing the new address to the neighbor. If the receptionist receives this package correctly it ACK's it by replying the same address (ACK I). The initiator then checks the content. In case of match it replies ACK II which terminates the flow-control on both sides. Otherwise the initiator retransmits the addressing package.

Fig. 3.24 illustrates the flow with transmission errors emphasized in gray. The waiting states are interrupted by timeouts which assures the system never remains in a locked state. This process is repeated until the retry counter is consumed, which ends up in an error state. The error state is a dead end state. The described initiator and receptionist flow-control implicitly ensures ARQs. The corresponding finite state machines (FSMs) are illustrated in fig. 3.25 and fig. 3.26.

For regular communication, when the network is already initialized, the proposed stop-and-wait-protocol produces too much overhead, since it is acknowledged twice (ACK I and ACK II). For that reason a FSM with lesser states is used. A further enhancement of the flow-control can be achieved by interacting with the data link layer's PRT bit which may be of interest in future work.

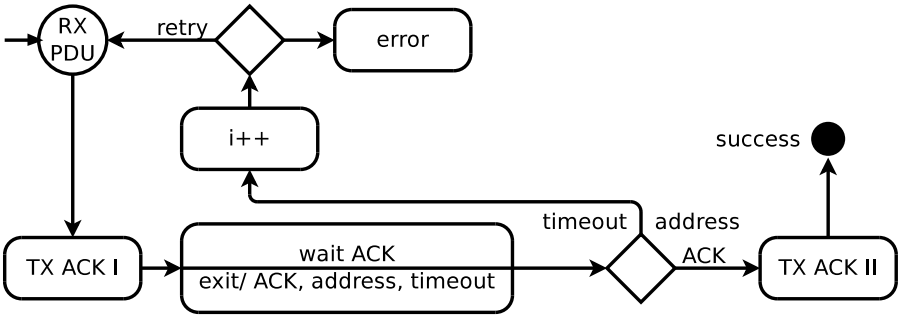
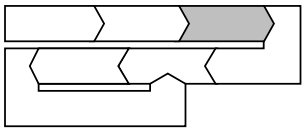


Fig. 3.26.: receptionist transmission (TX) flow-control

Synchronization



The protocol's focus is the synchronized execution of commands. Since the node design does not provide a real time clock (RTC) nor an external time synchronization method, the protocol must provide an accurate synchronization mechanism. The protocol also must consider the circumstance that MCUs are clocked by their rather inaccurate internal RC circuit. Thus we encounter two problems, a missing global clock and a possible clock skew. In the upcoming calculations, if not specified differently, the time base refers to MCU cycles rather than seconds.

The underlying counting mechanism is performed by a 16-bit TCNT which is coupled directly to the MCU clock without prescaler. This allows to us calculate directly with MCU clock cycles.

Regarding communication, during the synchronization phase, we see two possibilities. Packages can be spread through the network simultaneously (broadcast mode) or subsequently.

In both modes, broadcast and subsequent, the clock skew compensation mechanism is the same. When a PDU is completely received, except of the delivered data, it provides additional information such as start and end time stamp of the reception. The access to this information is vital for the whole synchronization process.

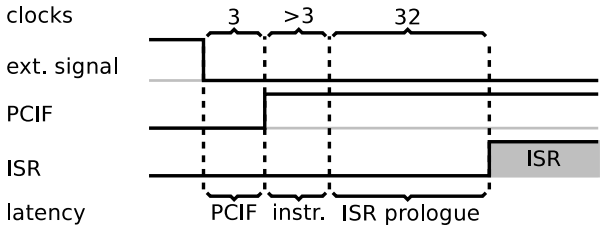


Fig. 3.27.: external pin change interrupt service routine (ISR) latency

Broadcast Mode In this mode the synchronization phase takes place while the network is set to broadcast mode. The origin node transmits a PDU containing several timing arguments. The transmission is performed simultaneously on both, the east port and south port. The simultaneous transmission introduces a minimal latency between both ports. When signals are received, nodes in broadcast mode firstly relay the signal to any connected port, then they record the time stamp. The relaying process, again introduces a minimal latency to each edge that passes the current node. Unfortunately this not only results in a simple shift that can be easily calculated according to the path length a signal has traversed, but also introduces a reception and transmission jitter at each node which is cumulated on each forwarding.

Time Synchronization Approach - Broadcast Mode For this method the whole transmission delay must be broke down to be able to recalculate the total duration from the timestamp the PDU was issued by the origin node. The transmission duration consists of the PDU transmission duration (d_{pdu}) and the introduced total signal latency ($t_{latency}$). The $t_{latency}$ can be seen as largely linear depending on the path length the message traversed. It consists of the latency introduced by the transmitter and the signal latency of one hop (t_{hop}) (equation (3.6)) the PDU experiences as formulated in equation (3.7). In small networks the introduced latency is neglect-able but it must be considered for larger networks.

The introduced per node latency t_{hop} can be split in two parts, a constant and a variable introduced latency. On the RX side the latency consists of four components: the pin change interrupt hardware timing, the instruction interrupting mechanism, the ISR prologue and the signal forwarding duration (t_{fwd}). The

pin change interrupt timing until the pin change interrupt flag (PCIF) is set is guaranteed to be constantly three clocks [14, pp. 50]. The prologue takes 32 clocks thus belongs to constant duration (t_{const}). The interrupt response time is at least four clocks [14, pp. 12] which corresponds to four clocks for the jump (t_{const}) and a variable duration (t_{var}) to interrupt the current instruction. Since instructions are atomic, they cannot be interrupted, but are executed trough. The instruction duration at the current MCU may take one up to four clocks [14, pp. 278], which introduces a jitter as illustrated in fig. 3.27. The encountered unpredictability of t_{var} problem is because one cannot make any assumptions about the instruction type executed when the interrupt flag is set. In this work no approximation models are applied to compensate the variating part instead an empirical value is used. On the TX side we face a similar problem, except the missing pin change interrupt flag latency (t_{pcif}).

32 clocks prologue: no documentation found

$$t_{hop} = t_{pcif} + t_{instr} + t_{prologue} + t_{fwd} \tag{3.6}$$

$$\begin{aligned} t_{latency} &= (-2 + row + column) * t_{hop} + (t_{instr} + t_{prologue} + t_{fwd}) \\ &= (-1 + row + column) * t_{hop} - t_{pcif} \end{aligned} \tag{3.7}$$

Since this approach is a far to complex, also having a cumulative error source no deep investigations for the synchronization in broadcast mode has been done.

Subsequent Mode In this mode the synchronization phase takes place while the network is not in broadcast mode. Again the origin node starts the synchronization by sending synchronization PDUs on both ports simultaneously. When these PDUs are received, they are interpreted, executed and lastly the current node prepares the transmission of a new synchronization package. This new package contains fresh timing arguments.

For this method only the transmission in between two consecutive nodes must be broken down, which is less error prone. Even if the RX and TX still suffer of a jitter, this approach does not cumulate the jitter illustrated in fig. 3.27.

Time Synchronization Approach - Subsequent Mode This synchronization approach is very simple. When a synchronization PDU is received the time

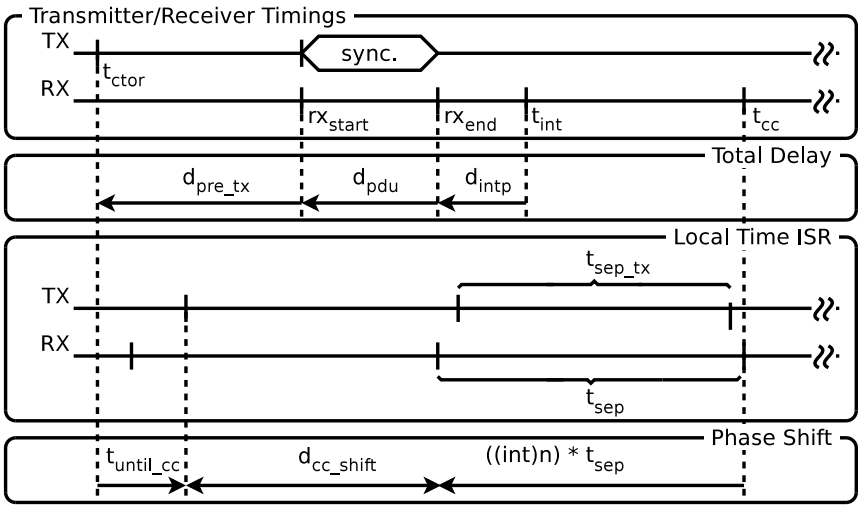


Fig. 3.28.: time synchronization and phase shift - step-by-step illustration of latencies a Time-Package experiences since it is constructed by transmitter until executed by the receiver

span since remote PDU construction until local execution (d_{ctor_intp}) by the receiver is the sum of the receiver's the interpreter delay (d_{intp}) followed by d_{pdu} and a constructor to transmission delay (t_{pre_tx}) as illustrated in fig. 3.28. Thus the time of PDU construction (t_{ctor}) is the time when the PDU is interpreted (t_{int}) minus d_{ctor_intp} as formulated in equation (3.8). With this information the amount of missed local time increments during the PDU transmission can be inferred.

$$\begin{aligned} d_{ctor\_intp} &= d_{intp} + d_{pdu} + d_{pre\_tx} \\ where & \\ d_{pre\_tx} &\approx 381\mu s \stackrel{\wedge}{=} 3 * t_{code} \\ t_{ctor} &= t_{int} - d_{ctor\_intp} \end{aligned} \tag{3.8}$$

To be more accurate also the local time counting ISR must be shifted to be in phase with the remote one as illustrated in fig. 3.28. Without shifting we would create a cumulative off-by-one error of the local time among all network

nodes. Thus the TimePackage carries data about the transmitter's local time (t_{tx}) when the PDU was constructed, the transmitter's local time tracking clock delay (t_{sep_tx}), the delay until next local time increment (t_{until_cc}), the force update local time flag (FU) and the end bit (EB). With this informations the shift in between t_{cc_tx} and t_{cc} (d_{cc_shift}) can be calculated. For this we take the difference of t_{until_cc} and t_{ctor} as basis. From this delay we take the rest of the division by local time counter clock delay (t_{sep}) as formulated in equation (3.9). For this calculation we assume that t_{sep}, or in other words the clock skew, is already correctly adjusted. The newly obtained value d_{cc_shift} can now be used to shift the local local time ISR compare value, equation (3.12) (t_{cc}) accordingly. Instead of shifting t_{cc} by the whole amount of d_{cc_shift}, a step-wise shifting has been implemented. This means if d_{cc_shift} exceeds a specific threshold the shift value is the threshold itself, d_{cc_shift} otherwise.

$$d_{cc\_shift} = (t_{cc} - t_{until\_cc})\%t_{sep} \tag{3.9}$$

The $t_{sep}[ms]$ at a MCU clock frequency (f_{cpu}) of $8MHz$ corresponds to $\approx 6.528ms$ as formulated in equation (3.11). Thus the current local time (t_{now}) overflows every ≈ 428 seconds.

$$t_{sep} = 51 * t_{code} \tag{3.10}$$

$$t_{sep}[ms] = \frac{t_{sep}}{f_{cpu}} \tag{3.11}$$

$$t'_{cc} = t_{cc} + \frac{t_{sep}}{2} \tag{3.12}$$

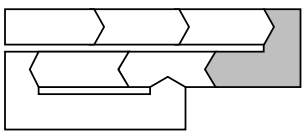
$$t_{pdu} = t_{code} * pdu\_bits \tag{3.13}$$

$$t_{pdu}[ms] = \frac{t_{pdu}}{f_{cpu}} \tag{3.14}$$

$$where$$

$$t_{code} = 1024 \tag{3.15}$$

Clock Skew Compensation A time skew may be introduced by many parameters. The most significant are inaccurate resistor-capacitor (RC) oscillator due to production factors, oscillator's temperature drift, VCC voltage drop,



ripple et cetera. To compensate a possible time skew we state that nodes adjust their local time counting along the origin node's counting speed. To obtain a clock speed from the origin node a fixed time span reference is needed as reference. As reference the data clock duration of the Manchester code (t_{code}) or the time package's d_{pdu} can be used. Due to accuracy we opted for the longer time interval d_{pdu}.

The compensation mechanism firstly observes d_{pdu} which holds equation (3.13) and updates all dependent values such as t_{sep} which holds equation (3.10) and the new baud rate by updating t_{code} in equation (3.15). By updating the baud rate the local changes are also exposed to the subsequent neighbor on the next TX.

In the optimum case, if two subsequent neighbors have the same f_{cpu}, t_{sep} resides at $\approx$ 80% of the TCNT maximum value which gives us $\pm$20% margin for adjustments. If even more margin is needed the constant in equation equation (3.10) can be tuned.

Tuning the the internal RC oscillator adjustment as proposed by [19] has also been considered. Because of the more complex, non linear internal RC oscillator calibration register (OSCCAL) to f_{cpu} correlation and the not so fine grained tuning possibility this option was discarded. No further investigations regarding OSCCAL has been done.

The environmental temperature is not considered in the adjustment model. However it is assumed that the synchronization process must be triggered frequently to stay consistent. If the system shares the same environmental temperature, this will cache the problem largely. For sporadic fast temperature changes of network parts the protocol provides no automatic adjustment.

For both, time synchronization and clock skew compensation, the protocol provides one PDU, the TimePackage which is illustrated in fig. 3.29. Each received package triggers the clock skew compensation, whereas the time synchronization only if the FU flag is set.

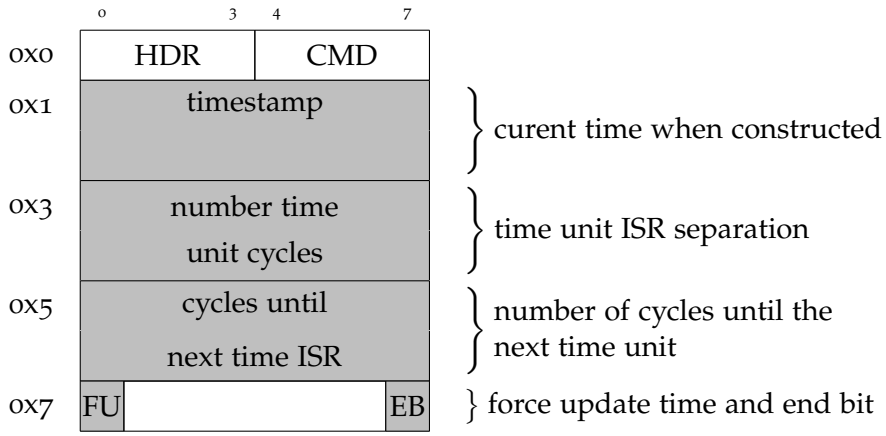
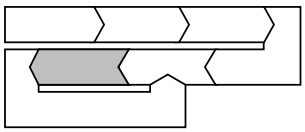


Fig. 3.29.: TimePackage

Actuation Command Scheduling



Actuation commands to be scheduled are defined by the command PDU which contains the destination node fig. 3.30 (or range of nodes fig. 3.31), a command start time (t_{start}), the command duration ($t_{duration}$) and the affected wires' flags (left wire flag (L), right wire flag (R)). Independent of t_{now} the command is scheduled for the next period which matches $start \leqslant t_{now} \leqslant (start + duration)$. When this period is passed the command is removed from the scheduler. During this period the affected wires are powered according to the actuator PWM setting, which per default is set to 50% duty cycle.

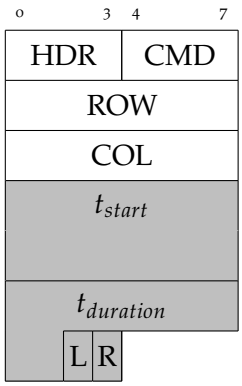


Fig. 3.30.: actuation command addressing one node

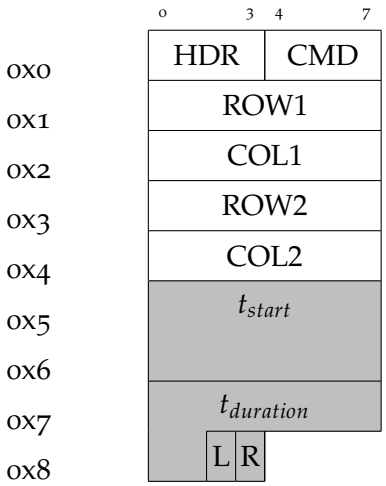


Fig. 3.31.: range actuation command addressing a node range

4. Implementation

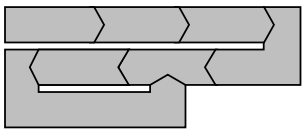
This chapter is focused to the implementation design decisions made to achieve the proposed protocol design. It also elaborates a set of optimization options. Based on the available flash size we focused to avoid a high level programming as C++ but also not to go to low level as with assembler. For the ATtiny1634 MCU we apply the free of charge AVR compiler avr-gcc of the GNU Compiler Collection (GCC) with gnu99 standard.

4.1. Software Implementation

To realize the desired protocol behavior the software is designed be to state driven. This eases implementation of context sensitive actions, which are natural for protocols. The FSM is oriented to the State pattern as proposed by [20, pp. 398]. Although the pattern is proposed for object oriented programming (OOP) we implement the behavior by means of C language.

4.1.1. Main Loop

The main process (*process*()) is programmed to check whether actions can be performed within the current state and the given context Thus it is called in a loop as illustrated in fig. 4.1. In general *process*() is called, eventually changes



```
void main() {
    while(true) { process(); }
}
```

Fig. 4.1.: invocation of the *process*() function

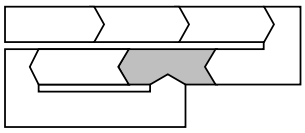
node states and returns With this design, states are effectively interruptible and can be cut short if necessary. For instance the TX/RX flow-control must never get stuck in a state without the possibility to recover. Thus timeout counters which are not required to be very accurate are based on the amount of calls to *process*().

The *process*() is independent of the TX coding and RX buffering. The interface between *process*() and TX/RX are two types of buffers at each communication port. For transmission a 9 byte buffer is provided onto which the PDU to be transmitted is written. In case of transmissions the *process*() writes data onto this buffer. For reception a 28 $UINT\_16$ buffer containing raw values to be decoded is available (equation (4.1)). In case of decoding the *process*() consumes data from this buffer. Both buffer types are allocated for each communication port.

$$\begin{aligned} buffer\_size &= \lceil buffering\_ratio * 2 * \mid PDU \mid_{max} \rceil \\ with & \\ buffering\_ratio &= 20\% \\ \mid PDU \mid_{max} &= 9 \end{aligned} \tag{4.1}$$

The FSM's states are sketched roughly in fig. 4.2. They are similar to the protocol stages proposed in fig. 3.2.

4.1.2. Reception and Decoding



The chosen post processing decode strategy has been investigated and discussed in 3.2.1. The process consists of a producer and consumer part as illustrated in fig. 4.3. A more exhaustive example is stated in the appendix fig. C.1.

Reception

The physical link layer reception is handled in the pin change ISR (PCI) which is basically capturing the occurred edge and storing to a ring buffer. The data

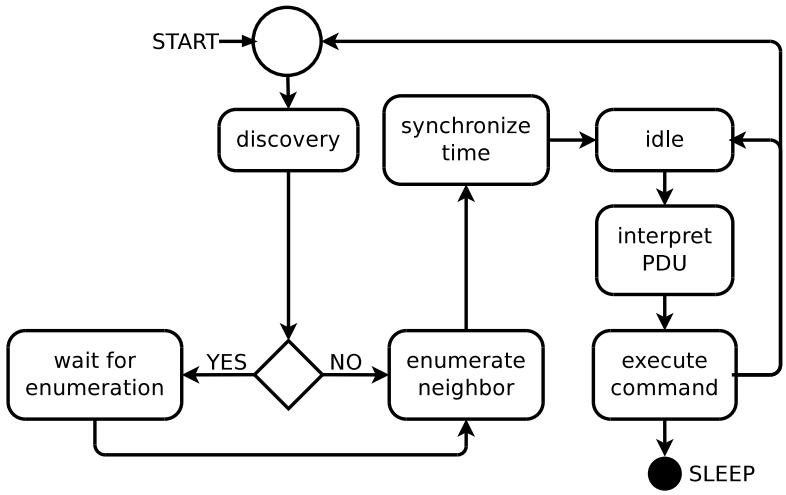


Fig. 4.2.: node's main states

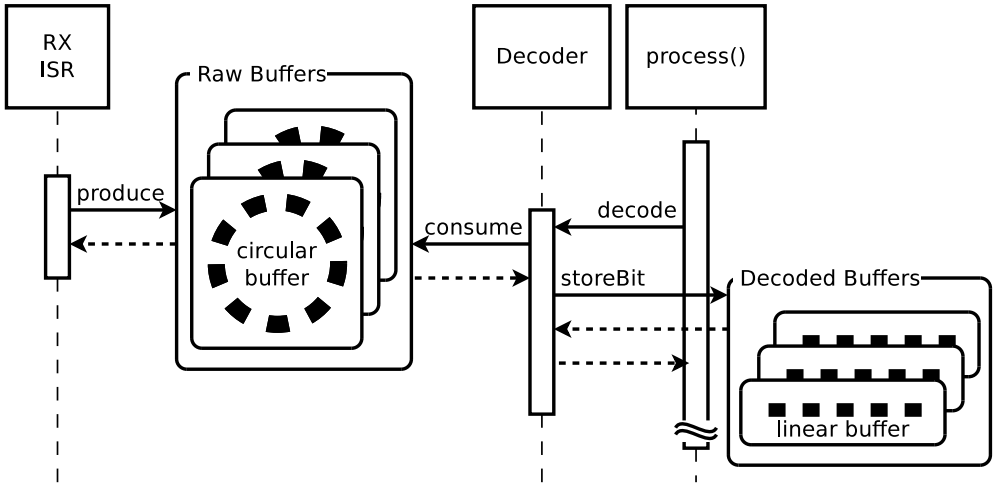


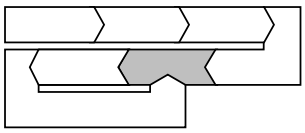
Fig. 4.3.: reception and decoding sequence diagram

produced is the 16 bit TCNT value when the ISR was triggered and the edge direction (rising/falling). For available SRAM memory reasons the LSB of the TCNT is sacrificed to store the edge direction, which results in a slight compression. To achieve more accuracy this feature can be turned off which results in storing each edge as a boolean. Thus occupies additionally one byte per edge of SRAM. The decoding is performed in the *process*() function.

Decoding

In reception states the *process*() function tests whether the RX ring buffers provide data for decoding. If yes, the data is consumed until the buffer is empty otherwise the decoder returns. This process occurs multiple times until a PDU is completely decoded. Decoding starts delayed to RX, runs partly parallel and partly beyond the PDU RX as illustrated in fig. 3.8. The post processing is investigated and proposed in section 3.2.1. This strategy needs an interruptible decoding implementation which requires a state driven approach. As already discussed in the Manchester code approach in section 3.2.1 applies a 1-bit counter for clock phase tracking. The clock phase is the binary FSM's state on which the decoded result is based on. In this producer/consumer implementation, although the producer (ISR) interrupts the consumer (*process*()), to avoid race conditions no additional locking is required.

4.1.3. Transmission and Coding



If data is to be transmitted, the *process*() function writes a PDU to the corresponding buffer's port. For transmitting with the introduced flow-control in section 3.2.5 each transmitting state implements it's own transmission handler. This is necessary to enable a per state flow-control implementation which allows to shortcut states if needed. For example the flow-control in the addressing phase differs from the flow-control in subsequent phases. The transmission handler puts the node into the correct sub-state (initiator or receptionist), enables the Manchester code generator and returns. To make this non-blocking behavior blocking, the handler is implemented state driven. It introduces sub states which are similar the introduced FSM's states illustrated in fig. 3.25 and fig. 3.26. An example is listed in the appendix, fig. D.1

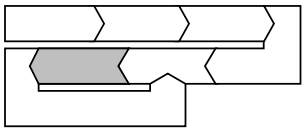
Manchester Code Signal Generator

When the physical link layer's transmission is enabled the, first generated signal, on TCNT compare match t_{cc}, is scheduled up to two t_{code} in the future starting from the current TCNT as formulated in equation (4.2). When the first TX ISR has triggered it generates the corresponding signal and schedules the next ISR in $t_{code}/2$ cycles (equation (4.3)).

$$t_{cc} = TCNT + 2 * t_{code} \tag{4.2}$$

$$t'_{cc} = t_{cc} + \frac{t_{code}}{2} \tag{4.3}$$

4.1.4. Interpreter



The interpreter is implemented stateless and very basic. It is called by the *process*() function and tests the decoded buffer if a PDU is interpretable. The buffer type is a C-Union consisting of all possible PDU types. This eases the data reinterpreting without an explicit C-Style cast. If a PDU is buffered the interpreter decides based on the CMD field which action is to be performed. If the PDU is associated to an executive function, it is called with the correctly casted PDU type. The invocation sequence diagram can be found in the appendix fig. C.1.

