Raoul RUBIEN, BSc

Daisy Chain Communication Protocol for Chains of Robotic Particles forming Shape-Shifting Displays

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

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Advisor
Dott. Dott. mag. Matteo Lasagni

Institute for Technical Informatics
Graz, December 2016

Affidavit

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, December 15th, 2016 _____ _____

Date Signature

Eidesstattliche Erklärung

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. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz am, 15. Dezember 2016 _____ _____

Datum Unterschrift

Abstract

We present design, implementation and evaluation of a lightweight extensible communication protocol that guarantees synchronization among nodes forming a predefined topology, despite their inaccurate clock source.

The ultimate goal of this project is to support the realization of a Shape-Shifting Display, a mechanical display that is able to approximate $3D$ surfaces. Such a display consists of multiple parallel chains able to change their curvature under software control. Composed of sequentially connected modular "robotic particles" that control the local curvature of the chain, each chain outlines the contour of a corresponding slice of the target $3D$ surface.

As the actuation of the robotic particles needs to be synchronized due to mechanical constraints, a communication protocol is developed to enable the communication among particles (which are the nodes in our system) and to ensure that their actuation is simultaneous.

The specific structure of a Shape-Shifting Display implies a predefined network topology, where sequences of nodes (i.e., chains) connected to a backbone forms a matrix-like layout.

The communication protocol performs an automatic unattended network discovery to provide an unique address to each node immediately after system boot-up. The specific network topology allows us to define different addressing modes particularly suitable to represent the actual spatial arrangement of nodes.

Our contribution includes the development and the implementation of the communication protocol from both the hardware and software point of views. In order to better support design decisions at every step, we also have extended an existing hardware simulator to make it suitable for our specific application. An evaluation of the performance of the communication protocol

is also included to show how the synchronization requirements are fully satisfied. Possible hardware extensions, such as the introduction of sensors, can be added to the system as our protocol natively supports bi-directional communication.

Zusammenfassung

Wir präsentieren Design, Implementierung und Evaluierung eines leichtgewichtigen erweiterbaren Kommunikationsprotokolls, welches es ermöglicht, formbare Knoten einer festgelegten Topologie trotz deren ungenauen Taktquelle zu synchronisieren.

Das grundsätzliche Ziel dieses Projektes ist die Realisierung eines Shape-Shifting Displays, also eines mechanischen Displays, welches in der Lage ist, $3D$ Flächen zu approximieren. Solch ein Display besteht aus mehreren parallelen Ketten, die in der Lage sind, ihre Krümmung mittels Software zu ändern. Jede Kette besteht aus mehreren, miteinander verbundenen modularen "robotischen Partikeln", welche die lokale Krümmung einer Kette bestimmen und stellt den Umriss eines Flächenschnittes einer $3D$ Zielfläche dar.

Da Aktuatoren aufgrund mechanischer Einschränkungen synchron aktiviert werden müssen, wurde ein Kommunikationsprotokoll entwickelt, welches die Kommunikation zwischen den Partikeln (in unserem System als Knoten bezeichnet) ermöglicht und so deren simultane Aktivierung sicherstellt.

Die spezielle Struktur des Shape-Shifting Displays beinhaltet eine vordefinierte Netzwerktopologie, in der mittels eines Backbone sequenziell verbundene Knoten (z.B. Ketten) eine matrixähnliche Form ergeben. Das Kommunikationsprotokoll führt unmittelbar nach dem Hochfahren des Systems autonom eine automatische Netzwerkerkennung aus, um jeden Knoten mit einer eindeutigen Adresse zu versorgen. Die spezielle Netzwerktopologie erlaubt es dabei, bestimmte Adressierungsmodi zu definieren, um die aktuelle räumliche Anordnung der Knoten zu bestimmen.

Unser Beitrag berücksichtigt bei der Entwicklung und Implementation des Kommunikationsprotokolls nicht nur die Belange der Software, sondern auch

der Hardware. Um jeden Schritt der Designentscheidungen besser unterstützen zu können, wurde ein bestehender Hardware Simulator erweitert und so besser an unsere spezielle Applikation angepasst. Eine Performanzanalyse des Kommunikationsprotokolls zeigt schlussendlich, wie die Synchronisationsbedürfnisse befriedigt werden. Hardwareerweiterungen wie beispielsweise Sensoren, können dem System hinzugefügt werden, da unser Protokoll bidirektionale Kommunikation unterstützt.

Contents

List of Figures

List of Tables

List of Tables

1. Introduction

Programmable matter indicates a novel class of materials that are able to change their physical properties under software control. Many different materials and approaches have been researched up to now. Changeable properties are for example shape, volume, viscosity and texture [1]. Programmable matter can be simple or complex material such as alloys or robotic entities. The matter is not necessarily stiff; it can be soft or liquid. A rather old programmable matter technology is the well-known Liquid Crystal Display (LCD). The programmable matter is the liquid crystal that aligns, and thus displays shapes, when electrically actuated. Although nano-scaled, the field does also consider larger scale programmable matter such as architectural robotics, intelligent and adaptive built environments [2] and gives a good example of coarse-grained material. The Animated Work Environment [3], 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-scale robotic entities are applied to achieve different properties. The base of our work are small-scale robotic particles which are used to implement a two-dimensional Shape-Shifting Display [4]. The Shape-Shifting Display is a network of many chains of robotic particles, which are able to form their curvature under software control. A chain consists of particles connected in a daisy chain manner. Hence the communication within a chain can be established to subsequent particles only. By controlling each chain, the display is able to approximate three-dimensional surfaces as illustrated in fig. 1.1.

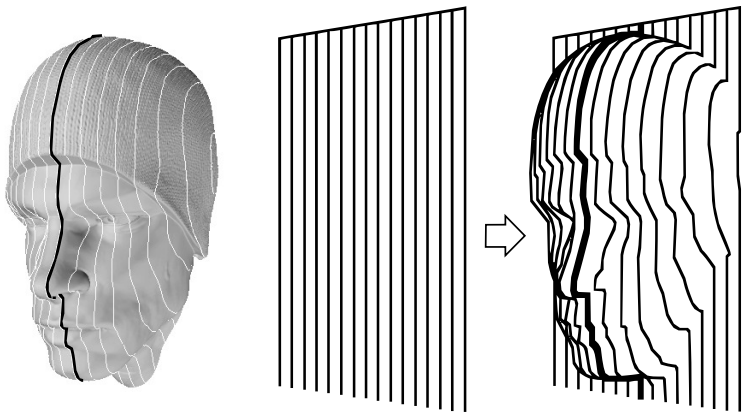


Fig. 1.1.: Shape shifting surface principle [4]; a grid of chains approximates a face structure

1.1. Background

The electrical design of the currently applied shape-shifting chain, as proposed by Lasagni et. al [5], consists of actuators, their electrical drivers and very basic communication controllers embedded in a mechanical body. The controller acts as a bus system participant and exploits the power supply wires as communication channel. However, the electrical and communication design bears many limitations which makes a system redesign necessary and requires a new network protocol for communication.

1.1.1. Mechanical Implementation

To realize chains, particles are connected by two links to each consecutive neighbor. Links, having in total three joints, can be collapsed but reside per default in a locked state. This state can be controlled upon activation of local actuators to unlock. This mechanism in combination with an externally applied force is used to collapse links which leads to folding consecutive particles, affecting the curvature of the chain. Fig. 1.2 illustrates the mechanical principle of a folded chain. When applying multiple chains, they share the

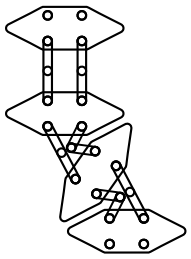


Fig. 1.2.: Folded chain

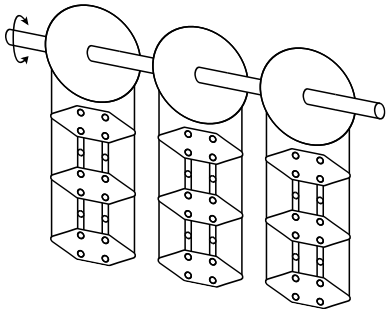


Fig. 1.3.: External force mechanics

same externally applied force. For this reason all chains of a Shape-Shifting Display are physically connected to a global mechanic system, which applies the needed external force by pulling two threads per chain as illustrated in fig. 1.3. Each thread is connected to the chain's end, while it slides through each other particle of the same chain. When applying the force, all chains are compressed. If chains contain unlocked links, they will fold at their location. Because of the mechanical implementation the compression of all chains occurs synchronous. Thus, particle actuation must be synchronous. With that structure and many fine-grained shape-able chains, a high resolution shape shifting surface can be implemented.

1.1.2. Electrical Implementation

The very basic controller located in particles provides a basic communication support 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 [6, 7], which contracts if the wire temperature rises. Two actuators connect consecutive particles. To activate the actuator the SMA wire must be heated, thus powered by consecutive particles.

The power supply of each particle is obtained from a common power line, all chains are connected to. Each chain is attached to the same power line. Particles are connected 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, as detailed in Lasagni et. al [5].

1.2. Limitations

1.2.1. Power Supply

In regular operation mode, particles receive commands and activate actuators in a repetitive manner. For this reason the power supply must be switched among two levels while communicating, and a third level for actuation. During communication the microcontroller's power supply must be buffered parasitic from the 1-Wire® bus. If during communication the network's total power consumption exceeds the 1-Wire® maximum specification, parasitic powering is not sufficient any more. A reliable 1-Wire® application allows a maximum current drain of approximately $2mA$ [8, 9, pp. 2], which does not scale for large particle networks.

1.2.2. Particle Localization

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 the network. To automate this task it is necessary to find all the 1-Wire® IDs and then actuate pairs of consecutive particles to discover their sequences. Since only consecutive particles can enable the connecting actuators, an increase of current consumption indicates their reciprocal connection. Unfortunately the complexity of this approach is $\mathcal{T}(n) = \mathcal{O}(n^2)$.

1.2.3. Unicast Communication

In the network's use case, the 1-Wire® protocol allows only one communication between a master and a particle at a time. This is due to the bus

characteristic, which does not allow local communication between consecutive particles. This means that, while two communication end points are performing their transaction, no other communication can take place in the network.

1.2.4. Concurrent Actuation

Particles having no possibility for local computations must be remotely triggered to actuate. Although by using the unicast bus system, no synchronization is required among particles, the number of concurrent actuations is limited. This implementation does not scale in large networks.

1.2.5. Remote Programming

For firmware development and activation a remote programming must be provided. We understand this procedure as programming 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.

1.3. Motivation

As a consequence of the current electrical implementation, the system comprises severe limitations that must be overcome to be scalable for bigger networks. Overcoming these limitations is our motivation 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 make both, the electrical and mechanical system, more complicated.

The idea is to use the already available actuator wires as communication channel. The actuator wires, having 150Ω per 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

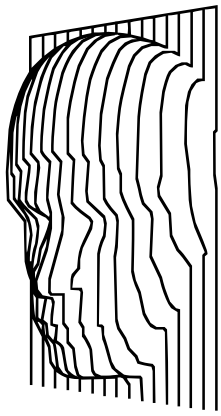


Fig. 1.4.: Folded shape shifting surface approximating a face shape

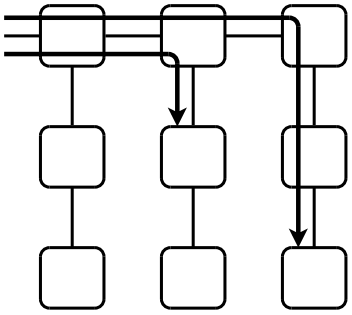


Fig. 1.5.: Network structure's communication paths example

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 exploiting actuators for both, actuation and communication. In terms of communication, the proposed network structure consists of daisy chain connected particles, see fig. 1.5. 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. 1.4 and explained in [4]. The network allows an external communication with the topmost leftmost network particle. Commands passed to this particle, with different destination than the particle itself, are routed accordingly to the next neighboring particle. With this Peer-to-Peer (P2P) [10, pp. 120] communication technique particles require to know where to relay transmissions. For that reason the particle's awareness of position in network and connectivity to neighbors is necessary.

1.4. Contribution

We propose a new protocol for networks of robotic chains connected linearly to a backbone to overcome the stated limitations. The protocol implementation relies on our preliminary work (attached in appendix section A), which proposes a new electrical implementation of the robotic particle without additional wires or changing the chain structure.

2. Protocol Design

In this chapter we elaborate the protocol design decisions with the focus on what are the requirements to be achieved.

The protocol design relies on our previous work, (appendix section A); a new network design to overcome the limitations of the current implementation has been proposed. The network design provides three communication ports for any particle, each consisting of transmission (TX) output and reception (RX) input, as illustrated in fig. 2.5. From the protocol's viewpoint a particle represents a node acting as network participant. The communication ports 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 [11, pp. 156] connection. The two wire connection of subsequent particles allows a lightweight full-duplex communication without the need of Carrier Sense Multiple Access (CSMA) [11, pp. 708] as the communication bus is shared by only two nodes. The network topology of chains connected linearly to a backbone can be described as a subset of an unweighted rooted tree with the left most, topmost node as root [12, pp. 24]. For simplicity we use the more general term, rooted tree, in the upcoming work. 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 [10, 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 as illustrated in fig. 2.1.

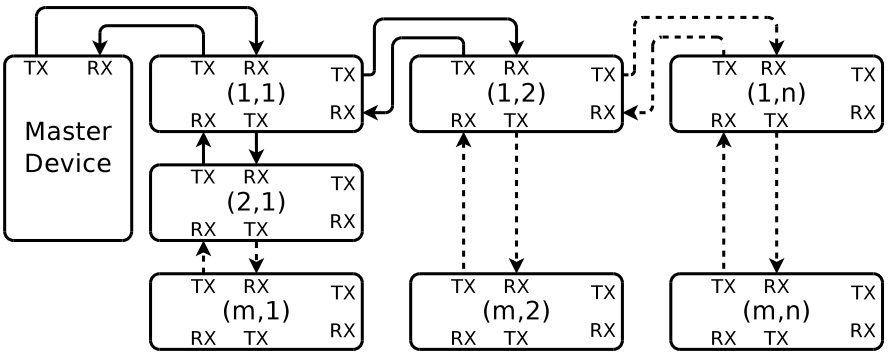


Fig. 2.1.: Proposed network structure; each subsequent chain is connected at the first particle to the previous chain; any node provides three connection ports, each consisting of transmission (TX) and reception (RX)

2.1. Requirements and Constraints

For the upcoming protocol development we have given requirements and constraints which we must consider during the whole process. The given requirements, such as network topology, physical link and much more, are adopted from our preliminary work.

Scalability The protocol implementation must be scale-able for big networks, since a high number of small-scale particles are to be addressed in real use cases. We assume an addressing range of $UINT8\_MAX$ rows and columns to be acceptable. With this upper limit 255^2 participants can be addressed.

Small-Scale and Lightweight With regards to physical forces and material stress, long chains of particles are only feasible using a lightweight construction design. By minimizing the particles' dimension the display's resolution can be improved.

Low Price Since particles will be used in large numbers, they need to be cheap. For that reason the amount of electronic parts is reduced to a minimum. Despite electronic parts, for local computations a Microcontroller Unit (MCU) must be applied that provides enough working memory and storage. With regards to the instruction set and capabilities the requirements are low.

Communication Throughput By using a simple MCU without dedicated hardware that implements the physical layer, we need to bit bang the communication. This means signal encoding, transmission, reception and decoding of the Physical Layer (PHY) will be completely software driven. We expect that a sophisticated connection oriented protocol stack, providing automatic error corrections, cannot be realized without harming the throughput. Thus it will cover basic use cases only.

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

Time Synchronization As each particle's clock is fed by the rather unstable internal RC circuit, time synchronization has the goal of providing a global time. 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 Localization The particle localization of the current implementation, a brute force method does not scale in large networks. It must be replaced by an optimized 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. 2.1, particles are identified by their $(row, column)$ coordinate just as the index of a matrix element. 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 Given the large number of particles, possible firmware update must be supported by the new software implementation. Otherwise particles must be reprogrammed manually. For this reason each particle must provide accessible Serial Programming Interface (SPI) connectors, even if mounted in the mechanical body. Because of the size requirement this is not possible and it also disagrees with the low price requirement. A better approach is to program each particle once before mounting them to chains. Later firmware updates must be feasible using the proposed network structure. This means an unattended replication programs the network subsequently after the origin node has been reprogrammed once. This strategy parallelizes

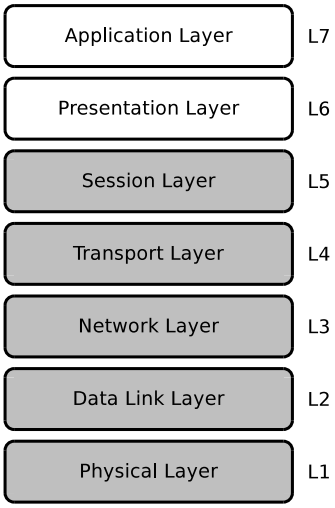


Fig. 2.2.: Open System Interconnect (OSI) layers

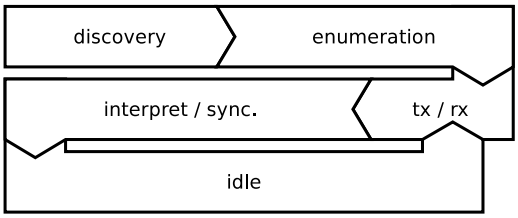


Fig. 2.3.: Protocol process stages

the programming process and skips the need of touching each single particle.

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

Self-Synchronizing Line Code In our preceding work (attached in appendix section A) we considered the pros and cons of applicable line codes. The proposed line code, the Manchester coding, combines both, clock and data into one signal. This transmission/reception method is self-synchronizing and has no need of additional clock wires.

2.2. Design

With respect to the requirements we decided to implement a lightweight bit oriented daisy chain communication protocol that ensures synchronous execution of commands. The protocol process, see fig. 2.3, autonomously detects particle's position in the chain and self-enumerates each network

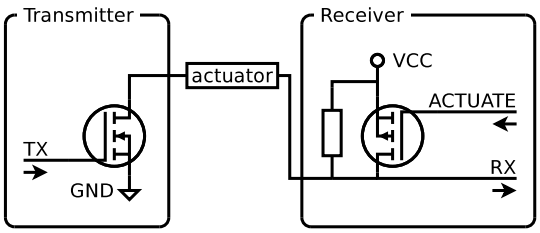


Fig. 2.4.: Simplex Peer-to-Peer (P2P) communication link with complementary MOSFETs at both ends, communication wire is replaced with an actuator, a Shape Memory Alloy (SMA) wire

participant so that after an initialization phase, the network is fully functional and requires only synchronization. The proposed protocol design follows the Open System Interconnect (OSI) [13, pp. 384-386] standard but is to be seen as just a subset of the standard as it does not implement all layers seen in fig. 2.2. The protocol stack [14, pp. 75] implements the physical layer, data link layer, network layer, transport layer and session layer as highlighted in illustration.

To keep track of the upcoming protocol description, highlighted margin notes indicate the belonging to the process or layer described in fig. 2.3 and fig. 2.2. In the following we will depict the corresponding margin note on the page's outer margin.

2.2.1. Physical Layer

Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

Hardware

At the physical layer we specify the electrical circuits and channel coding. For transmitting data an unipolar [11, pp. 131] Pulse-Code Modulation (PCM) is applied. The signal levels are switched between $0V$ and $5V$, which correspond to ground (GND) and supply voltage (V_{CC}).

The transmission/reception (TX/RX) hardware consists of a N-channel MOSFET on the TX side and a P-channel MOSFET on the RX side as shown in

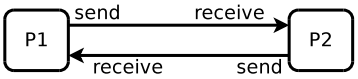


Fig. 2.5.: Full-duplex serial Peer-to-Peer (P2P) connection

fig. 2.4. The receiver pulls up the voltage level on the link wire to V_{CC} using a pull-up resistor. For generating a falling edge on the link the transmitter activates the TX wire, enabling the MOSFET, and thus pulls down the voltage level to GND. A rising edge on the communication line is achieved by pulling the TX wire to GND, which disables the MOSFET. 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, i.e. actuation wires, as shown in fig. 2.5.

Line Code

The physical 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), has been chosen: the Manchester coding. The advantages are the self-synchronizing nature and simplicity with regards to the implementation. Furthermore the Manchester coding does not require additional clock wires or a global clock.

The chosen encoding has some disadvantages. It does not provide error detection without introducing additional error detection bits into the payload [11, pp. 85-90]. Although the hardware allows unipolar signaling (positive voltage levels) only, a transmission must start with the inverted line value. Otherwise a leading edge will be missed. To overcome this issue we assume, that each transmission starts with a constant start bit value.

The Manchester coding belongs to the Return to Zero (RZ) coding group. Thus it needs as wider transmission bandwidth. The signal rate, i.e. the baud rate, is dependent on the data and is up to two times higher [15, pp. 75-78] the bit rate. The RZ group encodes signal levels which return to zero after consecutive bits with the same value. In contrast to RZ, Non Return to Zero (NRZ) does do not return to zero on consecutive bits with the same value. This leads to a lower transmission rate. A comparison among NRZ and RZ using

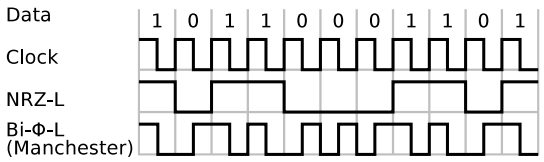


Fig. 2.6.: Non Return to Zero (NRZ) versus Return to Zero (RZ) by means of Manchester coding, also known as Bi-Phase-Level (Bi-ϕ-L)

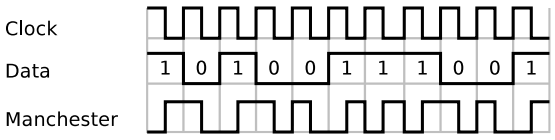


Fig. 2.7.: Manchester coding; the code level is obtained by clock exclusive-or ($\oplus$) data

NRZ-Level (NRZ-L) and Bi-Phase-Level (Bi-ϕ-L), better known as Manchester code, are lined up in fig. 2.6.

The Manchester coding combines data with clock in one signal. The encoding provides a clock transition in every bit interval. At the receiver's side this is used for receiving clock synchronization. The illustration in fig. 2.7 shows the principle of the Manchester coding. Clock and data are merged with the very basic exclusive-or ($\oplus$) operation as expressed in equation (2.1). With the inverse operation of $\oplus$ being $\oplus$ again, the separation of data and clock can be simply formulated as equation (2.2). With this knowledge the receiver just needs to detect the clock of a signal, to reproduce the data. Detailed encoding and decoding examples are provided in equation (2.3) and equation (2.4).

$$manchester = clock \oplus data \tag{2.1}$$

$$data = clock \oplus manchester \tag{2.2}$$

$$\begin{aligned} 101010 &= clock \\ \oplus\ 001100 &= data \\ 100110 &= manchester \end{aligned} \tag{2.3}$$

$$\begin{aligned} 101010 &= clock \\ \oplus\ 100110 &= manchester \\ 001100 &= data \end{aligned} \tag{2.4}$$

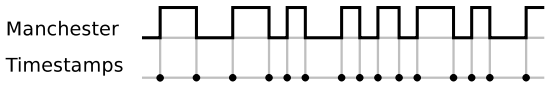


Fig. 2.8.: Manchester coding timestamps of edge occurrences

Implementation

Fortunately the Manchester coding is simple to implement and can be processed relatively fast. For signal de-/encoding we elaborated different bit bang approaches. For any approach we apply the MCU's 16 bit Timer/Counter (TCNT) periphery feature of the ATtiny1634.