4.1.5. Scheduler

For additional extra features, which are not vital to the protocol, a simple scheduler is provided. The scheduler accepts tasks to be registered, which are executed when a set of rules is satisfied. Such a set may consist of constraints as the local time, required node type, task start/end time and cyclic task characteristic. The scheduler is called once each *process*() loop and tests each task's rules set. The advantage of the scheduler is code deduplication and performance gain, since without each task is forced to apply its own specific counter and implementation for triggering. The scheduler is absolutely necessary for the protocol evaluation as, for measuring characteristics of network synchronization and actuation, tasks preparing these experiments will be triggered by the origin node's scheduler.

```
...
addCyclicTask(TASK_ID, toggleHeartbeatLed, 250, 100);
taskEnableNodeTypeLimit(TASK_ID, NODE_TYPE_ORIGIN);
taskEnableCountLimit(TASK_ID, 60);
...
```

Fig. 4.4.: registration of light emitting diode (LED) blinking task 250 time units after boot with a separation of 100 time units and 60 total executions until task deactivation; applies to origin node only

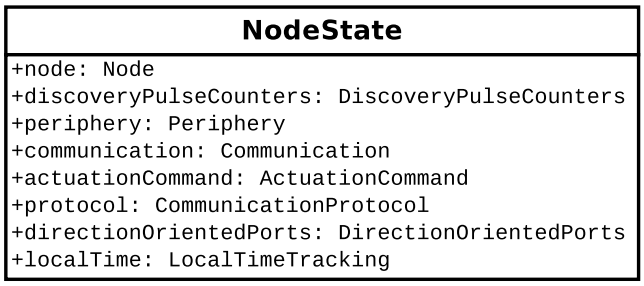


Fig. 4.5.: NodeState overview

For example, a 30 times executed cyclic light emitting diode (LED) blinking of the origin node is easily achieved by a cyclic task that is initially executed 250 time intervals after boot with a separation of 100 time units as illustrated in fig. 4.4.

4.1.6. Node Context

The node context including the FSM's states is packed into a global NodeState structure. Besides states it stores all information needed for all protocol layers such as buffering, scheduled data, local time et cetera.

Node Information about node connectivity, address in network and the node's FSM states. The complete FSM is illustrated appendix fig. F.1.

DiscoveryPulseCounters Discovery and port connectivity information.

Periphery Counters for non vital periphery control, such as LEDs, test points et cetera.

Communication Coding and decoding related buffers and adjustment parameters.
ActuationCommand Command scheduling and execution related parameters.
CommunicationProtocol Network layer and flow-control related parameters.
DirectionOrientedPorts Facade [20, pp. 212] of Communication, CommunicationProtocol and implementations bundled to direction aware communication ports (north port, east port and south port).
LocalTimeTracking Contains the current local time and adjustment parameters.

The complete NodeState's illustration is listed in appendix fig. E.1 - fig. E.4.

4.1.7. Synchronization

For reasons of protocol implementation and hardware restrictions, it is recommended to synchronize the network time and time clock speed periodically. For the same reason minimal measuring inaccuracy is to be expected when acquiring timings of TimePackage delays. This inaccuracy is expected to be non skewed normal distributed and fit the normal distribution model $\mathcal{N}(\mu, \sigma)$.

The synchronization mechanism introduced in section 3.2.5 gains accuracy when TimePackages are transmitted multiple times. Having access to various measurements provides a good base for several averaging methods. Thus we have implemented and evaluated the following approaches: Raw Observation Value (ROV), Simple Moving Average (SMAV), Weighted Movint Averate (WMA) and Moving Least Squares (MLS).

Basically the averaging extension requires just a simple protocol modification which bypasses observed d_{pdu} to a lightweight first in first out (FIFO) queue. According to the configuration, the queue delivers it's data to the corresponding averaging algorithm on each insertion. This means whenever a TimePackage is interpreted the data is bypassed throughout the FIFO to the averaging algorithm. The final outcome is then considered by the synchronization implementation that updates the timing dependencies as stated in equation (3.10) and equation (3.11).

The queue has a very lightweight implementation. It is iterable only when full. Thus it is necessary to pre-fill it with default expectation values which

reflect the current time clock speed. Otherwise the synchronization process is slightly delayed.

On cyclic network synchronization, the buffered values in the FIFO represent a moving window of measured observations. Besides smoothing this also provides the possibility of deeper data analysis such as trend, distribution, distribution skewness, median et cetera.

Additionally to the update methods, an outlier rejection feature can be combined with the some methods. The rejection filter can be activated to skip outliers before the mean is calculated. It does not affect the values captured and stored to the buffer. As outlier rejection strategy two options are provided. A normal distribution ($\mathcal{N}$) standard deviation (σ) based rejection, and an alternative implementation that constantly counts rejected and accepted values. With this counters it adjusts an acceptance window around the mean to fulfill a defined rejection percentage. The outlier rejection has been implemented due to the assumption that the buffer may not suffice to capture a statistically representative amount of observations.

Raw Observation Value The ROV is naive method which immediately updates the newly retrieved clock speed arguments. This method is used as baseline (equation (4.4)).

$$\overline{X} = x_0 \tag{4.4}$$

Simple Moving Average The SMAV approach takes each buffered value, considering eventually activated outlier rejection, and calculates the arithmetic mean (equation (4.5)).

$$\overline{X} = \frac{1}{n} \sum_{i=0}^{n-1} x_i \tag{4.5}$$

Weighted Moving Average A very simple implementation of the WMA algorithm with one buffered value. The weighted average of both, the current and newly observed value, represent the new current value (equation (4.6)) where $\overline{X}$ represents the history of previous values and x_0 is a new update.

$$\overline{X} = p * \overline{X} + (1 - p) * x_0 \tag{4.6}$$

Moving Least Squares The MLS method applies a least squares linear regression on the buffered values to obtain an averaged value. It also considers outlier rejection if configured. The regression fits a line through the given values trying to minimize the squared error $S(\beta_1, \beta_2)$ between the value and the fitted line. As values we use the timing observation made at each TimePackage (y_n) and the position in queue (x_n).

$$\begin{aligned} S(\beta_1, \beta_2) &= \sum_{i=0}^{n-1} (f(\beta_1, \beta_2, x_i) - y_i)^2 \\ with & \\ f(\beta_1, \beta_2, x_n) &= \beta_1 + \beta_2 * x_n \\ \beta_1 &= \overline{X} \end{aligned} \tag{4.7}$$

With MLS we have a tool that provides more than an averaged argument $\overline{X}$. Since it results in β_1 and β_2 of the linear function $f(\beta_1, \beta_2, x_n)$ one could infer a trend out of β_2 which describes the slope.

4.1.8. Optimization

Optimization can be tuned by several parameters but is always a speed-size tradeoff. We try to find a compromise between both, speed and size by using the inline keyword. Since the maximum program size is limited to the MCU's 16kB flash it is not possible to simply inline each function. Thus we are forced to select the most relevant functions to inline. To be more flexible we define an extend-able set of macro definitions to select the scopes to inline. Each function definition and it's declaration is prefixed with one of the following keywords FUNC_ATTRS, CTOR_ATTRS, DECODING_FUNC_ATTRS. The keyword values ("inline", "") are configured at compile-time in the framework's settings according to the use case needs. For example to define the scope of all decoder related functions we prepend the DECODING_FUNC_ATTRS

keyword to them. For speed optimization one may want to inline as many functions as possible as long the program fits onto the MCU's flash. In case of debugging it may be of benefit if all function calls are visible to the debugger. With a complete call trace, errors are easier to find.

4.1.9. Commands

The commands carried by PDUs are expressed by the CMD field. A detailed listing of CMD and payload values is presented in table 4.1.

AckPackage Acknowledge package for flow-control.

AckWithAddressPackage Acknowledge package for flow-control during addressing phase.

AnnounceNetworkGeometryPackage Automatic response containing the network geometry. It is replied onto the EnumerationPackage which has the network discovery breadcrumb flag (B) set.

EnumerationPackage Package containing the receptionist's address assignment. The B flag is set automatically in order to be forwarded to the right most, bottom most node only. The tail node receiving this flag replies an AnnounceNetworkGeometryPackage response.

ExtendedHeaderPackage This CMD is reserved. It allows protocol extension if further CMDs are necessary since the 4-bit width CMD is exhausted.

HeaderPackage Package transporting header flags to the neighbor. It is not relayed at all. A HeaderPackage is necessary for updating the BCT flag of all network nodes.

HeatWiresPackage The actuation command contains address, time and duration for scheduling an actuation. The package is routed to the corresponding ROW and COL. The command is executed in the next time interval matching t_{start} and $t_{duration}$. The L as well the R indicate the affected actuators.

HeatWiresRangePackage Similar to HeatWiresPackage except the addressing mode. The address is expressed as a rectangular range by the top left and lower right corner as ROW1, COL1 and ROW2, COL2.

HeatWiresModePackage Package for tuning the actuation power. The heating mode (M) can be set to maximum, strong, medium and weak as

listed in table 4.2. The values correspond 100%, 75%, 50% and 25% PWM duty cycle. The default is 50%.

ResetPackage Package initiating an immediate node reset.

RelayHeaderPackage Header which is relayed to east port and south port subsequently. A RelayHeaderPackage is necessary for updating the BCT flag of all network nodes.

SetNetworkGeometryPackage Package stating a new network geometry. It is relayed to the east port and east port. Nodes outside the new geometry switch to sleep mode. The new geometry network rows (ROWS) and network columns (COLS) must be within the bounds as reported by AnnounceNetworkGeometryPackage.

SyncNetworkTimeHeaderPackage This package is issued only by the master device to the origin node. The purpose is to trigger a network time synchronization.

TimePackage Time synchronization package contains the t_{tx}, t_{sep_tx}, t_{until_cc} and the flags FU and EB. The arguments are necessary for the receptionist to convert delays from the transmitter's to the local time computation, where t_{tx} is the transmitter's local time when the PDU is constructed, t_{sep_tx} is the current transmitter's time unit counting separation in cycles and t_{until_cc} is the number of clocks until the time is incremented in cycles when the PDU is constructed. The FU indicates the receiver to update the local time to t_{tx} with respect of transmission latency. The constant EB is used for PDU RX timing measuring. This package is issued by the origin node.

$$\begin{aligned} f_{actuator} &= \frac{f_{cpu}}{(2 * UINT8\_MAX * prescaler)} \\ with & \\ prescaler &= 64 \end{aligned} \tag{4.8}$$

4.1.10. Configuration

The source code structure allows configuration of several implementation parts which makes the firmware highly configurable. The configuration folder

update the config files/parameters according to last software updates

Command	CMD	Parameters	Figure
HeaderPackage	$0x01$	HDR, CMD	fig. B.7
RelayHeaderPackage	$0x03$	HDR, CMD	fig. B.8
ResetPackage	$0x04$	HDR, CMD	fig. B.9
AckPackage	$0x05$	HDR, CMD	fig. B.10
AckWithAddress-Package	$0x06$	HDR, CMD, ROW, COL	fig. B.11
AnnounceNetwork-GeometryPackage	$0x07$	HDR, CMD, ROWS, COLS	fig. B.12
SetNetworkGeometry-Package	$0x08$	HDR, CMD, ROW, COL	fig. B.13
EnumerationPackage	$0x09$	HDR, CMD, ROW, COL, B	fig. B.14
TimePackage	$0x10$	HDR, CMD, t_{tx}, t_{sep_tx}, t_{until_cc}, FU, EB	fig. B.15
HeatWiresPackage	$0x11$	HDR, CMD, ROW, COL, t_{start}, $t_{duration}$, L, R	fig. B.16
HeatWiresRange-Package	$0x11$	HDR, CMD, ROW1, ROW2, COL1, COL2, t_{start}, $t_{duration}$, L, R	fig. B.17
HeatWiresMode-Package	$0x12$	HDR, CMD, M	fig. B.18
ExtendedHeader-Package (reserved)	$0x15$	HDR, CMD	fig. B.19
SyncNetwork-TimeHeaderPackage	$0x02$	HDR, CMD, t_{tx}	fig. B.20

Table 4.1.: command id (CMD) listing and their corresponding parameters

M	Duty Cycle [%]	Frequency [Hz]	Power
$0x00$	100%	0	maximum
$0x01$	75%	≈ 244.1	strong
$0x02$	50%	≈ 244.1	medium
$0x03$	25%	≈ 244.1	weak

Table 4.2.: heating mode heating mode (M) listing, MCU clock frequency (f_{cpu})= $8MHz$, actuator frequency ($f_{actuator}$) is formulated in equation (4.8)

contains header files of each implementation part as shown in fig. G.1. The provided parameters can be split in two parts, protocol and hardware related. This section describes the protocol related configuration only. For the hardware configuration, which mainly sets up the wiring and internal MCU configuration registers the source code must be looked up. A complete listing of configurable parameters as well the MCU pinout are listed in the appendix from table G.1 and table G.2.

Actuation.h, ActuationTimer.h The actuation parameters define the duty cycle of the actuator wires. The levels weak, medium and strong are adjustable whereas the maximum level which corresponds to 100% cannot be changed. The higher the value the higher the duty cycle ($UINT8\_MAX \triangleq 100\%$, $0 \triangleq 0\%$). The PWM mode is phase correct [14, pp. 81] with TCNT prescaler 64 and can not be changed. The frequency is formulated in equation (4.8).

Communication.h The clock adjustment for the Manchester code coding and decoding can be tuned with this parameters. The values are based on the underlying TCNT and its setup. For transmission this means the clock cycle duration.
For the reception, the separation of subsequent signal edges must be classified into three groups: short separation ($t_{code}/2$), long separation (t_{code}) and timeout. The default settings adjust the TX/RX clock period to $2 * 1024$ MCU clocks, thus the $f_{xmission}$= $3906.25Hz$. Separations below 75% of the TX/RX clock cycle delay ($d_{xmission}$) are classified as short, below 125% as long or as timeout otherwise. The communication protocol settings define flow-control timeout and introduced PDU transfer latency which is used to approximate t_{now} during time synchronization.

The flow-control timeout is implemented as a $process()$ loop counter which is decremented until zero. On $counter = 0$ a timeout is detected.

Discovery.h, DiscoveryTimer.h, Particle.h The discovery phase's pulse generation and minimum received pulses can be tuned in this file. If more pulses than the defined discovery pulse counter are registered, the corresponding ports are marked as connected. If a node is found to be connected within minimum/maximum neighbors discovery pulse loops the discovery phase switches to pulsing only (turns RX off). If the maximum neighbors pulsing loops is reached the discovery is finished. The counter compare value parameter can be tuned to change the PWM frequency. The PWM is in clear timer on compare match mode (CTC) [14, pp. 78] with TCNT prescaler 8 and cannot be changed. The discovery signal frequency ($f_{discovery}$) can be calculated as in equation (4.9). Different $process()$ loop separation delay can be tuned for the phase until a node's connection is determined and the phase until the maximum pulses (post discovery pulsing) are reached.

$$\begin{aligned} f_{discovery} &\approx \frac{f_{cpu}}{8 * 0x80 * 2} \\ &\approx 3906.25Hz \end{aligned} \tag{4.9}$$

interrupts/ The settings defined in this folder are mostly related to the pinout and MCU configuration register. The hardware configuration is not covered in this document.

Periphery.h This file is meant for non vital hardware extensions that do not affect the protocol such as LEDs or test points (TPs). The heartbeat signal frequency (flashing LED) may be adjusted here.

Time.h For local time tracking the cyclical ISR separation can be adjusted. The detailed reasoning for the chosen value can be read in section 3.2.5.

4.1.11. Simulation

For simulation a tool that is capable of simulating a whole network of nodes is needed. It should provide the possibility of monitoring, logging and synchronous execution of node's firmware. From the evaluated frameworks the top two simulators are SimulAVR and Avrora[1]. The benefit of SimulAVR is the

[1] http://compilers.cs.ucla.edu/avrora

debugging capability in combination with GDB debugger and DDD graphical user interface. For a network simulation SimulAVR is not that convenient without workarounds. It does also not provide an adequate MCU, such as an ATtiny for our needs. The second candidate's documentation and features looked very promising. Avrora provides many features for instrumentation out of the box. The Avrora simulator covers all our listed needs and is flexible in regards of extensibility. The Avrora simulation framework is perfect for scale-able sensor network simulation. Unfortunately Avrora cannot be used in combination with GDB when simulating networks, but this circumstance becomes irrelevant with the features the framework provides. Although the lastest Avrora release is older than SimulAVR's we have opted to use the Avrora simulation framework.

Avrora Simulation Framework

The Avrora framework is a non-intrusive simulation framework [21] of the UCLA Compilers Group[2] written in Java. for experimentation, profiling and analysis. It allows target code to be written without the need of editing for instrumentation. In other words the compiled code to be run on a physical MCU can be used directly for simulation. The fundamental instrumentation approach is based on probes, watches and events. Simulation mechanisms such as monitoring are build on top of this instrumentation points. This allows the framework to be very flexible in regards of the provided features, since they can be easily extended. The simulation output is very detailed which makes it possible to reuse the data in many other tools such as automated testing framework, network visualization or even other simulations like force or friction simulations. The produced data is a very promising base for future work, since the work flow allows to simulate the whole process starting from the original firmware.

Synchronization The framework simulates each node in a separate thread. This introduces the need of a special synchronization mechanism that ensures

[2] http://compilers.cs.ucla.edu

the nodes having the same progress. This is vital for sensor networks, especially since communication relays on timing. Avrora considers two timing strategies [22].

Syncrhonization Intervals Each node's execution is divided into intervals. Each node runs until the end of it's interval and waits until all other nodes reach the same point of simulation, the interval end. For a cycle-by-cycle synchronization the interval length corresponds to 1.

Wait for neighbors This approach uses a sliding window strategy which assures all nodes are within a specified time interval. If a node's progress is beyond the specified window it is blocked until all nodes are again within the window interval.

The Avrora framework set up for protocol simulation synchronizes nodes in intervals of 4 MCU cycles. Good experimental values are within $[2, 32]$. This values are be justified with the approximative amount of instructions the simulation is allowed to drift. In this application the unwanted introduced jitter can be used positively to test the protocol robustness. A limitation of the framework is a missing clock drift model. The simulation framework's most important benefit is the possibility to abstract not only the MCU but also whole printed board circuits (PCBs). With connected nodes there is no need for synthetic signal generation. Otherwise for simulation and testing the MCU must be fed with manually generated samples which is to be avoided. In total this means a fully connected network of communicating nodes can be simulated with only one firmware and without externally applied signals.

Avrora Platform Extension To simulate a network of nodes we implemented a particle platform which abstracts the physical link layer as illustrated in fig. 3.4 and some LEDs. A MOSFET abstraction is not provided but could be easily implemented and adopted to the framework. The particle platform is implemented similar to the physical PCB's schematic diagram as shown in section H. For networking it provides two wires (TX/RX) per port (north port, east port and south port) which are connected respectively by a network builder before the simulation starts. The network builder is able to construct a $(rows \times cols)$ network with nodes having the same firmware and optionally appending one extra node to the netwok's origin node as master device having a different firmware as illustrated in fig. 3.1. With this system communication

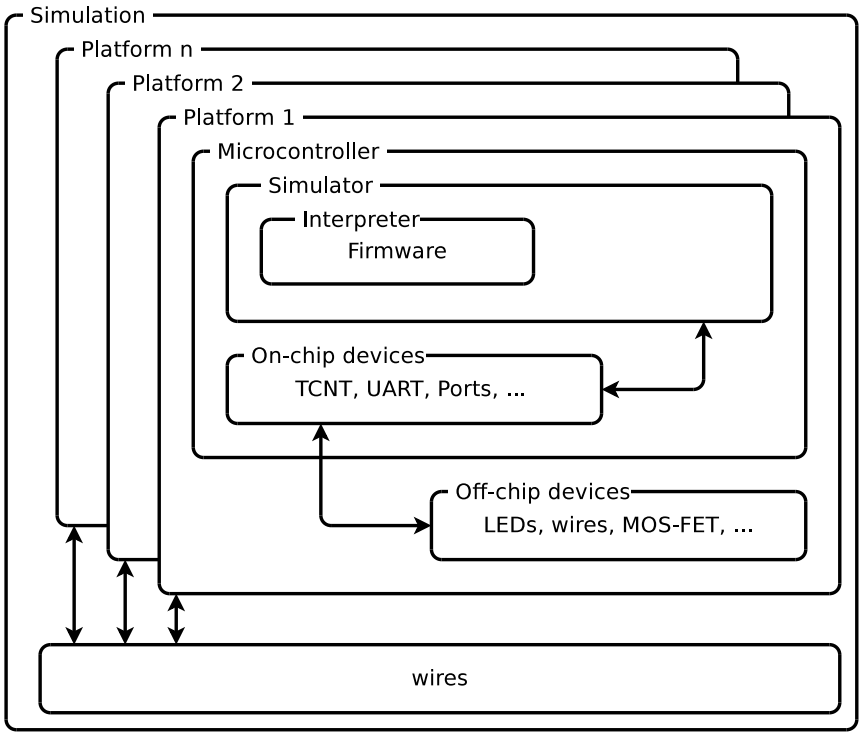


Fig. 4.6.: Avrora's software structure

signals generated by a specific MCU are propagated as follows: TX MCU pin → wire → MOSFET → wire connecting platforms → wire → MOSFET → RX MCU pin.

Avrora Monitor Extension The simulation framework provides many analysis monitors for example function calls, interrupts, memory profiling and much more. For the our specific needs we implemented a monitor that is exactly tailored for the node's platform - the particle monitor. Among others it monitors communication wires and the MCU's internal SRAM and translates events into readable logs. For example the default memory monitor reports in line 5 a write of $0x60$ onto the SRAM address $0x6a$ of platform 0 as $SRAM[6a] \leftarrow (60)$ whereas the extended particle monitor reports $SRAM[Particle.\ discoveryPulseCounters.\ loopCount] \leftarrow (96)$ as illustrated in fig. 4.8.

The particle monitor configurable to interpret changes of any SRAM address as one or multi byte width data and present it as signed/unsigned, hexadecimal, decimal, binary, character, float or double. On multi byte data the monitor watches any byte change and re-interprets the new value with respect to the current write offset. The platform number to address mapping and vice versa are formulated in equation (4.10) and equation (4.11).

The example configuration in fig. 4.7 defines the *loopCount* member address and data type and also the MCU port A pins' human readable names. With this feature we can watch the node's internal global state. This approach is limited to global variables only with a constant address, since local variables' addresses must be found out dynamically at simulation run time.

The complete configuration file length is ≈ 2500 lines of code, which makes it very inconvenient to maintain. If a small change is done in source, for instance one member is removed, the whole file must be re-edited. For that reason an auto generator was implemented in Python that creates a the Json configuration out of the C source code. For more details about the generator consult the development repository[3]. With the framework's built-in monitors and especially the tailored particle monitor there is no need for a debugger.

[3] https://github.com/ProgrammableMatter/cstruct-to-json

```
...
"Particle.discoveryPulseCounters": [
  {
    "property": "loopCount",
    "type"    : "unsigned",
    "address" : "globalStateBase+10"
  },
  ],
  "A.out": [
    {
      "property": "(EAST_TX | EAST_SW | TP3 | PA4 | \
                   SOUTH_TX | SOUTH_SW | TP2 | ERROR)",
      "type"    : "bit",
      "address" : 59
    },
],
...
```

Fig. 4.7.: particle monitor example configuration snippet

The gathered uniform monitor dump is easily parsed with simple regular expressions and used for automatic JUnit testing.

$$having\ (M \times N)\ network \quad | \quad id \in \mathbb{N},\ m,n \in \mathbb{N}_+,\ m \leq M,\ n \leq N$$

$$\begin{aligned} idToAddress(id) &\longmapsto (m,n) \\ idToAddress(id) &= ((id\%N)+1), \left\lfloor \frac{id}{N} \right\rfloor + 1) \end{aligned} \tag{4.10}$$

$$\begin{aligned} addressToId(m,n) &\longmapsto id \\ addressToId(m,n) &= (n-1) * M + m \end{aligned} \tag{4.11}$$

Extensions The particle related Avrora extensions[4] are not meant to be added to the simulation framework, but instead the framework to be added to the extension's Java project. To register the extended parts (particle platform,

[4] https://github.com/ProgrammableMatter/avrora-particle-platform

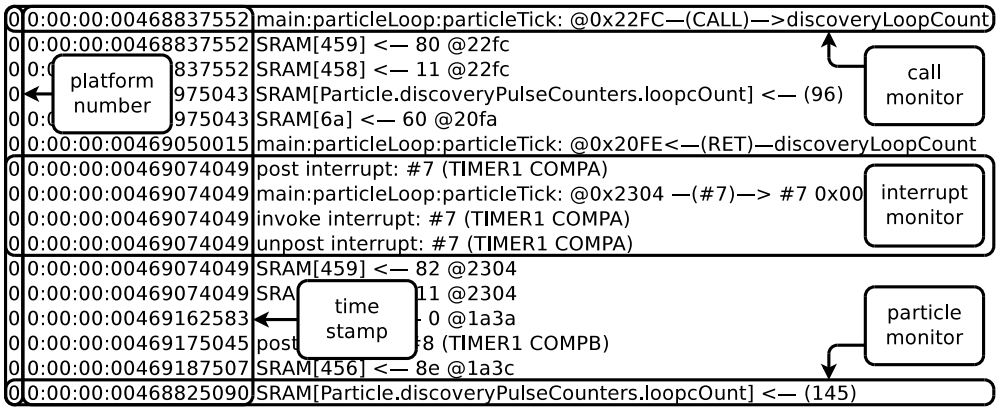


Fig. 4.8.: Avrora simulation trace of several monitors

particle monitor and the ParticleSimulation class) the Avrora framework's registration methods have been slightly refined. The registration can be simply achieved from outside the framework as illustrated in fig. 4.9. Hence other projects implementing different platforms/extensions are not forced to touch the simulation framework any more.

Due to the missing build manager the file structure has been modified slightly and configured for Maven. The modified framework[5] is available at the Open Source Sonatype Repository Hosting (OSSRH)[6] as Maven package. With this modification the framework can be started on command line with the additionally registered extensions. For unit testing purposes the registration must be done similarly before a simulation starts.

Testing

During development the continuous testability is an important criterion and has been always prioritized. The applied simulation framework produces enough information for JUnit testing. This removes the need of breakpoint debugging such as with GDB completely. It also speeds up the development

[5] https://github.com/avrora-framework

[6] https://oss.sonatype.org/#nexus-search;quick~avrora-framework

```
public class Main {
  public static void main(String[] args) {
    Defaults.addPlatform
      ("particle-platform", ParticlePlatform.Factory.class);
    Defaults.addSimulation
      ("particle-simulation", ParticleSimulation.class);
    Defaults.addMonitor
      ("particle-monitor", ParticlePlatformMonitor.class);
    edu.ucla.cs.compilers.avrora.avrora.Main.main(args);
  }
}
```

Fig. 4.9.: Avrora extension registration

process and minimizes the time for finding implementation errors. The work flow is very simple:

1. write JUnit test case
2. implement feature in firmware
3. run JUnit test which does automatically
   a) start the simulation
   b) capture the output
   c) evaluate the output
4. refine JUnit test case
5. refine the firmware and go to step 3 unless the test succeeds

Limitations are the simulated time and produced output. A (6×6) network simulation (36 nodes) can be quite exhaustive in terms of duration and memory occupied by the log output. The evaluation of the 36 nodes' output slows down the testing process significantly. To improve this tests must be parallelized and are run through even if assertions fail. Test case parallelizing speeds the evaluation process up by a factor of 3. The implementation inspects the log output for each test case in parallel using Java streams. The inspection does not assert but saves the result for later assertion. The real JUnit tests just evaluate the stored result after inspection to ensure that all results can be run through.

As simulation synchronization interval multiple setups have been evaluated. Synchronization intervals of 4 up to 8 MCU clocks (setup B and C in table 4.3)

set-up	synchronization interval $[cycles]$	simulation dur. $[s]$	inspection dur. $[s]$	total dur. $[s]$	CPU freq. $[MHz]$
A	2	≈ 238	≈ 98	≈ 336	8.0
B	4	≈ 176	≈ 91	≈ 267	8.0
C	8	≈ 136	≈ 93	≈ 229	8.0
D	12	≈ 132	≈ 98	≈ 230	8.0
E	16	≈ 128	≈ 91	≈ 219	8.0
F	20	≈ 127	≈ 94	≈ 221	8.0
G	32	≈ 123	≈ 92	≈ 215	8.0

Table 4.3.: duration vs. synchronization of a (6×6) network simulation - simulating $150ms$ and several synchronization interval arguments

are sufficient accurate for the line coding/de-coding. For example a 0.15 second simulation of 36 nodes takes ≈ 176 seconds followed by ≈ 94 seconds for log inspection (total 4.5 minutes) and produces $\approx 410MB$ logs (setup B in table 4.3). The JUnit tests are found in the default Maven folder *src/test/java* of the particle platform implementation[4].

Visualization

Because simulation output can be very exhaustive, having tens up to hundreds of megabytes, it is very time consuming to follow what is happening. Therefore a network visualization tool[7] has been implemented in Python. The tool visualizes wire signals, ISR actions or changes of SRAM registers on a time line. The visualization takes a user defined source (i.e. north port, ISR, TX wires or an arbitrary register address) and the simulation output as input. It filters all events of the specified sources from the input and renders the visualization. On the visualization chart each event is marked with a clickable bullet. On mouse click a label showing the event's value appears and the current time value is copied to clipboard. The visualization is straight forward for integer values such as digital wire signals, *uint*8_*t* or *uint*16_*t*. In case of

[7] https://github.com/ProgrammableMatter/network-visualization

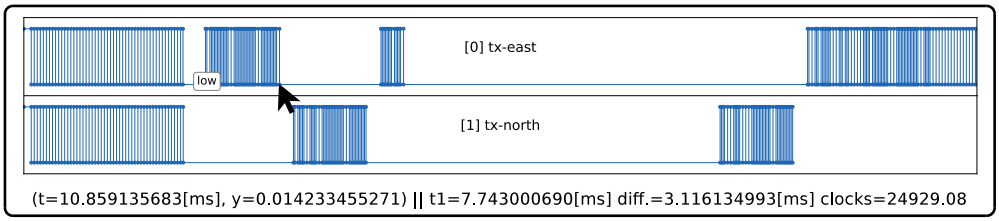


Fig. 4.10.: simulated (1×2) network visualization - communication signals of two neighbored particles showing a highlighted label and detailed information at the bottom of the chart

char and *enum* a user defined mapping can be applied to translate this events to integer. The mapping also allows to define a human readable string that is shown on mouse click at the respective position in chart. If no mapping is defined the tool toggles events between 0 and 1. The visualization tool provides more details in the status bar as illustrated in fig. 4.10. Besides showing a label on mouse click, the tool also indicates the

event time t in $[ms]$ when it occurred, the
numeric value y, the
crrent mouse position's time $t1$ in $[ms]$, the time
difference between both positions in $[ms]$ and the amount of
MCU cycles passed between the positions.

The chart in fig. 4.10 shows two time lines marked as $[0]tx\text{-}east$ and $[1]tx\text{-}north$ which represent the transmission wire signals of node 0 and 1. The time lines represent the discovery phase followed by the addressing phase and at the end the time synchronization phase. It labels the first PDU transition's event as "low" that occurred at $t \approx 10.9[ms]$, shows the difference until the mouse pointer which in this case is $diff. \approx 3.1[ms]$. The time span corresponds to $\approx 25k$ MCU cycles. In other words the PDU TX started at millisecond 10.9 past simulation start and took 3.1 milliseconds. The presented chart is a very basic visualization example. A more detailed visualization of an (3×3) network showing all phases until the time synchronization can be found in the appendix in fig. I.1

PDU	Purpose
HeatWiresPackage	actuation command
HeatWiresRangePackage	actuation command
HeatWiresModePackage	actuation command setup
ResetPackage	reboot network
RelayHeaderPackage	routed broadcast package to enable BCT
SyncNetworkTimeHeaderPackage	tell origin node to re-synchronize
ResetPackage	reboot network
SetNetworkGeometryPackage	re-define a new network geometry

Table 4.4.: protocol data units (PDUs) for master device to origin node communication

Network Use Case

The described network structure in fig. 3.1 of section 3 introduces a master device coordinating network activities. This master device is not required during the initialization phase and must not generate discovery signals, otherwise the network initialization will definitely fail. It seems natural to use the same but slightly enhanced protocol firmware and a particle node hardware as master device. This is possible as long the use case is realizeable with the available MCU resources. The only restriction is the limited set of PDUs allowed to be issued to the origin node $(1, 1)$ as listed in table 4.4. The origin node does not filter PDUs, thus any PDU will be captured, interpreted and executed which, for development and testing, may be helpful.

A simple use case: boot the network, wait until enumeration phase has finished, issue several SyncNetworkTimeHeaderPackages followed by HeatWiresRangePackages or HeatWiresRangePackages.

Build Environment

To speed up the deployment process, provide code analysis tools we configured a tool chain as illustrated in fig. 4.11 using CMake, a cross-platform build tool. To sustain a test driven development we decided to build multiple

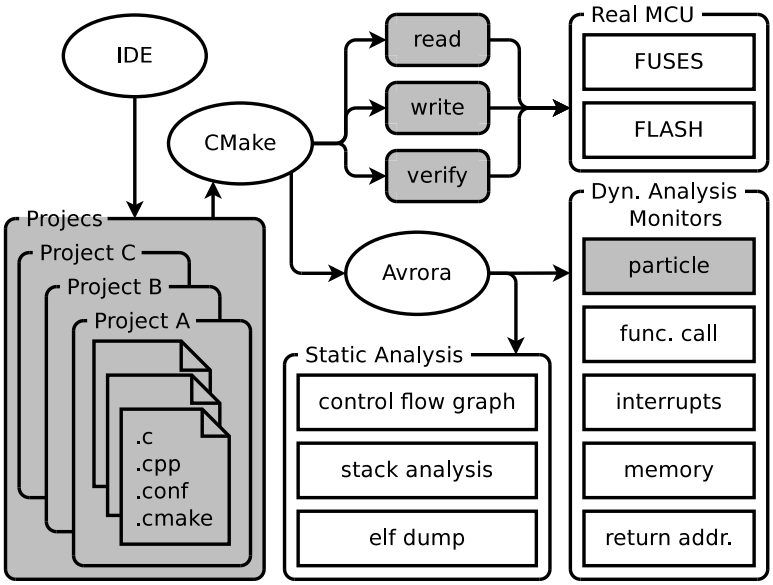


Fig. 4.11.: development tool chain

executables for each test case. Thus any project is set up to produce exactly one executable which, in case of test-able firmware, is configured to be in a predefined state according to the test-case needs.

The implementation's core parts are chosen not to be compiled as libraries since each project's target MCU can be configured separately. Thus the core implementation used by a certain project is linked the to the project folder. This reduces the need to compile each library for any target MCU used by projects.

A project allows to configure following parts:

Programmer Configuration of an Avrdude supported programmer to be used when deploying to the real MCU. This part also adds custom rules which stripes the corresponding sections (.text, .data, .bss, .fuses, et cetera) from the executable to read/write/verify the MCU's flash and fuses.

MCU Allows configuration of the Avr-gcc supported MCUs (-mmcu flag).

Compiler Configuration of many avr-gcc flags and optimization settings in detail.

Custom Make Rules Different predefined Make rules may be extended, i.e. Avrora related rules can be added by linking a project folder to the the rule folder.

Debugger Simple RS-232 debugging configuration such as baud rate and device.

Since each project provides it's own set of Make rules they are prefixed with the respective "ProjectName_". The global Make rules "all", "clean" and "help" are not prefixed. In the overview listing of table 4.5 we skip the prefix for simplicity. The Make rules are not designed to support JUnit testing, but rather allow to start a simulation process for dynamic analysis.

The highlighted parts in fig. 4.11 correspond to the thesis's practical development. For more details on how the project directory is structured one may consider consulting the source code repository[8].

[8] https://github.com/ProgrammableMatter/particle-firmware

Rule	Purpose
all	build all executables
clean	delete object files
help	lists all make rules
.elf	compile the executable
flash	writes executable to MCU
erase	erases MCU's flash
verify	compare executable with MCU flash
fuse	write all fuses
rfuse	read all fuses
rhfuse	read high fuse
rlfuse	read low fuse
refuse	read extended fuse
avrora-simulate	start simulation
avrora-elf-dump	print elf dump
avrora-inter-procedural-side-effect-analysis	invokes the inter-procedural side-effect analysis tool
avrora-analyze-stack	stack analysis tool to determine worst-case stack depth
avrora-cfg	shows the control flow graph
avr-cycles	shows the object dump decorated with MCU cycles per instruction

Table 4.5.: Make rules listing of non prefixed rules (first block) and project dependent rules (subsequent blocks)

5. Evaluation

The evaluation focuses on protocol timings, behavior and other findings observed during the evaluation process. The software development was largely sustained by the Avrora simulator, thus measurements obtained by simulation are compared to measurements of real hardware. Unfortunately some parts are difficult to evaluate in hardware or simulation, hence this comparison cannot be made of each and any evaluation.

Simulated results are performed with the simulation setup **C** as stated in table 4.3. Hardware evaluations are performed on a network having (12×1) geometry, supplied with a voltage regulated power supply at VCC$= 5.1V$ connected at node $(1, 1)$ unless specified differently.

PDU	**Len.** $[bits]$	**Min.** $[\mu s]$	**Max.** $[\mu s]$	**Avg.** $[\mu s]$
AckPackage	8	277	294	**285**
AckWithAddressPackage	24	305	323	**314**
EnumerationPackage	25	295	309	**302**
TimePackage I	56	286	314	**298**
TimePackage II	56	294	331	**308**

Table 5.1.: protocol data unit (PDU) length vs. simulated decoder's post processing delay

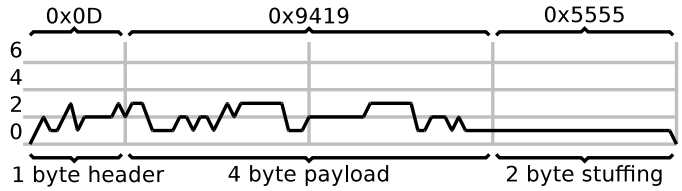


Fig. 5.1.: simulated buffer size versus protocol data unit (PDU) length - TimePackage I

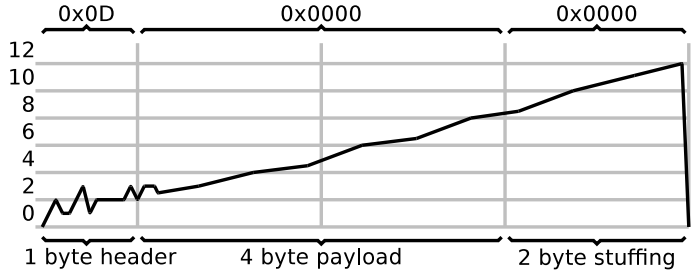


Fig. 5.2.: simulated buffer size versus protocol data unit (PDU) length - TimePackage II

5.1. Memory Consumption

5.1.1. Manchester Decoding

Simulated Evaluation

Contrary to our expectations the evaluation of the post processing strategy (discussed in section 3.2.1) shows no relevant increasing decoding delay (illustrated in fig. 3.8) nor relevant increasing of the decoder's buffer consumption for different $\mid PDU \mid$. Table 5.1 shows a summary of measured delays between RX PDUs and the decoding process' end. We observe a rather constant average post processing delay regardless of the $\mid PDU \mid$. This is based on the Manchester code's nature and the varying baud rate which is dependent of the encoded data. The real data carried by these PDUs results in more or less random bits to be encoded. A lower baud rate leaves more space in between RX ISRs for decoding which results in a faster and less buffer consuming decoding.

For evaluating the decoder's buffer worst-case consumption we compare

a typical and a prepared TimePackage, which with $\mid PDU \mid = 7$ byte is a rather long PDU. In fig. 5.1 we present the decoder's buffer consumption of the typical TimePackage. The PDU has kind of random bits in the first 5 bytes followed by $0x5555$, which in binary representation are alternating bits $0b101010$.... This results in the coded data having a fluctuating baud rate for the first 5 byte but a lower and constant for the last 2 byte. The figure shows the correlation between the described coding frequency and the buffer consumption. In contrast, the prepared TimePackage causes a linear increasing buffer consumption as illustrated in fig. 5.2. This is due to the manually forced higher line code frequency for the bytes carrying the bits $0b000000$.... The buffer consumption evaluation shows that it is strongly dependent on the data carried by the PDU, whereas assumptions based only on the $\mid PDU \mid$ are weak. In the worst-case we observe the decoder buffer consumption is increasing by $\approx 10/6$ byte per PDU byte. With this we are able to state that a buffer size for only $\approx 20\%$ of the maximum events that may occur in a PDU having $\mid PDU \mid = 9$ byte is more than enough (equation (4.1)). Furthermore we see that the decoder's post process delay is correlated to the buffer consumption which in combination with fig. 5.1 explains the rather low average post process delay of the TimePackage I in table 5.1.

5.2. Timing Evaluation

5.2.1. Discovery

In general the discovery duration depends on two aspects: the number of calls to *process*() and the ISR load. The limit of calls to *process*() reflects the discovery timeout. On timeout the discovery phase is forced to terminate. The duration of *process*() is influenced by the ISR load. To reduce the ISR load, the discovery pulse period is configured to be the same as the default Manchester code's clock period as stated in equation (4.9). We observe a longer *process*() delay at nodes having a higher connectivity degree, except one special case: discovery of fully connected nodes has a shorter delay. This is due all neighbors are discovered before the discovery timeout occurs.

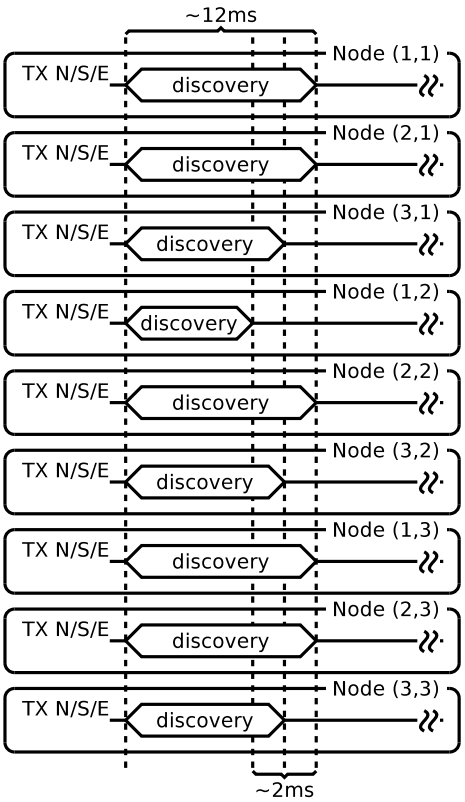


Fig. 5.3.: simulated (3×3) discovery phase - discovery duration differs according to node's connectivity

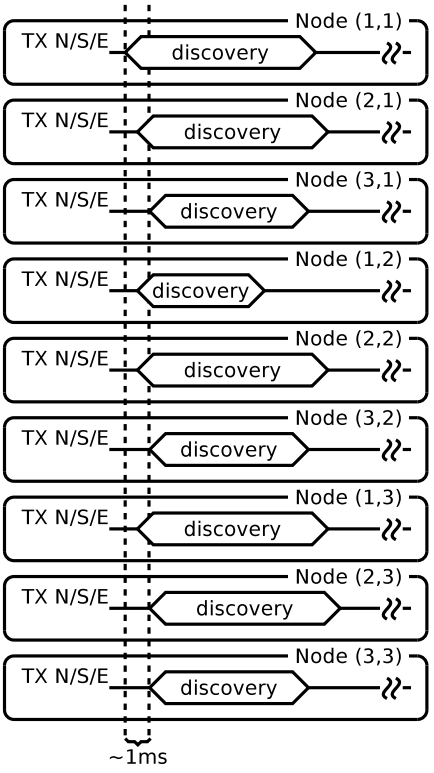


Fig. 5.4.: measured (3×3) discovery phase - introduced supply voltage (VCC) drop causes discovery shifts

Simulation Evaluation

In the simulated (3×3) network example of fig. 5.3 we see a fully connected node $(1,2)$ (inter head) finishing the discovery at first, followed by the least connected nodes $(3,1)$ $(3,2)$ and $(3,3)$ (tail nodes). The longest duration we observe at nodes having two connections: $(1,1)$, $(2,1)$, $(2,2)$, $(1,3)$ and $(2,3)$ (inter nodes). The discovery pulsing takes $\approx 12ms$.

Hardware Evaluation

In our experiments we see varying boot delays among network nodes. This happens due to the falling VCC of the power supply from the least to the most distant network node. In the evaluated (3×3) network example we see a Voltage drop of $\approx 100mV$ at node $(3,3)$ without evidence of ripple. This affects the discovery phase's start, which takes place within an interval of $\approx 1ms$ as illustrated in fig. 5.4. The interval is expected to be higher the higher the VCC voltage drop among particles. Fortunately the shuffled discovery start does not cause any protocol errors when executed on hardware.

The evaluation shows that the internal clock's RC circuit is very sensitive to the supplied VCC. Except of the slightly shuffled discovery start we see no relevant discrepancy between simulation and hardware evaluation. Although the simulation is able to simulate a random start, we did not use this feature extensively to reduce the simulation time.

5.2.2. Addressing

Enumeration The addressing follows the flow-control as illustrated in fig. 3.24 in section 3.2.5. The enumerator sends a new address, the receiver acknowledges the address value by sending the same back and waits for ACK from the enumerator. If the enumerator receives the correct address as transmitted before the transmission was received without errors. The enumerator acknowledges the transaction, otherwise it falls back and re-transmits the enumeration PDU. This flow-control is necessary to prevent mistakenly interpreting discovery signals as communication attempt. Such cases may easily occur due

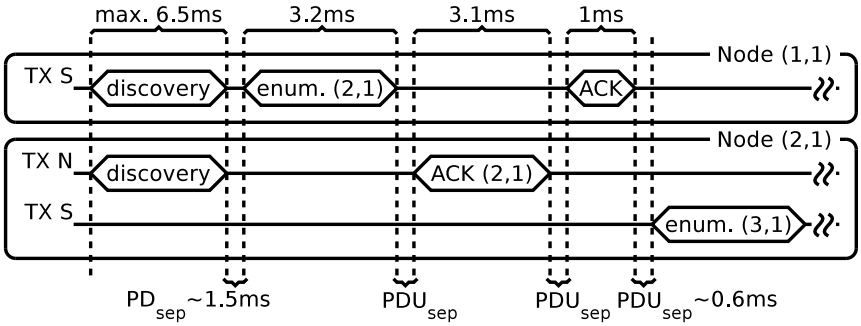


Fig. 5.5.: simulated (3×1) network enumeration of node $(2,1)$ showing PDU transmission duration (d_{pdu}) of several protocol data units (PDUs)

to VCC voltage drops as described in section 5.2.1. Until the communication initiator does not receive an ACK the addressing transaction is not finished, unless the transaction is interrupted by a timeout.

Simulation Evaluation The simulation output of a (3×1) network, illustrated in fig. 5.5, shows a simple enumeration example. Since the simulation suffers of minimal jitter the presented PDU TX delays are the result of averaged measurements. The separation between a PDU is received and the corresponding response (PDU_{sep}) depends on the line code decoding which is partially post processed. The post processing duration again depends on the number of remaining data to be processed when the currently received PDU is completely received. In the illustrated example of fig. 5.5 EnumerationPackage, AckWithAddressPackage and AckPackage having 25, 24 and 8 bits, the $\mid PDU \mid$ differ relevantly but the introduced PDU_{sep} delay having $\approx 0.6ms$ does not, which confirms the explanation in section 5.1.1.

Hardware Evaluation The hardware evaluation of the same (3×1) network geometry at a specific MCU shows slightly different results. The TX delay of EnumerationPackage, AckWithAddressPackage, AckPackage differ for instance by $\pm 3\%$. If we consider the current transmitting MCU's f_{cpu} deviation from $8.0MHz$, we obtain the same percentage of deviance. Thus we can state that both evaluations, hardware and simulated, correlate well.