Manchester Decoding For both approaches an internal 16 bit TCNT is used. The TCNT is set up to fast Pulse Width Modulation (PWM) mode [16, pp. 80]. When received signals are processed we reference the current TCNT as a timestamp which is used for calculations. Both approaches are time-memory trade-off, different in baud rate but have the same throughput as illustrated in fig. 2.9.

On-the-fly Decoding In this approach the signal is decoded on-line in the Interrupt Service Routine (ISR). Each edge change triggers an ISR where the current edge timestamp is decoded according to the previous edge timestamp (see fig. 2.8). This method does not need any explicit buffer except of the previous timestamp value. 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 coding self-synchronization property for global time synchronization. This method poorly sustains the synchronization needs, because without buffering one can hardly infer timing adjustments, just from the last timestamp. A better approach is to buffer timestamps and decode later.

Post-Processing In this approach a timestamp (see fig. 2.8) when the signal edge occurs is buffered as fast as possible in the corresponding ISR and decoded later. The disadvantage of this method is the large buffer needed to store the signal.
The buffer size ($buffer\_size$) per connection (also denoted as port) depends on the amount of decoded data bytes the protocol should receive

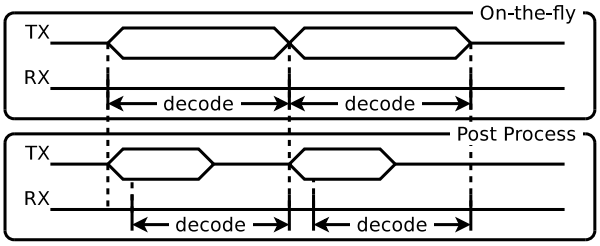


Fig. 2.9.: On-the-fly decoding versus post-processing

at once, the bit rate (two voltage transitions per bit) and the timestamp resolution (16 bit). If the protocol must buffer 9 byte completely before they are decoded, the $buffer\_size$ must be 288 byte (see equation (2.6)). Additionally if a full-duplex communication for all three ports should be achieved, three buffers must be allocated, which in total occupy 864 byte. The calculation can be seen as an upper bound approximation. However the decoding process takes place partially simultaneous to the reception. Therefore the effectively needed buffer can be reduced by about 80% of the calculation above. The reduction is expected to be linearly dependent on the received package size. The package size we denote as protocol data unit size PDU size ($\mid PDU \mid$). The mentioned fraction holds for a $\mid PDU \mid$ of approximately 9 byte.
With about $1kB$ static RAM (SRAM) available on the MCU, this approach is limited but a Protocol Data Unit (PDU) of 9 byte 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.

$$\begin{aligned} \mid buffer \mid &= byte_{rx} \cdot sizeof(uint16\_t) \cdot 8 \cdot transitions \\ &= 9 \cdot 2 \cdot 8 \cdot 2 = 288 \end{aligned} \tag{2.5}$$

$$\begin{aligned} &with \\ byte_{rx} \quad &... \quad number\ of\ received\ bytes\ to\ buffer \end{aligned}$$

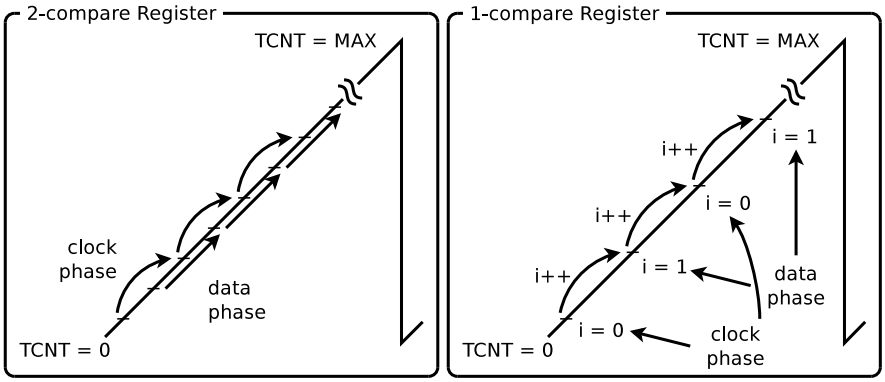


Fig. 2.10.: Signal generator scheduling; two compare register versus one compare register approach

Manchester Encoding 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. 2.10. In both methods the TCNT value is incremented by the MCU's PWM periphery but never changed manually. 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 coding by terms of equation (2.1). The second interrupt rectifies the signal according to the next data to be transmitted (see fig. 2.7). 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 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.

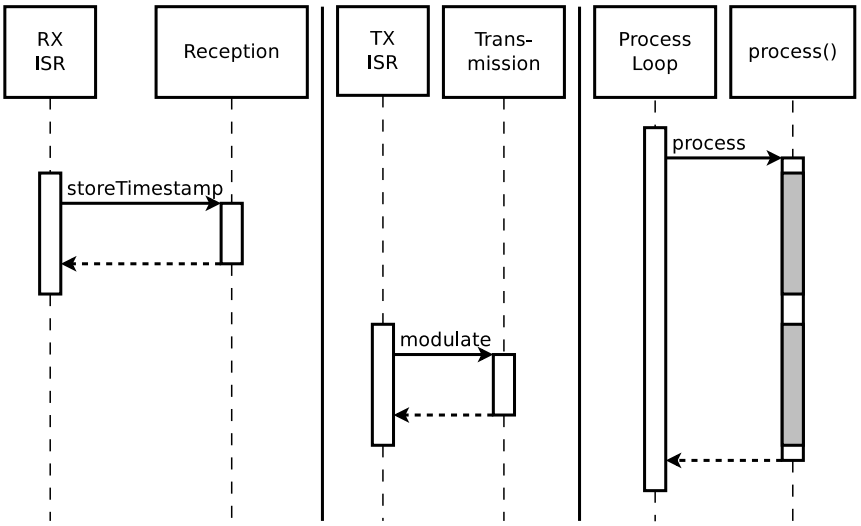


Fig. 2.11.: Reception and decoding sequence diagram; gray highlighted areas are interrupted intervals

Design Decision

We have chosen to use a post-processing decoding method, although it requires a large amount of SRAM [17, pp. 315]. Because buffering is much faster than online decoding, this approach allows a higher communication baud rate. The overall decoding throughput, consisting of both, buffering and decoding, remains the same as with the online method (see fig. 2.9). With this approach we have retrospection into the reception buffer which allows a 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 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. 2.11. Gray highlighted areas are interrupted intervals due to ISR calls.

TX/RX ISRs calls do never overlap.

2.2.2. Data Link Layer

The data link layer is responsible for collecting streams of received bits [15, 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. 2.12. An extensive package description can be found in section B. The HDR field contains start bit (STB), parity bit (PRT) and broadcast bit (BCT). The remaining bit is not used.

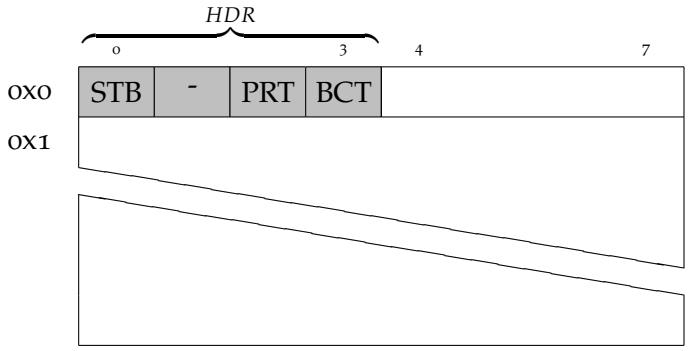


Fig. 2.12.: Header field

STB The start bit (STB) is constantly set to one and necessary for the Manchester coding. Without STB the decoding of a frame starting with a zero would fail, because the first signal edge cannot be detected on the line as described in section 2.2.1.

PRT The parity bit (PRT) is used for error detection of up to 1 bit. To assure an even amount of total ones the PDU's PRT 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 broadcast bit (BCT) bit changes the reception mode. A received package having BCT set affects the subsequent reception only. The following PDU is indeed forwarded to any active connection port. Thus

the signal is broadcasted with a minimum latency. This state remains until the BCT flag is set. 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 minimum 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 [18, pp. 35]. Any data package is constructed as C-Union/Struct [19, pp. 27] which, for transmission, is iterated bit-wise beginning at C-Struct's base address. A little endian byte order example starting with the C-Struct's least significant bit or byte (LSB) is illustrated in fig. 2.14. The corresponding bit stream is illustrated in fig. 2.15.

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

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

Fig. 2.13.: Data structure example

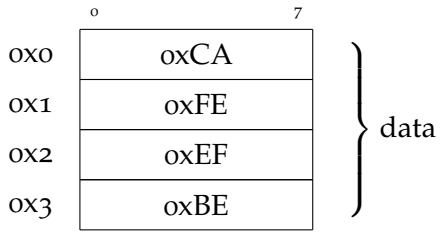


Fig. 2.14.: Little endian data structure on Microcontroller Unit (MCU)

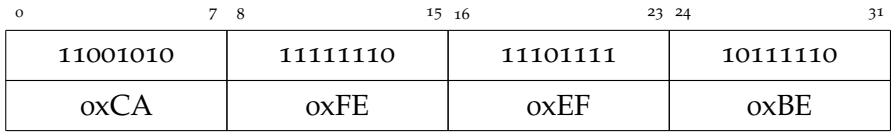


Fig. 2.15.: Transmission bit stream example containing the structure data as explained in fig. 2.13

2.2.3. Network Layer

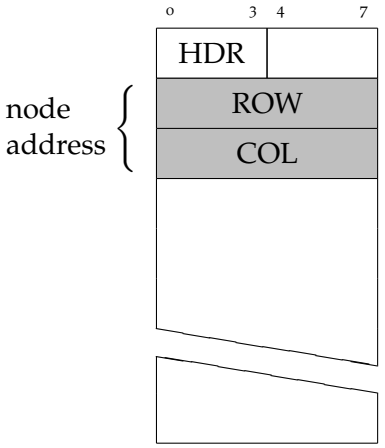


Fig. 2.16.: Unicast package

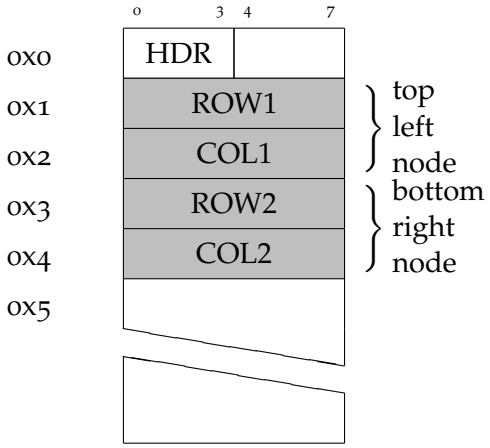
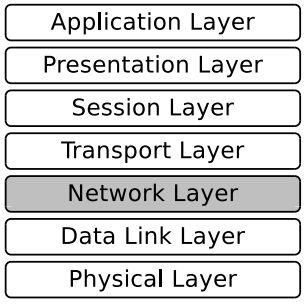
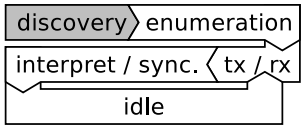


Fig. 2.17.: Multicast package



The network layer is responsible for discovery, 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. 2.16 and fig. 2.17. The number of address fields varies depending on the use case: unicast (one address), multicast (two addresses) or transmission to neighbor (no address).

Discovery



When the system boots up, nodes need to retrieve their position in the network before any address can be assigned. For this reason we introduce a node classification scheme which determines a rough node position, depending to the node's connectivity. Basic types are nodes at the top, within or at the end of chains: head nodes, inter nodes and tail nodes as the illustrated classification matrix in fig. 2.18. An exhaustive description of all types is listed in table 2.1.

The discovery phase starts immediately after nodes are powered. In the discovery phase nodes generate pulse on the transmission wires, which

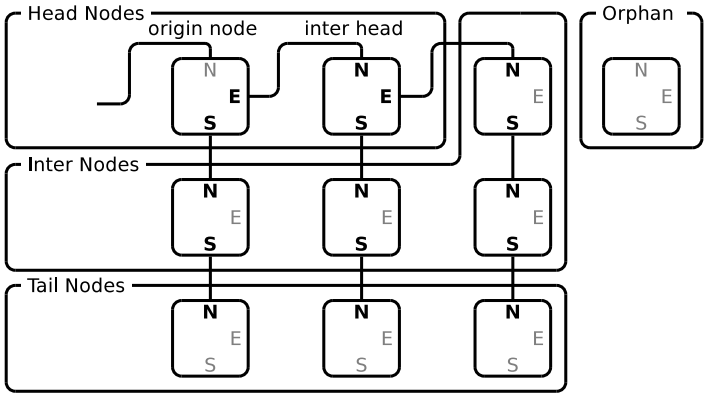


Fig. 2.18.: Node classification matrix highlighting the possible node types: origin node, inter head, inter node, tail node and orphan node

Node Type	Description
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.

Table 2.1.: Listing of classifiable node types

are sensed by neighboring nodes. For proper sensing the origin node (top left 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. 2.19: the counting-phase, classification-phase and post-classification-phase. During the whole counting-phase, incoming pulses are counted for each port separately. If a counter exceeds a specific threshold, the respective 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 in table 2.1.

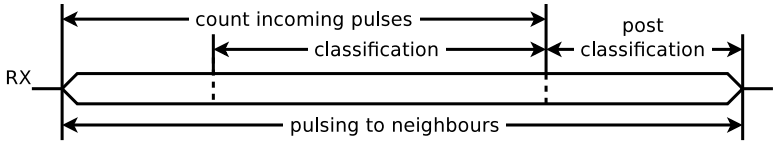
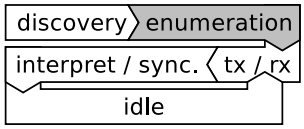


Fig. 2.19.: Discovery phase

Addressing



As addressing scheme we orient on a matrix manner numbering which uses a $(row \times column)$ coordinate for each cell 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 is rectangular, the addressing scheme does not limit network size. The scheme may also be used to address networks having irregular chain lengths such as rooted tree topologies. For simplicity we consider addressing rectangular shaped networks only. Addressing modes 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. 2.16.

Range Addressing Mode 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. 2.17.

Broadcast With this addressing mode messages are forwarded to all nodes. No additional address information is required.

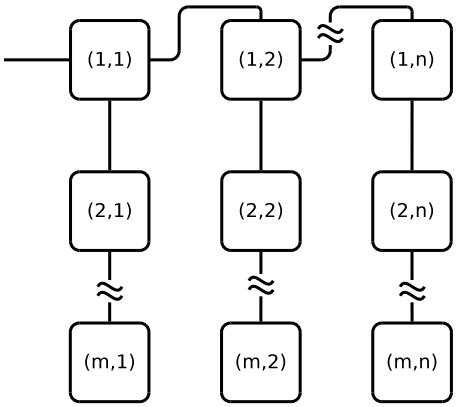


Fig. 2.20.: $(rows \times columns)$ network addressing schema

Routing

Since we have a rather simple, static and well defined network structure, as illustrated in fig. 2.20, we opted for a global routing algorithm. The algorithm needs no further information except the network topology, which is a subset of unweighted rooted tree with the origin node and the node's address. With this definition and the addressing scheme we are able to define the global routing algorithm through a simple set of rules.

According to the package direction we split the undirected network (fig. 2.20) into two directed spanning trees (fig. 2.21 and fig. 2.22). The directed out-tree shows all possible paths from the origin node to the leaf nodes, whereas the directed in-tree 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.

In the following we denote routing rules flowing along directed out-tree paths as $R_{out}(...)$ and $R_{in}(...)$ the rules flowing along directed in-tree paths. The routing rule result is a subset of ports: $\{north, east, south\}$. Generally seen, the algorithm applies $R_{out}(...)$ for forward routing. In case $R_{out}(...)$ does not provide any result backward routing $R_{in}(...)$ is applied.

Unicast Routing In this use case packages carry one address field. An intuitive set of directed out-tree and directed in-tree routing rules is

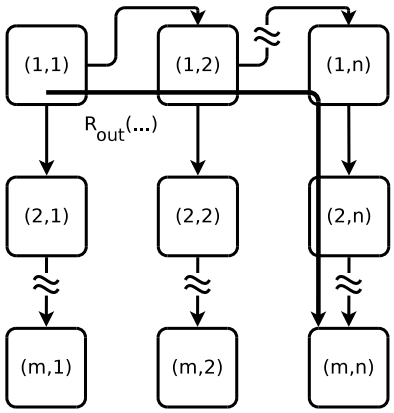


Fig. 2.21.: Directed out-tree highlighting a unicast route example $R_{out}(...)$

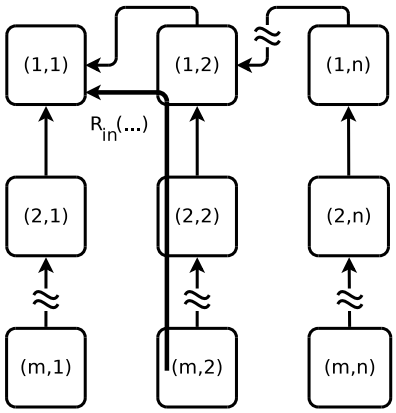


Fig. 2.22.: Directed in-tree highlighting a unicast route example $R_{in}(...)$

described by $R_{out}(...)$ in equation (2.6) and $R_{in}(...)$ in equation (2.7). If the broadcast bit (BCT) is set, no routing is done by this layer. The unicast PDU is illustrated in fig. 2.23.

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

$$R_{out}(...) = \begin{cases} \{\} & if \quad (BCT = 1) \\ \{east\} & elif\ (local.COL < dest.COL) \\ \{south\} & elif\ (local.COL = dest.COL)\ and \\ & \quad (local.ROW < dest.ROW) \\ \{\} & otherwise \end{cases} \tag{2.6}$$

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

$$R_{in}(...) = \begin{cases} \{north\} & if\ (dest.COL < local.COL) \\ \{\} & otherwise \end{cases} \tag{2.7}$$

Range Routing In this case we assume multicast transmissions are issued only by the origin node. Thus packages traverse just the directed out-tree which simplifies the routing rules (equation (2.8)). Backward routing along the directed in-tree is not considered (equation (2.9)).

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. 2.24.

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

$$R_{out}(...) = \begin{cases} \{\} & if \quad (dest2.ROW < local.ROW) \\ \{east\} & elif\ (local.COL < dest1.COL) \\ \{east,\ south\} & elif\ (dest1.COL \leq local.COL)\ and \\ & \quad (local.COL \leq dest2.COL) \end{cases} \tag{2.8}$$

$$R_{in}(local,\ dest1,\ dest2) \mapsto\subset \{north\}$$

$$R_{in}(...) \quad = \quad \{\} \tag{2.9}$$

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. Otherwise the routing rules $R_{out}(...)$ and $R_{in}(...)$ are applied for the corresponding PDUs. Broadcast is spread just along the directed out-tree (equation (2.10)), backward routing is not considered (equation (2.11)). In both use cases no address fields are carried at all.

$$R_{out}(local) \mapsto\subset \{north, east\}$$

$$R_{out}(...) \quad = \quad \{north, east\} \tag{2.10}$$

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

$$R_{in}(...) \quad = \quad \{\} \tag{2.11}$$

When a PDU is passed over the last link it can be transmitted without the destination address, since this field is redundant. The protocol design intends this method of communication to neighboring nodes but this part is intended for future port, thus not implemented. The software design is prepared for this kind of extension. For simplicity packages pass the last link unmodified.

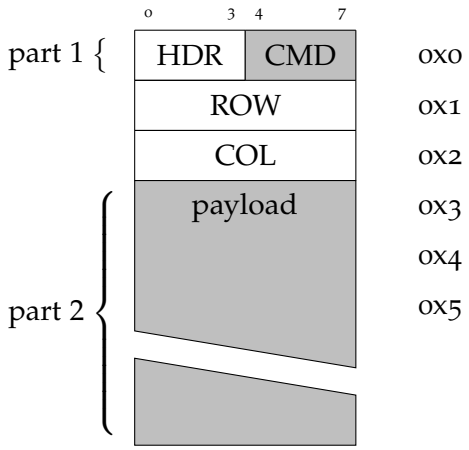


Fig. 2.23.: Unicast Protocol Data Unit (PDU)

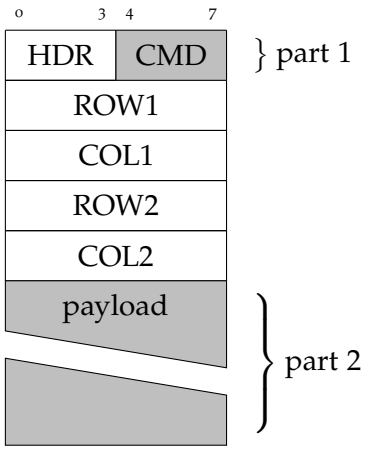


Fig. 2.24.: Multicast Protocol Data Unit (PDU)

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 [20, pp. 501].

2.2.4. Transport Layer

Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical 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 C-Struct's base until the end. This decision requires a proper field alignment within the structure which leads to placing the CMD field into the unused 4 bits in the HDR, while appending the remaining payload as illustrated

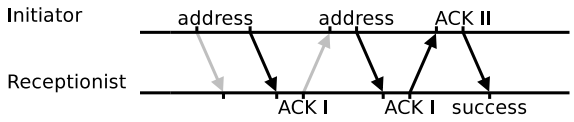


Fig. 2.25.: Flow control in addressing phase, highlighted arrows represent Protocol Data Units (PDUs) followed by reception (RX) timeout

in fig. 2.23 and fig. 2.24. Commands received by this layer are passed to an interpreter which is discussed in the implementation part, section 3.

2.2.5. Session Layer

Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical 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.

Flow Control

A correlation between the node indegree and the discovery process duration has been observed. On tail nodes the discovery phase ends approximately 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 the protocol's flow control implements a stop-and-wait 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.

The stop-and-wait-protocol implementation distinguishes between the node initiating a communication (initiator) and the node waiting for reception

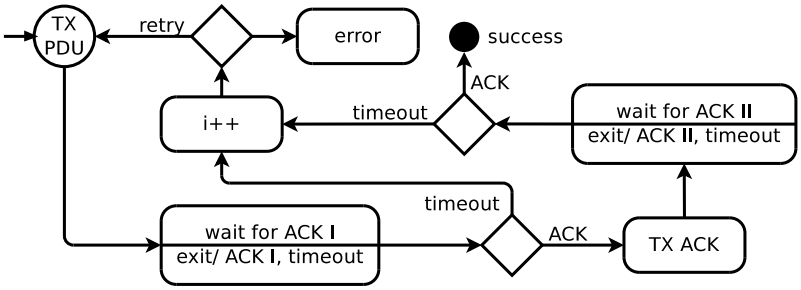


Fig. 2.26.: Initiator transmission (TX) flow control state diagram with *i* being the timeout/retry counter

(receiver). In the addressing phase the initiator sends a package containing the new address to the neighbor. If the receiver receives this package correctly it acknowledges (ACK I) by replying the same address. 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. 2.25 illustrates the flow with transmission errors highlighted in gray. The corresponding Finite State Machines (FSMs) are illustrated in fig. 2.26 and fig. 2.27. Waiting states are interrupted by timeouts which assures the system never remains in a locked state. This process is repeated until a "retry" counter is consumed, which ends up in an error state. The error state is a dead end state. ARQs are implicitly ensured by the described initiator and receiver flow control.

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.

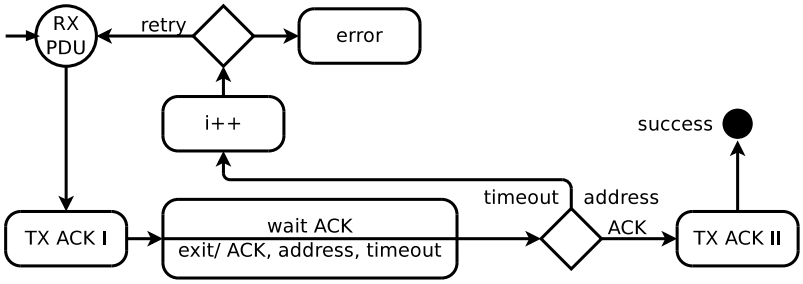
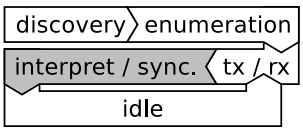


Fig. 2.27.: Receiver transmission (TX) flow control state diagram with *i* being the timeout/retry counter

Synchronization



The protocol's focus is the synchronized execution of commands. Due to the node design does not provide a Real Time Clock (RTC) or 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 challenges, a missing global clock and a possible clock skew.

The underlying timing mechanism is sustained by a timer/counter MCU feature, which allows us to calculate delays in clock cycles. Thus in equations the time base is rather MCU cycles than seconds, if not specified differently.

Regarding communication, during the synchronization phase, we see two possibilities. Packages can be spread through the network simultaneously (broadcast mode) or subsequently (subsequent mode). In simultaneous mode signals are relayed immediately when received, whereas in subsequent mode a Protocol Data Unit (PDU) is received completely until it is retransmitted. In both cases, 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 time" and "end time" of the reception. The access to this information is vital for the whole synchronization process.

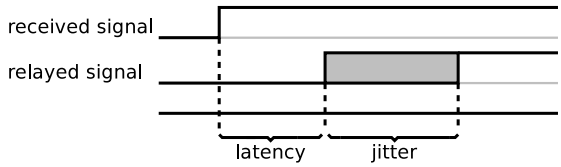


Fig. 2.28.: Received edge versus forwarded edge timing in broadcast mode; forwarded edge is delayed by a constant latency plus an unpredictable jitter

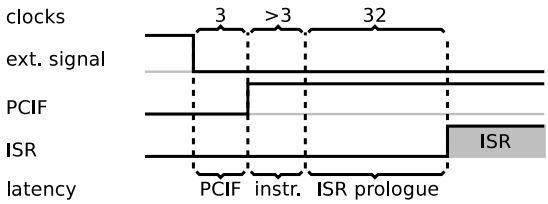


Fig. 2.29.: External pin change Interrupt Service Routine (ISR) latency

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 first relay the signal to any connected port and then they record the timestamp. The relaying process, introduces a minimal latency to signals passing the current node as illustrated in fig. 2.28. 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 For this method the whole transmission delay must be broken 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 ($d_{latency}$). The $d_{latency}$ can be assumed to be linearly dependent on the path length the message traverses. It consists of the latency introduced by the transmitter and the signal latency of one hop (d_{hop}) that the PDU experiences as formulated in equation (2.12) and equation (2.13). In small networks the introduced

latency is neglect-able but it must be considered for larger networks.

$$d_{hop} = d_{pcif} + d_{instr} + d_{prologue} + d_{fwd} \tag{2.12}$$

$$\begin{aligned} d_{latency} &= (-2 + row + column) \cdot d_{hop} + (d_{instr} + d_{prologue} + d_{fwd}) \\ &= (-1 + row + column) \cdot d_{hop} - d_{pcif} \end{aligned} \tag{2.13}$$

The latency d_{hop} can be split in two parts: a constant and a variable 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 (d_{fwd}) as illustrated in fig. 2.29. The pin change interrupt timing until the Pin Change Interrupt Flag (PCIF) is set is guaranteed to be constantly three clocks [16, pp. 50]. The prologue takes constantly 32 clocks thus belongs to the constant duration (d_{const}). The interrupt response time is at least four clocks [16, pp. 12] which corresponds to four clocks for the jump (d_{const}) and a variable duration (d_{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 [16, pp. 278], which introduces a jitter. The encountered unpredictability of d_{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 model is applied to compensate the non-constant part, while instead an empirical value is used. On the TX side we face a similar problem, except for the missing pin change interrupt flag latency (d_{pcif}). However this approach is a far to complex and bears also a cumulative error source. Thus no deep investigations for the synchronization in broadcast mode have been done. For that reason we focus on the subsequent mode.

Subsequent Mode In this mode the synchronization phase takes place while the network is not in broadcast mode. 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.

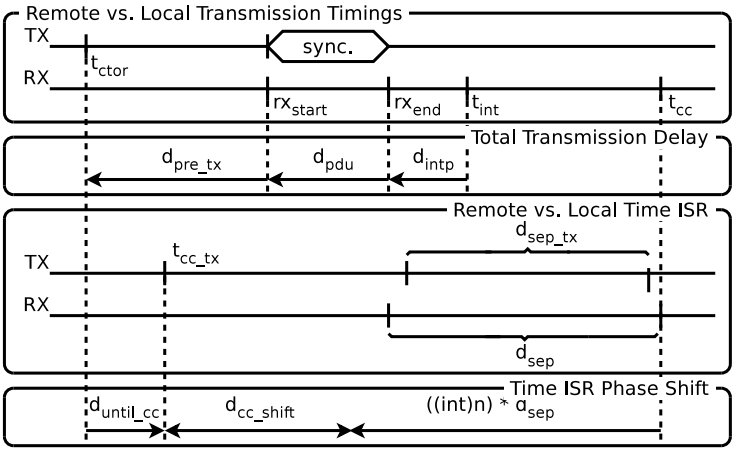


Fig. 2.30.: Time synchronization and phase shift; step-by-step illustration of latencies a Time-Package experiences because it is constructed by transmitter until executed by the receiver

For this method only the transmission 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. 2.29.

Time Synchronization Approach When a synchronization PDU is received we focus on the time span since remote PDU construction until local execution (d_{ctor_intp}) of the package. This delay is the sum of the receiver's the interpreter delay (d_{intp}), PDU transmission duration (d_{pdu}) and a constructor to transmission delay (d_{pre_tx}) as illustrated in fig. 2.30 (Total Transmission Delay: TX time line illustrates the constructor and PDU transmission, RX time line illustrates the receiver's reception, interpreter and local time ISR timings). Thus the time of PDU construction (t_{ctor}) is the time when the PDU is interpreted (t_{int}) minus d_{ctor_intp} (equation (2.14)) as formulated in equation (2.15). By considering these delays we can conclude the amount of local time increments during the PDU

transmission.

$$
\begin{aligned}
d_{ctor\_intp} &= d_{intp} + d_{pdu} + d_{pre\_tx} && (2.14)\\
with & \\
d_{pre\_tx} &\approx 381\mu s \stackrel{\wedge}{=} 3 \cdot d_{code} \\
t_{ctor} &= t_{int} - d_{ctor\_intp} && (2.15)
\end{aligned}
$$

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. 2.30 (Remote vs. Local Time ISR: TX time line illustrates the initiator's local time ISR, RX time line illustrates the receiver's local time ISR). Without shifting we would create a cumulative off-by-one error of the local time among all network nodes. The TimePackage carries data about the transmitter's local time (t_{tx}) when the PDU was constructed, the transmitter's local time clock delay (d_{sep_remote}), the current delay until next local time increment (d_{until_cc}), the force update local time flag (FU) and the end bit (EB). With this information the shift in between t_{cc_tx} and t_{cc} (d_{cc_shift}) can be calculated. For this we take the difference of d_{until_cc} and t_{ctor} as basis. From this delay we take the remainder of the division as formulated in equation (2.16). For this calculation we assume that local time counter clock delay (d_{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 time ISR compare value (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.

$$
\begin{aligned}
d_{cc\_shift} &= (t_{cc} - d_{until\_cc}) \mod d_{sep} && (2.16)\\
d_{until\_cc} &= d_{sep} \cdot \frac{d_{until\_cc\_remote}}{d_{sep\_remote}}
\end{aligned}
$$

The $d_{sep}[ms]$ at a MCU clock frequency (f_{cpu}) of $8MHz$ corresponds to approximately $6.528ms$ as formulated in equation (2.18). Thus the

current local time (t_{now}) overflows every 428 seconds.

$$d_{sep} = 51 \cdot d_{code} \tag{2.17}$$

$$d_{sep}[ms] = \frac{d_{sep}}{f_{cpu}} \tag{2.18}$$

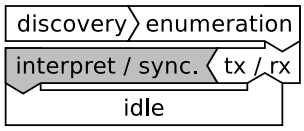
$$t'_{cc} = t_{cc} + \frac{d_{sep}}{2}$$

$$d_{pdu} = d_{code} \cdot pdu\_bits \tag{2.19}$$

$$d_{pdu}[ms] = \frac{d_{pdu}}{f_{cpu}}$$

$$with$$

$$d_{code} = 1024 \tag{2.20}$$



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, V_{CC} 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 the data clock duration of the Manchester code (d_{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} (equation (2.19)) and updates all dependent values such as d_{sep} which holds (equation (2.17)) and the new baud rate by updating d_{code} in equation (2.20). By updating the baud rate the local changes are also exposed to the subsequent neighbor on the next TX.

In the optimal case, if two subsequent neighbors have the same f_{cpu}, d_{sep} resides at about 80% of the maximum value which gives us $\pm 20\%$ margin for adjustments. If even more margin is needed the constant in equation (2.17) can be tuned.

Tuning the internal RC oscillator adjustment as proposed by [21] has also been considered. Although the applied MCU provides an internal RC oscillator

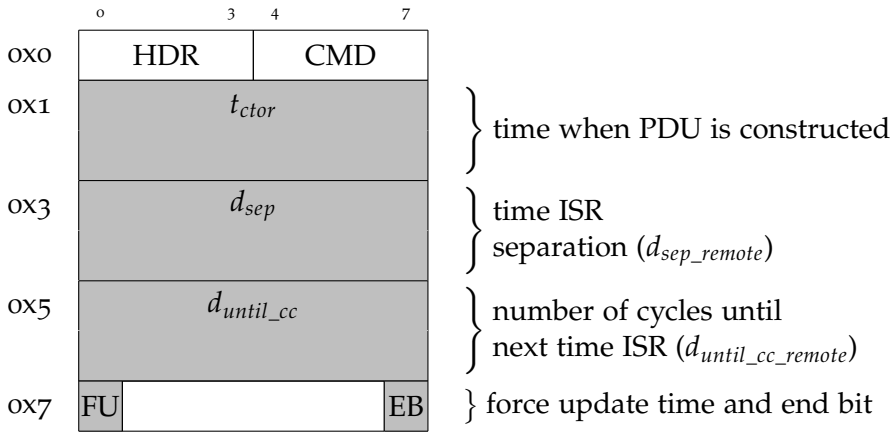


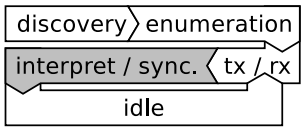
Fig. 2.31.: TimePackage

calibration register (OSCCAL), we do not use this feature to adjust the f_{cpu} since it has two major issues: the tuning possibility is not so fine grained and a linear adjustment of the OSCCAL does not cause a linear f_{cpu} change. Because of these issues no further investigation 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. 2.31. Each received package triggers the clock skew compensation, whereas the time synchronization only if the FU flag is set.

Actuation Command Scheduling



Actuation commands to be scheduled are defined by the command PDU which contains the destination node fig. 2.32 (or range of nodes fig. 2.33), a command start time (t_{start}), the actuation command duration (d) and the

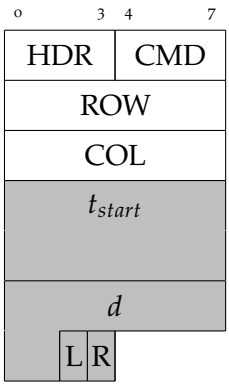


Fig. 2.32.: Actuation command addressing one node

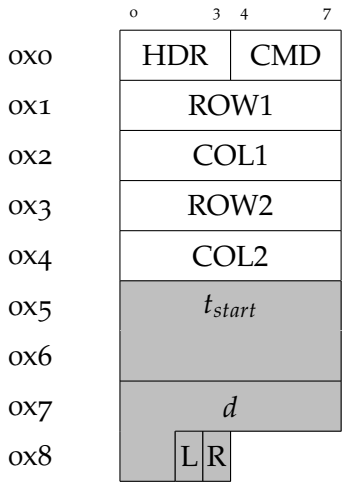


Fig. 2.33.: Range actuation command addressing a node range

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.

3. Implementation

This chapter is focused on the software implementation made to achieve the proposed protocol design and elaborates the following items:

1. overall software structure
2. line encoding and decoding
3. network time synchronization
4. programming code optimization
5. protocol commands
6. protocol configuration
7. simulation and testing

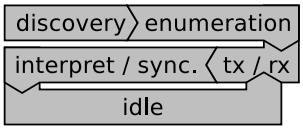
Based on the available flash size we focus at avoiding a high level programming languages such as C++. For the ATtiny1634 MCU we apply the free of charge AVR compiler avr-gcc of the GNU Compiler Collection (GCC) with gnu99 standard. The basic key data of the ATtiny1634 MCU are $16kB$ flash (program memory), $1kB$ SRAM (working memory), one 8 bit and one 16 bit Timer/Counter (TCNT). As clock source the factory calibrated internal oscillator at a frequency of $f_{cpu} = 8MHz$ is used.

To realize the desired protocol behavior the software is designed to be state driven. This eases implementation of context sensitive actions, which are natural for protocols. The node's context corresponds to any possible state the protocol can reach. The implemented Finite State Machine (FSM) is oriented to the State pattern as proposed in [22, pp. 398]. A rough overview of all FSM states and transitions can be found in the appendix in fig. E.1. Although the pattern is proposed for Object Oriented Programming (OOP) we implement the behavior in C language.

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

Fig. 3.1.: Invocation of the *process*() function

3.1. 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. 3.1. In general *process*() is called which checks for executable current tasks, eventually changes node states and finally 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. For timeout implementation we apply counters which are incremented 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. Due to parallel decoding the least reception buffering ratio (*buffering_ratio*) is assumed to be below 10%. For safety reasons twice the expectation value is used. For reception a 28 times $UINT\_16$ buffer containing raw values to be decoded is available (equation (3.1)). In case of decoding, the *process*() consumes data from this buffer. Both buffer types are allocated for each communication port.

$$\begin{aligned} buffer\_size[B] &= \lceil buffering\_ratio \cdot 2 \cdot \mid PDU \mid_{max} \cdot 8 \cdot 2 \rceil \\ &\text{with} \\ buffering\_ratio &= 20\% \\ \mid PDU \mid_{max} &= 9 \end{aligned} \tag{3.1}$$

The FSM's states are sketched in fig. 3.2. They are similar to the protocol stages proposed in fig. 2.3.

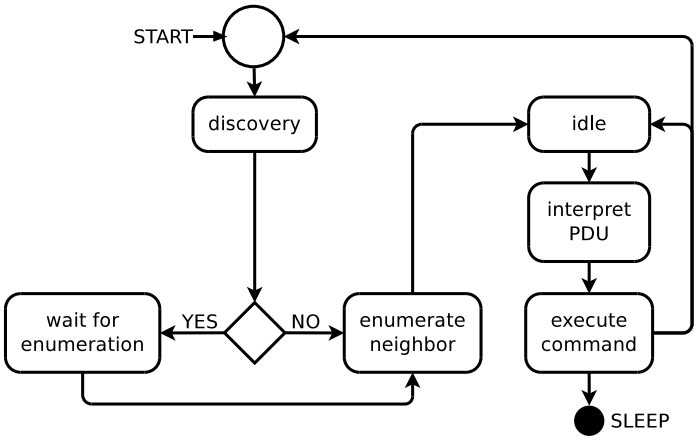
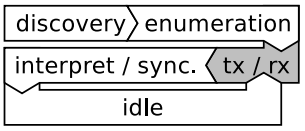


Fig. 3.2.: Main Finite State Machine (FSM) states of the node's context

3.2. Receiver and Decoder

The chosen post-processing decode strategy has been investigated and discussed in 2.2.1. The process consists of a producer and consumer part as illustrated in fig. 3.3.



3.2.1. Receiver

The physical layer reception is handled in the Pin Change ISR (PCI) which is basically capturing any rising or falling edge and storing to a ring buffer. The data produced is the current 16 bit Timer/Counter (TCNT) value whenever the ISR is triggered and the edge direction (rising/falling). Because of limited SRAM memory reasons the LSB of the TCNT is ignored 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.

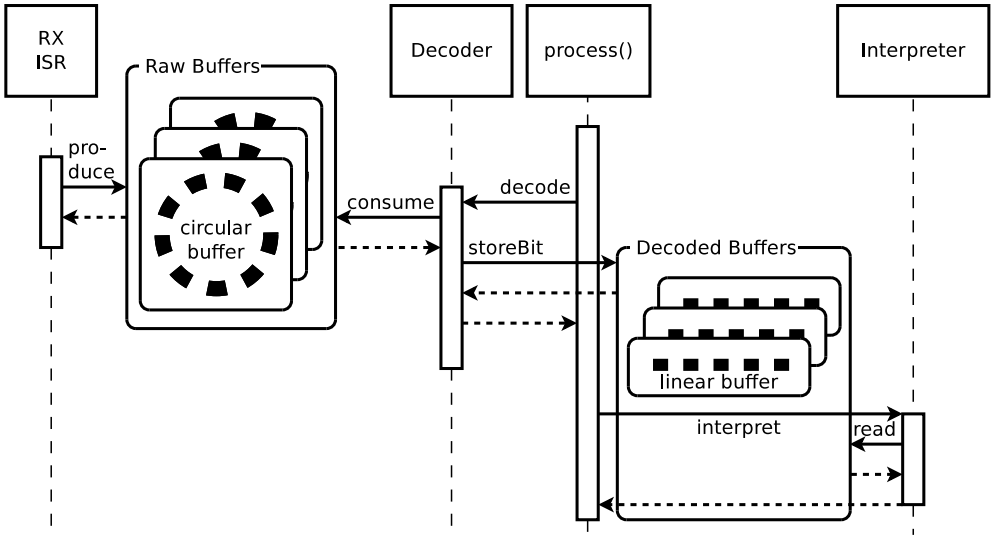
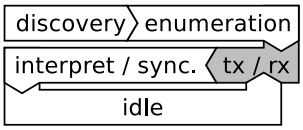


Fig. 3.3.: Receiver, decoder and interpreter sequence diagram illustrating the producer consumer mechanism

3.2.2. Decoder

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 ends. This process repeats multiple times until a PDU is completely decoded. The decoder starts after the reception ISR stored the first TCNT timestamp. It decodes parallel to the ongoing reception as long timestamps can be consumed from the buffer as illustrated in fig. 2.9. Decoding terminates when the buffer stays empty for longer than $2 \cdot d_{code}$. The post-processing is investigated and proposed in section 2.2.1. This strategy needs an interruptible decoding implementation which requires a state driven approach. As already discussed in the Manchester coding approach in section 2.2.1, a 1 bit counter for clock phase tracking is applied. 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 mechanism is required.

3.3. Transmitter and Encoder



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 2.2.5 each transmitting state implements its 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 receiver), enables the Manchester coding 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. 2.26 and fig. 2.27. An example is listed in the appendix, fig. C.1

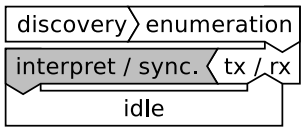
3.3.1. Manchester Code Signal Generator

When the physical layer's transmission is enabled, the first generated signal on TCNT compare match t_{cc}, is scheduled up to two d_{code} in the future starting from the current TCNT as formulated in equation (3.2). When the first transmission ISR has triggered it generates the corresponding signal and schedules the next ISR in $d_{code}/2$ cycles (equation (3.3)). The resulting maximum baud rate is about $15.63kBd$ (equation (3.4)) which corresponds to a transmission rate of approximately $0.98kB/s$. The resulting communication speed, being about 1.6 times faster than proposed in M-TRAN: Self-Reconfigurable Modular Robotic System [23], is assumed to be fast enough for scalable network

communication.

$$
\begin{aligned}
t_{cc} &= TCNT + 2 \cdot d_{code} && (3.2)\\
t'_{cc} &= t_{cc} + \frac{d_{code}}{2} && (3.3)\\
baud\_rate &= \frac{2 \cdot f_{cpu}}{d_{code}} && (3.4)\\
baud\_rate &= 15.625kBd \\
with & \\
d_{code} &= 1024 \\
f_{cpu} &= 8.0MHz
\end{aligned}
$$

3.4. Interpreter



The interpreter is implemented stateless and very minimalist. It is called by the *process*() function and validates the parity of the decoded data and whether it is interpretable as PDU. 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 fig. 3.3.

3.5. 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 from the *process*() loop and tests each task's rules set. The advantage of the scheduler is programming code deduplication and performance gain. Otherwise, without scheduling, each task is forced to apply its own specific counter and implementation

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

Fig. 3.4.: 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

for triggering at the desired time. The scheduler is absolutely necessary for the protocol evaluation as, for measuring the characteristics of the network synchronization and actuation. Tasks carrying out these experiments will be triggered by the origin node's scheduler.

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. 3.4.

3.6. Node Context

As node context we denote all protocol relevant resources a node is able to access. These are buffers, synchronization parameters, counters, FSM states and much more. The node context is packed into a global NodeState structure. The information stored in NodeState is necessary for the implementation of all protocol layers such as buffering, scheduled data, local time et cetera. An overview is illustrated in fig. 3.5.

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

DiscoveryPulseCounters Discovery and port connectivity information.

Communication Coding and decoding related buffers and adjustment parameters.

CommunicationProtocol Network layer and flow control related parameters.

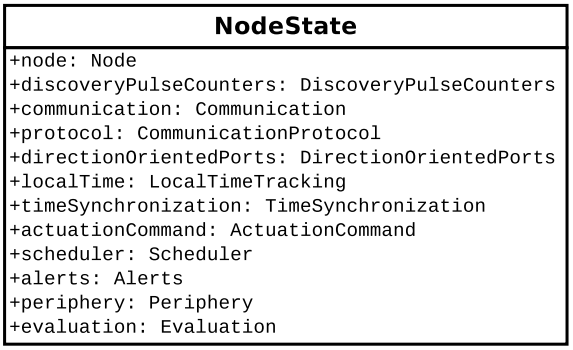


Fig. 3.5.: NodeState overview

DirectionOrientedPorts Facade [22, 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.