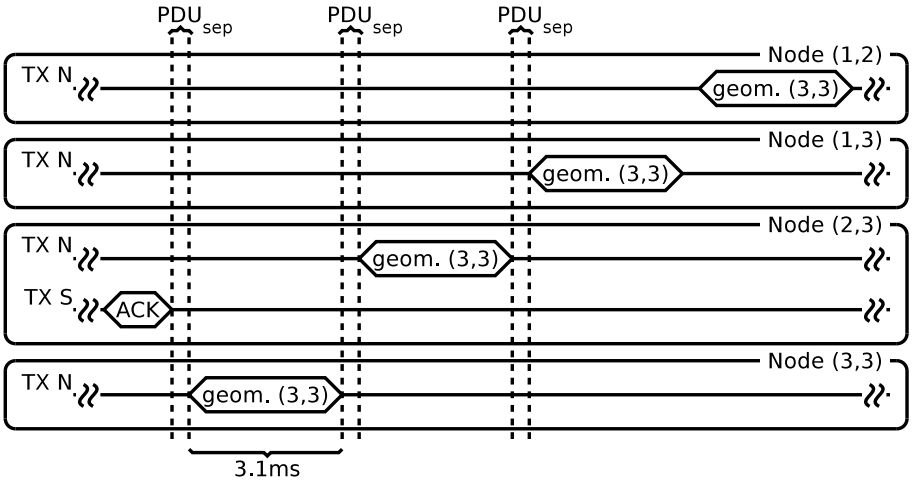


Fig. 5.6.: simulated (3×3) network geometry disclosure of node $(3,3)$ showing AnnounceNetworkGeometryPackage's PDU transmission duration (d_{pdu})

Apart of the package timings we were forced to increase the discovery phase to 150% of the duration used in simulation, as well as the post discovery separation between discovery and the subsequent PDU (PD_{sep}) up to $1.5ms$ to overcome the varying boot delay introduced due to the VCC voltage drop among nodes. Also alerting mechanisms, such as parity error and buffer overflow alerting, have to be turned off temporarily until the enumeration phase starts. The short discovery timings work are necessary in simulation where simulated real time is very costly, whereas in big real networks it is expected to be necessary to adjust both parameters. Due to the extent we do not investigate more details related to the boot timings in this work.

Network Geometry Disclosure The geometry disclosure is initiated by bottom most, right most node which sends it's local address back to the origin node, immediately after the enumeration transaction is finished. Fig. 5.6 shows node $(2,3)$ acknowledging the enumeration transaction. Node $(3,3)$ responds with a AnnounceNetworkGeometryPackage which is routed to the origin node.

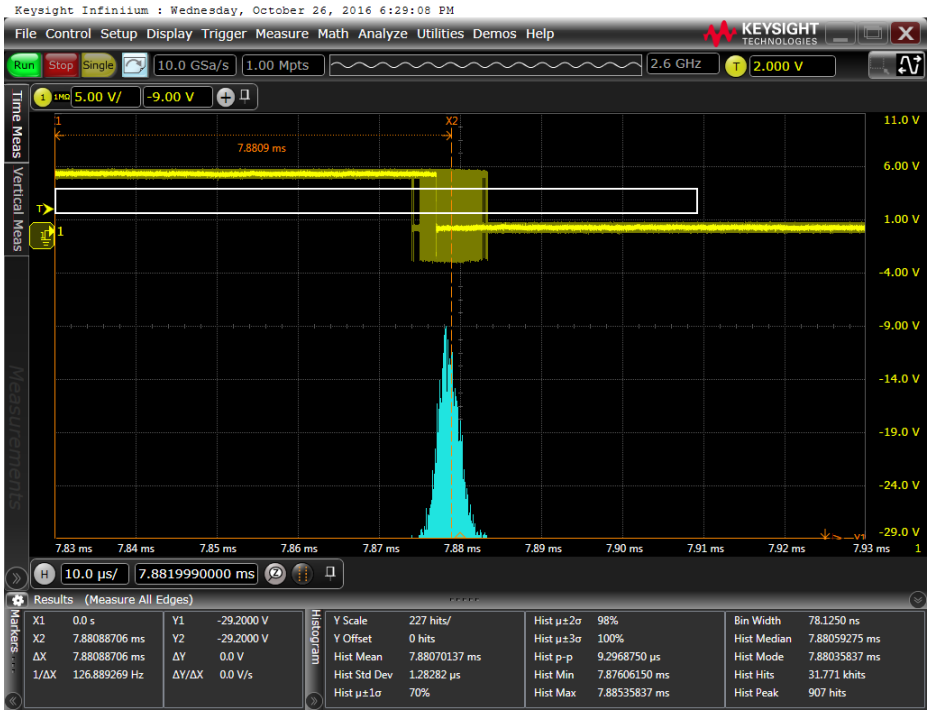


Fig. 5.7.: TimePackage's PDU transmission duration (d_{pdu}) jitter of last falling edge, triggered first falling protocol data unit (PDU) edge

Simulation and Hardware Evaluation We observe same coherence in between simulated results and results when the protocol is executed on real hardware as discussed in the enumeration evaluation in section 5.2.2.

5.2.3. Timings Acquisition

As stated in section 4.1.7 we expect measured timings to hav a minimal measuring error. However the distribution is assumed to be a non skewed $\mathcal{N}$ having a specific variance around a mean. To prove this assumption the MCU's f_{cpu} jitter and the TimePackage d_{pdu} has been investigated. The f_{cpu} illustrated in fig. 5.8 shows a non skewed $\mathcal{N}$ having $\mathcal{N}(\mu = 40.2167\mu s, \sigma = 10.08ns)$. The measurement is taken at the mean (μ), μ= $40.2167\mu s$ at a falling edge after

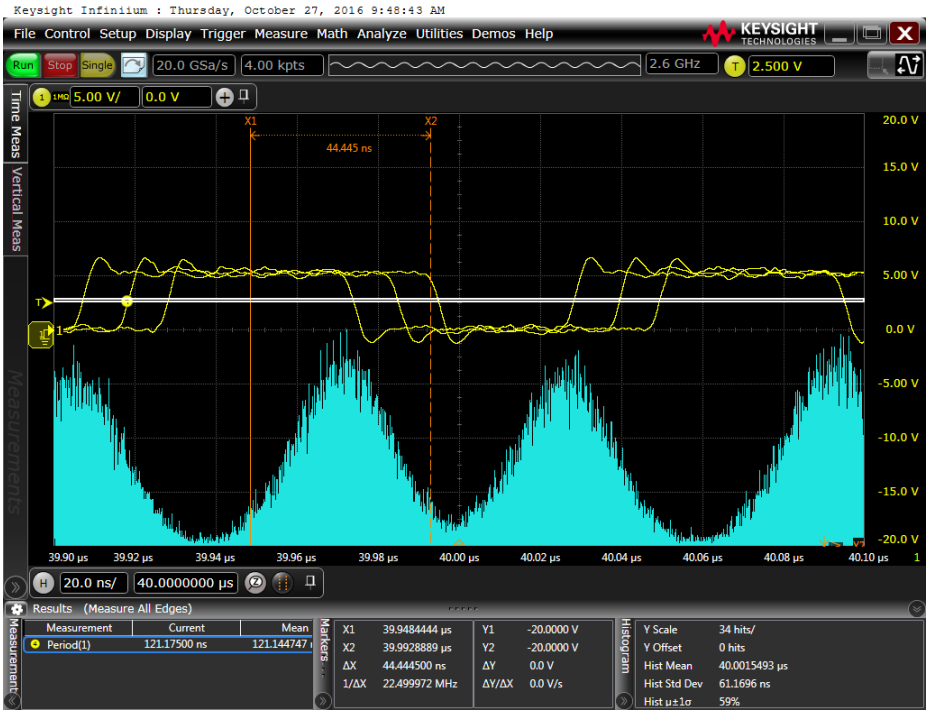


Fig. 5.8.: MCU clock frequency (f_{cpu}) jitter of several edges at $\approx 40\mu s$ after trigger

the trigger (on falling edge).

At protocol layer the d_{pdu} of a TimePackage has been measured. Fig. 5.7 shows the distribution of the last falling edge of the TimePackage. The trigger is set at the first falling PDU edge. The total delay in between trigger and highlighted edge is $63 * t_{code}$ which corresponds to 64512 cycles, or $\approx 8ms$. This jitter, again shows a non skewed $\mathcal{N}$ having $\mathcal{N}(\mu = 7.88, \sigma = 1.283\mu s)$ With this results we are able to state that the described averaging strategies ROV, SMAV, WMA and MLS suit our application.

5.2.4. Network Time Synchronization

The network time synchronization as described by the protocol can be achieved in two ways: in broadcast or subsequent mode.

Delay	Min. [μs]	Max. [μs]	Avg. [μs]
east-south signal generation shift	0.249	0.250	**0.250**
$BCTE_{delay}$	5.875	7.875	**6.969**
$BCTS_{delay}$	6.126	8.126	**7.219**

Table 5.2.: simulated introduced forwarding delay in broadcast mode (BCT_{delay}) evaluation summary of (6×6) network simulation, see also table J.1

Broadcast Mode

In this mode the first time-synchronization can be issued by the origin node as soon the AnnounceNetworkGeometryPackage is received. In this state the network must be in broadcast mode which means that any incoming signal edge at each node's north port (except of origin node) is forwarded to the east port and south port before it is captured for de-coding. Hence the TimePackage is transmitted simultaneously to any node with an introduced forwarding delay in broadcast mode (BCT_{delay}), see table 5.2.

Simulation Evaluation In fig. 5.9 for simplicity the BCT_{delay} is assumed to be constant. The reason for the TimePackages shift is the node's position in network and the data propagation. In fig. 5.9 the origin node transmits to $(1,2)$, and $(2,1)$ simultaneously, these nodes broadcast to their subsequent neighbors and so on. The detailed parts the delay consists of, are discussed in section 3.2.5.

If we investigate the measured BCT_{delay} values we can split them into two distinct groups. This is due to a time shift of $\approx 0.25\mu s$ between the east port and south port signal generation. The east port forwarding delay ($BCTE_{delay}$) and south port forwarding delay ($BCTS_{delay}$) differ exactly by this time shift as listed in table 5.2.

Hardware Evaluation An overall comparison of the introduced $BCTE_{delay}$ of a simulated network ($\approx 7\mu s$) to an optimized test implementation on a physi-

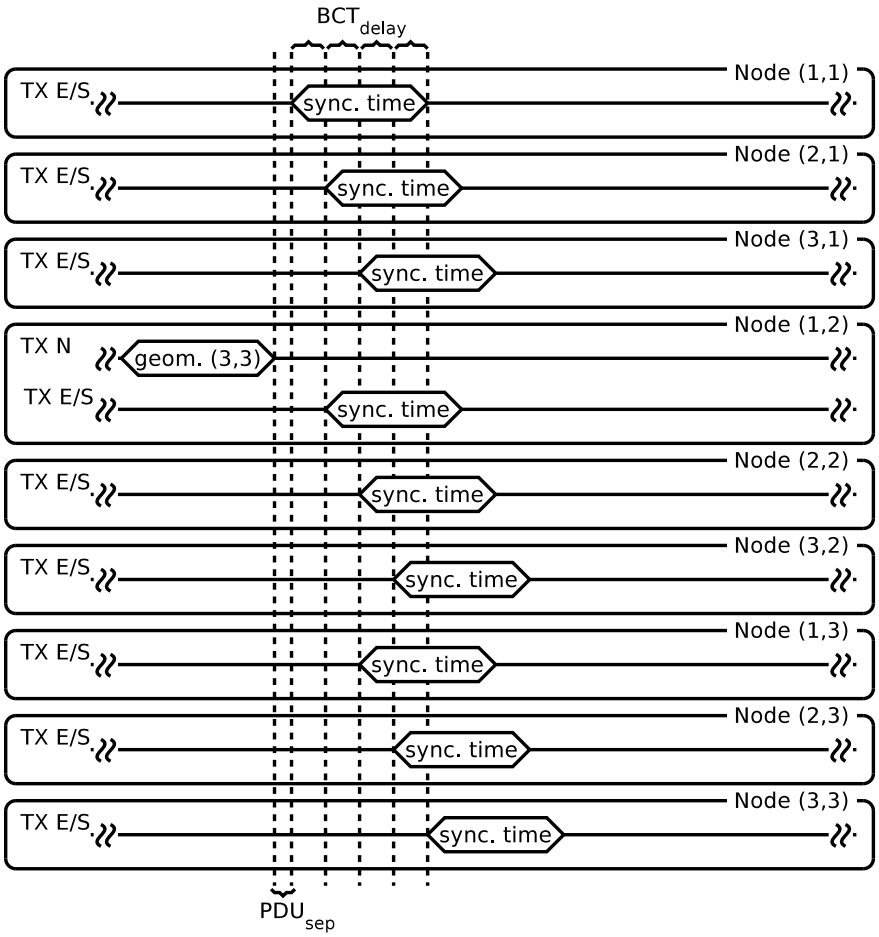


Fig. 5.9.: simulated (3×3) network time synchronization in broadcast mode showing the introduced forwarding delay in broadcast mode (BCT_{delay}) spread among nodes

cal MCU (4.4178μs) proves that the values are still reasonable. The difference we can explain due one extra function call in the simulated implementation versus the optimized implementation run on the real MCU. More details on the evaluated BCT_{delay} values are listed in the appendix in table 5.2.

According to our observations we consider the broadcast mode synchronization to be not well scaling since it shifts signal edges where the shift is not well predict-able.

Shift Any forwarded signal edge is shifted by a specific BCT_{delay}. We observe an obvious jitter on signal forwarding. If forwarded PDUs' delays are distorted by each MCU's specific BCT_{delay} plus a normal distributed ISR jitter, the PDU length cannot be seen as accurate time span reference any more.

Predictability The BCT_{delay} is not well predict-able. On each forwarding it dependents on the current MCU's f_{cpu}. Even if the delay is very short it should not be neglected in large networks.

Of course the broadcast method provides sufficient synchronization accuracy in small networks, but we aim to rely on as less as possible adjust-able parameters built in the protocol and not to cope with overcomplicated calculations. Thus we favor the subsequent synchronization mode over the broadcast synchronization mode.

Subsequent Mode

In this mode the synchronization is performed in between two subsequent nodes only. The transaction is kept very simple and does not implement any ARQ's. The initiator sends a TimePackage without expecting any response. The receptionist considers the carried data and the PDU reception duration for local clock skew, local time ans phase synchronization.

To be more precise on time measurements we favor falling over rising edges to achieve a bit more accuracy. The applied MOSFETs' gate capacity is quite high, which makes rising edges less steep and may lead to a slightly higher inaccuracy of PCI timing measurements. The observed magnitude of the time taken, since a high level is assigned to the line until it reaches VCC, is $\approx 1 * f_{cpu}$.

With respect to the online Manchester code implementation we observed asymmetric delays when generating a clock or data signal. In other words when the corresponding ISR calls the line code implementation to generate the next signal it takes longer until the response is visible on the line if the signal is a clock signal. Otherwise if the signal is a bit signal the response delay is shorter. The total difference in between both is ≈ 8 instructions.

For these reasons and simplicity we opted for a solution that takes the first and last bit edge of the TimePackage as time span reference.

5.2.5. Clock Skew Compensation

To compensate the local frequency to comply with the neighbor we adopt new frequency parameters from the time span reference which again is obtained from any TimePackage. Once the total TimePackage length is known the relative time span reference can be used into calculate several parameters needed for the clock skew compensation. The new coding speed t_{code} is updated according to observed equation (3.13). After the update the new frequency parameters are exposed immediately via the line code to subsequent neighbors.

When measuring the reference time span of the TimePackage we observed that is is necessary to not start measuring at a Manchester code clock edge. This is due to the online implementation of the Manchester code coding algorithm, which introduces two different delays before generating an edge: delay of Manchester code clock edge and delay of bit-edge. Unlike stated in the protocol design and with respect to the observations so far we update equation (3.13) and simplify it to equation (5.1). The update considers measuring the delay in between the second and last but one edge of the TimePackage. In other words the delay from the first to the last bit-edge.

$$\begin{aligned} t_{code} &= \frac{t_{pdu}}{63} \\ where & \\ 63 &= number\ of\ measured\ t_{code}\ intervals\ in\ TimePackage \end{aligned} \tag{5.1}$$

Fig. 5.10.: clock skew compensation without averaging algorithm; beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, purple node $(12,1)$, network setup network configuration setup 1 ($net1$)

Unfortunately the evaluation reveals that the MCU's f_{cpu} is very sensitive to VCC. A VCC ripple of $\pm 100mV$ destabilizes the f_{cpu} too much. This means, if the origin node's f_{cpu} is affected, also the whole network synchronization mechanism is affected. For this reason, we evaluate several averaging algorithms, as stated in section 4.1.7.

In the following figures we compare by means of a (12×1) network. The measurements are taken at nodes $(1,1)$ (first node), $(4,1)$, $(7,1)$ and $(12,1)$ (last node). The measured output represents the local time counting frequency at the respective node.

Raw Observation Value

With this method, the baseline, the clock skew is adjusted immediately after the first TimePackage is received, as illustrated in fig. 5.10. To compensate the whole network's clock skew only one TimePackage needs to be propagated throughout the network. Since there is no averaging we observe a jittering after each interpreted TimePackage. The disadvantage of this method is the sensitivity to outliers. It the result always dependent on the lastly received TimePackage. An other disadvantage of the abrupt adjustment is a possible communication disruption. If the correction is so big, the new baud rate may be out of bound for the subsequent neighbor.

Simple Moving Average

The SMAV, which makes use of four buffered values, shows a better result compared to the baseline method. The measured outcome is illustrated in fig. 5.11. In contrast to the baseline it overcomes the possible communication disruption because the adjustment takes place step-wise. Any obtained timing argument is weighted equally throughout each buffered value. This means it takes exactly as many received TimePackages as the buffer size, until the time skew is completely compensated according to the TimePackage transmitting neighbor. This method reduces the jittering which makes it hardly observable.

Thus compensation in between subsequent neighbors takes longer the bigger the buffering is. For applying this algorithm with outlier detection, a representative set of buffered values is needed which roughly estimated is > 40 measured values. In huge networks this can lead to unwanted long delays until the time synchronization is stable. Apart from this, since the results with outlier detection does not improve relevantly, no further evaluation details about this investigation are presented.

Fig. 5.11.: clock skew compensation with Simple Moving Average (SMAV) and 4 buffered values without outlier detection; beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, purple node $(12,1)$, network setup network configuration setup 1 (*net*1)

Fig. 5.12.: clock skew compensation with averaging using Weighted Movint Averate (WMA); beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, purple node $(12,1)$, network setup network configuration setup 1 ($net1$)

Weighted Moving Average

In our evaluation the WMA, which buffers only one value, we found to scale well with a weight argument of $p = 0.75$ (equation (4.6)). In figure fig. 5.12 we observe longer delay until the clock skew is compensated as with the SMAV. If we investigate the curvatures more detailed we observe an asymptotic convergence towards the target. Thus compared to SMAV we find the WMA to perform slightly poorer. On systems lacking of random access memory (RAM) we would be satisfied with WMA, otherwise we favor the SMAV over WMA.

Moving Least Squares

The evaluation of MLS showed very poor results. In the experiment we observed an overshooting of the clock skew compensation. For a better visualization the buffer of averaged values has been increased ten times. This helps smoothing the result for a better explanation. The result of fig. 5.13 shows that overshooting occur even on short network paths. This is visible at the green curvature, which represents the internal time counting frequency of node $(4, 1)$ which is only three nodes away from the origin node(beige curvature).

The overshooting occurs due to the nature of the MLS algorithm, which tries to keep the sum of squared errors minimal. This means, outliers having higher divergence to the mean are weighted quadratic. Thus the fitting function is very strong influenced by outliers.

An other serious problem is the communication disruption, which happens due to the overshooting. The first overshooting peeks after network boot, which are the highest, show a divergence of $\approx \pm 800\mu s$. This leads to moving the baud rate outside the limits, which can be seen at the purple curvature of node $(12, 1)$ in fig. 5.13. The curvature moves outside the measurement window. On a long term measurements we see that this state never recovers.

The outlier rejection lets the previous results to perform even worse. A rejection bound of $\mu \pm 2{*}\sigma$ and $\mu \pm 5{*}\sigma$ resulted in even more overshooting, whereas a boundary of $\mu \pm 10{*}\sigma$ produces again similar results as without outlier

Fig. 5.13.: clock skew compensation with averaging using Moving Least Squares (MLS) and 40 buffered values without outlier detection; beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, purple node $(12,1)$, network setup network configuration setup 1 (*net*1)

rejection. A boundary of $\mu \pm 10 * \sigma$ cannot be justified, thus outlier rejection does not gain performance at all.

Averaging Strategies Comparison

The implemented averaging strategies have been compared with respect to the memory usage, calculation complexity, adjustment parameters quantity, convergence duration and accuracy gain compared to the baseline method ROV.

MLS All tested MLS based setups produce unusable results. Even with outlier rejection this method seems to be hardly adjustable for this application. This method cannot be recommended at all.

SMAV Experiments with large buffer heavily extend the convergence delay. Large buffers may be useable in small networks but do not scale well. Out of both outlier strategies, the $\mathcal{N}$ σ dependent and the adaptive rejection, we definitely favor the σ dependent rejection even if the calculation is more costly as the adaptive method's calculation. The adaptive rejection method implies too many adjustable parameters. SMAV performs well with a buffer size of four.

WMA The WMA method performed well, except of the asymptotic convergence. The convergence duration is longer compared to SMAV but still an option on systems having less RAM or flash memory.

ROV This method is not recommendable since the possible accuracy gain, of multiple synchronization PDUs, remains unused.

Conclusion For productive usage we opted for the SMAV method with a FIFO buffer size of 4 and no outlier rejection mechanism. In the upcoming experiments, if not specified differently, this configuration is enrolled onto the (12×1) test network.

Setup	Method	FiFo Size	Outlier Detection	Performance
1	ROV	-	-	useable (baseline)
2	WMA	-	-	good
3	SMAV	4	-	good
4	SMAV	40	2σ	long convergence delay
5	SMAV	40	adaptive	many tuning parameter
6	MLS	4	-	poor
7	MLS	40	2σ,5σ,10σ	very poor

Table 5.3.: averaging strategy performance listing

5.3. Other Observations

VCC Ripple

With the current hardware design and the applied MCU we observe a very high f_{cpu} sensitivity to the applied VCC. In our experiments, for usability reasons, we used the on-board LEDs for signaling. In the (12×1) test network this lead to a VCC ripple of VCC$\pm 100mV$, which was measurable at the first node $(1,1)$. We observed a local time clock speed drift according to the ripple. In further investigations we found that at the f_{cpu} of an ATtiny1634 drifts $\pm 32kHz$ if VCC drifts $\pm 100mV$ away from $5.0V$ (measured at node $(1,1)$ with network configuration setup 0 ($net0$)).

However the internal RC oscillator is realized, we decided to not modify the hardware, but instead skip the LEDs and omit pulsing loads especially during synchronization. This immediately affected the accuracy when compensating the clock skew and synchronizing network time. Other steps to reduce this problem are: operate the MCU at $3.3V$ where the f_{cpu} function is flat, assure a stable power VCC at each node and shorter periods for network synchronization. In total this means, even with no pulsing LEDs, the network must be synchronized after actuations, since they drain a multiple of the LEDs power as shown in fig. 5.14.

add frequency curve of tiny1634 to appendix, highlight at 3.3v and 5.0v

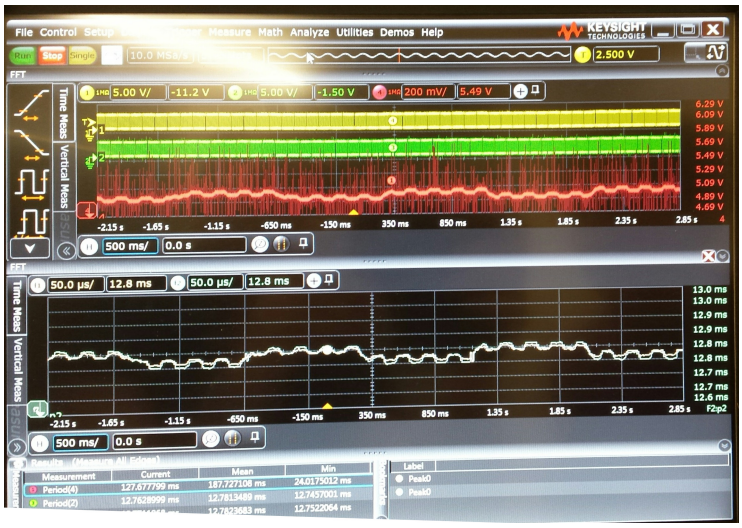


Fig. 5.14.: MCU clock frequency (f_{cpu}) sensitivity to supply voltage (VCC)

Measurement Discretization

In fig. 5.15 we see the last falling edge of the TimePackage as transmitted by node $(6,1)$, whereas in the optimum case the edge should meet the marker. The marker's position outlines the same edge of the TimePackage as transmitted by the first node $(1,1)$.

Apart from the jitter mentioned so far, the analysis of d_{pdu} of the whole test network shows a decision problem in between discrete values. When a reference time span of an incoming TimePackage is measured, it is used to re-calculate the skew compensation and baud rate as formulated in equation (3.10) and equation (5.1). This down-scaling necessarily introduces a discretization error when casting the measured floating point value to integer which manifests as the decision problem as seen in fig. 5.15.