TimeSynchronization Stores statistical information of measured reference time spans. Used by the synchronization algorithm to update the time and compensate the clock skew locally.

ActuationCommand Command scheduling and execution related parameters.

Scheduler Executes registered tasks at specified local timestamps considering also the node state and type.

Alerts Alerting implementation for severe errors. The implementation is not essential thus skipped when release compiling.

Periphery Non-vital periphery control, such as LEDs, test points et cetera.

Evaluation Implementation of evaluation tasks which are registered to the scheduler to provide basic functionality for measuring experimental results. For example: boot network, synchronize time, compensate clock skew and execute actuation cyclically.

The complete structure diagram of NodeState including associated network layers is sketched in the appendix in fig. D.1.

3.7. 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 2.2.5 gains accuracy when TimePackages are transmitted multiple times. Having access to various measurements provides a base for several averaging methods. Thus we have implemented and evaluated the following approaches: Raw Observation Value (ROV), Simple Moving Average (SMAV), Weighted Moving Average (WMA) and Moving Least Squares (MLS). The aim is to calculate an accurate reference time span ($\overline{X}$) which is necessary for time synchronization and clock skew compensation.

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 its data to the averaging algorithm on each insertion. The compile time configuration allows only one algorithm to be chosen. Concatenation of multiple algorithms is not supported. 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 (2.17) and equation (2.18).

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 values buffered in the FIFO represent a moving window of measured observations. Besides smoothing, this also provides the possibility to perform 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 these 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.

3.7.1. Raw Observation Value

The ROV is naive method which immediately updates the newly retrieved clock speed arguments. This method is used as baseline (equation (3.5)).

$$\begin{aligned} \overline{X} &= x_0 \\ &with \\ x_0 &\dots \textit{last observed value} \end{aligned} \tag{3.5}$$

3.7.2. Simple Moving Average

The SMAV approach takes each buffered value, considering eventually activated outlier rejection, and calculates the arithmetic mean (equation (3.6)).

$$\begin{aligned} \overline{X} &= \frac{1}{n}\sum_{i=0}^{n-1} x_i \\ &with \\ x_i &\dots \textit{buffered observed values} \\ n &\dots \textit{number buffered values} \end{aligned} \tag{3.6}$$

3.7.3. 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 (3.7)) where $\overline{X}$ represents the history of previous values and x_0 is a new update.

$$\begin{aligned} \overline{X} &= p \cdot \overline{X}_{old} + (1-p) \cdot x_0 \\ with & \\ x_0 &\ldots \textit{last observed value} \\ p &\ldots \textit{weight of smoothed observations} \\ 1-p &\ldots \textit{weight of newly observed value} \end{aligned} \tag{3.7}$$

3.7.4. Moving Least Squares

The method is inspired by Fine-Grained Network Time Synchronization using Reference Broadcasts, [24] where observed broadcast PDUs are used to synchronize a network time. 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).

$$\begin{aligned} S(\beta_1, \beta_2) &= \sum_{i=0}^{n-1} (f(\beta_1, \beta_2, x_i) - y_i)^2 \\ & with \\ f(\beta_1, \beta_2, x_n) &= \beta_1 + \beta_2 \cdot x_n \\ \beta_1 &= \overline{X} \\ \beta_2 &\ldots \textit{gradient of } f(\beta_1, \beta_2, x_n) \\ x_i &\ldots \textit{buffered observed values} \\ y_i &\ldots \textit{position of} x_i \textit{ in buffer} \end{aligned} \tag{3.8}$$

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.

3.8. Optimization

Optimization can be tuned by several parameters but is always a speed-size trade-off. 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 is advantageous if all function calls are visible to the debugger as with a complete call trace, errors are easier to find. Despite of an exhaustive usage of the inline keyword at early stage of development, we face the problem of too less flash memory. Since the maximum program size is limited to the MCU's $16kB$ flash size, we are forced to make use of inline in very rare cases.

The most performant effects towards speed and size we observed with the arguments listed in table 3.1.

3.9. Commands

The commands carried by PDUs are expressed by the CMD field. A detailed listing of CMD and payload values is presented in table 3.2.

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 receiver'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.

Flag	Description	Scope
-std=gnu99	GNU dialect of ISO C99	global
-Os	optimize for size	global
-fpack-struct	pack all structure members together without holes	global
-funsigned-bitfields	let bit-fields be unsigned	global
-funsigned-char	let the type char be unsigned	global
-fdce	perform dead code elimination	global
-ffunction-sections	place each function item into its own section	global
-fdata-sections	place each function item into its own section	global
-fshort-enums	allocate to an enum type only as many bytes as it needs	release
-gstabs	produce debugging information	debug

Table 3.1.: Applied options for avr-gcc of the GNU Compiler Collection (GCC) for simulation and release compilation

ExtendedHeaderPackage This CMD is reserved. It allows protocol extension if further CMDs are necessary due to the fact that 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 d. 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 3.3. The values correspond 100%, 75%, 50% and 25% PWM duty cycle. The default is 50%. The corresponding PWM frequency can be calculated as stated in equation (3.9).

$$\begin{aligned} f_{actuator} &= \frac{f_{cpu}}{(2 \cdot UINT8\_MAX \cdot prescaler)} \\ with & \\ prescaler &= 64 \end{aligned} \tag{3.9}$$

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 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 network time synchronization.

TimePackage The time synchronization PDU contains the fields t_{tx}, d_{sep_remote}, d_{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, d_{sep_remote} is the current transmitter's time unit counting separation in cycles and d_{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.

3.10. Configuration

The source code structure allows tuning of many implementation parts which makes the firmware highly configurable. The configuration folder contains header files of each implementation part as shown in the appendix in fig. F.1 and fig. F.2. 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 F.1 and table F.3.

Actuation.h The actuation parameters define the PWM duty cycle of the actuator SMA 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 [16, pp. 81] with TCNT prescaler 64 and can not be changed.

Evaluation.h For obtaining reproducible experimental results, experiments have been implemented as tasks and registered to the scheduler. This file contains arguments relevant to evaluation only.

communication/[Communication.h, ManchesterDecoding.h] The frequency is formulated in equation (3.9). The clock adjustment for the Manchester coding (encoding and decoding) can be tuned with this

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}, d_{sep_remote}, d_{until_cc}, FU, EB	fig. B.15
HeatWiresPackage	$0x11$	HDR, CMD, ROW, COL, t_{start}, d, L, R	fig. B.16
HeatWiresRange-Package	$0x11$	HDR, CMD, ROW1, ROW2, COL1, COL2, t_{start}, d, L, R	fig. B.17
HeatWiresMode-Package	$0x12$	HDR, CMD, M	fig. B.18
ExtendedHeaderPack-age (reserved)	$0x15$	HDR, CMD	fig. B.19
SyncNetworkTime-HeaderPackage	$0x02$	HDR, CMD, t_{tx}	fig. B.20

Table 3.2.: Command id (CMD) listing and 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 3.3.: Heating mode heating mode (M) listing, MCU clock frequency (f_{cpu})= $8MHz$, actuator frequency ($f_{actuator}$) is formulated in equation (3.9)

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 (line coding half bit delay), long separation (line coding clock delay) and timeout. The classification is needed for decoding the Manchester coding. The default settings adjust the TX/RX clock period to $2 \cdot 1024$ MCU clocks, thus the $f_{xmission}$= $3906.25Hz$. Separations $\leq d_{code} \cdot 0.75$ are classified as short, separations $\leq d_{code} \cdot 1.25$ as long and separations $> d_{code} \cdot 1.25$ as timeout.

CommunicationProtocol.h The communication protocol settings define flow control timeout and retransmission counters. The flow control counters are implemented as a $process()$ loop counter which are decremented until zero accordingly. On $counter = 0$ a timeout is detected.

Discovery.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) [16, pp. 78] with TCNT prescaler 8 and cannot be changed. The discovery signal frequency ($f_{discovery}$) can be calculated as in equation (3.10). 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.

$$f_{discovery} \approx \frac{f_{cpu}}{8 \cdot 0x80 \cdot 2} \approx 3906.25Hz \tag{3.10}$$

IoPins.h, Leds.h, Periphery.h This files are meant for non-vital hardware extensions that do not affect the protocol such as LEDs or test points (TPs). The frequency of LED signals such as heartbeat and many more can be tuned in this files.

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.

Scheduler.h The scheduler implementation requires an array of tasks. The static array size and task identifiers are defined in this file.

Stdout.h The function $printf(...)$ redirects the output to $UART1$. The format is 8 data bits, no parity bit and 1 stop bit ($8N1$). The configuration allows only setting the baud rate.

synchronization/* The protocol implementation bears many strategies for processing measured reference time spans which directly affects synchronization and clock skew compensation. Which strategy is applied and how it is tuned can be configured in this folder. A detailed list of arguments is found in the appendix.

Time.h For local time tracking the cyclical ISR separation is adjusted in this file. The detailed reasoning for the chosen value can be read in section 2.2.5.

3.11. Simulation

To reduce the time spending on development, minimize the risk of design errors, enhance the code quality and especially for analysis reasons we apply a simulation framework. The simulation, providing complete insight into the MCU's, is necessary for a highly diagnostic evaluation, which otherwise is not feasible just on hardware. Another reason for simulation is the aim for holistic simulation of Shape-Shifting Displays. Different future simulations, i.e. forces, mechanical stress, shape shift planning etc., can be adopted to

use the output of a network simulation. In general concatenating multiple simulation frameworks is possible by using the simulation output as input for the next simulation framework. With such a concatenation an entire simulation, beginning with network communication layer, is feasible.

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 debugging capability in combination with GDB debugger and DDD graphical user interface. For a network simulation SimulAVR cannot be used out of the box. It does also not provide any MCU, such as the ATtiny we need. Instead Avrora provides many features for instrumentation out of the box. The Avrora simulator covers all our listed needs and is easily extensible. The Avrora simulation framework is excellent for scale-able sensor network simulation. Unfortunately, Avrora cannot be used in combination with GDB when simulating networks, but this circumstance can be overcome with the features the framework provides. Although the latest Avrora release is older than SimulAVR's we have opted to use the Avrora simulation framework.

3.11.1. Avrora Simulation Framework

The Avrora framework is a non-intrusive simulation framework [25] of the UCLA Compilers Group[2] written in Java for experimentation, profiling and analysis. It allows a target code to be written without the need of inserting extra code 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 built on top of this instrumentation points. This allows the framework to be very flexible with respect to 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 in future work such as simulating physical forces, friction et cetera. The produced data is a

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

very promising base for future work, as long 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 the nodes having the same progress. This is vital for networks, especially because communication relies on timing. Avrora considers two timing strategies [26].

Synchronization Intervals Each node's execution is divided into intervals. Each node runs until the end of its 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 approximate 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. It abstracts the particle hardware concept (see appendix section G) which includes

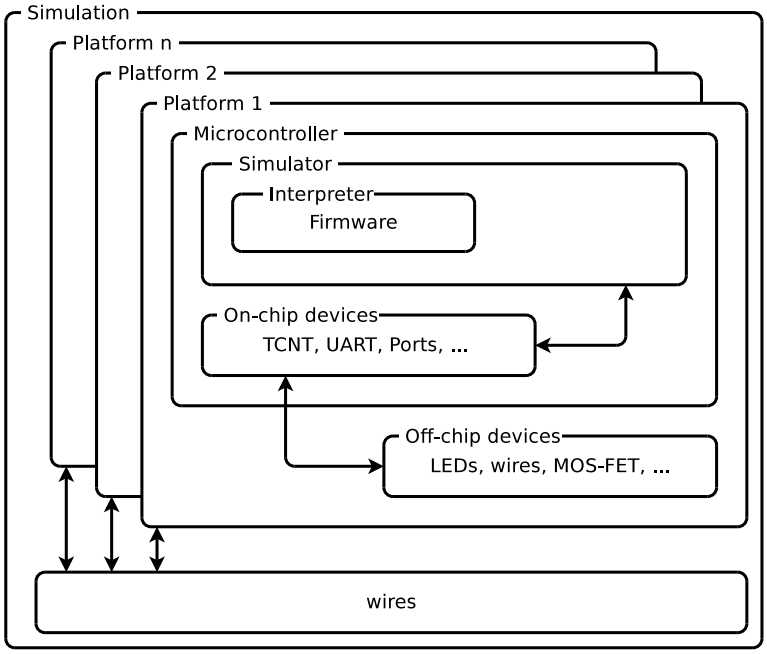


Fig. 3.6.: Avrora's software structure; platforms connected by wires to allow inter-platform communication

the physical layer as illustrated in fig. 2.4 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 G. 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 network's origin node as master device having a different firmware as illustrated in fig. 2.1. With this system communication signals generated by a specific MCU are propagated as follows: MCU TX pin $\rightarrow$ wire $\rightarrow$ MOSFET $\rightarrow$ wire connecting platforms $\rightarrow$ wire $\rightarrow$ MOSFET $\rightarrow$ MCU RX pin. An illustration of Avrora's internal software structure shows the inter-platform communication in fig. 3.6.

Avrora Monitor Extension

The simulation framework provides many analysis monitors for example function calls, interrupts, memory profiling and much more. For our specific needs we implemented a monitor that is exactly tailored to 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. 3.7.

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 reinterprets the new value with respect to the current write offset. The platform number to address mapping and vice

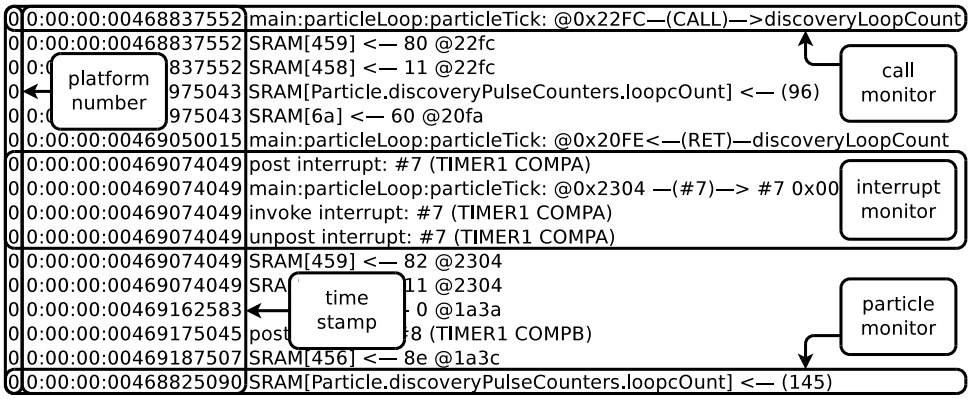


Fig. 3.7.: Avrora simulation trace of several monitors

versa are formulated in equation (3.11) and equation (3.12).

$$
\begin{aligned}
given\ (M \times N)\ network \quad &| \quad id \in \mathbb{N},\ m,n \in \mathbb{N}^{+},\ m \leq M,\ n \leq N \\
idToAddress(id) \quad &\mapsto \quad (m,n) \\
addressToId(m,n) \quad &\mapsto \quad id \\
idToAddress(id) \quad &= \quad ((id \mod N)+1),\ \left\lfloor \frac{id}{N} \right\rfloor + 1) \qquad (3.11) \\
addressToId(m,n) \quad &= \quad (n-1) \cdot M + m \qquad (3.12)
\end{aligned}
$$

The example configuration in fig. 3.8 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 about 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 reedited. For that reason an auto generator was implemented in Python that creates a JavaScript Object Notation (JSON) configuration file out of the C source code. For more details about the generator refer to the development repository[3]. With the

[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. 3.8.: Particle monitor's JavaScript Object Notation (JSON) configuration file example snippet

framework's built-in monitors and especially the tailored particle monitor there is no need for a debugger. The gathered uniform monitor dump is easily parsed with simple regular expressions and used for automatic JUnit testing.

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, 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. 3.9. 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

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

```
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. 3.9.: Avrora extension registration

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.

3.11.2. 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 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

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

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 3.4.: Duration versus synchronization of a (6×6) network simulation; simulation of $150ms$ with Microcontroller Unit (MCU) frequency $f_{cpu} = 8.0MHz$ using different synchronization interval arguments

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 3.4) are sufficient accurate for the line de-/coding. For example a 0.15 second simulation of 36 nodes takes about 176 seconds followed by approximately 94 seconds for log inspection (total 4.5 minutes) and produces about $410MB$ logs (setup B in table 3.4). The JUnit tests are found in the default Maven folder *src/test/java* of the particle platform implementation[4].

3.11.3. 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, $uint8\_t$ or $uint16\_t$. In case of *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. 3.10. 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
current 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. 3.10 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 about $25k$ 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)

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

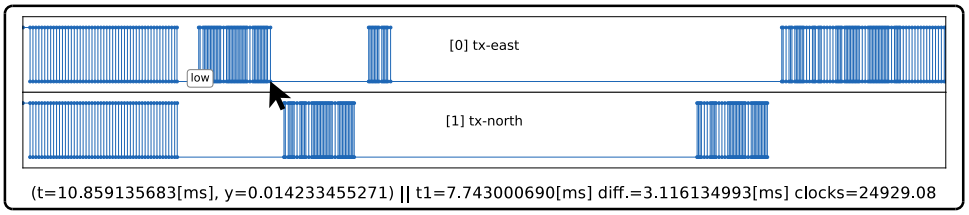


Fig. 3.10.: Simulated (1×2) network visualization; communication signals of two neighbored particles showing a highlighted label and detailed information at the bottom of the chart

network showing all phases until the time synchronization can be found in the appendix in fig. H.1

3.11.4. Network Use Case

The described network structure in fig. 2.1 of section 2 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 realizable 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 3.5. 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.

3.11.5. Build Environment

To speed up the deployment process, provide code analysis tools we configured a tool chain as illustrated in fig. 3.11 using CMake, a cross-platform

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 resynchronize
ResetPackage	reboot network
SetNetworkGeometryPackage	redefine a new network geometry

Table 3.5.: Protocol Data Units (PDUs) for master device to origin node communication

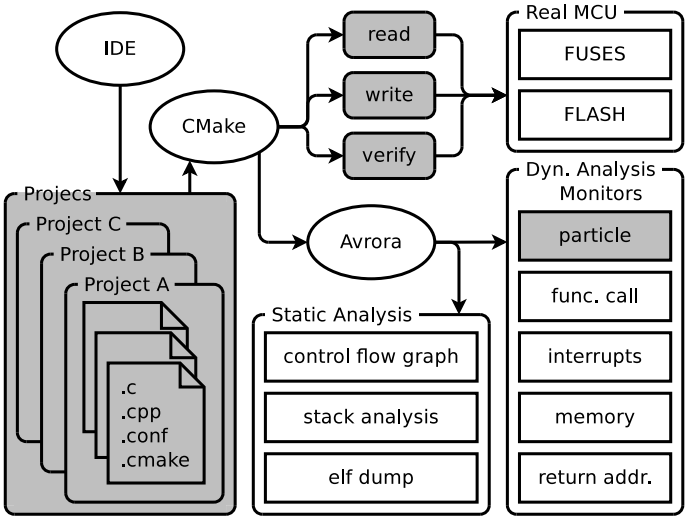


Fig. 3.11.: Development tool chain; gray highlighted items reflect developed parts of our work

build tool. To sustain a test driven development we decided to build multiple 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 the 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 rule folder.

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

Considering each project provides its 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 3.6 we skip the prefix for simplicity. The Make rules are not designed to support JUnit testing, but rather allow starting a simulation process for dynamic analysis.

The highlighted parts in fig. 3.11 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 3.6.: Make rules listing of non-prefixed rules (first block) and project dependent rules (subsequent blocks)

4. Experimental Results

The evaluation focuses on memory consumption, communication timings, time synchronization accuracy and other unexpected findings observed during the evaluation process. By combining measured data and protocol behavior we prove the protocol scalability. The evaluation will pass through in a bottom-up approach, beginning with basics and finishing at the time synchronization experiments. The following items will be discussed:

1. Manchester decoding memory consumption
2. timing evaluation
    a) discovery phase
    b) enumeration phase
    c) general timing acquisition
3. experiments
    a) clock skew compensation
    b) time synchronization
    c) actuation of actuators

The software development was largely sustained by the Avrora simulator, thus measurements obtained by simulation are compared to measurements of real hardware. This should prove the correlation of simulation and execution on real hardware. Unfortunately some parts are difficult to evaluate in hardware or simulation, hence this comparison cannot be made of each and any evaluation.

All simulated results are performed with the synchronization setup as stated in table 3.4, setup **C**: platform thread synchronization each 8 MCU cycles, MCU frequency f_{cpu}= 8.0MHz. Hardware evaluations are performed on a network having a geometry of (12×1), supplied with a regulated voltage V_{CC}= 5.1V connected at origin node node $(1, 1)$ (unless specified differently).

PDU	Len. [*bits*]	Min. [μs]	Max. [μs]	Avg. [μs]
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 4.1.: Protocol Data Unit (PDU) length versus simulated decoder's post-processing delay

4.1. Manchester Decoding Memory Consumption

The line code decoding requires buffering of received signals. Because of the limited working memory, the implementation buffers just a fraction ($buffering\_ratio$) of the total buffer needed to capture the largest possible PDU. The evaluation proves that a buffering ratio of 20% as stated in equation (3.1) is sufficient.

4.1.1. Evaluation Based on Simulation

Contrary to our expectations the evaluation of the post-processing strategy (which is discussed in section 2.2.1) shows no relevant decoding delay (illustrated in fig. 2.9) nor relevant increment of the decoder's buffer consumption for different PDU size ($\mid PDU \mid$). Table 4.1 shows a summary of measured delays between received PDUs and the decoding process' end. We observe a rather constant average post-processing delay regardless of the $\mid PDU \mid$. This is due to the Manchester coding and the varying baud rate, which is dependent on the encoded data. The real data carried by these PDUs results in rather mixed bits to be encoded (fig. 4.1). A lower baud rate leaves more space in between reception ISRs for decoding, which results in a less buffer consuming decoding.

For evaluating the decoder's buffer worst-case consumption we compare a typical TimePackage (section 3.9) against a prepared one. The TimePackage,

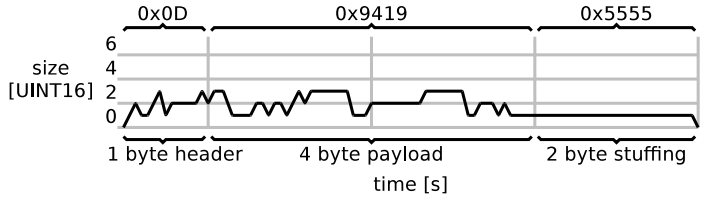


Fig. 4.1.: Simulated buffer size versus Protocol Data Unit (PDU) length of TimePackage I decoding; average case

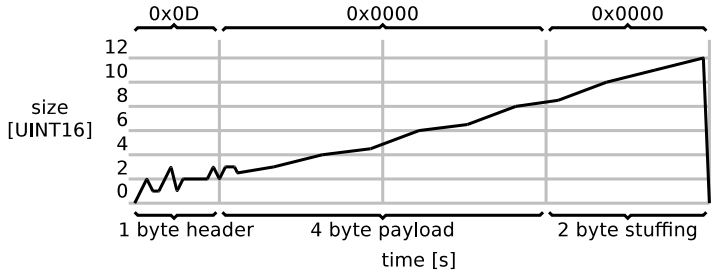


Fig. 4.2.: Simulated buffer size versus Protocol Data Unit (PDU) length of TimePackage II decoding; worst case

having a PDU length of $\mid PDU \mid = 7$ byte, is a rather long PDU. In fig. 4.1 we present the average case decoder's buffer consumption of the typical TimePackage. The PDU contains mixed bits in the first 5 byte followed by two bytes $0x5555$, 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 baud rate for the last 2 byte. Fig. 4.1 shows the correlation between the described coding frequency and the buffer consumption. In contrast, the prepared TimePackage II causes a linear increasing buffer consumption as illustrated in fig. 4.2. This is due to the manually forced higher baud rate for the bytes carrying the bits $0b000000....$ In the worst-case we observe the decoder buffer consumption increases by approximately $10/6$ byte per PDU byte as formulated in equation (4.1).

$$\begin{aligned} buffer\_size &\approx 2 \cdot k \cdot \mid PDU \mid \\ with & \\ k &\approx \frac{10}{6} \end{aligned} \tag{4.1}$$

The buffer consumption evaluation shows that it is strongly dependent on the data carried by the PDU, whereas assumptions based only on just the PDU size ($| PDU |$) are weak. Furthermore we can state that the decoder's post-process delay (see table 4.1 TimePackage I and II) is proportional to the buffer consumption (fig. 4.2). Since in the average case, as depicted in fig. 4.1, has a rather constant and low buffer consumption, it explains the low post-process delay of not prepared PDUs: AckPackage, AckWithAddressPackage, EnumerationPackage and TimePackage I (see table 4.1).

4.1.2. Conclusion

Considering the worst case (fig. 4.1) we are able to conclude that a buffering ratio of only $buffering\_ratio = 20\%$ is more than sufficient: equation (4.2).

$$buffering\_ratio = 20\% \tag{4.2}$$

4.2. Timing Evaluation

In the timing evaluation we investigate communication delays of the discovery and addressing phase to prove the scalability of our protocol. In addition we analyze which difficulty we face when the implementation measures the length of reference time spans and the result of the time synchronization and clock skew compensation.

4.2.1. Discovery

The discovery phase is the time period within each node listens to signals generated by their neighbors, to detect their rough position in the network. This position is necessary for the subsequent addressing process.

In general the discovery duration per node 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 coding's clock period, as earlier stated in equation (3.10). 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 to all neighbors are discovered before the discovery safety timeout occurs and thus are not obligated to wait until the discovery timeout. On a network level, the total discovery phase duration is the time span from the first node entering the phase until the last node leaves the discovery phase. As illustrated in fig. 4.3, the simulated (3×3) network example shows a total discovery phase duration of $12ms$. In contrast, the hardware based evaluation shows a total duration of $13ms$.

Evaluation Based on Simulation In the simulated (3×3) network example of fig. 4.3, 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 can be observed at nodes having two connections which are: $(1,1)$, $(2,1)$, $(2,2)$, $(1,3)$ and $(2,3)$ (inter nodes). The discovery phase takes $12ms$.

Evaluation Based on Hardware In our experiments we see varying boot up phase delays among network nodes. This happens due to the falling V_{CC} of the power supply from the least to the most distant network node. In the evaluated (3×3) network example we see a V_{CC} drop of about $100mV$ at node $(3,3)$ without evidence of ripple. Since the particles are timed by their internal RC oscillator, a minor voltage drop of $50mV$ is more than sufficient to slow down the clock and thus extend the boot up delays relevantly. This affects the discovery phase's start, which takes place within an interval of $1ms$ as illustrated in fig. 4.4. The interval is expected to be longer the higher the V_{CC} voltage drop among particles. Despite the asynchronous boot up, the shifted discovery start does not cause any protocol errors when executed on hardware.

Conclusion

The evaluation shows that the internal clock's RC circuit is very sensitive to the supplied V_{CC}. Except for the asynchronous boot up we see no relevant discrepancy between simulation and hardware evaluation. Although the

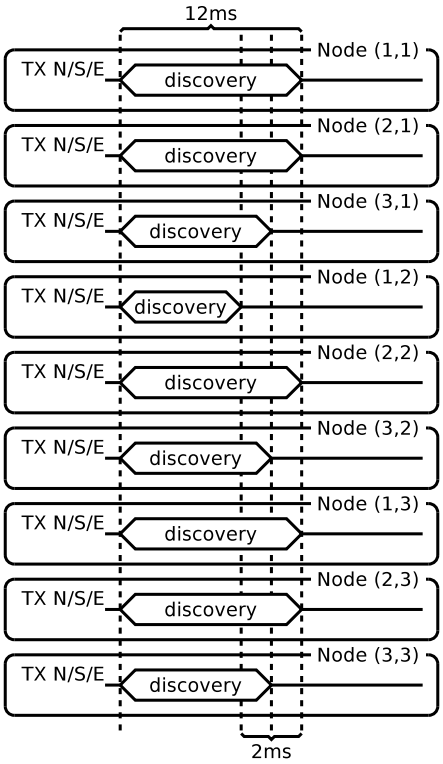


Fig. 4.3.: Simulated (3×3) discovery phase; discovery duration differs according to node's connectivity

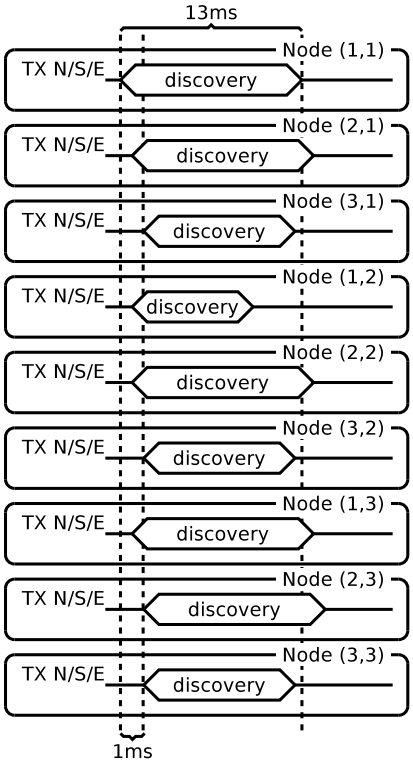


Fig. 4.4.: Measured (3×3) discovery phase; supply voltage (V_{CC}) fluctuation causes discovery shifts

simulation is able to simulate a random start, we did not use this feature extensively to reduce the simulation time.

4.2.2. Addressing

In the addressing phase, addresses are assigned to all network nodes. The phase is initiated by the top most, left most node (origin node) and thus the enumeration process is based on the discovery phase's result. The origin node switches to enumeration mode (enumerator) assigns the address to its neighbors and leaves this mode. Subsequently, the neighbors switch to enumeration mode, assign addresses to their neighbors and so forth. This address propagation scheme does parallelize addressing of nodes. Therefore the total addressing duration depends on the longest path, which again is dependent on the network geometry. The addressing phase is finished by the AnnounceNetworkGeometryPackage PDU. This package is issued by the bottom most, right most node of the network to inform the origin node of its position. The origin node considers this information as the network geometry.

Addressing The addressing follows a flow control as illustrated in fig. 2.25 in section 2.2.5. The enumerator sends a new address, the receiver acknowledges the address value and waits for ACK from the enumerator. If the enumerator receives the correct address, it means that the transmission had no errors. The enumerator can either acknowledge the transaction, or fall back and retransmit the enumeration message. This flow control is necessary to avoid misinterpreted discovery signals as messages. Such cases may easily occur due to V_{CC} voltage fluctuation as described in section 4.2.1. Until the enumerator does not receive an ACK the addressing transaction is not finished successfully. The transaction can be interrupted by timeout on both nodes.

Network Geometry Feedback The network geometry disclosure is initiated by bottom most, right most node which sends its local address back to the origin node, immediately after the enumeration transaction is finished. Fig. 4.5 shows node $(2,3)$ acknowledging the enumeration transaction. Node $(3,3)$

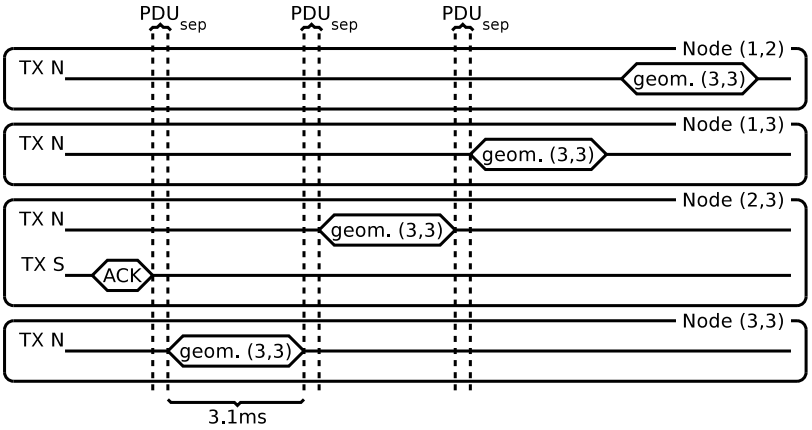


Fig. 4.5.: Simulated (3×3) network geometry disclosure of node $(3,3)$ showing AnnounceNetworkGeometryPackage's PDU transmission duration (d_{pdu})

responds with an AnnounceNetworkGeometryPackage which is routed to the origin node.

Evaluation Based on Simulation The simulation output of a (3×1) network, illustrated in fig. 4.6, shows a simple enumeration example. After all the simulation suffers of minimal jitter, thus presented PDU transmission delays are no absolute values. The measurements are averaged values of multiple simulations. The time delay 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 size of remaining data to be processed at the timestamp when the currently receiving PDU is completely received. In the example illustrated in fig. 4.6, the PDUs EnumerationPackage, AckWithAddressPackage and AckPackage having a size of 25, 24 and 8 bits, the size differs relevantly but the introduced PDU_{sep} delay having about $0.6ms$ does not. This confirms the explanation in section 4.1.1.

Evaluation Based on Hardware The hardware evaluation of the same (3×1) network geometry at a specific MCU shows slightly different results. The transmission delay of EnumerationPackage, AckWithAddressPackage, AckPackage differ for instance by $\pm 3\%$. If we consider the current transmitting MCU's clock frequency deviation at a base frequency of

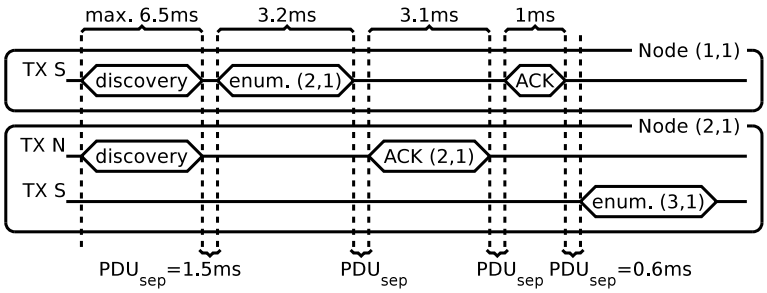


Fig. 4.6.: Simulated (3×1) network enumeration of node $(2,1)$ showing PDU transmission duration (d_{pdu}) of several Protocol Data Units (PDUs)

f_{cpu}= 8.0MHz, we obtain the same percentage of deviance. Thus we can state that both evaluations, hardware and simulated, correlate well. Apart from 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. This was to overcome the asynchronous boot up delay introduced by the V_{CC} voltage difference among nodes. Also alerting mechanisms, such as parity error and buffer overflow, have to be turned off temporarily until the enumeration phase starts. Short discovery timings are necessary in simulation where simulated real time is very costly. Instead in large real hardware networks it is expected to be necessary to adjust both parameters.

4.2.3. Conclusion

Despite the need of specific tuning of the hardware based evaluation, comparing the simulation and hardware based protocol timings we see a strong correlation in between both. Due to the applied parallelized addressing strategy, the addressing duration can be notated as equation (4.3). With this linear

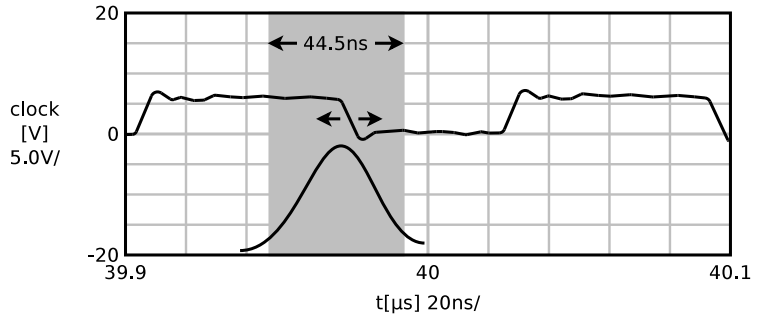


Fig. 4.7.: MCU clock frequency (f_{cpu}) jitter of one falling edge at approximately 40μs after trigger

notation $\mathcal{T}(n) = \mathcal{O}(ROWS + COLS)$ we prove the addressing scalability.

$$\begin{aligned} \mathcal{T}(n) &= \mathcal{O}(c' + c'' \cdot (ROWS + COLS) + c''' \cdot (ROWS + COLS)) \\ &= \mathcal{O}(ROWS + COLS) \end{aligned} \tag{4.3}$$

with

c' ... *total discovery delay*
c'' ... *per node address transaction delay*
c''' ... *per node network geometry transmission delay*
$ROWS$ ... *total network rows*
$COLS$ ... *total network columns*

4.2.4. General Timing Acquisition

In this section we investigate distribution of measurements as taken by the protocol implementation as well the cumulative effect among nodes.

As stated in section 3.7 we expect measured timings to have a minimal measuring error. However the distribution is assumed to be a non-skewed normal distribution ($\mathcal{N}$) having a specific variance around a mean (μ). To prove this assumption we investigate two measurements in detail: the MCU's clock frequency (f_{cpu}) jitter and the TimePackage transmission delay (d_{pdu}). The MCU's clock frequency illustrated in fig. 4.7 shows a non-skewed normal

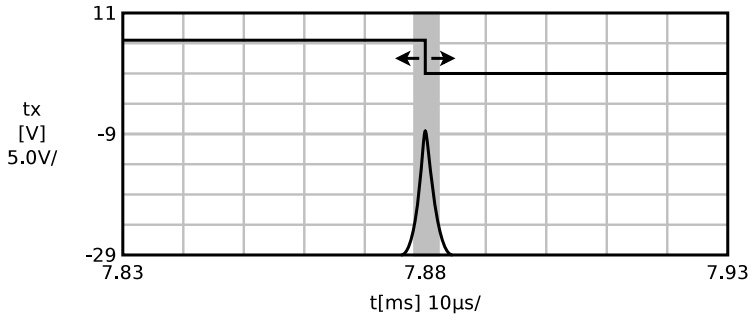


Fig. 4.8.: TimePackage's PDU transmission duration (d_{pdu}) jitter of last falling edge distribution $\mathcal{N}(\mu = 7.88ms, \sigma = 1.28\mu s)$, triggered first falling Protocol Data Unit (PDU) edge

distribution having $\mathcal{N}(\mu = 40.2167\mu s, \sigma = 10.08ns)$. The measurement is taken at μ= 40.2167μs at a falling edge.

At protocol layer the PDU transmission duration of a TimePackage has been measured. Fig. 4.8 shows the distribution of the last falling edge of the TimePackage. The trigger is set at the first falling PDU edge, since we observe falling edges to be steeper than riding. The total delay in between trigger and highlighted edge is $63 \cdot d_{code}$ which corresponds to 64512 cycles, or approximately $8ms$. This jitter, again shows a non-skewed normal distribution ($\mathcal{N}$) having $\mathcal{N}(\mu = 7.88ms, \sigma = 1.283\mu s)$.

Conclusion

With the above results we are able to prove that that the MCU's clock jitter propagates up to protocol layer measurements and that the jitter observed at protocol layer still remains a non-skewed normal distribution. Furthermore, having a normal distribution, the described averaging strategies Raw Observation Value (ROV), Simple Moving Average (SMAV), Weighted Moving Average (WMA) and Moving Least Squares (MLS) suit our application.

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 4.2.: Simulated introduced forwarding delay in broadcast mode (BCT_{delay}) evaluation summary of (6×6) network simulation, see also table I.1

4.2.5. Network Time Synchronization

The main goal of our work is network time synchronization which is vital for synchronous actuation. The time synchronization process as described by the protocol can be achieved in two ways: in broadcast or subsequent mode. In this section we evaluate the strength and weaknesses of both methods and conclude why one approach is chosen over the other.

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 decoding. Hence the TimePackage is transmitted simultaneously to any node with an introduced forwarding delay in broadcast mode (BCT_{delay}), see table 4.2.

Evaluation Based on Simulation In fig. 4.9 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. 4.9 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 2.2.5.
If we investigate the measured BCT_{delay} values we can split them into two distinct groups. This is due to a time shift of approximately $0.25\mu s$

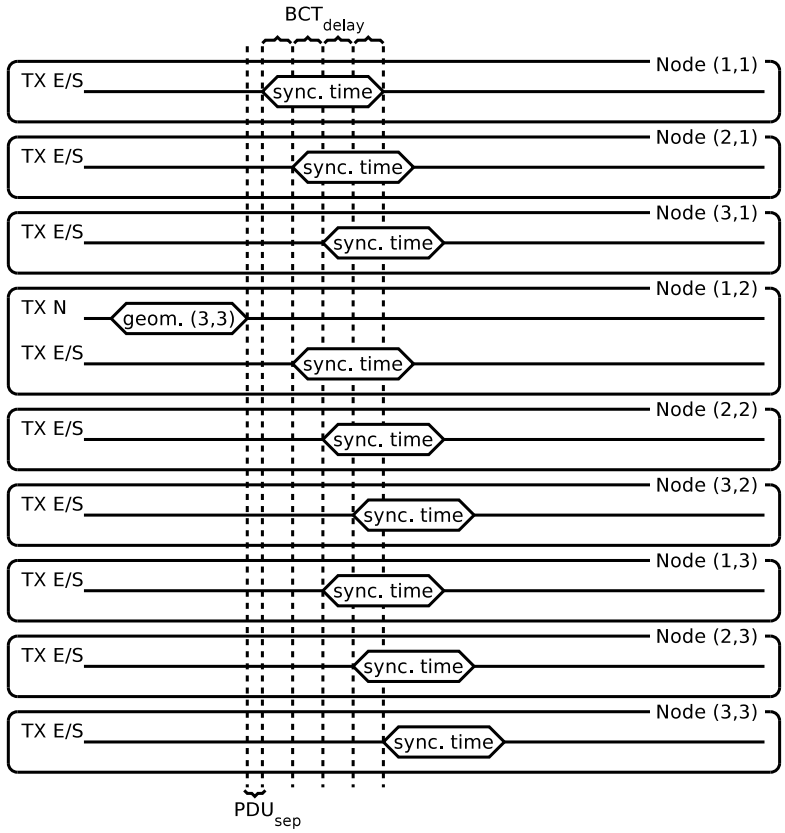


Fig. 4.9.: Simulated (3×3) network time synchronization in broadcast mode showing the introduced forwarding delay in broadcast mode (BCT_{delay}) spread among nodes

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 4.2.

Evaluation Based on Hardware An overall comparison of the introduced $BCTE_{delay}$ of a simulated network ($7\mu s$) to an optimized test implementation on a physical MCU ($4.4178\mu s$) 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 4.2.
According to our observations we consider the broadcast mode synchronization to be not well scaling. The reason for that is the not well predict-able signal edge shift.

Signal 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.

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 receiver considers the carried data and the PDU reception duration for local clock skew, local time and 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 V_{CC}, is about one MCU cycle at a base frequency of $8MHz$: $1 \cdot \frac{1s}{8MHz}$.

With respect to the online Manchester coding 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 approximately 8 instructions.

Conclusion

Despite the broadcast method provides sufficient synchronization accuracy in small networks, we aim on (i) relying on as less as possible adjust-able parameters built in the protocol and not to cope with overcomplicated calculations and (ii) scalability. The subsequent mode, considering the first and last signal edge of subsequently sent TimePackages as time reference, obtains a more accurate result. Thus we favor synchronization using the subsequent over the broadcast mode regardless of the network size. The time synchronization propagation delay can be notated as equation (4.4).

$$\begin{aligned} \mathcal{T}(n) &= \mathcal{O}(c' \cdot (ROWS + COLS)) \\ &= \mathcal{O}(ROWS + COLS) \end{aligned} \tag{4.4}$$

with

c' ... *per node TimePackage transmission delay*

$ROWS$ ... *total network rows*

$COLS$ ... *total network columns*

4.2.6. Clock Skew Compensation

The clock skew compensation is an essential part to sustain the time synchronization accuracy over long periods. Without clock skew compensation, once the time is synchronized among particles, it will drift over time. In this section we will discuss implementation modifications which, during hardware based evaluation, turned out to be necessary such as: (i) reference time span

measurement and (ii) averaging algorithms. The clock skew compensation evaluation is based on hardware only.

For compensating the local MCU frequency to comply with the north port neighbor's time counting speed, a node needs to adopt its frequency parameters. The parameters are deduced from a time span reference which is obtained from any TimePackage. Once the total TimePackage PDU length (equation (2.19)) is known, a relative time span reference is used to calculate several parameters to compensate the local clock skew. After the new frequency parameters are updated locally, to expose this new calculations, the communication baud rate (d_{code}) is updated accordingly. Thus the next TimePackage, passed to the subsequent south port neighbor, propagates the new timing.

When measuring the reference time span of the TimePackage we observe the necessity to not start measuring at a Manchester coding clock edge. This is due to the online implementation of the Manchester coding algorithm, which introduces two different delays before generating an edge: delay of Manchester coding clock edge and delay of bit-edge. Unlike stated in the protocol design and with respect to the observations so far we update equation (2.19) and simplify it to equation (4.5). The update considers measuring the delay in between the second and last but one edge of the TimePackage PDU. In other words the delay from the first bit-edge to the last bit-edge.

$$\begin{aligned} d_{code} &= \frac{d_{pdu}}{63} \qquad (4.5) \\ &with \\ 63 &\ldots \; number\ of\ measured\ d_{code}\ intervals\ in\ TimePackage \end{aligned}$$

In addition the evaluation reveals a very sensitive reaction of the MCU clock frequency (f_{cpu}) to the applied power supply voltage (V_{CC}). A V_{CC} ripple of $\pm 100mV$ destabilizes the MCU clock frequency too much. All together this means, a fluctuation of the origin node's f_{cpu} affects the whole network synchronization. For this reason, we evaluate several averaging algorithms, as stated in section 3.7.

In the following figures we compare averaging algorithms by means of a (12×1) network. The measurements are always taken at nodes $(1,1)$ (first node), $(4,1)$, $(7,1)$ and $(12,1)$ (last node), unless specified differently. The

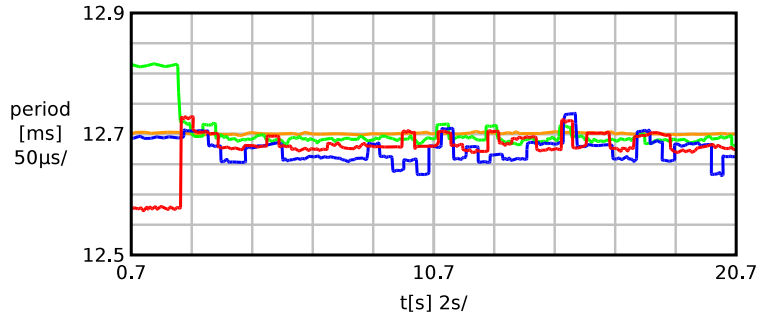


Fig. 4.10.: Clock skew compensation without averaging algorithm; beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, red node $(12,1)$, network setup network configuration setup 1 (*net*1)

measured output represents the local time counting frequency as time period (y-axis) at the respective node over time (x-axis).

Raw Observation Value

With this method, the baseline, the clock skew is adjusted immediately after the first TimePackage is received, as illustrated in fig. 4.10. To compensate the whole network's clock skew only one TimePackage needs to be propagated throughout the network. Hence there is no averaging we observe a jittering after each interpreted TimePackage. The disadvantage of this method is the sensitivity to outliers. The result always dependent on the lastly received TimePackage. Another 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. 4.11. 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

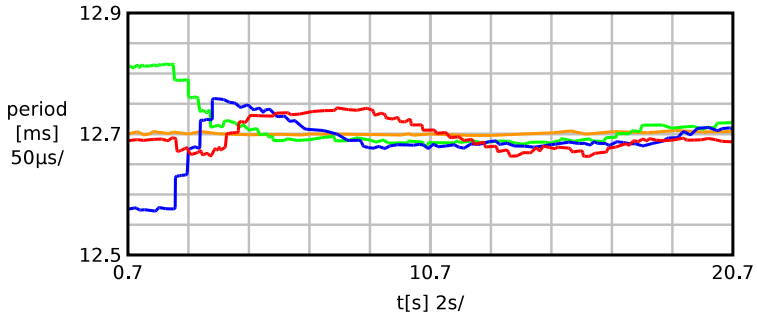


Fig. 4.11.: Clock skew compensation with Simple Moving Average (SMAV) and 4 buffered values without outlier detection; beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, red node $(12,1)$, network setup network configuration setup 1 (*net*1)

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, the results with outlier detection do not improve relevantly, thus no further evaluation details about this investigation are presented.