For transmission this means if the new d_{pdu} is ± 1 the total integer discretization delay ($d_{discrete}$) of one PDU is $\approx \pm 8\mu s$ and can be formulated as equation (5.2). The calculated $d_{discrete}$ fits perfectly to the span in between the

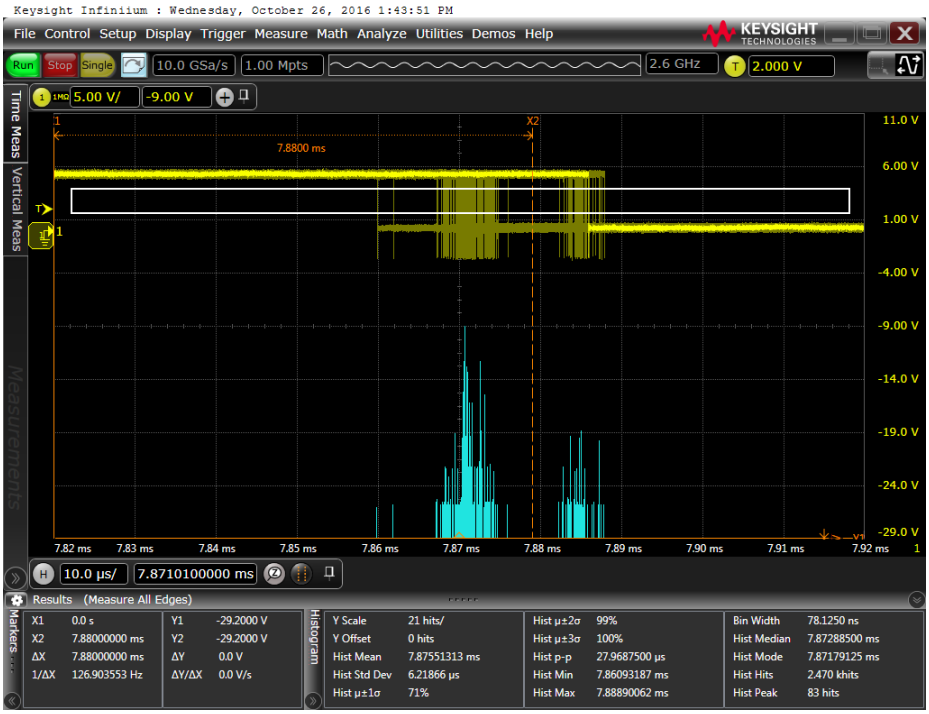


Fig. 5.15.: PDU transmission duration (d_{pdu}) discretization

Fig. 5.16.: PDU transmission duration (d_{pdu}) discretization in clock skew compensation; beige node $(1,1)$, green node $(4,1)$, blue $(7,1)$ purple node $(12,1)$setup network configuration setup 1 ($net1$)

discretization centers of fig. 5.15.

$$\begin{aligned} d_{discrete} &= \pm 64 * \frac{1}{f_{cpu}} \\ &\approx \pm 8\mu s \end{aligned} \tag{5.2}$$

Since the discretization error occurs at each node, the error cumulates the longer the path. In general, even if the standard distribution rises, we observe a new $\mathcal{N}'$, which in detail, is a Gaussian mixture consisting of multiple $\mathcal{N}$ of each discretization. The new expectation value μ' of $\mathcal{N}'$ resides the at the same marker's position, which we formulate as equation (5.3).

$$E(d_{pdu}|node = (1,1)) \stackrel{\triangle}{=} E(d_{pdu}|node = (n,1)) \tag{5.3}$$

The same cumulative discretization error is observable in a frequency trend analysis as illustrated in fig. 5.16 which shows the compensated time clock frequency after a long term synchronization run. The offset between green and purple curves of $\approx 10\mu s$ is classifiable as the discretization error. The decision problem in between two discrete values, can be perfectly followed in the time span of $-1s$ to $300\mu s$ of the purple curvature.

Given the specified hardware requirements, the observable discretization error shows that with a 16-bit TCNT, on which the whole protocol implementation relies, the limits of possible accuracy are reached.

5.4. Experiments

5.4.1. Clock Skew Compensation

In this experiment the first node $(1,1)$ is prepared to change it's f_{cpu} continuously. This is realized by incrementing and decrementing the OSCCAL. The deviation is bounded to OSCCAL± 8 which in frequency is within $[7.920, 8.566]MHz$. The rather coarse-grained step-wise f_{cpu} change is illustrated in fig. 5.17 as beige curvature.

Fig. 5.17.: clock skew compensation experiment with moving MCU clock frequency (f_{cpu}) of node $(1,1)$ (beige); green node $(4,1)$, blue node $(7,1)$ purple node $(12,1)$, setup network configuration setup 1 ($net1$)

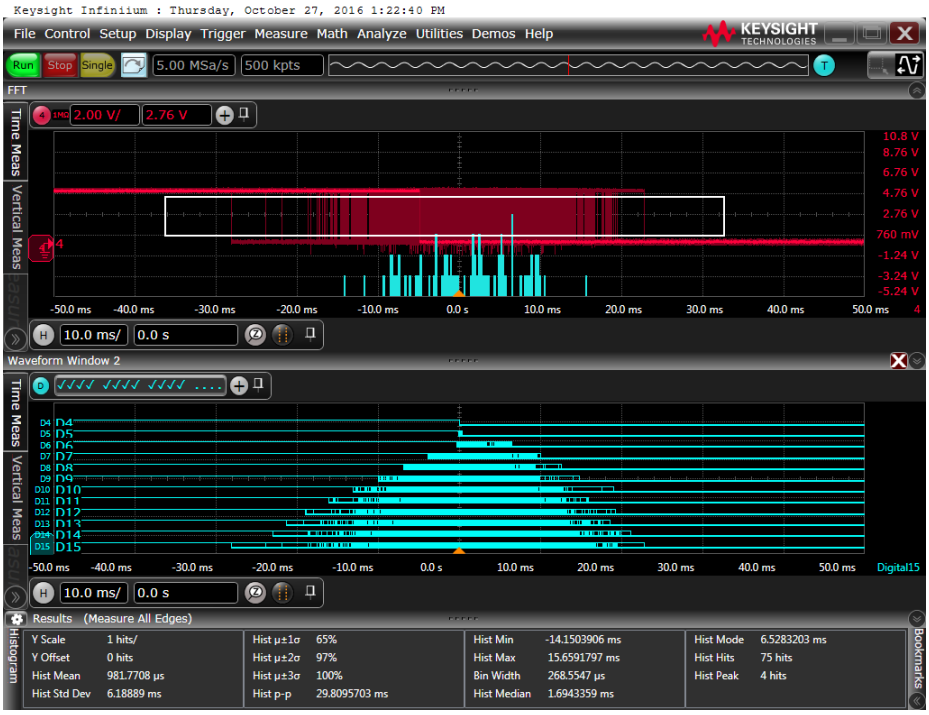


Fig. 5.18.: time synchronization distribution; purple: time distribution of last node, cyan D1-D15: time distribution of all nodes, origin node as D1, all curvatures having infinite persistence

The continuous compensation propagation manifests as delay since the change of f_{cpu} of node $(1,1)$ until a subsequent nodes reacts to it. For nodes $(4,1)$, $(7,1)$ and $(12,1)$ this is visible in the delayed response of the green, blue and purple curvatures. In the experiment the response delay from first to last node is $\approx 5s$. The delay can be decreased by more frequent synchronizations.

The experiment also reveals the coarse-grained f_{cpu} tuning characteristic of OSCCAL as discussed in section 3.2.5. Compared to our implementation the granularity is roughly 8 times smaller than with OSCCAL.

5.4.2. Time Synchronization

In this experiment we compare the network time deviance among all test network's nodes. The nodes are configured to toggle an output signal each 64 time clock intervals. This duration corresponds to $\approx 410ms$. When the network is synchronized, it receives TimePackages exactly in between these long interval toggles. This leaves enough time to propagate and execute the TimePackages until the next toggle. An other reason for the extremely long interval is ensure the measurement does not include edges of strongly shifted time intervals of the next or previous interval. The measurement illustrated in fig. 5.18 is taken after a long stabilization phase, to ensure the network is in a stable state.

Due to meter limitations and experiment setup we cannot measure the $\mathcal{N}$ arguments μ and σ. However, based on the time span of the edge distribution, the formulated distribution model of equation (5.3) is identifiable.

The observed time clock frequency shifting is a result of the measurement discretization as described in section 5.3. As outcome of several measurement repetitions we observe 90% of the synchronization deviation of any node to the origin node to be within $\mu \pm 10ms$, with μ being the origin node's current local time.

Actuation

In the actuation experiment the network is prepared to firstly synchronize then actuate each left actuator simultaneously. Since per default the communication line resides at a high level, we are only able to measure the actuator's terminal which is switched to GND. In fig. 5.19 a VCC voltage drop, according to the load of all activated actuators, is visible. A more detailed investigation showed a voltage drop of $900mV$ at node $(1,1)$ and $1100mV$ at node $(12,1)$ at the same experiment. On such heavy VCC impacts a network resynchronization is urgently necessary.

Fig. 5.19.: actuation accuracy; yellow actuator (1-2, 1), green actuator (3-4, 1), blue actuator (6-7, 1) and purple actuator (11-12, 1), cyan D4-D14 all actuators, (1-2, 1) as D1

6. Discussion

Originally the protocol was meant to also sustain remote programming of nodes. This means once the origin node is flashed with newer firmware, it replicates the same onto the subsequent nodes. This feature is very desirable since a lot of time has been spent onto enrolling updates. Unfortunately the protocol development extent became enormous and this nicely feature had to be skipped.

Regarding error detection and fault tolerance no huge efforts has been made. The protocol only implements ARQ during a short initialization phase and detects up to one bit flip in regular communications. The error detection could be greatly enhanced. For instance an error reporting to the origin node would be very helpful. Error correction strategies are more complicated since it bears also many questions: how can the protocol still remain a real time protocol, how to not increase the communication overhead too much et cetera.

The clock skew compensation is the crucial point of the whole protocol. We expect the compensation to be improvable by using a hybrid strategy. Averaging strategies perform well with a larger buffer, but this also vastly increases the synchronization delay from boot until the network is stable. A combination of WMA and SMAV would shorten the delay until network is stable and allow larger buffering.

At an early phase we had also a flooding communication mechanism in mind. Such an communication requires the implemented simultaneous communication mode. We left this part out, due to the major focus on scalability. However, for applications in small networks this may be an essential mode. It is expected to outperform the current synchronization accuracy in small networks.

The difficulty we face on clock skew compensation is the sensitive oscillator stability. We faced a tightly VCC coupled clock speed at the operating voltage. To overcome this issue one may synchronize the network more frequently but

we also suggest to stabilize the supply voltage at each node, suppress any kind of supply voltage ripples or eventually also switch to a crystal oscillator driven approach.

7. Conclusion

Exploiting actuators as communication channels turned out to work very stable even with the chosen baud rate, that is to be understood as the maximum bound the implementation is able to handle reliable. With respect to the needed accuracy, which is in the range of seconds, the requirements and limitations we can present promising results. The major pitfall for this protocol is the oscillator stability which, in our application, is tightly coupled to the power supply voltage. Even small ripples lead to severe de-synchronization.

The simulator sustained development process turned ot to be more powerful than expected. The realized test driven development, memory inspection, visualization and timing metering massively gained the development speed as well the protocol quality. Roughly more than 95% of the total development could be achieved by help of simulation only, until it was enrolled on physical hardware. An other powerful aspect is detailed and uniform simulator log which is suitable to feed other simulations such as physics, forces, friction et cetera.

The produced output consisting of modular development hardware, the protocol itself and several simulation tools makes the application quite generic. It may be applicable in many different environments as just in shape shifting displays only. Especially in systems where a scalable lightweight hardware implementation is vital.

***** end of writing ***** begin of notes *****

* the cascaded interrupt response time $\mathcal{N}$ is $\mathcal{N}(\mu = 6.15\mu s, \sigma = 167ns)$

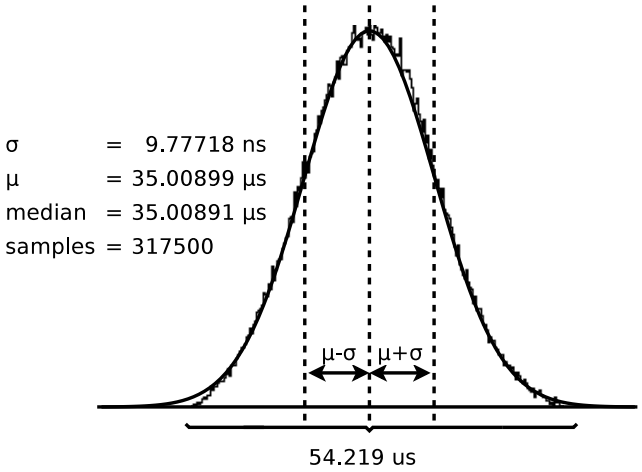


Fig. 7.1.: RC clock jitter distribution measured at $t = 35\mu s$

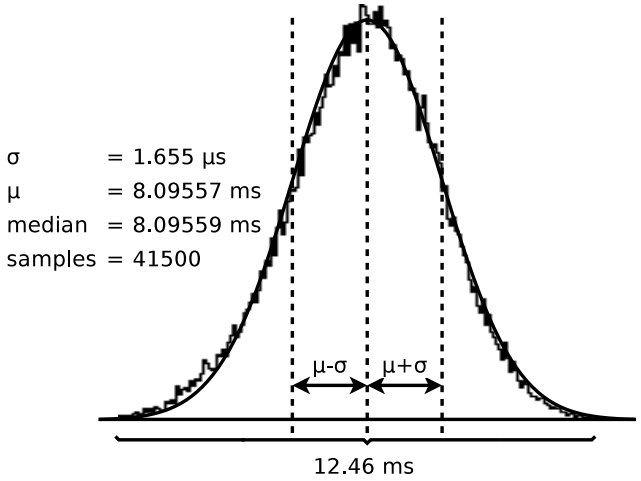


Fig. 7.2.: internal time tracking jitter distribution measured at $t = 8.1ms$

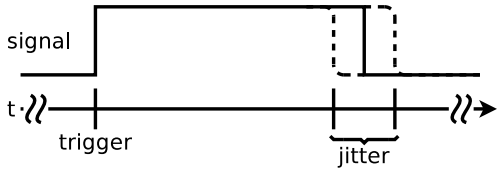


Fig. 7.3.: signal frequency jitter

issues when deploying to hw * pull up wrongly configured - tn1634 has PUE but mega16 has: conf. pin as input, write 1 to output
* east and south PCI were switched in real mcu config
* boot time differs in real life: pos in network but also from mcu to mcu. thus the discovery must be extended to 3-5 times than in sim.
* PDUsep seems to be some times slightly but nor relevant higher than in sim: due to extra 8instructions when applying the rectified signal to line -> consider just bit to bit signal delay
* ack is slightly shorter than 1ms, 950us
* disclose network geometry 3.0 instead of 3.1 ms
* BTCSdelay 7.826us seems to be okay, also in comparison with deltaT measured by matteo (4.42us).
* east to south signal generation shift is not measurable
* μ from samples is too low by $\approx$ 13 clocks
* issue with interrupt jitter

Todo list

Part I.

Appendix

A. Hardware and Network Design Proposal

Daisy Chain Communication for long Chains of Robotic Particles

Raoul Rubien, BSc
rubienr@sbox.tugraz.at

***Abstract*—Chains of robotic particles are able to change their shape in a programmatically. By applying multiple chains surfaces capable of shifting their form may be realized. The sum of such chains can be viewed as programmable matter. Programmable matter is a collection of small scaled units integrating computing, sensing, actuation, and locomotion mechanisms [1] that is able to change physical properties on command [2] and thus a universal material. It usually consists of a high volume of units.**

We apply particle chains as presented by Lasagni et al. in [3] and present a method to exploit the actuators in the system as communication channel. This method helps minimizing the particle size and thus the weight which extends the maximum chain length. It overcomes also other limitations of that work.

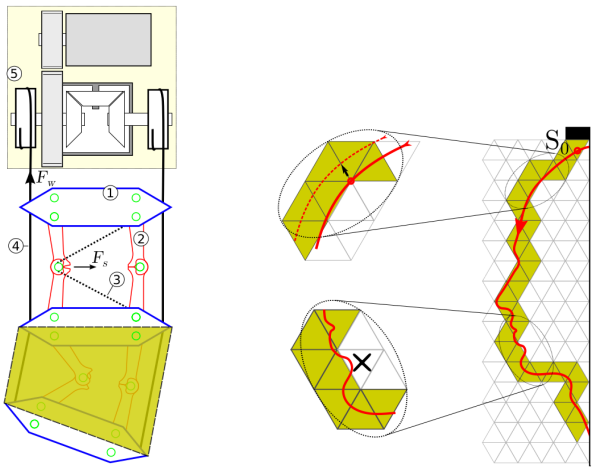
This work elaborates the design and assembly of a particle prototype and also the necessary programming tool chain. It focuses on the hardware implementation. Software details as communication protocol, addressing or network discovery will be part of the upcoming work based on the outcome of this project.

I. Introduction

Unlike several in literature proposed folding methods [1, 2, 4] the force guided chain nodes do not utilize motors, magnetic adherence fields or origami folding to achieve folding. Force guided chains are made of tied particles that are fold-able in two directions of one geometric dimension similarly to a snake's shape when moving on a plane surface. With multiple parallel mounted chains, shape shifting surfaces may be realized. Such surfaces are able to approximate 3-dimensional models at least partly in a 2.5-dimension.

A. Functionality [3]

All nodes of a free-hanging particle chain are naturally pulled down by gravity. Thus the chain's natural state is unfolded and the particles are ordered in a line. Between neighboring particles two connections, one at the left and a second at the right-side, are realized with monostable hinged edges, as illustrated in fig. 1a. When the chain is straight, the edges reside in a locked state. In this state an utilization of the external force (4) and (5) in fig. 1a would lead to a contraction of the chain but will not shape it. Only if hinges are unlocked in advance the force would lead to shape forming. As an example let us see what happens if the chain is contracted but one left hinge in-between two particles is unlocked and all other are locked. The applied force contracts the chain and thus compresses the particle's left-side (shorten the distance) at the unlatched hinge position whereas the right-side remains at the same length. This ends in folding of two particles to the direction where the hinge was previously unlocked as shown in the highlighted zone of fig. 1a.



(a) Mechanical design of chained particles with hinges (2) that are unlocked actuators (3). When applying external force (4) unlocked hinges are folded.

(b) Shaped chain example. Chain is aligned to a grid following a shape.

Figure 1: Mechanical design and folding example [3] of a chain.

B. Limitations

The applied particles do not operate autonomously. They have to be coordinated at a higher level. Thus they must at least communicate with a bus master. In Lasagni et al. as communication protocol the Dallas 1-Wire®[1] bus is utilized. Among others, for this application the bus brings significant disadvantages. 1) The protocol's maximum current limitation: If the current consumption of the attached devices exceeds the limitation communication must be decoupled from the power supply. To overcome this issue in Matteo et al. the communication is decoupled through time division. The system switches in between power supply and communication mode accordingly. As a consequence of that the bus master looses synchronization with the slaves which introduces a delay after each operation to restore the bus communication. During communication the particle power supply must be

[1] https://en.wikipedia.org/wiki/1-Wire (01/2016)

buffered beforehand with a capacitor which introduces a more electronic parts per particle. 2) Large addressing overhead of 64 Bits: The addressing is rather huge according to the needed payload for this application thus the addressing field produces a lot of overhead. 3) Lack of advanced features: The bus provides no advanced addressing features which were desirable in a particle network such sending a datum to a range of nodes. 4) Also the network discovery comes with some limitations. Network addresses can be easily retrieved with the Dallas 1-Wire® bus but not the placement of nodes. Thus the network positions must be probed in a brute-force way. Beside the bus limitations a chain's maximum length is physically limited by its weight.

II. Motivation

The optimal particle design would be very small. Combining a optimal hardware design, a customized communication protocol and the chained arrangement we want to present a daisy chain communication method which in contrast to Matteo et al. [3] exploits the actuators embedded in the system.

Our primary motivation is to minimize the size and weight of particles. Therefore we attempt to lessen the number of electronic components per particle since this physical parameters help chains to be miniaturized and extend the physically limited maximum length. Although we decouple communication from power supply we cannot eliminate the need of time division multiplexing as transmissions and actuating must never overlap.

Our secondary motivation is to enhance the communication overhead, the duration and also the network discovery. Thus we set up a daisy chain protocol which allows to use the underlying physical infrastructure as bus or peer to peer network. For the upcoming work this ensures enough freedom to choose one or both option/s for data transmission according to the use case scenario. With two actuator wires per particle pair the the communication protocol can be developed to support full duplex operation.

III. Goals

As the new protocol's physical layer differs from the current one there is no chance to re-use the circuity. This circumstance forces us to build a new prototype that is able to sustain the upcoming work. The outcome we are interested in are a combination of hardware and software infrastructure that sustains the protocol development. 1) Hardware related: a) a fully functional PCB project (schema and routed PCB) that can be chained, that allows b) modifying rapidly the number of network nodes and c) a simple debugging method (for example test point pinout). 2) Software related: a) a Unix-based toolchain, b) a test software that can be used to check newly assembled boards for errors, c) a simple debug possibility and d) a convenient method to invoke test cases on a sensor network simulator.

IV. Requirements

With respect to the principal requirements illustrated in fig. 2 the project requirements can be listed as: 1) Building a development particle prototype that substitutes the design of fig. 3 during development. In contrast to the currently existent particle, the new prototype must be capable of transmitting data to adjacent nodes using the actuator wires. 2) A new prototype must be able to control the actuators and provide the support for serial full-duplex communication via a N/P-Channel MOSFET. 3) Prototypes have to be handy, offer access to test points such as rx, tx before and after the MOSFET transistors and 4) several spare test points directly connected to the MCU. Despite of the productive particle the development PCB size is not required to be at minimum. 5) CLKO [5] pin must be connected to one test point for potential internal RC-oscillator calibration. 6) Particles have to signal their internal state such as heartbeat, status, error by means of LEDs. 7) The ISP programming interface must be easily accessible. 8) Self programming: Particles need to be capable of enrolling their firmware on their next neighbors. 9) Actuators should be replaced by 5V light bulbs with similar electrical characteristics. 10) The network topology is a combination of tree [6] [7] and linear daisy chain [7] network. Additionally added chains must be connect-able to the network as illustrated in fig. 4a.

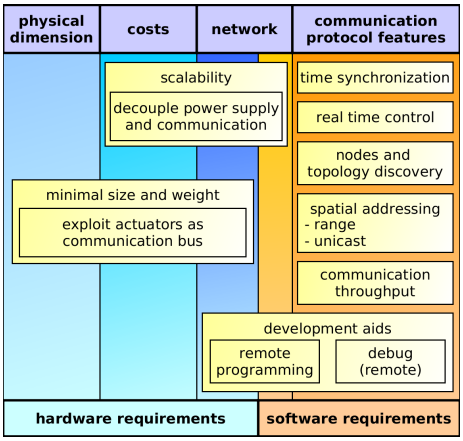


Figure 2: Principal Requirements

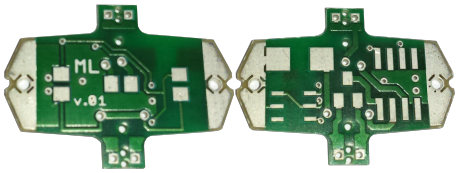


Figure 3: Front and back-side of the current unequipped particle PCB layout [3]. The dimensions are approximately $2cm \times 1cm$.

V. Materials and Methods

This section elaborates the hardware development and software simulation approach with regard to the listed requirements in section IV. Decisions made are explained in detail.

A. MCU selection

With respect to the requirements in section IV we started searching a low level economically priced MCU. When comparing MCUs of different manufacturer we chose Atmel® because of several reasons. The most significant are the availability of 1) a free of charge usable open-source compiler, 2) a variety of inexpensive programmer hardware, 3) good documentation and many examples and 4) the increasingly used MCU family. With the Tiny (ATTiny) MCU category Atmel® provides small sized 8-Bit low level micro controllers which perfectly meets our needs.

1) First prototype: The firstly created development particle board applied an ATTiny20 MCU. This MCU provides 2 Kilobytes of flash memory and 128 Bytes of static random access memory (SRAM). To proof the protocol concept with the chosen MCU a quick survey demonstrated that 2 Kilobytes of flash will not be sufficient. Just the implementation of a simple neighbor discovery exhausted up to 50% of the flash memory. Regarding SRAM size we did no extra survey but the experience we made showed up that this resource may be critically low.