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 (3.7)). In figure fig. 4.12 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.

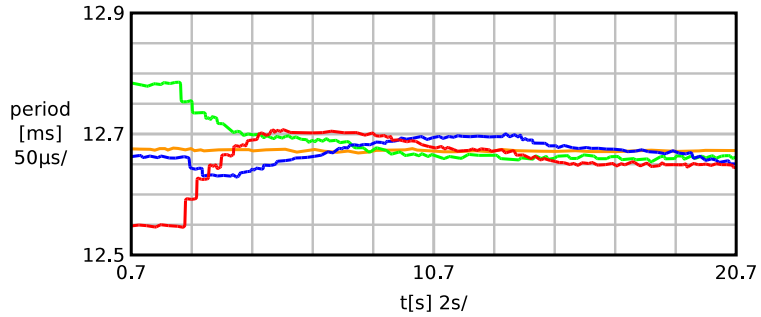


Fig. 4.12.: Clock skew compensation with averaging using Weighted Moving Average (WMA); beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, purple node $(12,1)$, network setup network configuration setup 1 (*net*1)

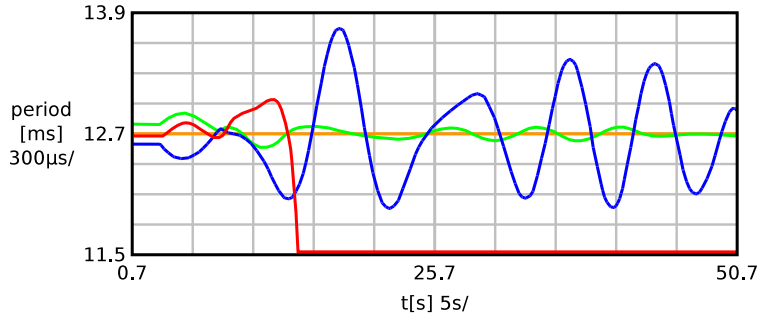


Fig. 4.13.: Clock skew compensation with averaging using Moving Least Squares (MLS) and 40 buffered values without outlier detection; beige node $(1,1)$, green node $(4,1)$, blue node $(7,1)$, red node $(12,1)$, network setup network configuration setup 1 (*net*1)

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. 4.13 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.

Another 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 about $\pm 800 \mu s$. This leads to moving the baud rate outside the limits, which can be seen at the red curvature of node $(12, 1)$ in fig. 4.13. 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 $\mu \pm 2 \cdot \sigma$ and $\mu \pm 5 \cdot \sigma$ resulted in even more overshooting, whereas a boundary of $\mu \pm 10 \cdot \sigma$ produces again similar results as without outlier rejection. A boundary of $\mu \pm 10 \cdot \sigma$ 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.

Moving Least Squares 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.

Simple Moving Average 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.

Weighted Moving Average 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.

Raw Observation Value This method is not recommendable as the possible accuracy gain, of multiple synchronization PDUs, remains unused.

Conclusion

For productive usage we opt for the SMAV method with a FIFO buffer size of 4 measured values and no outlier rejection mechanism. A short performance overview is listed in table 4.3. In the upcoming experiments the same configuration is enrolled onto the (12×1) test network, unless specified differently.

With the SMAV method activated, the clock skew compensation propagation delay among nodes can be noted as equation (4.6).

$$\begin{aligned} \mathcal{T}(n) &= \mathcal{O}(c' \cdot (ROWS + COLS) \cdot c'') \\ &= \mathcal{O}(ROWS + COLS) \end{aligned} \tag{4.6}$$

with

c' ... *per node TimePackage transmission delay*

c'' ... $FIFO\ buffer\ size$

$ROWS$ ... *total network rows*

$COLS$ ... *total network columns*

4.2.7. Scalability

In our evaluation we have proven the scalability of the unattended addressing method (equation (4.3)), the time synchronization (equation (4.4)) and the clock skew compensation (equation (4.6)) propagation. The delay of all these strategies together, again has an linear scaling dependent on the network geometry: $\mathcal{T}(n) = \mathcal{O}(ROWS + COLS)$.

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 4.3.: Averaging strategy performance listing

4.3. Other Observations

Vcc Ripple

With the current hardware design and the applied MCU we observe a very high f_{cpu} sensitivity to the applied V_{CC}. In our experiments, for usability reasons, we used the on-board LEDs for signaling. In the (12×1) test network this lead to a V_{CC} ripple of $V_{CC} \pm 100mV$, 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 V_{CC} 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 (see appendix fig. I.1), 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. 4.14.

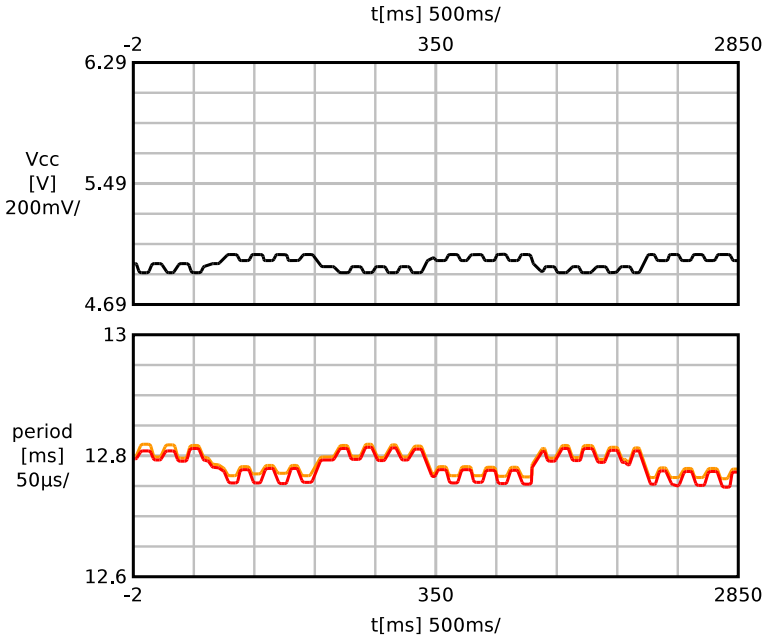


Fig. 4.14.: MCU clock frequency (f_{cpu}) sensitivity versus supply voltage (V_{CC}) as measured at the local time counting speed period duration; supply voltage (top chart) versus time counting speed (bottom chart); beige node $(1,1)$, red node $(12,1)$

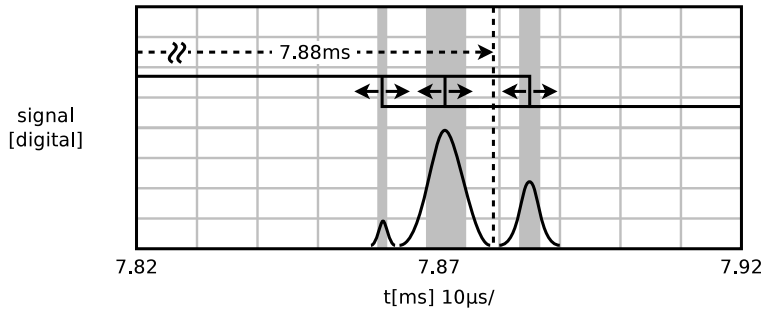


Fig. 4.15.: PDU transmission duration (d_{pdu}) discretization as observed on retransmission when tuning the clock skew compensation using minimal adjustment step; target $7.88ms$, actual values within gray areas

Measurement Discretization

In fig. 4.15 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 recalculate the skew compensation and baud rate as formulated in equation (2.17) and equation (4.5). 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. 4.15.

For transmission this means if the new d_{pdu} is ± 1, the total integer discretization delay ($d_{discrete}$) of one PDU is about $\pm 8\mu s$ and can be formulated as equation (4.7). The calculated $d_{discrete}$ fits perfectly to the span in between the discretization centers of fig. 4.15.

$$\begin{aligned} d_{discrete} &= \pm 64 \cdot \frac{1}{f_{cpu}} \\ &\approx \pm 8\mu s \end{aligned} \tag{4.7}$$

As 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

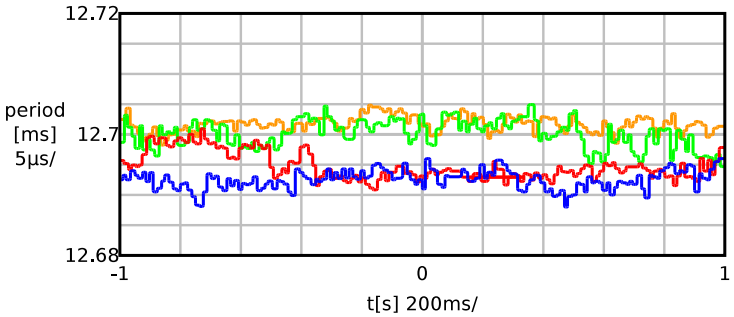


Fig. 4.16.: PDU transmission duration (d_{pdu}) discretization of clock skew compensation using minimal adjustment step; time counting period as y-axis; beige node $(1,1)$, green node $(4,1)$, blue $(7,1)$ red node $(12,1)$; applied network configuration setup 1 ($net1$)

$\mathcal{N}'$, 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 (4.8).

$$E(d_{pdu}|node = (1,1)) \stackrel{\wedge}{=} E(d_{pdu}|node = (n,1)) \tag{4.8}$$

The same cumulative discretization error is observable in a frequency trend analysis as illustrated in fig. 4.16 which shows the compensated time clock frequency after a long term synchronization run. The offset between green and red curves of approximately $10\mu s$ 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 red 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 with the given constraint formulated in equation (2.17). By skipping this constraint, the discretization can be minimized to a fraction and equation (4.7) does not hold any more.

4.4. Experiments

The main target of our experiments is to measure the clock skew compensation and time synchronization. For this reason we evaluate three experiments:

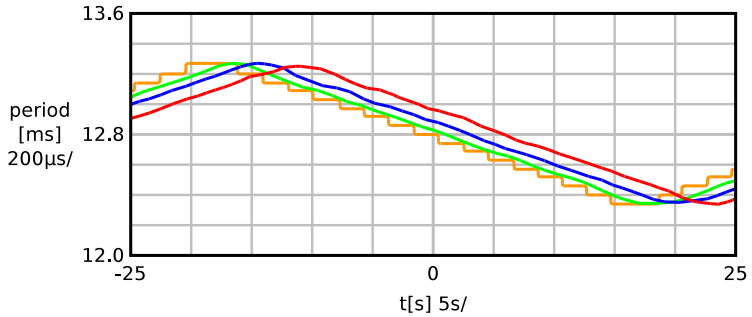


Fig. 4.17.: Clock skew compensation experiment with moving MCU clock frequency (f_{cpu}) of node $(1,1)$ (beige); green node $(4,1)$, blue node $(7,1)$ red node $(12,1)$; applied network configuration setup 1 ($net1$)

(i) clock skew experiment, (ii) time synchronization experiment and (iii) actuation experiment. In experiment (i) the origin node's clock speed factory calibration is changed on purpose, to prove the clock skew compensation of subsequent nodes compensates the new frequency change. In experiment (ii) we measure the synchronicity of the time counting among nodes. The last experiment (iii) shows the synchronicity of actuator commands, which is our main goal.

4.4.1. Clock Skew Compensation

In this experiment the first node $(1,1)$ is prepared to change its 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. 4.17 as beige curvature.

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 red curvatures. In the experiment the response delay from first to last node is about $5s$. The delay can be decreased by more frequent synchronizations.

The experiment also reveals the coarse-grained f_{cpu} tuning characteristic of

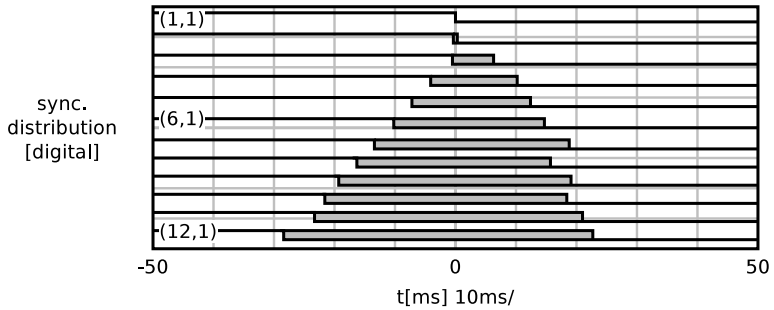


Fig. 4.18.: Time synchronization distribution among nodes (2-12,1) relative to origin node (1,1); gray areas highlight the minimum to maximum distribution; measurement duration approximately 15 minutes

OSCCAL as discussed in section 2.2.5. Compared to our implementation the granularity is roughly 8 times smaller than with OSCCAL.

4.4.2. Time Synchronization

In this experiment we compare the network time deviance among all test networks' nodes. The nodes are configured to toggle an output signal each 64 time clock intervals. This duration corresponds to approximately $410ms$. 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. Another 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. 4.18 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 (4.8) is identifiable.

The observed time clock frequency shifting is a result of the measurement discretization as described in section 4.3. 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.

4.4.3. Actuation

In the actuation experiment the network is prepared to firstly synchronize then actuate each left actuator simultaneously. As 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. 4.19 a V_{CC} 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 V_{CC} impacts a network resynchronization is urgently necessary.

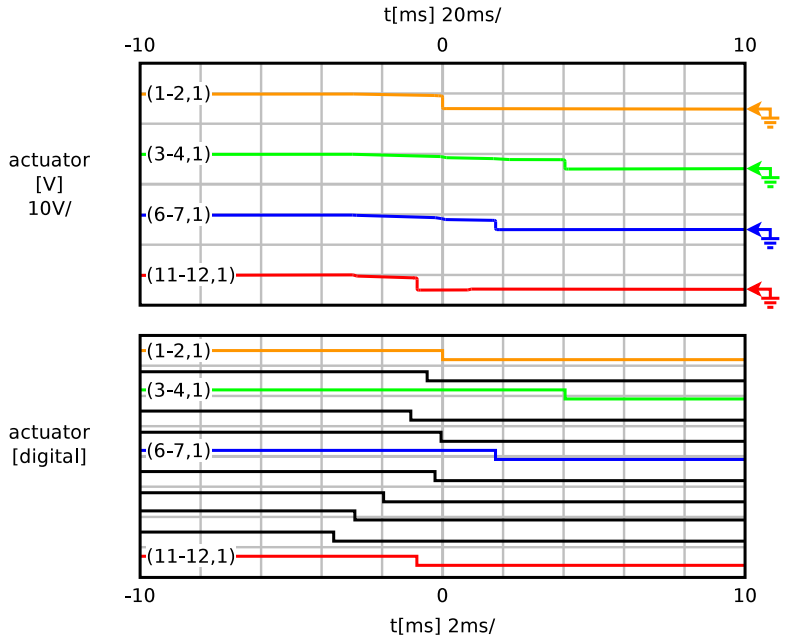


Fig. 4.19.: Actuation accuracy; yellow actuator (1-2, 1), green actuator (3-4, 1), blue actuator (6-7, 1) and red actuator (11-12, 1), cyan D4-D14 all actuators, (1-2, 1) as D1

4.4.4. Conclusion

We could prove the clock skew compensation algorithm is compensating MCU clock frequency discrepancies successfully. The algorithm is expected to cover a wider range than $650kHz$ as shown in the experiment. The necessary time synchronization accuracy, which is enough to be in range of seconds, can be achieved as shown by the results of time synchronization and actuation experiments.

5. Related Work

Nowadays many different approaches to enable programmable matter exist. Programmable matter raises complex problems, that can be seen from different perspectives: changeable property (i.e. shaping) by software, system structure (chain/lattice based [27–29]) actuator type and stimulus, actuation environment (i.e. sliding, floating [30]), hardware implementation (i.e. self-contained, power supply, externally applied forces), communication, actuation planning (decentralized, centralized, meta programming), user interface [31, 1, 32] (i.e. soft materials [33], posing [34], placing [35]), system reconfigurability, self-reproducibility and degree of freedom (DOF). A short exploration of these viewpoints reveals that in the programmable matter field they are very closely related. Thus design decisions are hardly based on just one viewpoint but rather the sum of them. In our short survey we review the most important aspects related to our work: hardware implementation, connectivity, actuation, collective actuation, and software. The survey should bring clearer understanding and emphasize general difficulties of programmable matter to be mastered.

5.1. Hardware Implementation

Many proposed systems apply self-contained robotic particles that have enough features implemented to enable autonomous actuation of these robots. The strength of self-contained robots is autonomy in terms of independent computations and actuations. Robotic parts having an own microcontroller unit are capable of local computations or scheduling of specific tasks. With regards to actuation they are able to perform actuations without externally applied forces or direct cooperation with neighbored robotic particles. This

does not necessarily state that lightweight systems, lacking these features, are weak because the true potential of a system depends on the use.

Miche [36], a good example of local computation, outsources the communication to a dedicated unit to save computational capacity of the main computation unit. Another hardware implementation with focus on scalability to very large numbers of interacting units is proposed in Claytronics [37] utilizing Catoms.

5.2. Connectivity

As connectivity we refer to the software or communication connectivity, instead of mechanical connectivity. Finding an adequate communication design for a specific programmable matter implementation is a very sensitive task. The most prioritized questions are: is the system reconfigurable [23, 38–40] or static, is unidirectional communication sufficient, is sensor data to be transported, is the communication limited to subsequent neighbors or is it multi-layered?

Reconfigurable systems imply that particles are able to move from one physical position of the network to another and integrate at the new position. Such systems require a more sophisticated network protocol that is able to handle these dynamic links efficiently. However, static systems having their particles residing at the same position in network require at least automatic position detection without the need of manually specifying network addresses of any particle. Having thousands of particles, doing this manually is not reasonable.

Depending on the needs, a protocol may not necessarily need to implement a bi-directional communication for each transmitted data. In real-time systems the responses on received data such as "acknowledge" or "task done" may be redundant. In fault tolerant systems even the response "error occurred" may be not of interest. On the other hand, one may be interested in extensibility of the system and allow particles to be extended with sensors to introduce the capability of interaction with humans. Other systems may inherently introduce the need of sensor data to allow interaction in between the target and actual state. This is especially necessary for systems capable of posing.

Posing means providing a Tangible User Interface (TUI), where a user is able to modify shapes by hand.

In reconfigurable systems, where particles do not necessarily know at which place they reside and where the next neighbor is, a multi-layered communication is advantageous. By communicating in layers such as neighbor, local and global the communication complexity can be reduced. Each layer introduces its own context, constraints and eventually its own communication channel, which also allows duplex communication without interfering. Examples are the Catoms [41, 42] proposing neighbored communication via inductive coupling, local communication via infrared, and global via IEEE 802.15.4 ZigBee and Posey [34] utilizing infrared and ZigBee.

5.3. Actuation

Many actuation technologies, the unit generating force for actuation, such as pneumatic, hydraulic, motoric, SMA [43] have been proposed. A very important group of actuators are the shape changing materials. This field includes actuators that are rather weak compared to motoric or hydraulic actuators and may be too weak for use in everyday systems. They outperform the strong ones in terms of weight, size, power consumption and environmental influences [44]. In environments lacking of electric power supply other stimuli for actuation such as pH, heat, light etc. are of interest. Further important properties of shape changing materials are: deformation strength and power requirement, speed and resolution [45], number of memory shapes [46] and environment compatibility [47].

5.4. Collective Actuation

With a collective of robotic particles, programmable matter's performance can be enhanced in many regards. The programming and communication complexity can be decomposed and assigned to groups of particles which can be more autonomous. The work proposed in [48], explains how to increase forces and physical robustness, simplify control and communication

by local control and fault isolation by means of a cell, which is a cluster of reconfigurable particles. Collective robotic particles can be also found in other programmable matter scientific fields, with focus on meta programming languages and decentralization of programs.

5.5. Software

Most scientific programmable matter publications propose great systems but apply just a small set of particles for evaluation. The answer to how the performance scales in systems having thousands of such units remains unclear. The software part of programmable matter addresses how particles can be controlled efficiently.

A simple method to control programmable matter is to apply a centralized program communicating and controlling each unit, which performs well in small systems but does not scale with system growth. However, finding the optimal actuation plan to achieve a target state given an arbitrary start state is a hard calculation problem. For this reason one may tend to break down the whole problem to smaller ones and delegate them to subordinate units or groups of units (meta modules) [49].

A meta module grants the responsibility for the subset it consists of plans, executes and controls the set in a decentralized manner. This requires more effort when programming the global program, which then is compiled to distributable program parts, distributed to meta modules and executed by particles. On the other hand, this method gains performance in different areas: it reduces the calculation complexity for planning, lowers the amount of needed communication and communication links in between particles and also reduces the error handling complexity or even may introduce a fault tolerance. Newly introduced constraints by meta modules, i.e. illegal movements, are easier manageable at module level than globally. An approach for programming meta modules using a meta language that produces distributable programs is Meld [50] and [51–53].

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 to save a lot of time, otherwise spent on reprogramming nodes multiple times.

Regarding error detection and fault tolerance, no deep investigation 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 enhanced. For instance, an error reporting to the origin node would be very helpful. Error correction strategies are more complicated since they bear also many questions: how can the protocol still remain a real time protocol, how to keep the communication overhead low, et cetera.

At an early phase we had a simultaneous data stream broadcasting method in mind. As highlighted during evaluation, we do not recommend simultaneous communication throughout whole networks. However a method where data is transmitted as a stream of subsequent packages is reasonable. Since this is rather a streaming than flooding method, a stream position to network address mapping is necessary as formulated in equation (6.1) and vice versa in equation (6.2). Due to the extent this part suits well for future work.

$$given\ (M \times N)\ network,\ stream\ position\ p \mid m, n \in \mathbb{N}^+;\ p \in \mathbb{N},\ p < M \cdot N$$

$$\begin{aligned} networkAddress(p) &\mapsto (m, n) \\ networkAddress(p) &= \begin{pmatrix} M - (p \mod M) \\ N - \left\lfloor \frac{p}{N} \right\rfloor \end{pmatrix}^T \qquad (6.1) \\ streamPosition(m, n) &\mapsto p \\ streamPosition(m, n) &= (M \cdot N - (m + (n-1) \cdot M)) \qquad (6.2) \end{aligned}$$

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 Weighted Moving Average and Simple Moving Average would shorten the delay until the network is stable and allows larger buffering.

The difficulty we face on clock skew compensation is the sensitive oscillator stability. We faced a tightly V_{CC} coupled clock speed at the operating voltage. To overcome this issue, one may synchronize the network more frequently; but we also suggest stabilizing the supply voltage at each node, suppressing any kind of supply voltage ripples or eventually also switch to a crystal oscillator driven approach. To improve the synchronization accuracy the requirement as formulated in equation (2.17), using an integer multiplier of the Manchester coding's clock as clock rate base, should be modified to use a TimePackage PDU duration as clock rate base. This prevents the discretization error of reference time spans.

7. Conclusion

In this work we focused on some limitations that inhibit the realization of a Shape-Shifting Display as proposed in Lasagni et. al [4, 5]. A Shape-Shifting Display is a mechanical display composed of programmable robotic particles, which can spatially rearrange in order to show arbitrary $3D$ shapes. Robotic particles form a large network. The communication between these particles needs to be scalable to support a growing number of nodes, and time synchronization among the particles must be guaranteed for proper actuation.

According to the imposed requirements, we had to exploit the actuators as communication medium. This allows stable communication with the chosen baud rate of $15.63kBd$ ($0.98kB/s$), which is the fastest stable communication rate our protocol implementation is able to handle.

With respect to the needed accuracy – in the range of seconds – and the given requirements, we have been able to present promising results. Statistically seen, the achieved synchronization looses about $2ms$ of accuracy per node. In a maximum sized network of (255×255) the introduced inaccuracy is expected to be approximately ± 1 second.

Among the major challenges, we had to face the instability of the internal oscillator, which is the local clock source of each node and is tightly coupled to the power supply voltage. Even small ripples of the supply voltage lead to severe nodes de-synchronization.

In order to aid the protocol development and easily assess the performance of our protocol, we have extended an existing AVR simulator to support our network topology. This allowed us to carry out multiple tests, to inspect memory, to visualize the communication packets and estimate the synchronization among nodes. This has constantly driven and sped up our development. Roughly more than 95% of the total development has been achieved with the

simulator only, while the real physical hardware has been used to validate the results.

The outcome consists of a modular hardware, the protocol itself and several simulation tools that make the application quite generic. It is applicable in many different environments not just in a Shape-Shifting Display only, especially in systems where a scalable lightweight hardware implementation is vital.

With the evaluated experiments, we could prove the proposed synchronization mechanism as functional and the protocol as scalable. Beside these results, our experiments emphasized the microcontroller oscillator's frequency sensitivity on supply voltage fluctuations. The specific microcontroller used in our application, ATtiny1634, introduces a relevant microcontroller frequency change, even if the supply voltage fluctuates $50mV$. This experience should be considered from an electrotechnical viewpoint in future work.

Part I.

Appendix

A. Hardware and Network Design Proposal

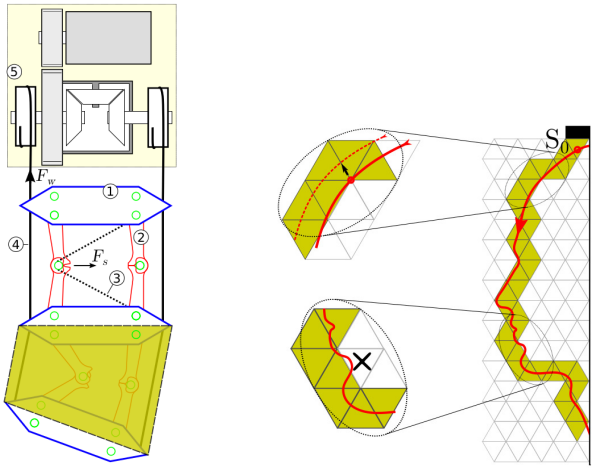
Daisy Chain Communication for long Chains of Robotic Particles

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.



(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.

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

All nodes of a free-hanging particle chain [3] 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.

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 using 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 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

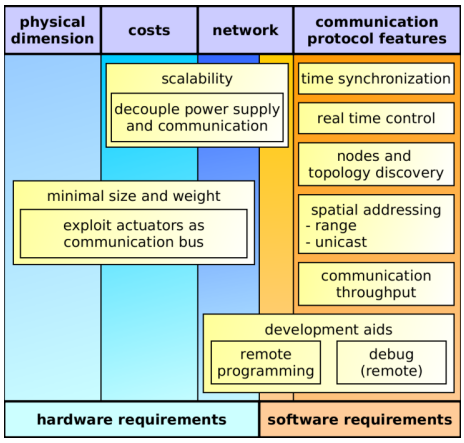


Figure 2: Principal Requirements

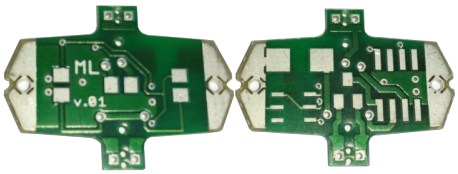


Figure 3: Front and back-side of the current unequipped particle PCB layout [3]. The dimensions are approximately $2cm \times 1cm$.

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.

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 com-

paring 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 its 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.

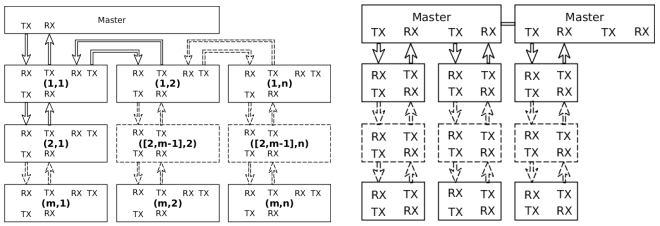
[2]www.sparkfun.com (01/2016)
[3]http://www.nongnu.org/avrdude/ (01/2016)
[4]https://en.wikipedia.org/wiki/Bit_banging (01/2016)

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 be 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 inputting 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.

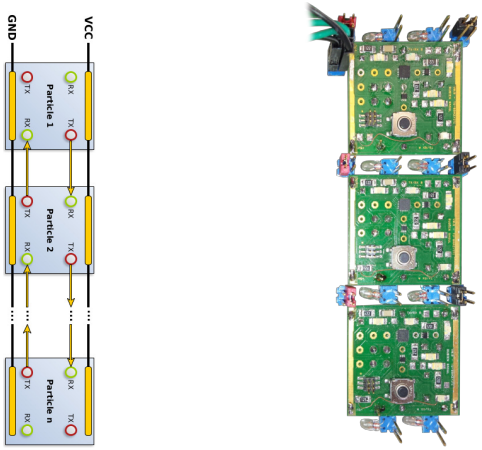


(a) Particle network with cascaded particle chains. They are connected their first particle.

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

Figure 4: Cascaded chains versus direct chain communication. Dashed rectangles represent 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.

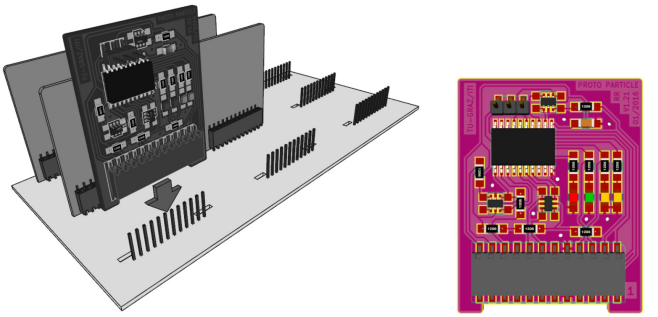


(a) Proposed model of a particle chain. Power supply can be soldered on a stable rail. Actuators may be replaced by 5V light bulbs.

(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 these 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 exemplary 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 Listing

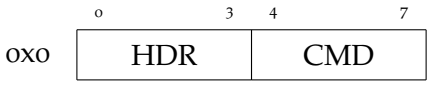


Fig. B.1.: Command

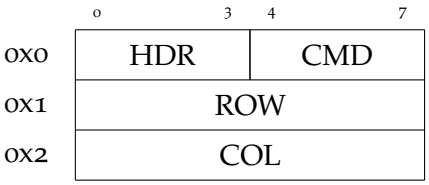


Fig. B.2.: Node command

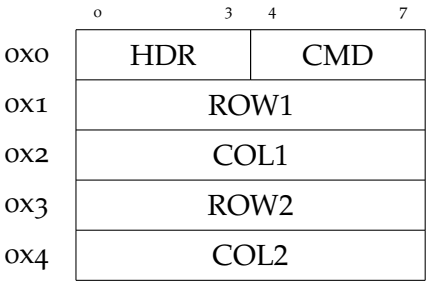


Fig. B.3.: Node range command

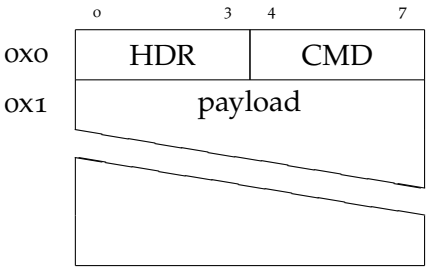


Fig. B.4.: Command with payload

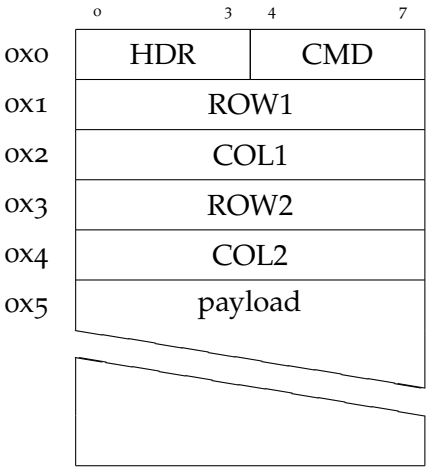


Fig. B.5.: Node range cmd. with payload

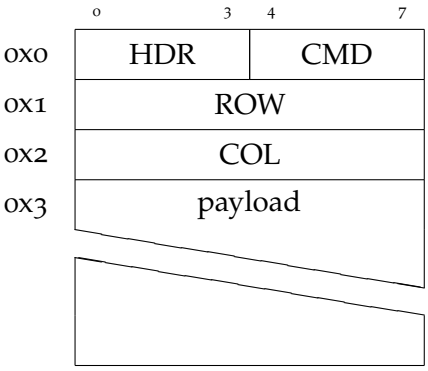


Fig. B.6.: Node command with payload

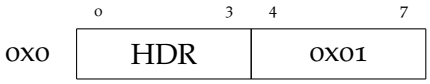


Fig. B.7.: HeaderPackage

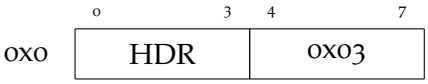


Fig. B.8.: RelayHeaderPackage

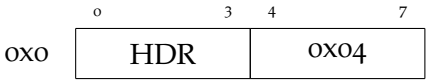


Fig. B.9.: ResetPackage

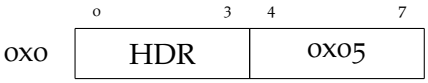


Fig. B.10.: AckPackage

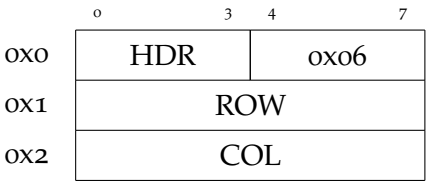


Fig. B.11.: AckWithAddressPackage

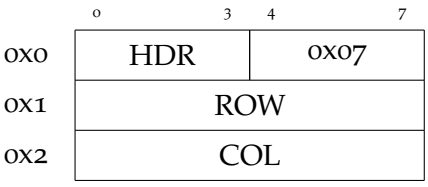


Fig. B.12.: AnnounceNetworkGeometry-Package

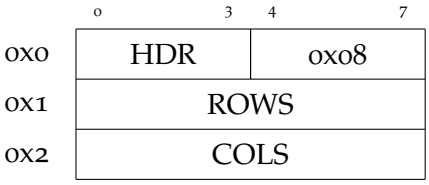


Fig. B.13.: SetNetworkGeometryPackage

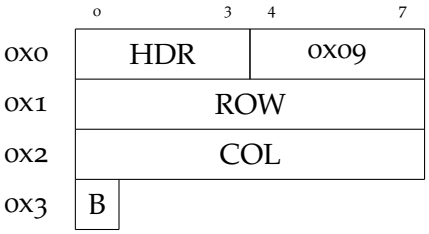


Fig. B.14.: EnumerationPackage

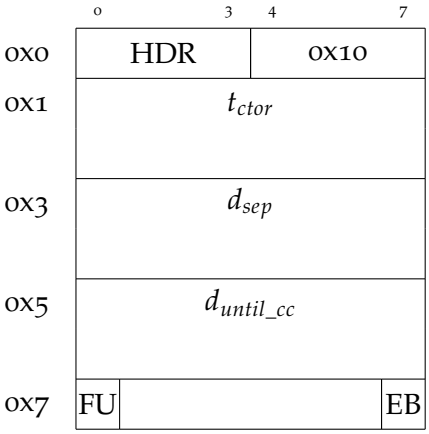


Fig. B.15.: TimePackage

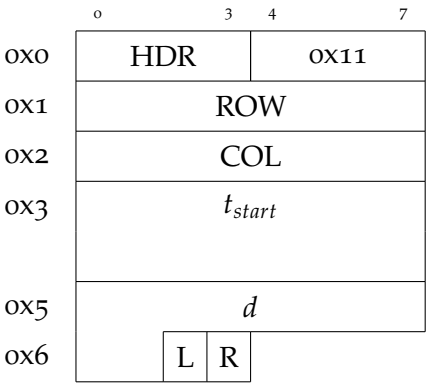


Fig. B.16.: HeatWiresPackage

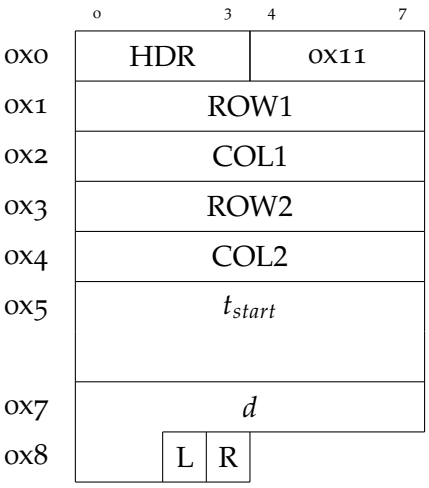


Fig. B.17.: HeatWiresRangePackage

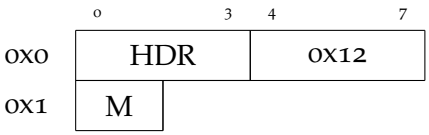


Fig. B.18.: HeatWiresModePackage

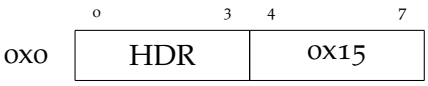


Fig. B.19.: ExtendedHeaderPackage (reserved)

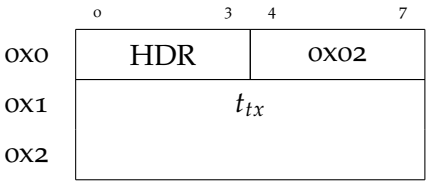


Fig. B.20.: SyncNetworkTimeHeaderPackage

C. Code Snippets

```
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. C.1.: Flow control handling example with Automatic Repeat Request (ARQ) shortcut

D. Node Context

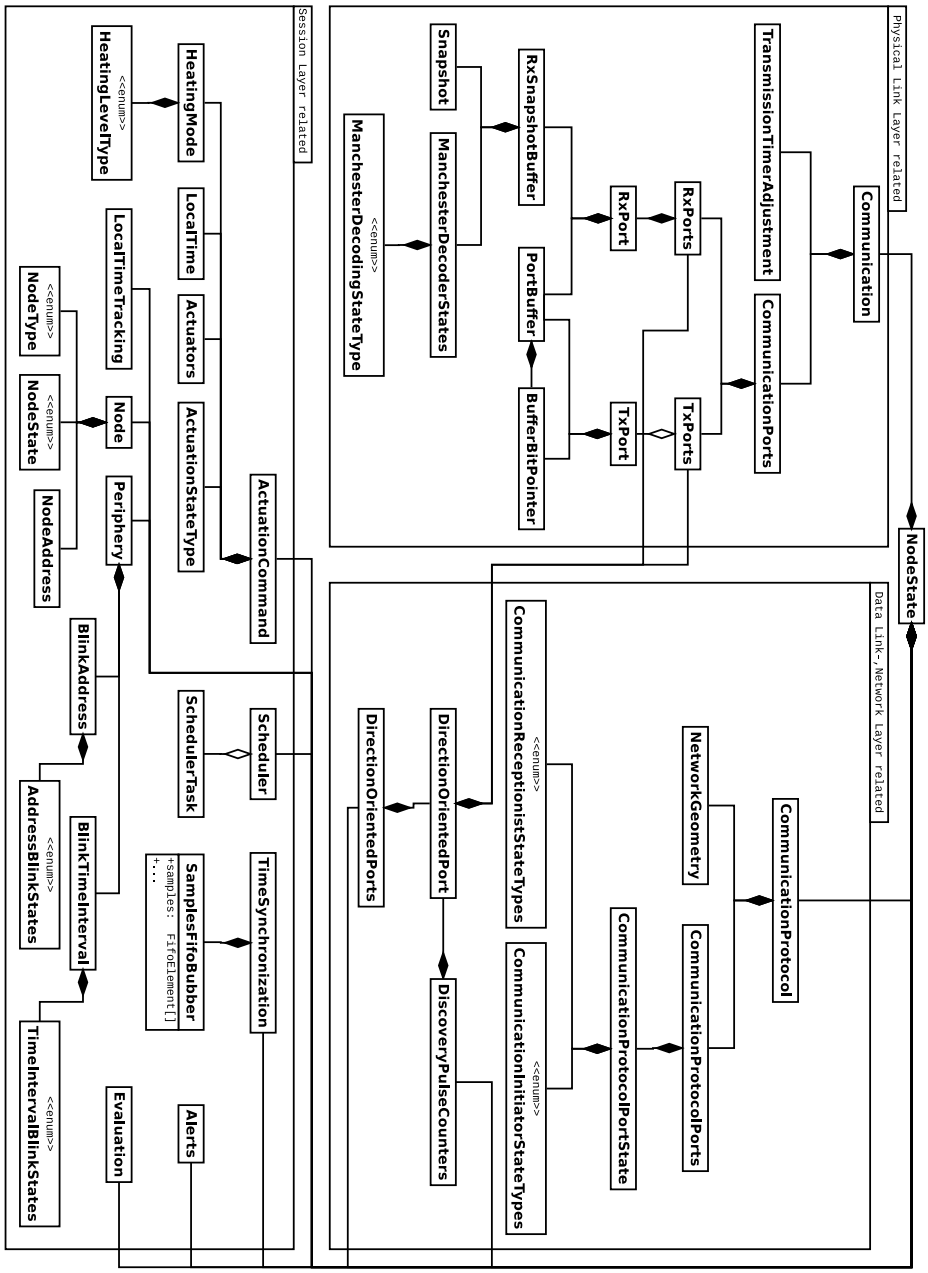


Fig. D.1.: Node's context overview categorized by layers

E. Node States

E. Node States

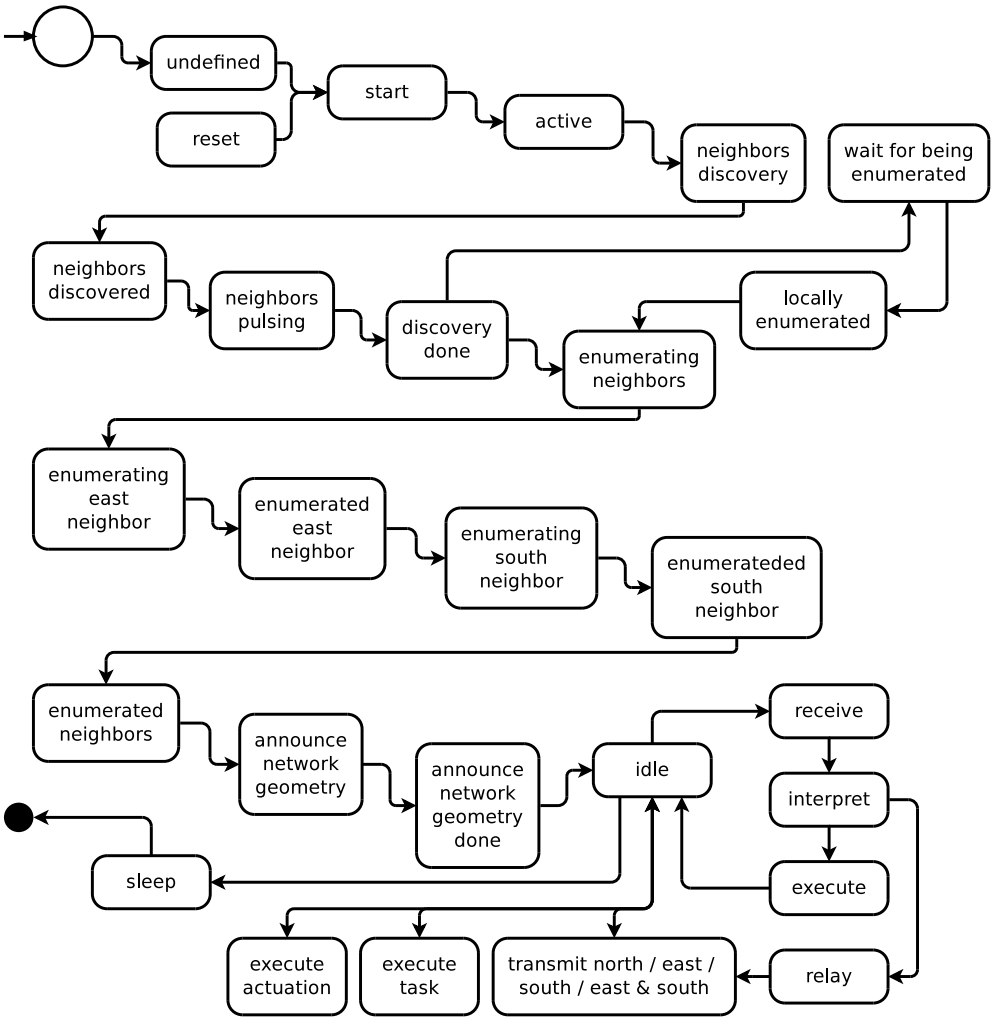


Fig. E.1.: Node's Finite State Machine (FSM) states

F. Configuration Parameter Listing

```
<project>
├─libs/
│ ├─common/
│ ├─simulation/
│ └─uc-core/
│   ├─actuation/
│   ├─communication/
│   ├─communication-protocol/
│   ├─configuration/
│   ├─delay/
│   ├─discovery/
│   ├─evaluation/
│   ├─interrupts/
│   ├─parity/
│   ├─particle/
│   ├─periphery/
│   ├─scheduler/
│   ├─stdout/
│   ├─synchronization/
│   └─time/
└─main/
```