2) Modified requirements: During investigating the communication protocol requirements have been refined slightly. A chain communication port was introduced to connect whole chains to the network without the need of additional master device per chain. For that reason three independent pin change interrupts, one per reception wire, are necessary. Hence the ATTiny20 is not be applicable any more. Also self programming is desirable at a later moment when the firmware of a whole network of particles has to be exchanged. This can be solved using a customized boot loader that receives, writes and forwards a firmware. Further we desire a big MCU package on the development board since it is very handy to mount and access for later measurements.

3) Result: Taking in account that the MCU package on a productive particle should be as small as possible we chose the ATTiny1634. This MCU comes with 16 Kilobyte of flash memory, 2 Kilobyte of SRAM, enough pin change interrupts and also 2 UART ports.

4) Side benefit: The shift away from the ATTiny20 also eases the firmware flashing. In case of ATTiny20 flashing a firmware is very costly since it supports no Serial Peripheral Interface (SPI) but only a Tiny Protocol Interface (TPI). A modified RS232 breakout board from SparkFun[2] with a customized avrdude[3] configuration using BitBang[4] protocol had to be applied. Fortunately this is not necessary for the SPI supported by the ATTiny1634.

B. Network topology

For this project many network topologies may be applicable but as mentioned our motivation is to exploit the already available actuator wires. They can be safely used as communication channel. Since each particle is connected via two actuators to it's neighbors we can use them to build a dual cable network system [8]. The network system uses one wire as up-link channel and the second as down-link channel.

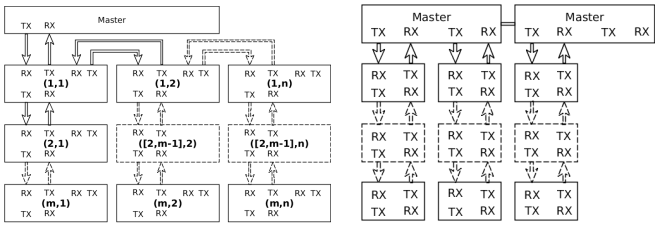
1) Network topology: We also opted for building a daisy-chained network where nodes are connected as peer to peer nodes. With that decision particles can be connected as linear network to achieve a particle chain. Due to the fact that a high number of chains is to be expected within an application the communication with chains needs to be bundled. In case of chains being connected directly to a master device (no bundled communication) each chain occupies two I/O pins. That also implies that multiple master devices need to bee employed if the number of available pins is exceeded as illustrated in fig. 4b. It also complicates the protocol by adding the necessity of master to master communication. A bundled method lowers the amount of occupied I/O pins at the network master regardless of the network size (fig. 4a). Thus we embed three identical communication channels per particle: 1) north - the communication port to the upper particle, 2) south - the communication port to the lower particle and 3) chain - the communication port to the next chain. The chosen network topology is a tree structure with chained nodes. This involves some risks. If a particle malfunctions the network is split into two parts. The interrupted segment is then not able to communicate with the root any more. Furthermore the daisy chained nodes' nature is to work as repeater. Each received data must be intercepted and forwarded. This adds a specific delay per node during data forwarding. Nevertheless the delay can be minimized by forwarding each received signal immediately to the next communication ports. Preliminary experiments showed the time shift between incoming signal and forwarded signal is about $2.2\mu s$ per MCU. The test was set up with an ATMega2560 using an external crystal oscillator at $16MHz$. This delay is expected to be longer in the real application since the MCU frequency is lower and the main routine is not remaining empty as in our test. Hence a slightly increased interrupt latency is caused by the jitter of multi cycle operations being executed when the inputing signal arrives.

2) Linking the network: To realize the network, particles chains are connected at the first chain's particle (later addressed as head particle or head), to the next chain. As an example two chains a and b are connected by linking the head particle's chain port of a to the head's north port of chain b. With this arrangements we can construct a network as illustrated in fig. 4a. The only communication entry point to the network is the north communication port of first chain's head particle which we term as the origin node.

C. Hardware layout

To sustain the upcoming development of a new communication protocol which uses actuators as communication lines a custom PCB that is capable of switching the actuator working mode (communication or actuating) is necessary. Also effortless access to hardware is desirable to be less time consuming when analyzing the physical communication. For that reasons a new prototype PCB needs to be developed and assembled. It should permit the developer to have fast access to several important test points and provide some visual signals as well.

[2]www.sparkfun.com (01/2016)
[3]http://www.nongnu.org/avrdude/ (01/2016)
[4]https://en.wikipedia.org/wiki/Bit_banging (01/2016)

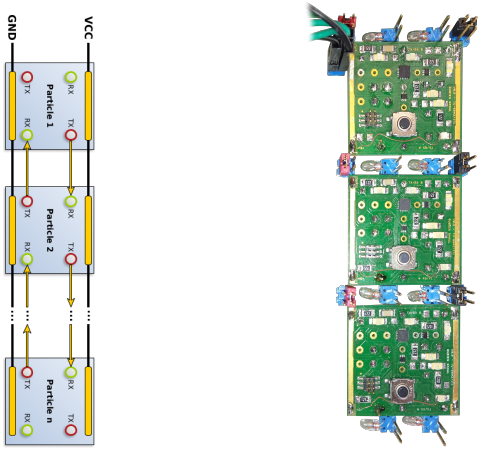


(a) Particle network with cascaded particle chains. They are connected their first particle.

(b) Particle network without cascaded chains. Each chain communicates directly to one master device.

Figure 4: Cascaded chains versus direct chain communication. One dashed rectangle represents a set of nodes.

1) Preparatory work: The current particle development board (V1.21) has past several versions. The first idea was to chain particles without using an underlying frame. Development nodes were conceptually designed to be connected at their power supply pads by using strong inflexible wires as depicted in fig. 5a. This should give enough stability to handle short chains and protect the light bulb terminals from breaking. Therefore the pads were realized stable and placed along the whole adjacent PCB sides. As a consequence of that, once a chain is assembled segments cannot be detached any more. Though detaching chain parts is desirable. To deal with that we mounted particles on a matrix board as shown in fig. 5b. All particle to particle connections were passed via jumpers. The result was stable and handy but unfortunately the fixing consumed too much time.

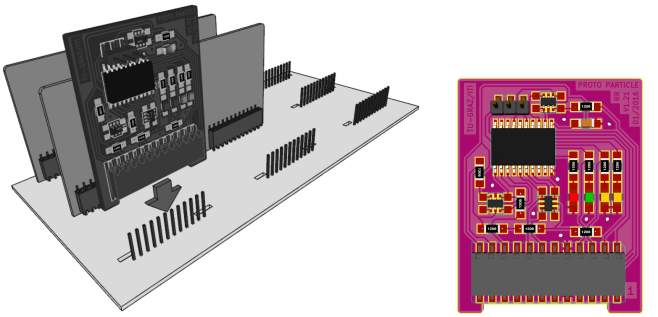


(a) Proposed model of a particle chain. Power supply can be soldered on a stable rail. Actuators may be replaced by 5V light bulbs.

(b) First working version (1.0) of a development particle chain. Actuators are substituted by light bulbs.

Figure 5: First implementation of a development particle chain.

2) Current result: For the reasons mentioned before we decided to build a grid board (see fig. 6a and fig. 6b) that provides the network connections for each particle. The idea is to make the network configurable by plugging/unplugging particles to/from the grid board. The In-System Programmable port is outsourced to the board which makes the particle design more uncomplicated. The upcoming protocol development will for sure need debugging capabilities. For this tasks the board provides three arbitrarily usable test pints and several LEDs. It is most likely that the protocol needs to synchronize the particles. How this is solved in detail is not part of this work. Never the less if the internal oscillator has to be calibrated at runtime the CLKO pin is needed to reflect the internal oscillator frequency. Therefore the CLKO pin is connected to one test point. If at a later moment an additionally serial communication is desired it can be derived from the SPI port since the MOSI/MISO pins also provide UART.



(a) An examplary 3×3 grid board. It provides the network infrastructure.

(b) Pluggable particle board. Two leads provide additional stability when plugged into the grid board.

Figure 6: A grid board exemplar and the particle board in detail.

D. Software simulation

To speed up the upcoming software development a software simulation is desired. Fortunately a particle network does not need synthesized input samples if any network node can be depicted in a network simulation. Hence we just need a simulation framework that is capable of simulating whole networks. Anyway if a network can be simulated there are still issues to investigate. For example how are particle's clock synchronized within the framework? Is the framework capable of scaling the clock or introduce clock drift per particular particles? Since one network does not only consist of MCUs but also some periphery components per particle's PCB we want to simulate a particle as a whole. With this desires we investigated available simulations and opted for the Avrora [9] sensor network simulation. Among others the framework is capable of simulating an ATTiny16 MCU and it is possible to simulate particles as a whole. The specific implementation of a particle including actuators, test points and LEDs is achieved by implementing Avrora's Platform interface. A proof of concept has been done with particle prototype hardware version 1.0. A neighbor discovery has been implemented by use of the simulation and then successfully tested on hardware.

E. Tool chain

The current state of the project also covers software implementations for concept proving. We organized the source with

CMake and a couple of Unix tools. With that we constructed a build chain to easily launch builds, flash particles, start sensor network simulations, retrieving simulation statistics or debug particles via UART and much more.

VI. Future work

In our future work we plan to develop a communication protocol that provides a way to communicate to each chain's particle. The protocol will span the first three layers of the OSI model: 1) Physical Layer, 2) Data Link Layer and 3) Network Layer and will address the network coding [10], self enumeration and addressing, scheduling of actuator tasks and time synchronization.

For the time synchronization it is to be determined if exploiting the synchronization of a Manchester coding (layer 2) is accurate enough or if it has to be solved in layer 3. Since particles use their internal oscillator it is of high interest if calibrating the internal oscillator at runtime [11] is feasible.

Also enrolling of particles' firmware we plan to achieve by using a customized boot loader to speed up the deployment in networks. The idea is to replicate on particles firmware to its next neighbor et cetera.

VII. Acknowledgements

This project was supported by the Institute for Technical Informatics of Graz University of Technology and Matteo Lasagni who has always been sincere and helpful and assisted this project.

References

[1] Seth Copen Goldstein; Jason D. Campbell; Todd C. M. Programmable Matter.

[2] Ara N. Knaian, Kenneth C. Cheung, Maxim B. Lobovsky, Asa J. Oines, Peter Schmidt-Neilsen, and Neil a. Gershenfeld. The Milli-Motein: A self-folding chain of programmable matter with a one centimeter module pitch. *IEEE International Conference on Intelligent Robots and Systems*, pages 1447–1453, 2012.

[3] Matteo Lasagni and Kay Römer. Force-guiding particle chains for shape-shifting displays. *CoRR*, abs/1402.2507, 2014.

[4] E Hawkes, B An, N M Benbernou, H Tanaka, S Kim, E D Demaine, D Rus, and R J Wood. Programmable matter by folding. *Proceedings of the National Academy of Sciences of the United States of America*, 107(28):12441–12445, 2010.

[5] Atmel. *8-bit Atmel tinyAVR Microcontroller with 16K Bytes In-System Programmable Flash*, 2 2014. Rev. 8303H.

[6] Joseph Kizza. *Guide to computer network security*. Springer, London, 2015.

[7] Barrie Sosinsky. *Networking bible*. Wiley, Indianapolis, IN, 2009.

[8] Andrew Tanenbaum. *Computer networks*. Prentice-Hall, Englewood Cliffs, N.J, 1988.

[9] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *2005 4th International Symposium on Information Processing in Sensor Networks, IPSN 2005*, volume 2005, pages 477–482, 2005.

[10] Andrew Tanenbaum. *Computernetzwerke*. Pearson, Munchen, 2012.

[11] Atmel. *AVR054: Run-time calibration of the internal RC oscillator*, 4 2008. Rev. 2563C-AVR-04/08.

List of Figures

List of Tables

Contents

B. Package Listings

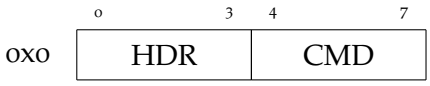


Fig. B.1.: command

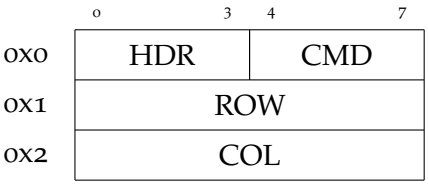


Fig. B.2.: node command

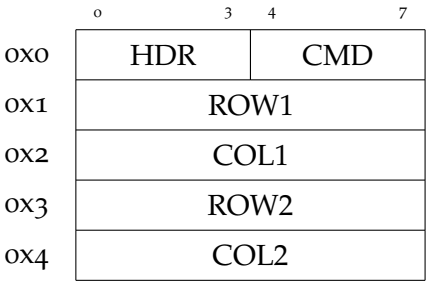


Fig. B.3.: node range command

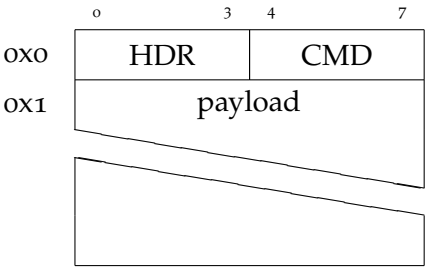


Fig. B.4.: command with payload

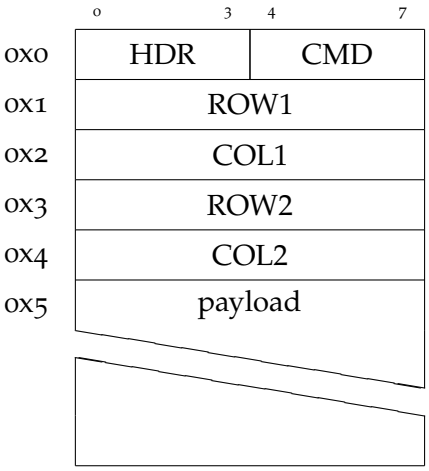


Fig. B.5.: node range cmd. with payload

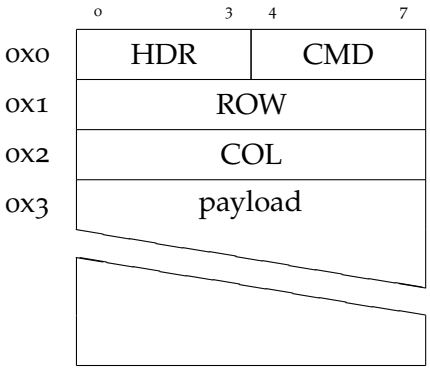


Fig. B.6.: node command with payload

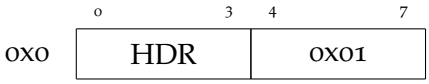


Fig. B.7.: HeaderPackage

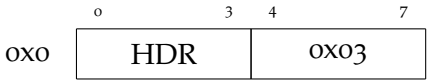


Fig. B.8.: RelayHeaderPackage

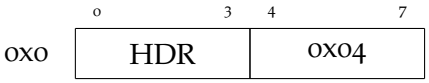


Fig. B.9.: ResetPackage

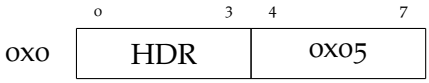


Fig. B.10.: AckPackage

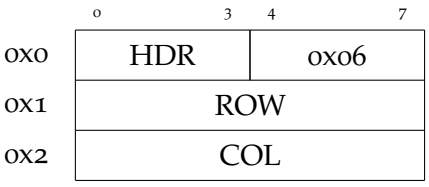


Fig. B.11.: AckWithAddressPackage

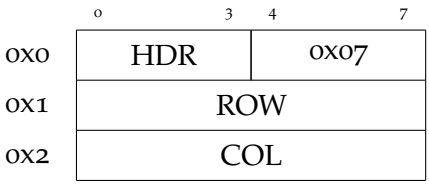


Fig. B.12.: AnnounceNetworkGeometry-Package

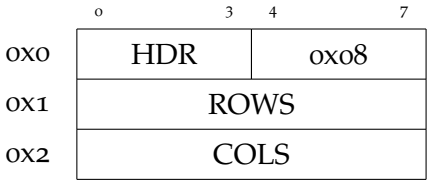


Fig. B.13.: SetNetworkGeometryPackage

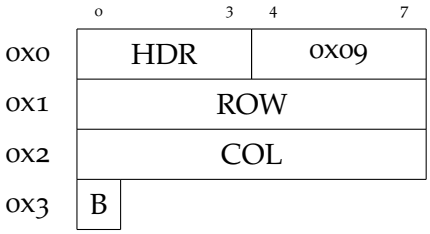


Fig. B.14.: EnumerationPackage

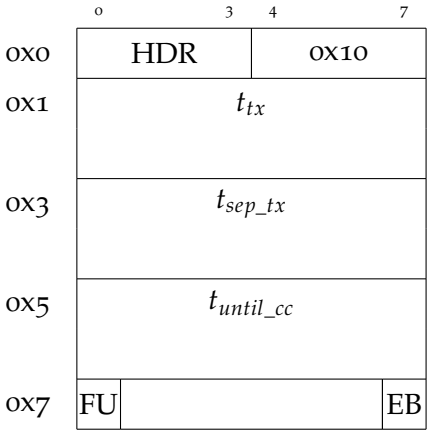


Fig. B.15.: TimePackage

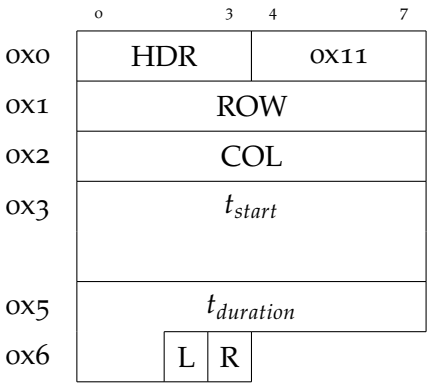


Fig. B.16.: HeatWiresPackage

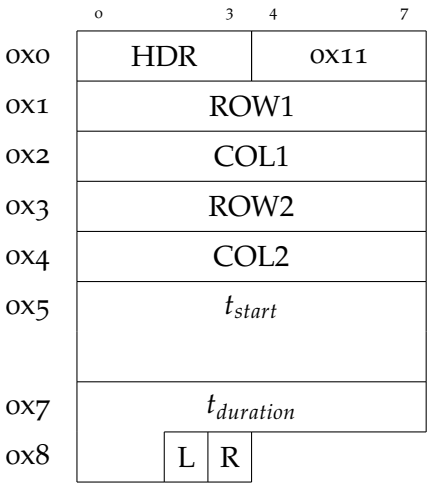


Fig. B.17.: HeatWiresRangePackage

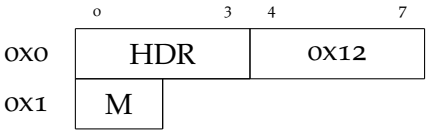


Fig. B.18.: HeatWiresModePackage

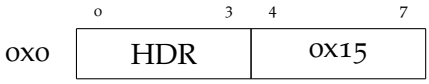


Fig. B.19.: ExtendedHeaderPackage (reserved)

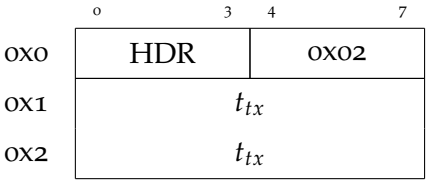


Fig. B.20.: SyncNetworkTimeHeader-Package

C. Sequence Diagrams

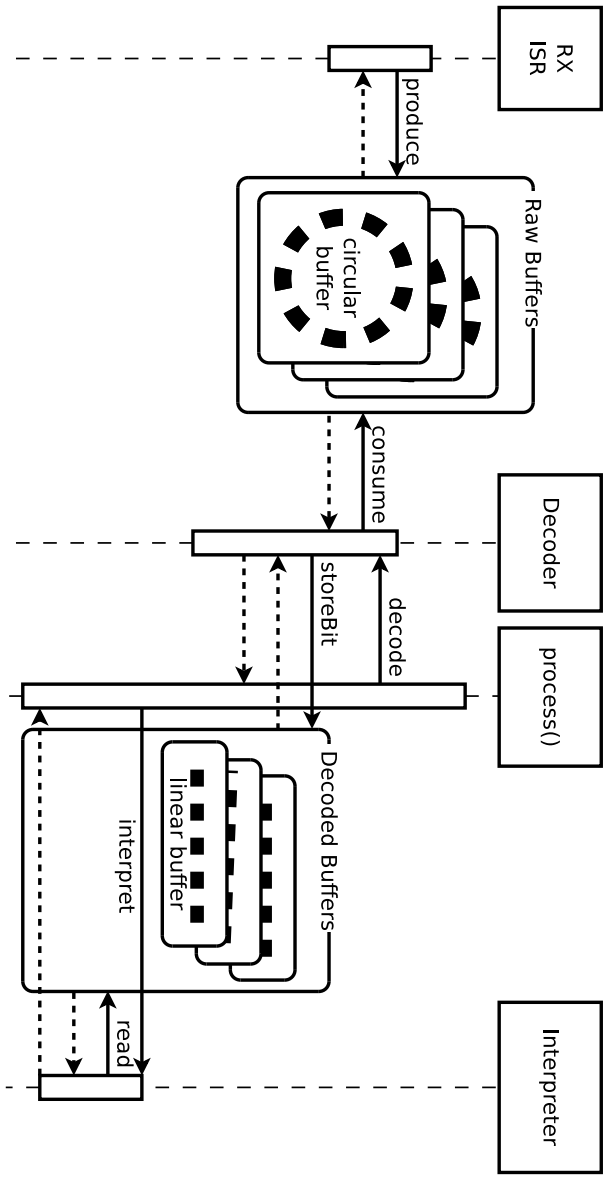


Fig. C.1.: reception, decoder and interpreter sequence diagram

D. Code Snippets

```
FUNC_ATTRS void __handleSendAnnounceNetworkGeometry
  (StateType endState) {
  volatile TxPort *txPort =
    &ParticleAttributes.communication.ports.tx.north;
  volatile CommunicationProtocolPortState *commPortState =
    &ParticleAttributes.protocol.ports.north;
  switch (ParticleAttributes.protocol.ports.north.\
    initiatorState) {
    // enable tx
    case INITIATOR_TRANSMIT:
      constructAnnounceNetworkGeometryPackage(
        ParticleAttributes.node.address.row,
        ParticleAttributes.node.address.column);
      enableTransmission(txPort);
      commPortState->initiatorState =
        INITIATOR_TRANSMIT_WAIT_FOR_TX_FINISHED;
    break;
    // wait for tx finished
    case INITIATOR_TRANSMIT_WAIT_FOR_TX_FINISHED:
      if (txPort->isTransmitting) { break; }
      commPortState->initiatorState =
        INITIATOR_IDLE;
      goto __INITIATOR_IDLE;
    break;
    // tx finished
    case INITIATOR_WAIT_FOR_RESPONSE:
    case INITIATOR_TRANSMIT_ACK:
    case INITIATOR_TRANSMIT_ACK_WAIT_FOR_TX_FINISHED:
    case INITIATOR_IDLE:
    __INITIATOR_IDLE:
      ParticleAttributes.node.state = endState;
    break;
  }
}
```

Fig. D.1.: flow control handling example with automatic repeat request (ARQ) shortcut

E. Structure Diagrams

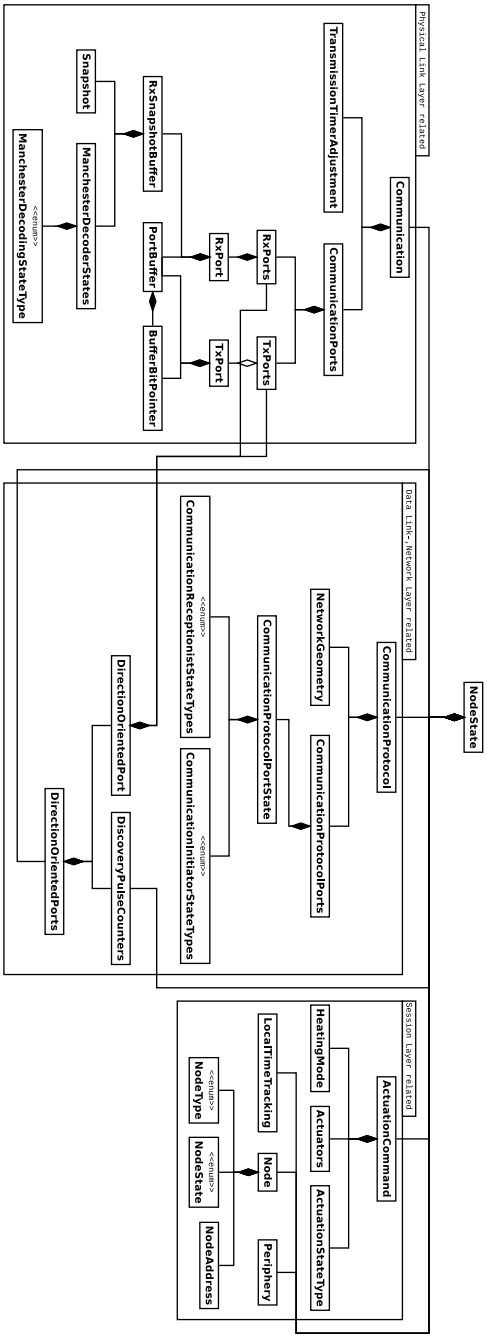


Fig. E.1.: node's context overview

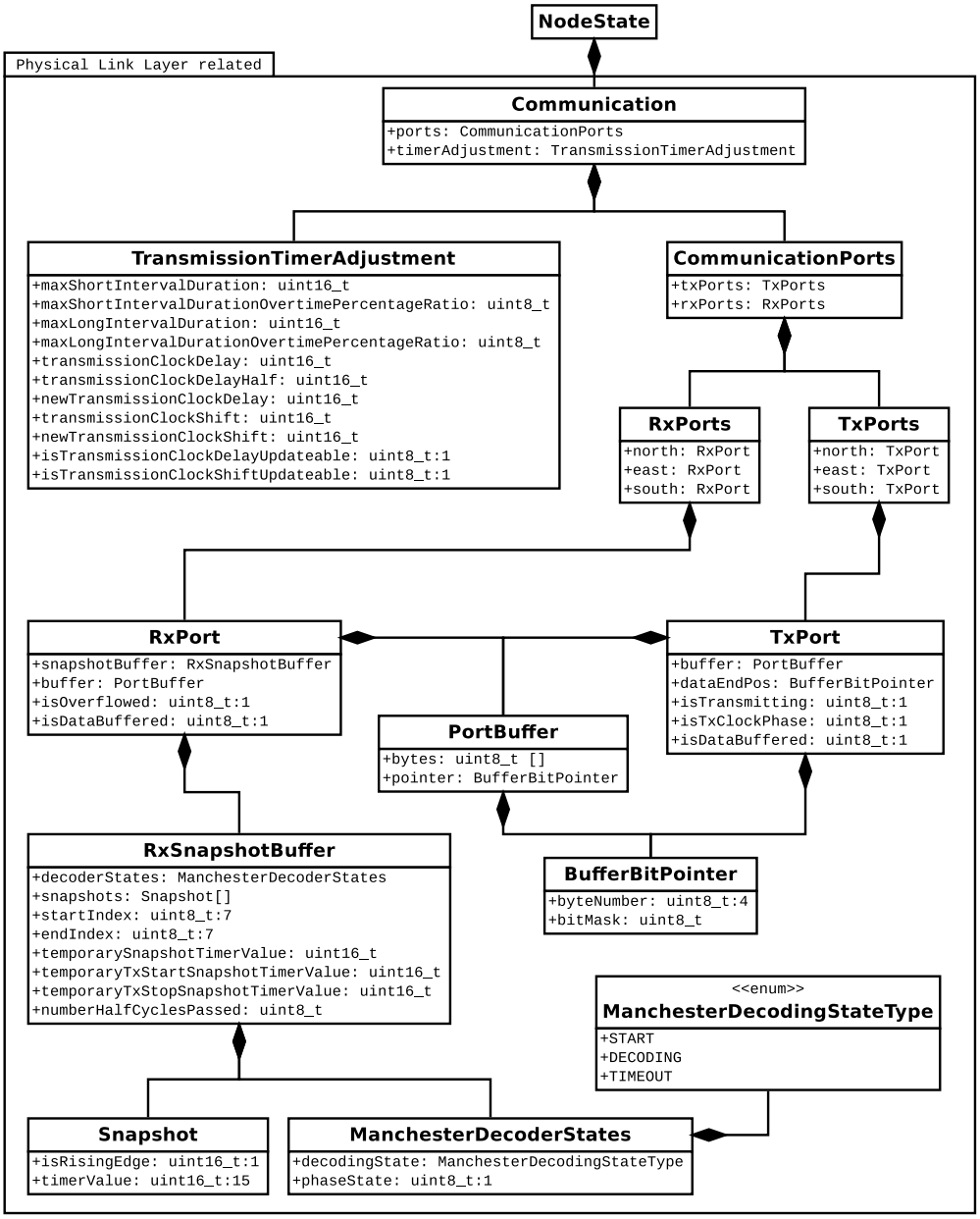


Fig. E.2.: node's physical link layer context

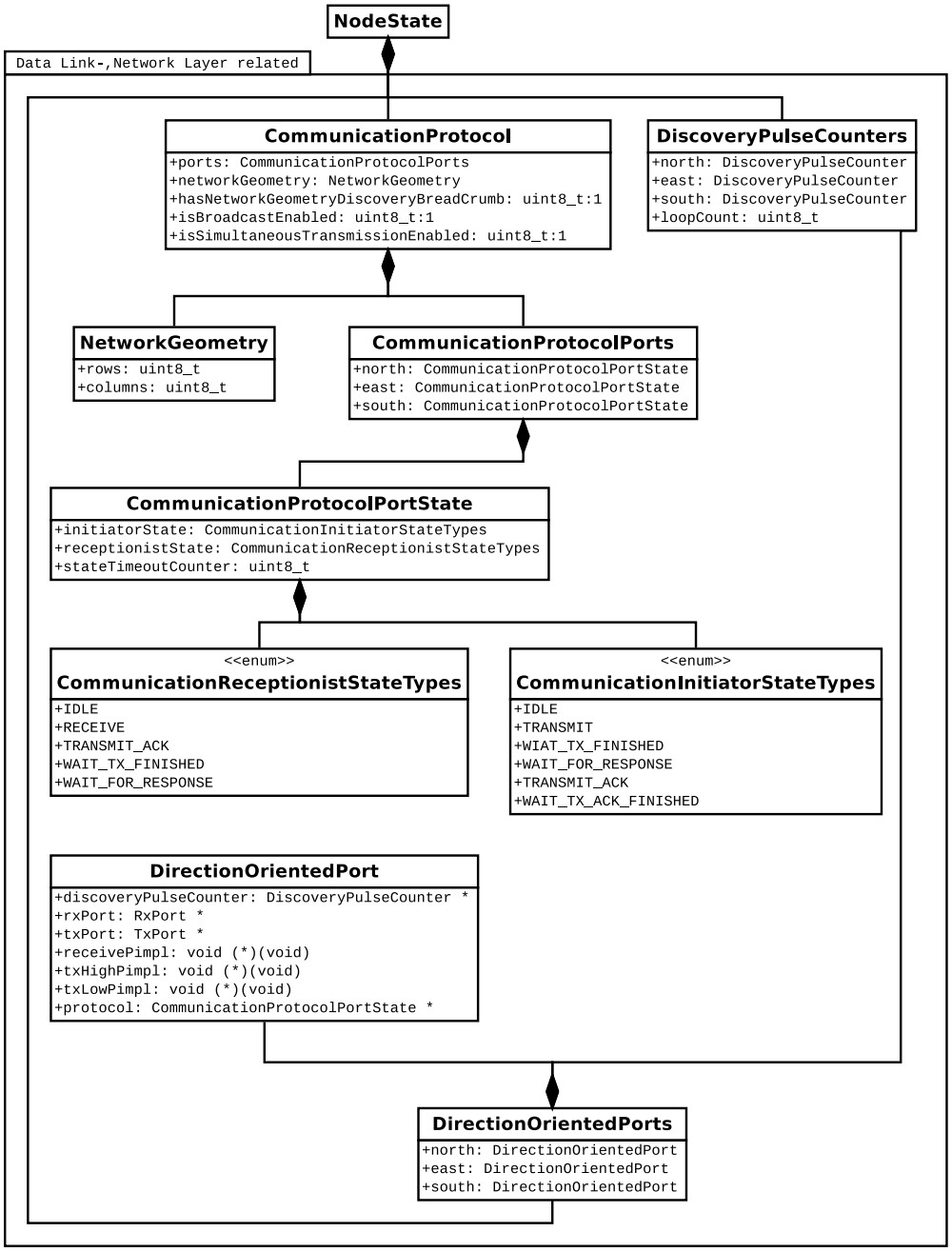


Fig. E.3.: node's data link layer and network layer context

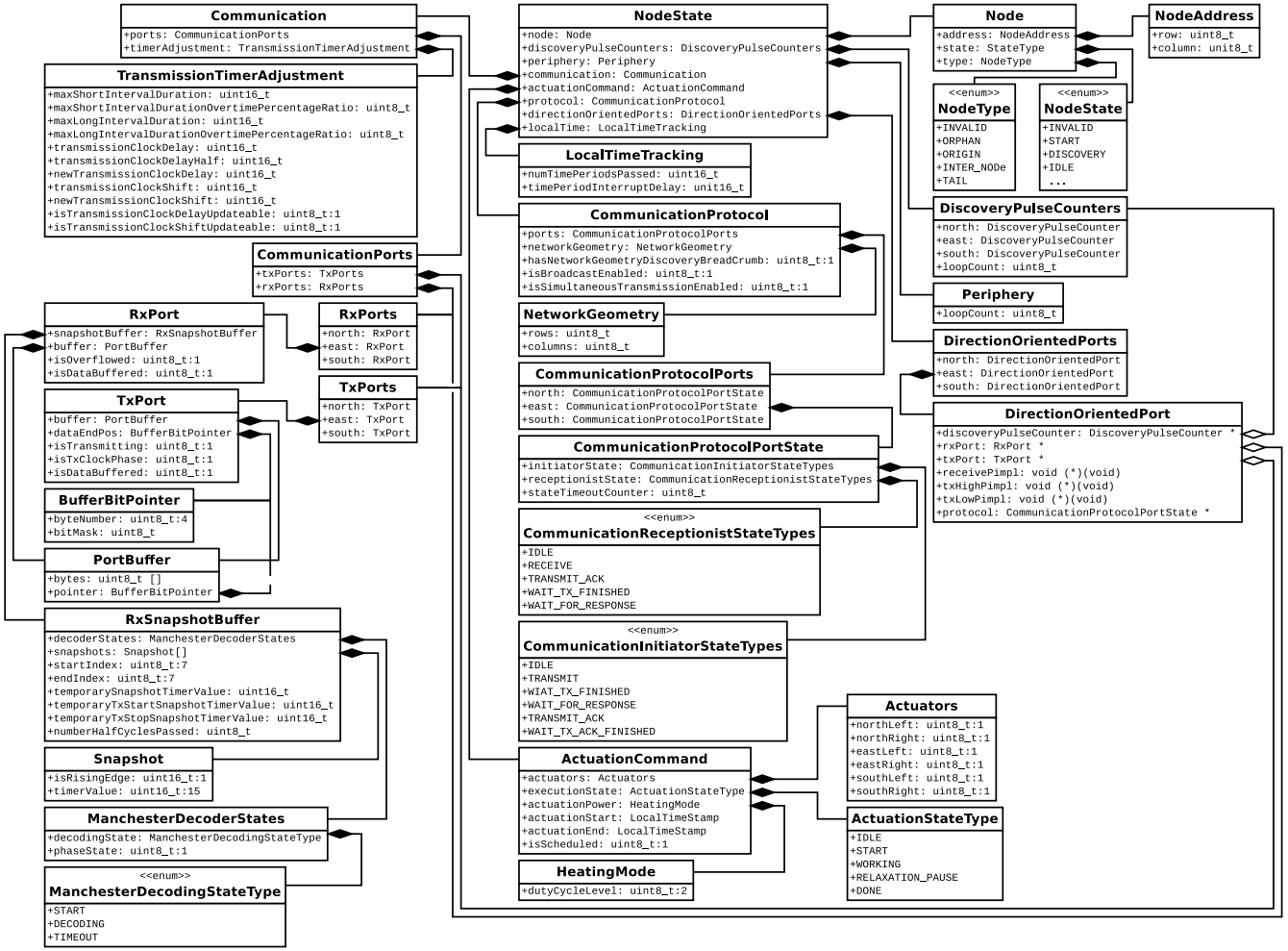


Fig. E.4.: node's detailed context

F. State Diagrams

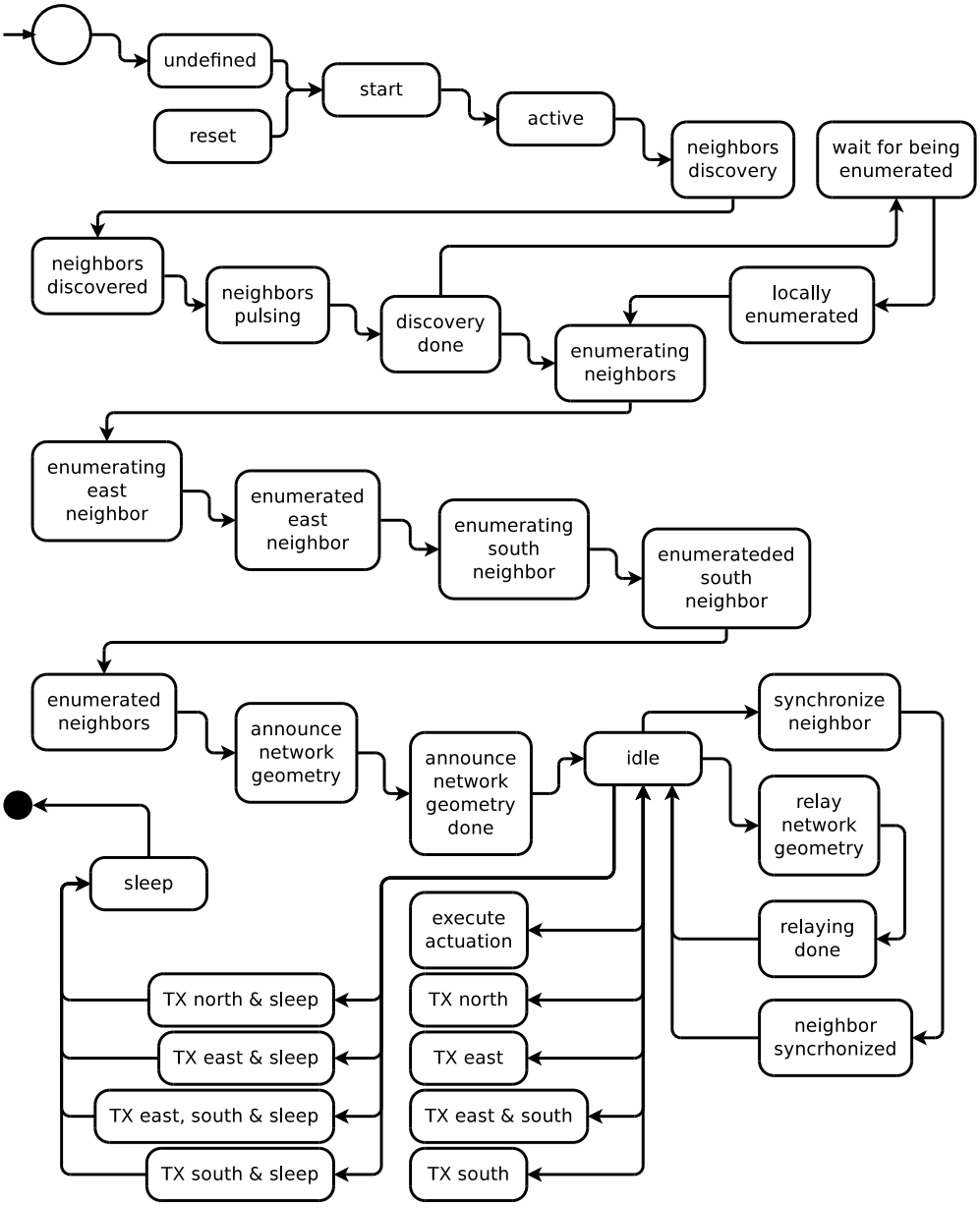


Fig. F.1.: node's finite state machine (FSM) states

G. Configuration Parameter Listing

update according to last implementation

update according to last implementation

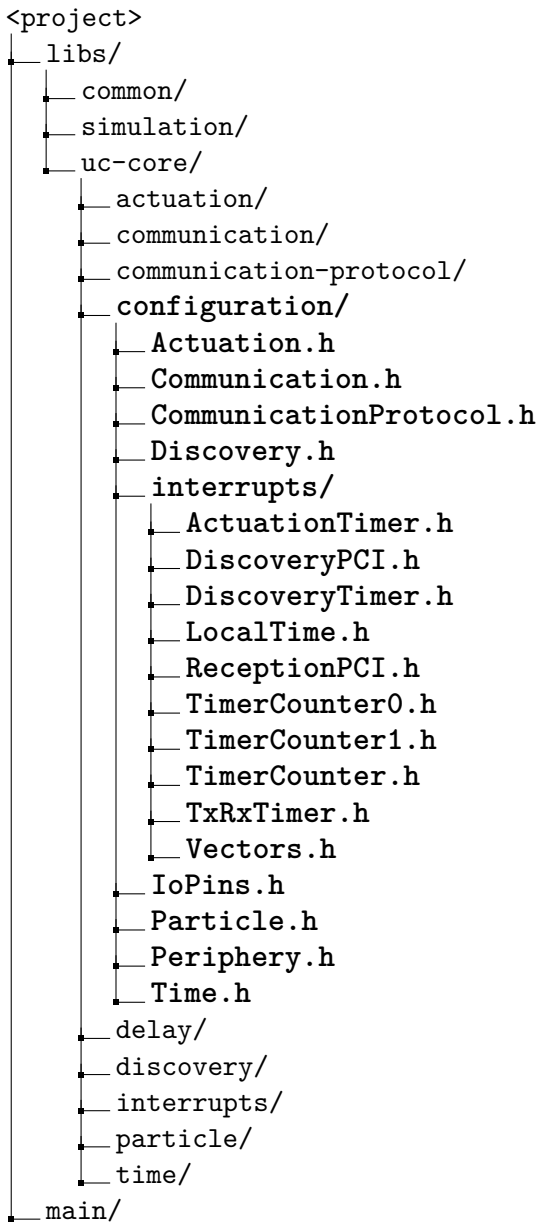


Fig. G.1.: configuration files structure

Parameter	Default Value	File
ACTUATION_COMPARE_VALUE_POWER_STRONG	$((UINT8\_MAX/4)*3)$	Actuation.h
ACTUATION_COMPARE_VALUE_POWER_MEDIUM	$(UINT8\_MAX/2)$	Actuation.h
ACTUATION_COMPARE_VALUE_POWER_WEAK	$(UINT8\_MAX/4)$	Actuation.h
DEFAULT_TX_RX_CLOCK_DELAY	$((uint16\_t)1024)$	Communication.h
DEFAULT_MAX_SHORT_RECEPTION_OVERTIME_PERCENTAGE_RATIO	$((uint8\_t)59)$	Communication.h
DEFAULT_MAX_LONG_RECEPTION_OVERTIME_PERCENTAGE_RATIO	$((uint8\_t)112)$	Communication.h
TX_RX_NUMBER_BUFFER_BYTES	$((uint8\_t)9)$	Communication.h
RX_NUMBER_SNAPSHOTS	$((uint8\_t)28)$	Communication.h
COMMUNICATION_PROTOCOL_TIME_SYNCHRONIZATION_PER_NODE_INTERRUPT_LAG	$((uint16\_t)0)$	CommunicationProtocol.h
COMMUNICATION_PROTOCOL_TIME_SYNCHRONIZATION_PACKAGE_RECEPTION_DURATION	$((uint16\_t)57320)$	CommunicationProtocol.h
COMMUNICATION_PROTOCOL_TIME_SYNCHRONIZATION_PACKAGE_EXECUTION_LAG	$((uint16\_t)2827)$	CommunicationProtocol.h
COMMUNICATION_PROTOCOL_TIME_SYNCHRONIZATION_MANUAL_ADJUSTMENT	$((uint16\_t)2346)$	CommunicationProtocol.h
RX_DISCOVERY_PULSE_COUNTER_MAX	$((uint4\_t)10)$	Discovery.h
MIN_NEIGHBOURS_DISCOVERY_LOOPS	$((uint8\_t)(50))$	Discovery.h
MAX_NEIGHBOURS_DISCOVERY_LOOPS	$((uint8\_t)(100))$	Discovery.h
MAX_NEIGHBOUR_PULSING_LOOPS	$((uint8\_t)(254))$	Discovery.h
DEFAULT_NEIGHBOUR_SENSING_COUNTER_COMPARE_VALUE	$((uint16\_t)0x80)$	Discovery.h
__ACTUATOR_COUNTER_PRESCALER_VALUE	$\_\_TIMER\_COUNTER\_PRESCALER\_64$	ActuationTimer.h
__TIMER_NEIGHBOUR_SENSE_PRESCALER_VALUE	$\_\_TIMER\_COUNTER\_PRESCALER\_8$	DiscoveryTimer.h
PARTICLE_DISCOVERY_LOOP_DELAY	$(DELAY\_US\_30)$	Particle.h
PARTICLE_DISCOVERY_PULSE_DONE_POST_DELAY	$(DELAY\_US\_500; DELAY\_US\_150)$	Particle.h
HEARTBEAT_LOOP_COUNT_TOGGLE	$((uint8\_t)20)$	Periphery.h
LOCAL_TIME_DEFAULT_INTERRUPT_DELAY	$((uint16\_t)57344)$	Time.h

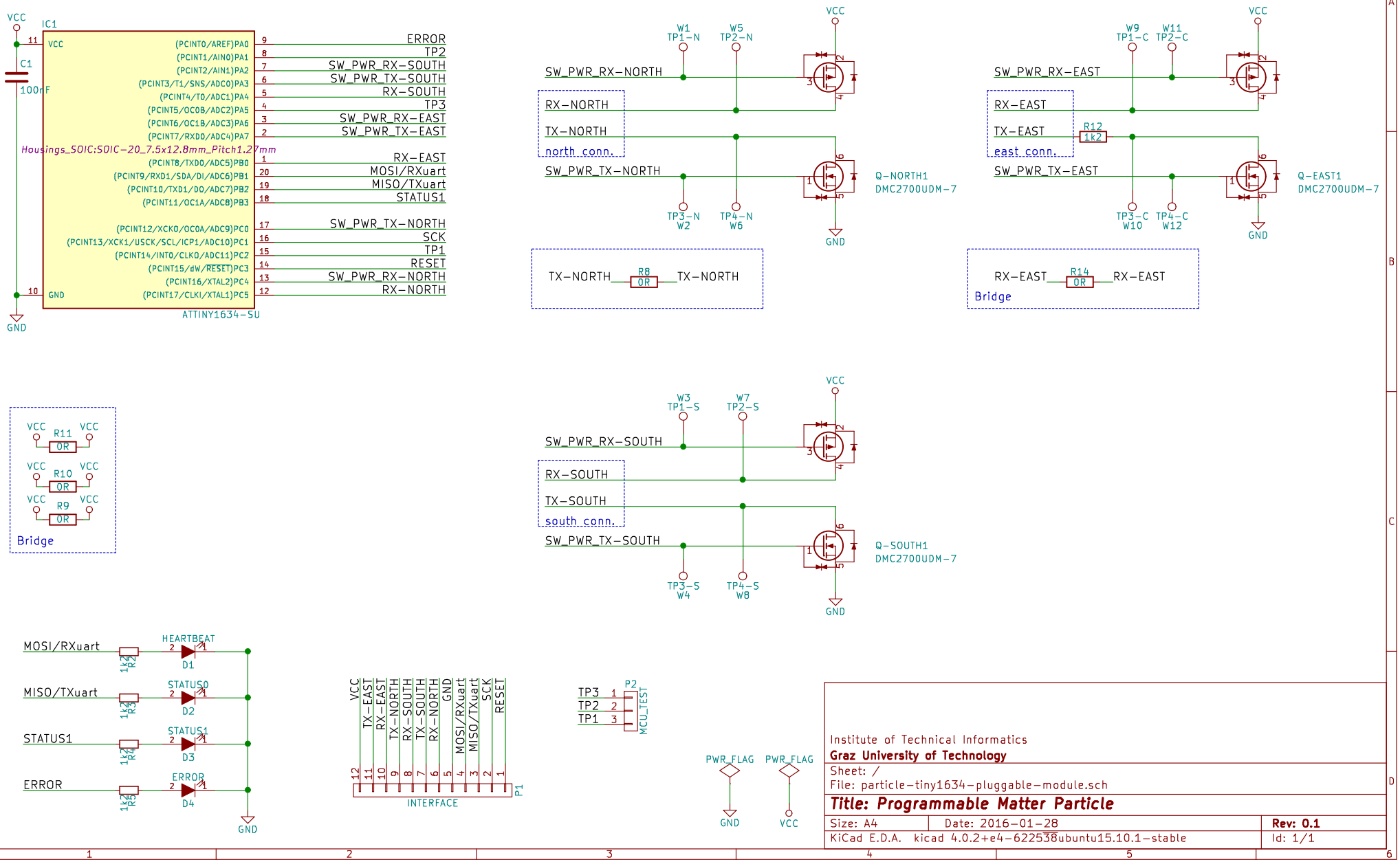
Table G.1.: protocol parameter listing

Parameter	Default Value	Port Pin
__NORTH_TX_PIN	*Pin*0	D 0
__NORTH_TX_DIR	*CDir*	
__NORTH_TX_OUT	*COut*	
__NORTH_TX_IN	*CIn*	
__NORTH_RX_PIN	*Pin*5	C 5
__NORTH_RX_DIR	*CDir*	
__NORTH_RX_OUT	*COut*	
__NORTH_RX_IN	*CIn*	
__NORTH_RX_SWITCH_PIN	*Pin*4	C 4
__NORTH_RX_SWITCH_DIR	*CDir*	
__NORTH_RX_SWITCH_OUT	*COut*	
__NORTH_RX_SWITCH_IN	*CIn*	
__EAST_TX_PIN	*Pin*7	A 7
__EAST_TX_DIR	*ADir*	
__EAST_TX_OUT	*AOut*	
__EAST_TX_IN	*AIn*	
__EAST_RX_PIN	*Pin*0	B 0
__EAST_RX_DIR	*BDir*	
__EAST_RX_OUT	*BOut*	
__EAST_RX_IN	*BIn*	
__EAST_RX_SWITCH_PIN	*Pin*6	A 6
__EAST_RX_SWITCH_DIR	*ADir*	
__EAST_RX_SWITCH_OUT	*AOut*	
__EAST_RX_SWITCH_IN	*AIn*	
__SOUTH_TX_PIN	*Pin*3	A 3
__SOUTH_TX_DIR	*ADir*	
__SOUTH_TX_OUT	*AOut*	
__SOUTH_TX_IN	*AIn*	
__SOUTH_RX_PIN	*Pin*4	A 4
__SOUTH_RX_DIR	*ADir*	
__SOUTH_RX_OUT	*AOut*	
__SOUTH_RX_IN	*AIn*	
__SOUTH_RX_SWITCH_PIN	*Pin*2	A 2
__SOUTH_RX_SWITCH_DIR	*ADir*	
__SOUTH_RX_SWITCH_OUT	*AOut*	
__SOUTH_RX_SWITCH_IN	*AIn*	

Table G.2.: microcontroller unit (MCU) pinout parameter listing of IoPins.h

H. Schematic Diagram

highlight optional east bridge


VCC
IC1
11
VCC
C1
100nF
10
GND
GND
(PCINT0/AREF)PA0
(PCINT1/AIN0)PA1
(PCINT2/AIN1)PA2
(PCINT3/T1/SNS/ADC0)PA3
(PCINT4/T0/ADC1)PA4
(PCINT5/OC0B/ADC2)PA5
(PCINT6/OC1B/ADC3)PA6
(PCINT7/RXD0/ADC4)PA7
Housings_SOIC:SOIC-20_7.5x12.8mm_Pitch1.27mm
(PCINT8/TXD0/ADC5)PB0
(PCINT9/RXD1/SDA/DI/ADC6)PB1
(PCINT10/TXD1/DO/ADC7)PB2
(PCINT11/OC1A/ADC8)PB3
(PCINT12/XCK0/OC0A/ADC9)PC0
(PCINT13/XCK1/USCK/SCL/ICP1/ADC10)PC1
(PCINT14/INT0/CLKO/ADC11)PC2
(PCINT15/dW/RESET)PC3
(PCINT16/XTAL2)PC4
(PCINT17/CLKI/XTAL1)PC5
9
8
7
6
5
4
3
2
1
20
19
18
17
16
15
14
13
12
ERROR
TP2
SW_PWR_RX-SOUTH
SW_PWR_TX-SOUTH
RX-SOUTH
TP3
SW_PWR_RX-EAST
SW_PWR_TX-EAST
RX-EAST
MOSI/RXuart
MISO/TXuart
STATUS1
SW_PWR_TX-NORTH
SCK
TP1
RESET
SW_PWR_RX-NORTH
RX-NORTH
ATTINY1634-SU
W1
TP1-N
W5
TP2-N
SW_PWR_RX-NORTH
RX-NORTH
TX-NORTH
north conn.
SW_PWR_TX-NORTH
TP3-N
W2
TP4-N
W6
Q-NORTH1
DMC2700UDM-7
TX-NORTH
R8
0R
TX-NORTH
W9
TP1-C
W11
TP2-C
SW_PWR_RX-EAST
RX-EAST
TX-EAST
R12
1k2
east conn.
SW_PWR_TX-EAST
TP3-C
W10
TP4-C
W12
Q-EAST1
DMC2700UDM-7
RX-EAST
R14
0R
RX-EAST
Bridge
R11
R10
R9
0R
Bridge
W3
TP1-S
W7
TP2-S
SW_PWR_RX-SOUTH
RX-SOUTH
TX-SOUTH
south conn.
SW_PWR_TX-SOUTH
TP3-S
W4
TP4-S
W8
Q-SOUTH1
DMC2700UDM-7
MOSI/RXuart
MISO/TXuart
STATUS1
ERROR
1k2
R1
R2
R3
R5
HEARTBEAT
STATUS0
STATUS1
ERROR
D1
D2
D3
D4
GND
VCC
TX-EAST
RX-EAST
TX-NORTH
RX-SOUTH
TX-SOUTH
RX-NORTH
GND
MOSI/RXuart
MISO/TXuart
SCK
RESET
12
11
10
9
8
7
6
5
4
3
2
1
P1
INTERFACE
TP3
TP2
TP1
P2
MCU_TEST
PWR_FLAG
PWR_FLAG
GND
VCC
Institute of Technical Informatics
Graz University of Technology
Sheet: /
File: particle-tiny1634-pluggable-module.sch
Title: Programmable Matter Particle
Size: A4
Date: 2016-01-28
Rev: 0.1
KiCad E.D.A. kicad 4.0.2+e4-622538ubuntu15.10.1-stable
Id: 1/1

I. Network Visualization

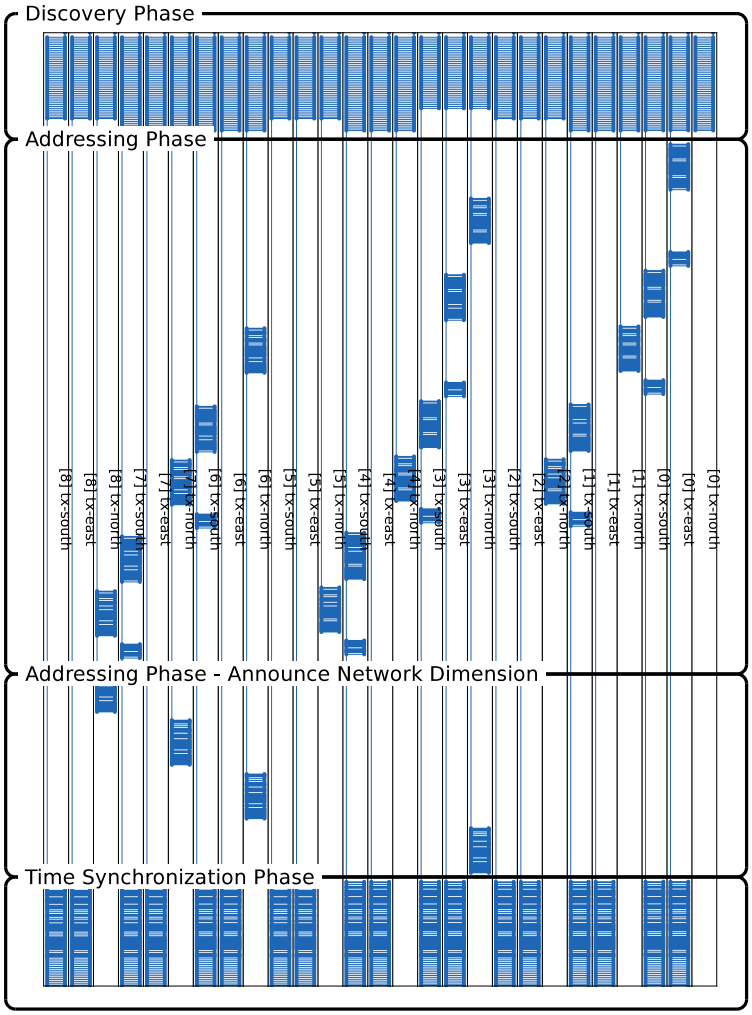


Fig. I.1.: (3×3) network visualization - communication signals

J. Evaluation

Node	**Wire**	**Time** $[ms]$	east-south signal shift $[ms]$	$BCTE_{delay}$ $[\mu s]$	$BCTS_{delay}$ $[\mu s]$
$(1,1)$	E	58.495750359			
	S	58.496000599	0.00025024	7.37496	7.6251
$(2,1)$	E	58.503375559			
	S	58.503625699	0.00025014	6.62524	6.87458
$(3,1)$	E	58.510250939			
	S	58.510500279	0.00024934	6.50062	6.74976
$(1,2)$	E	58.502250979			
	S	58.502500119	0.00024914	5.87543	6.12577
$(2,2)$	E	58.508375549			
	S	58.508625889	0.00025034	7.74972	7.99996
$(3,2)$	E	58.516375609			
	S	58.516625849	0.00025024	7	7.24914
$(1,3)$	E	58.509250979			
	S	58.509500119	0.00024914	6.75087	7.00011
$(2,3)$	E	58.516250989			
	S	58.516500229	0.00024924	7.87544	8.12568
$(3,3)$	E	58.524375669			
	S	58.524625909	0.00025024		
		BCT_{delay} minimum	0.24914	5.87543	6.12577
		BCT_{delay} maximum	0.25034	7.87544	8.12568
		BCT_{delay} **average**	**0.2497844444**	**6.969035**	**7.2187625**

Table J.1.: introduced forwarding delay in broadcast mode (BCT_{delay}) evaluation of (6×6) network simulation with setup C as listed in table 4.3

Particle ID	Nominal $f_{cpu}[MHz]$	Network Address
1	8.178	$(1,1)$
2	8.180	$(2,1)$
3	8.191	$(3,1)$
4	8.211	$(4,1)$
5	8.241	$(5,1)$
6	8.254	$(6,1)$
7	8.279	$(7,1)$
8	8.284	$(8,1)$
9	8.386	$(9,1)$
10	8.303	$(10,1)$
11	8.355	$(11,1)$
12	8.382	$(12,1)$

Table J.2.: nodes' physical enumeration and nominal MCU clock frequency (f_{cpu}) at $VCC = 5.0V$

Network Name	Particle ID Ordre	Network Geometry
$net0$	$\{6\}$	$(1,1)$
$net1$	$\{6,3,9,1,7,4,11,2,10,5,12,8\}$	$(12,1)$

Table J.3.: evaluation networks and order setup

Glossary

1-bit even parity uneven sum of 1-bits results in $PRT = 1$, $PRT = 0$ otherwise
$BCTE_{delay}$ east port forwarding delay
$BCTS_{delay}$ south port forwarding delay
BCT_{delay} introduced forwarding delay in broadcast mode
PDU_{sep} separation between a PDU is received and the corresponding response
PD_{sep} separation between discovery and the subsequent PDU
$\mathcal{N}$ normal distribution
$\mid PDU \mid$ PDU size
$\mid buffer \mid$ buffer size
μ mean
$\oplus$ exclusive or
σ standard deviation
d_{cc_shift} shift in between t_{cc_tx} and t_{cc}
d_{ctor_intp} time span since remote PDU construction until local execution
$d_{discrete}$ integer discretization delay
d_{intp} interpreter delay
d_{pdu} PDU transmission duration
$d_{xmission}$ TX/RX clock cycle delay
$f_{actuator}$ actuator frequency
f_{cpu} MCU clock frequency
$f_{discovery}$ discovery signal frequency
$f_{xmission}$ TX/RX clock frequency
$net0$ network configuration setup 0
$net1$ network configuration setup 1
$process()$ main process
t_{cc} local time ISR compare value, equation (3.12)
t_{code} data clock duration of the Manchester code
t_{const} constant duration
t_{ctor} time of PDU construction

$t_{duration}$ command duration
t_{fwd} signal forwarding duration
t_{hop} signal latency of one hop
t_{int} time when the PDU is interpreted
$t_{latency}$ total signal latency
t_{now} current local time
t_{pcif} pin change interrupt flag latency
t_{pre_tx} constructor to transmission delay
t_{sep_tx} transmitter's local time tracking clock delay
t_{sep} local time counter clock delay
t_{start} command start time
t_{tx} transmitter's local time
t_{until_cc} delay until next local time increment
t_{var} variable duration
1-Wire® a communication bus system; 1-Wire is a registered trademark of Maxim Integrated Products, Inc
ACK acknowledgement
AckPackage acknowledgement package
AckWithAddressPackage acknowledgement package with address fields
Actuation.h actuation configuration file
ActuationTimer.h actuation timing configuration file
actuator a SMA wire that contracts when heated; it is also used as communication wire
AnnounceNetworkGeometryPackage automatic response package containing the network geometry
ARQ automatic repeat request
ATtiny AVR microcontroller for applications that need performance but a small package
ATtiny1634 ATtiny MCU with 16kB flash and 1kB SRAM
avr-gcc AVR C compiler
avrdude driver program for simple Atmel AVR MCU programmer
Avrora AVR simulation and analysis framework http://compilers.cs.ucla.edu/avrora/
B network discovery breadcrumb flag
baud rate the rate at which the signal changes
BCT broadcast bit
Bi-ϕ-L bi-phase-level

bit bang software driven input/output to emulate an interface
bit-oriented a bit-oriented protocol framing is not bound to byte boundaries
broadcast sending to all network participants
broadcast mode if BCT is set signals are passed through
C-Style cast data type conversion from one type into a different type
chain a sequence of connected particles using north or south connection ports
CMake cross-platform build tool
CMD command id
COL address column
COL1 top left address column
COL2 bottom right address column
COLS network columns
Communication.h communication and line coding configuration file
CommunicationProtocol.h protocol timing configuration file
CSMA carrier sense multiple access
CTC clear timer on compare match mode
daisy chain sequential wired network participants without loops
data link layer defines flow control and error detection, also termed layer 2
DDD data display debugger https://www.gnu.org/software/ddd/
decoder line code de implementation
directed in-tree if any unique path from a network node to a given node s is a directed path
directed out-tree if any unique path from the given node s to every other network node is a directed path
Discovery.h discovery configuration file
DiscoveryPCI.h discovery ISR configuration file
DiscoveryTimer.h discovery pulse generator timing configuration file
east port particle's right/east TX/RX connection wires
EB end bit
endianness order of multi-byte values
EnumerationPackage package containing the node's address assignment
event triggered depending on simulation time
ExtendedHeaderPackage reserved package CMD for extensibility
FIFO first in first out
flash programmable program memory
flow-control defines the communication sequence mechanism to ensure communication reliability

FSM finite state machine
FU force update local time flag
fuse essential MCU configuration bits/flags
GCC GNU Compiler Collection
GDB GNU Project debugger https://www.gnu.org/software/gdb/gdb.html
global routing-algorithm routing algo. which uses the total knowledge of a network
GND ground
gnu99 the C99 with GNU extensions
HDR package header
HeaderPackage package without address and payload fields
HeatWiresModePackage actuation mode package
HeatWiresPackage actuation command package
HeatWiresRangePackage actuation command package referring to a rectangular range
initiator the node initiating a transmission
inline the keyword that indicates to duplicate function code rather than translate to function calls
inter head any chain's 1^{st} particle having also the east port connected
inter node a particle between 1^{st} and last particle of a chain
interpreter associates the CMD field value to executive implementation
interrupt response time latency between PCIF is set and ISR execution
IoPins.h pinout configuration file
ISR interrupt service routine
jitter time variation in a series of time intervals
JUnit testing Java unit testing framework
L left wire flag
LED light emitting diode
LED light emitting diode
line code method of coding data on a transmission line
Little-Endian least significant byte stored at lowest address
LSB least significant bit or byte
M heating mode
Make GNU make utility to maintain groups of programs
Manchester code line code incorporating data and clock in one signal, also termed PE
Marshalling transformation of data to a transport-able format

master device a device sending commands to the network
Maven Java build manager
MCU microcontroller unit
MLS Moving Least Squares
MOSFET metal-oxide-semiconductor field-effect transistor
MTU maximum transfer unit
multicast sending to group of network participants
network layer defines package routing, also termed layer 3
node network participant, usually referred to as particle
node indegree number of incoming connections
NodeState the global node context structure
north port particle's upper/north TX/RX connection wires
NRZ non return to zero
NRZ-L NRZ-Level
online algorithm that processes piece-by-piece
OOP object oriented programming
origin node top most, left particle having at least one connection at the east port or south port
orphan node particle without any connection
OSCCAL internal RC oscillator calibration register
OSI Open System Interconnect
OSSRH Open Source Sonatype Repository Hosting
P2P peer-to-peer
particle a shape shifting chain link
particle monitor monitor watching and reporting events of the extended particle platform
particle platform particle PCB simulator abstraction
Particle.h particle loop configuration file
ParticleSimulation class particle simulation implementation for the Avrora framework
PCB printed board circuit
PCI pin change ISR
PCIF pin change interrupt flag
PCM pulse-code moudlation
PDU protocol data unit
PE phase encoding
Periphery.h non vital periphery configuration file

PHY physical layer
physical link layer defines voltage level and wiring, also termed PHY or layer 1
pin change interrupt timing latency between pin change until PCIF is set
port particle's TX/RX connection wires
probe triggered when a particular location in the program is reached
programmer hardware to write the MCU flash
protocol stack set of protocol layers
PRT parity bit
PWM pulse width modulation
R right wire flag
RAM random access memory
RC resistor-capacitor
RC circuit RC circuit
real time protocol a protocol that ensures responses within specific time constraints
receptionist the node receiving a transmission
RelayHeaderPackage automatically forwarded package to north and east port
ResetPackage package to reset a network node
rooted tree rooted tree network is a tree with a specially designated root node
ROV Raw Observation Value
ROW address row
ROW1 top left address row
ROW2 bottom right address row
ROWS network rows
RS-232 standard for serial communication
RTC real time clock
RX reception
RZ return to zero
session layer ensures reliability and automatic recovery, also termed layer 5
SetNetworkGeometryPackage package stating a new network geometry
SimulAVR simulator for the Atmel AVR family http://www.nongnu.org/simulavr/
SMA shape memory alloy
SMAV Simple Moving Average

south port particle's bottom/south TX/RX connection wires
spanning tree a spanning sub graph of a graph
SPI serial programming interface
SRAM static RAM
State pattern design pattern to implement behavior changes according to a context
STB start bit
stop-and-wait-protocol simple flow control, the sender waits for ACK after each PDU TX
SyncNetworkTimeHeaderPackage package header causing the origin node to re-synchronize the network time
tail node last particle of a chain
TCNT timer/counter
Time.h local time tracking configuration file
TimePackage synchronization package
TP test point
transport layer transforms packets to data, also termed layer 4
tree network a connected network that contains no cycle
TX transmission
TX/RX transmission/reception
unicast sending to one network participant
unipolar a signal that uses one polarity according to a reference point
Unmarshalling transformation of transport-able format to data
VCC supply voltage
watch triggered when a particular location in the memory is modified
WMA Weighted Movint Averate

Bibliography

[1] H. Ishii, D. Lakatos, L. Bonanni, and J.-B. Labrune, "Radical atoms: Beyond tangible bits, toward transformable materials," *Interactions*, vol. 19, no. 1, pp. 38–51, Jan. 2012, ISSN: 1072-5520. DOI: 10.1145/2065327.2065337. [Online]. Available: http://doi.acm.org/10.1145/2065327.2065337 (cit. on p. 3).

[2] M. Lasagni and K. Römer, "Force model of a robotic particle chain for 3d displays," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, Salamanca, Spain: ACM, 2015, pp. 314–319, ISBN: 9781450331968. DOI: 10.1145/2695664.2695932. [Online]. Available: http://doi.acm.org/10.1145/2695664.2695932 (cit. on p. 3).

[3] ——, "Force-guiding particle chains for shape-shifting displays," *CoRR*, vol. abs/1402.2507, 2014. [Online]. Available: http://arxiv.org/abs/1402.2507 (cit. on p. 4).

[4] G. Song, B. Kelly, and B. N. Agrawal, "Active position control of a shape memory alloy wire actuated composite beam," *Smart Materials and Structures*, vol. 9, no. 5, p. 711, 2000. [Online]. Available: http://stacks.iop.org/0964-1726/9/i=5/a=316 (cit. on p. 5).

[5] B. L. Kelly, "Beam shape control using shape memory alloys.," DTIC Document, Tech. Rep., 1998. [Online]. Available: http://faculty.nps.edu/agrawal/docs/theses/kelly_MS.pdf (cit. on p. 5).

[6] Maxim, *How to power the extended features of 1-wire® devices*, Appnote4255, Maxim Integrated Products, Jan. 2008. [Online]. Available: http://www.maxim-ic.com/an4255 (cit. on p. 6).

[7] ——, *Guidelines for reliable 1-wire networks*, Appnote148, Maxim Integrated Products, Nov. 2001. [Online]. Available: http://www.maxim-ic.com/an148 (cit. on p. 6).

[8] D. Comer, *Computernetzwerke und Internets: Mit Internet-Anwendungen*. München: Pearson Studium, 2002, ISBN: 382737023x (cit. on pp. 7, 9).

[9] B. Sklar, *Digital communications: Fundamentals and applications*. Upper Saddle River, N.J: Prentice-Hall PTR, 2001, ISBN: 0130847887 (cit. on pp. 9, 12, 14).

[10] R. Ahuja, *Network flows: Theory, algorithms, and applications*. Englewood Cliffs, N.J: Prentice Hall, 1993, ISBN: 013617549x (cit. on p. 9).

[11] R. Williams, *Computer systems architecture: A networking approach*. Harlow, England New York: Addison-Wesley, 2001, ISBN: 0201648598 (cit. on p. 11).

[12] G. Coulouris, *Distributed systems : Concepts and design*. Harlow, England New York: Addison-Wesley, 2005, ISBN: 0321263545 (cit. on p. 12).

[13] L. Peterson, *Computer networks: A systems approach*. Amsterdam Boston: Morgan Kaufmann Publishers, 2003, ISBN: 155860832x (cit. on pp. 14, 18).

[14] Atmel, *8-bit atmel tinyavr microcontroller with 16k bytes in-system programmable flash*, ATtiny1634, Rev. 8303H, Atmel, Feb. 2014. [Online]. Available: http://www.atmel.com/images/atmel-8303-8-bit-avr-microcontroller-tinyavr-attiny1634_datasheet.pdf (cit. on pp. 15, 31, 49, 50).

[15] J. Reichardt, *Lehrbuch Digitaltechnik: Eine Einführung mit VHDL*. München: Oldenbourg, 2011, ISBN: 9783486706802 (cit. on p. 17).

[16] U. Hammerschall, *Verteilte Systeme und Anwendungen : Architekturkonzepte, Standards und Middleware-Technologien*. München Boston u.a: Pearson Studium, 2005, ISBN: 3827370965 (cit. on p. 20).

[17] K. Schmaranz, *Softwareentwicklung in C++*. Berlin u.a: Springer, 2003, ISBN: 3540443436 (cit. on p. 20).

[18] J. Kurose, *Computernetze : Ein Top-Down-Ansatz mit Schwerpunkt Internet*. München: Pearson Studium, 2002, ISBN: 3827370175 (cit. on p. 26).

[19] Atmel, *Avr054: Run-time calibration of the internal rc oscillator*, Application Note, Rev. 2563C-AVR-04/08, Atmel, Apr. 2008. [Online]. Available: http://www.atmel.com/Images/doc2563.pdf (cit. on p. 34).

[20] E. Gamma, *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. München Boston u.a: Addison-Wesley, 2004, ISBN: 3827321999 (cit. on pp. 37, 43).

[21] B. L. Titzer and J. Palsberg, "Nonintrusive precision instrumentation of microcontroller software," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '05, Chicago, Illinois, USA: ACM, 2005, pp. 59–68, ISBN: 1-59593-018-3. DOI: 10.1145/1065910.1065919. [Online]. Available: http://doi.acm.org/10.1145/1065910.1065919 (cit. on p. 51).

[22] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, ser. IPSN '05, Los Angeles, California: IEEE Press, 2005, ISBN: 0-7803-9202-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1147685.1147768 (cit. on p. 52).

Index