Fig. F.1.: Project files structure

```
configuration/
├─Actuation.h
├─Communication.h
├─CommunicationProtocol.h
├─Discovery.h
├─interrupts/
│ ├─ActuationTimer.h
│ ├─DiscoveryPCI.h
│ ├─DiscoveryTimer.h
│ ├─LocalTime.h
│ ├─ReceptionPCI.h
│ ├─TimerCounter0.h
│ ├─TimerCounter1.h
│ ├─TimerCounter.h
│ ├─TxRxTimer.h
│ └─Vectors.h
├─IoPins.h
├─Particle.h
├─Periphery.h
└─Time.h
```

Fig. F.2.: Configuration files structure

Parameter	Default Argument	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
COMMUNICATION_DEFAULT_TX_RX_CLOCK_DELAY	$((uint16\_t)1024)$	communication/Communication.h
COMMUNICATION_DEFAULT_MAX_SHORT_RECEPTION_OVERTIME_PERCENTAGE_RATIO	$((float)0.75)$	communication/Communication.h
COMMUNICATION_DEFAULT_MAX_LONG_RECEPTION_OVERTIME_PERCENTAGE_RATIO	$((float)1.25)$	communication/Communication.h
COMMUNICATION_TX_RX_NUMBER_BUFFER_BYTES	$((uint8\_t)9)$	communication/Communication.h
RX_NUMBER_SNAPSHOTS	$((uint8\_t)28)$	communication/Communication.h
MANCHESTER_DECODING_RX_NUMBER_SNAPSHOTS	9	communication/ManchesterDecoding.h
consult implementation	–	interrupts/*
DEVIATION_MATH_SQRT	defined	synchronization/Deviation.h
SAMPLE_FIFO_NUM_BUFFER_ELEMENTS	4	synchronization/SampleFiFoTypes.h
TIME_SYNCHRONIZATION_SAMPLES_FIFO_BUFFER_ITERATOR_END	$((uint8\_t)(SAMPLE\_FIFO\_NUM\_BUFFER\_ELEMENTS+1))$	synchronization/SampleFiFoTypes.h
SAMPLE_FIFO_ADAPTIVE_REJECTION_REDUCE_COUNTERS_LIMIT	$((uint16\_t)2000)$	synchronization/SampleFiFoTypes.h
SAMPLE_FIFO_ADAPTIVE_REJECTION_UPDATE_REJECTION_INTERVAL_THRESHOLD	$((uint8\_t)25)$	synchronization/SampleFiFoTypes.h
SAMPLE_FIFO_ADAPTIVE_REJECTION_UPDATE_REJECTION_STEP	$((uint8\_t)10)$	synchronization/SampleFiFoTypes.h
SAMPLE_FIFO_ADAPTIVE_REJECTION_UPDATE_REJECTION_MIN_INTERVAL	$((int8\_t)10)$	synchronization/SampleFiFoTypes.h
SAMPLE_FIFO_ADAPTIVE_REJECTION_UPDATE_REJECTION_MAX_INTERVAL	$((int16\_t)20000)$	synchronization/SampleFiFoTypes.h
SAMPLE_FIFO_ADAPTIVE_REJECTION_ACCEPTANCE_RATIO	$((uint16\_t)9)$	synchronization/SampleFiFoTypes.h
TIME_SYNCHRONIZATION_SAMPLE_OFFSET	$((uint16\_t)INT16\_MAX)$	synchronization/Synchronization.h
SYNCHRONIZATION_TIME_PACKAGE_DURATION_COUNTING_FIRST_TO_LAST_BIT_EDGE	defined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_RAW_OBSERVATION	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_MEAN	defined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_PROGRESSIVE_MEAN	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_MEAN_WITHOUT_OUTLIER	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_MEAN_WITHOUT_MARKED_OUTLIER	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_ENABLE_ADAPTIVE_MARKED_OUTLIER_REJECTION	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_LEAST_SQUARE_LINEAR_FITTING	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_LEAST_SQUARE_LINEAR_FITTING_WITHOUT_OUTLIER	undefined	synchronization/Synchronization.h
SYNCHRONIZATION_OUTLIER_REJECTION_SIGMA_FACTOR	$((CalculationType)2.0)$	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_MEAN_OLD_VALUE_WEIGHT	$((float)0.75)$	synchronization/Synchronization.h
SYNCHRONIZATION_STRATEGY_MEAN_NEW_VALUE_WEIGHT	$((float)0.25)$	synchronization/Synchronization.h
SYNCHRONIZATION_TYPES_CTORS_FIRST_SYNC_PACKAGE_LOCAL_TIME	$((uint16\_t)350)$	synchronization/SynchronizationTypesCtors.h
SYNCHRONIZATION_TYPES_CTORS_FAST_SYNC_PACKAGE_SEPARATION	$((uint8\_t)40)$	synchronization/SynchronizationTypesCtors.h
SYNCHRONIZATION_TYPES_CTORS_SYNC_PACKAGE_SEPARATION	$((uint16\_t)80)$	synchronization/SynchronizationTypesCtors.h
SYNCHRONIZATION_TYPES_CTORS_TOTAL_FAST_SYNC_PACKAGES	$((uint16\_t)30)$	synchronization/SynchronizationTypesCtors.h

Table F.1.: Protocol configuration parameter and default arguments listing (continued in table F.1)

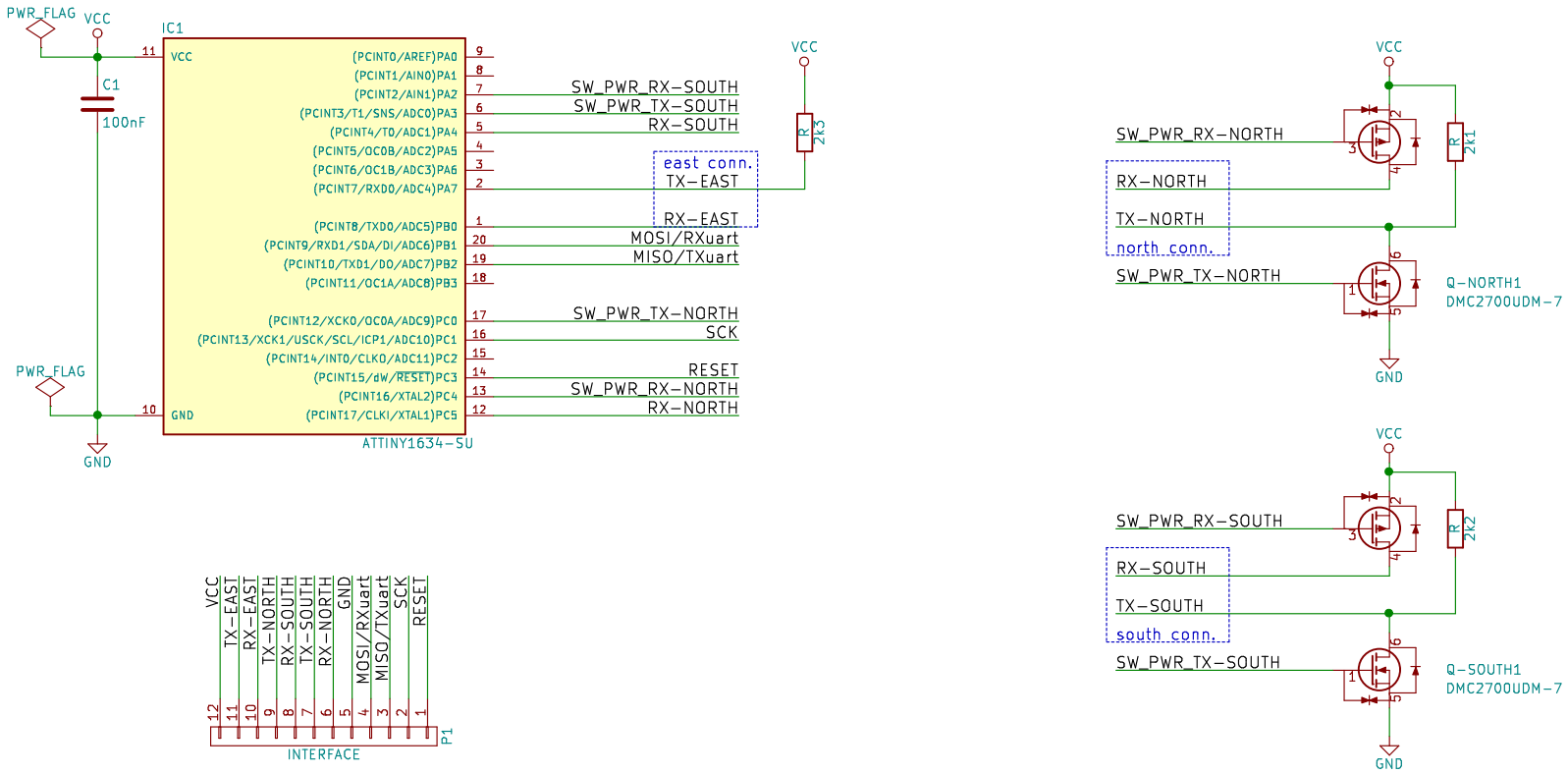
Parameter	Default Argument	File
ACTUATION_COMPARE_VALUE_POWER_STRONG	$((UINT8\_MAX/4)*3)$	Actuation.h
ACTUATION_COMPARE_VALUE_POWER_MEDIUM	$(UINT8\_MAX/2)$	Actuation.h
ACTUATION_COMPARE_VALUE_POWER_WEAK	$(UINT8\_MAX/4)$	Actuation.h
COMMUNICATION_PROTOCOL_TIMEOUT_COUNTER_MAX	$((uint8\_t)250)$	CommunicationProtocol.h
COMMUNICATION_PROTOCOL_RETRANSMISSION_COUNTER_MAX	$((uint8\_t)3)$	CommunicationProtocol.h
RX_DISCOVERY_PULSE_COUNTER_MAX	$((uint4\_t)30)$	Discovery.h
MIN_NEIGHBORS_DISCOVERY_LOOPS	$((uint8\_t)(50))$	Discovery.h
MAX_NEIGHBORS_DISCOVERY_LOOPS	$((uint8\_t)(250))$	Discovery.h
MAX_NEIGHBOR_PULSING_LOOPS	$((uint8\_t)(254))$	Discovery.h
DEFAULT_NEIGHBOR_SENSING_COUNTER_COMPARE_VALUE	$((uint16\_t)0x80)$	Discovery.h
EVALUATION_SIMPLE_SYNC_AND_ACTUATION	defined	Evaluation.h
EVALUATION_SYNC_CYCLICALLY	undefined	Evaluation.h
EVALUATION_SYNC_WITH_CYCLIC_UPDATE_TIME_REQUEST_FLAG	undefined	Evaluation.h
EVALUATION_SYNC_WITH_CYCLIC_UPDATE_TIME_REQUEST_FLAG_IN_PHASE_SHIFTING	undefined	Evaluation.h
EVALUATION_SYNC_WITH_CYCLIC_UPDATE_TIME_REQUEST_FLAG_THEN_ACTUATE_ONCE	undefined	Evaluation.h
see table F.3	–	IoPins.h
LEDS_SUPPRESS_OUTPUT	defined	Leds.h
PARTICLE_DISCOVERY_LOOP_DELAY	$delay\_us(30)$	Particle.h
PARTICLE_DISCOVERY_PULSE_DONE_POST_DELAY	$delay\_ms(3.5)$	Particle.h
PERIPHERY_REMOVE_IMPL	defined	Periphery.h
ADDRESS_BLINK_STATES_LED_ON_COUNTER_MAX	$((uint8\_t)30)$	Periphery.h
ADDRESS_BLINK_STATES_LED_OFF_COUNTER_MAX	$((uint8\_t)30)$	Periphery.h
ADDRESS_BLINK_STATES_LED_SEPARATION_BREAK_COUNTER_MAX	$((uint8\_t)90)$	Periphery.h
ADDRESS_BLINK_STATES_LED_SEPARATION_FLASH_COUNTER_MAX	$((uint8\_t)7)$	Periphery.h
ADDRESS_BLINK_STATES_LED_SEPARATION_LONG_BREAK_COUNTER_MAX	$((uint8\_t)140)$	Periphery.h
TIME_INTERVAL_BLINK_STATES_PERIOD_MULTIPLIER	$((uint8\_t)60)$	Periphery.h
SCHEDULER_MAX_TASKS	5	Scheduler.h
SCHEDULER_TASK_ID_ENABLE_ALERTS	$((uint8\_t)0)$	Scheduler.h
SCHEDULER_TASK_ID_SETUP_LEDS	$((uint8\_t)1)$	Scheduler.h
SCHEDULER_TASK_ID_SYNC_PACKAGE	$((uint8\_t)2)$	Scheduler.h
SCHEDULER_TASK_ID_HEARTBEAT_LED_TOGGLE	$((uint8\_t)3)$	Scheduler.h
SCHEDULER_TASK_ID_HEAT_WIRES	$((uint8\_t)4)$	Scheduler.h
STDOUT_UART_BAUD_RATE	(19200)	Stdout.h
LOCAL_TIME_IN_PHASE_SHIFTING_ON_LOCAL_TIME_UPDATE	defined	Time.h
LOCAL_TIME_IN_PHASE_SHIFTING_MAXIMUM_STEP	$((uint16\_t)2000)$	Time.h
LOCAL_TIME_TRACKING_INT_DELAY_MANCHESTER_CLOCK_MULTIPLIER	$((uint8\_t)51)$	Time.h

Table F.2.: Protocol configuration parameter and default arguments listing (continued)

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 F.3.: Microcontroller Unit (MCU) pinout parameter listing of IoPins.h configuration file

G. Schematic Diagram



Institute of Technical Informatics
Graz University of Technology

Sheet: /
File: particle-tiny1634-pluggable-module.sch

Title: Programmable Matter Particle Concept

Size: A4	Date: 13.12.2016	**Rev: 0.1**
KiCad E.D.A. kicad 4.0.2+e4-622538ubuntu15.10.1-stable		Id: 1/1

H. Network Visualization

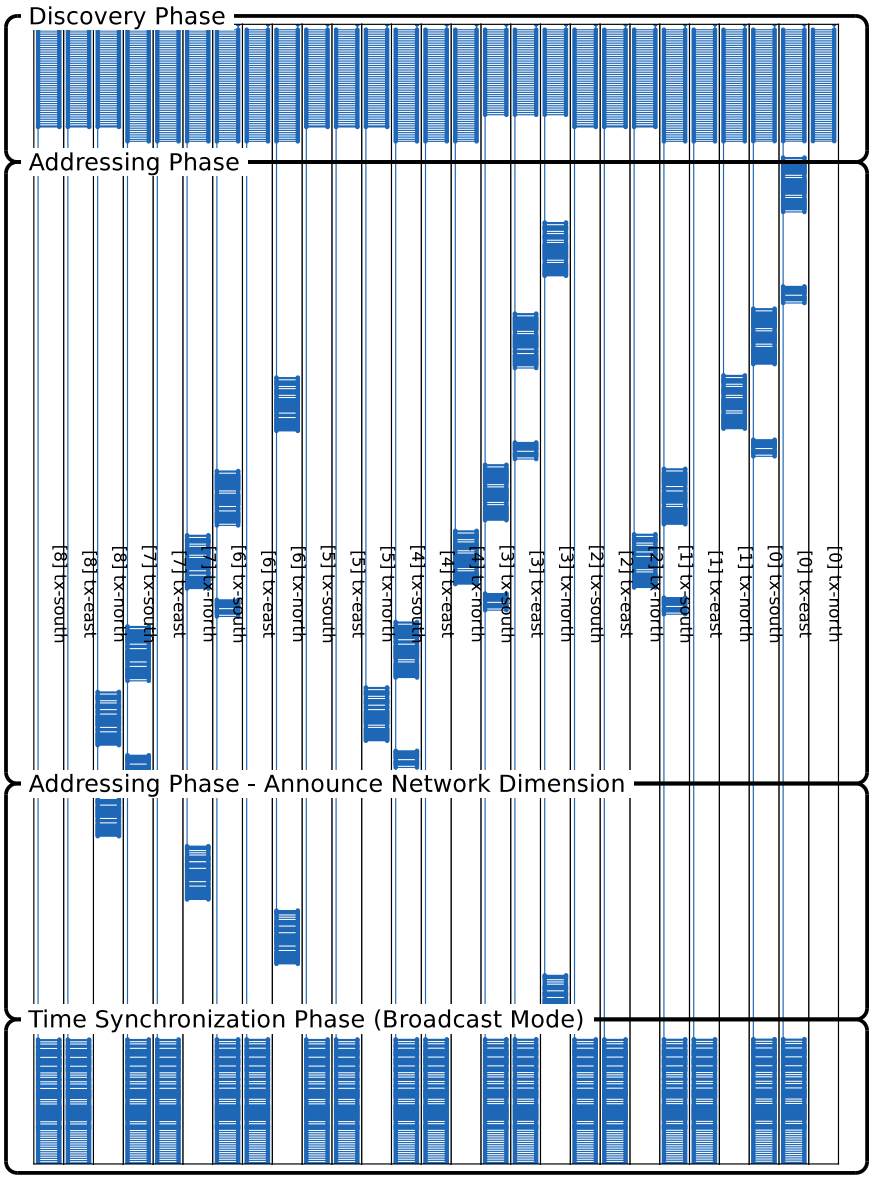


Fig. H.1.: Downscaled (3×3) network visualization showing the communication wires' signals of the network initialization phases applying network time synchronization using broadcast mode; frequent communication signals changes appear as rectangular box

I. 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 I.1.: Introduced forwarding delay in broadcast mode (BCT_{delay}) evaluation of (6×6) network simulation with setup C as listed in table 3.4

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 I.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 I.3.: Evaluation networks and order setup

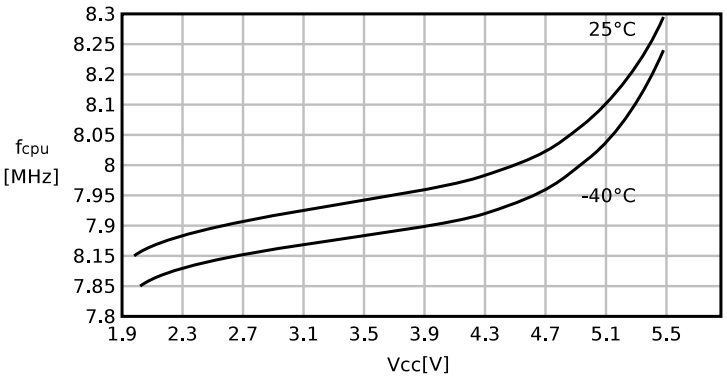


Fig. I.1.: ATtiny1634 MCU clock frequency (f_{cpu}) versus supply voltage (V_{CC}) [16, pp. 272]

Nomenclature

$(m \times n)$ network geometry notation of a network having m rows and n columns
(m, n) network address notation referring to the node at row m and column n
B byte
$BCTE_{delay}$ east port forwarding delay
$BCTS_{delay}$ south port forwarding delay
BCT_{delay} introduced forwarding delay in broadcast mode
Bd baud
PDU_{sep} time delay 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
μ mean
$\oplus$ exclusive-or
$\overline{X}$ reference time span
σ standard deviation
b bit
$buffer\_size$ buffer size
$buffering\_ratio$ reception buffering ratio
d actuation command duration
d_{cc_shift} shift in between t_{cc_tx} and t_{cc}
d_{code} data clock duration of the Manchester code
d_{const} constant duration
d_{ctor_intp} time span since remote PDU construction until local execution
$d_{discrete}$ integer discretization delay
d_{fwd} signal forwarding duration
d_{hop} signal latency of one hop

d_{instr} current instruction duration
d_{intp} interpreter delay
$d_{latency}$ total signal latency
d_{pcif} pin change interrupt flag latency
d_{pdu} PDU transmission duration
d_{pre_tx} constructor to transmission delay
$d_{prologue}$ ISR prologue duration
d_{sep_remote} transmitter's local time clock delay
d_{sep} local time counter clock delay
$d_{until\_cc\_remote}$ delay until next remote local time increment
d_{until_cc} delay until next local time increment
d_{var} variable 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
kB kilo byte
kBd kilo baud
kb kilo bit
$net0$ network configuration setup 0
$net1$ network configuration setup 1
$process()$ main process
t_{cc_tx} remote time ISR compare value
t_{cc} local time ISR compare value
t_{ctor} time of PDU construction
t_{int} time when the PDU is interpreted
t_{now} current local time
t_{start} command start time
t_{tx} transmitter's local time

B network discovery breadcrumb flag
BCT broadcast bit

CMD command id
COL address column
COL1 top left address column

COL2 bottom right address column
COLS network columns

EB end bit

FU force update local time flag

GND ground

HDR package header

L left wire flag

M heating mode

PRT parity bit

R right wire flag
ROW address row
ROW1 top left address row
ROW2 bottom right address row
ROWS network rows

STB start bit

$\mathbf{V_{CC}}$ supply voltage

Glossary

1 **bit even parity** uneven sum of 1 bits results in $PRT = 1$, $PRT = 0$ otherwise
1-**Wire®** a communication bus system; 1-Wire is a registered trademark of Maxim Integrated Products, Inc
$normpdf(\mu, \sigma)$ normal probability distribution function with $mean = \mu$ and standard deviance σ

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
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/

baud rate the rate at which the signal changes
bit bang software driven input/output to emulate an interface
bit oriented a bit oriented protocol framing is not bound to byte boundaries
bit rate the rate of bits per time interval
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
Communication.h communication and line coding configuration file
CommunicationProtocol.h protocol timing configuration file

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, see also https://www.gnu.org/software/ddd/
decoder line code decoder
Deviation.h standard deviation configuration file
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
endianness order of multi-byte values
EnumerationPackage package containing the node's address assignment
Evaluation.h evaluation configuration file
event triggered depending on simulation time
ExtendedHeaderPackage reserved package CMD for extensibility

flash programmable program memory
flow control defines the communication sequence mechanism to ensure communication reliability
frame chunk of data, PDU
fuse essential MCU configuration bits/flags

GDB GNU Project debugger, see also https://www.gnu.org/software/gdb/gdb.html
global routing algorithm routing algo. which uses the total knowledge of a network
gnu99 the C99 with GNU extensions

head node the origin node or any chain's first node having also the east port connected

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 first node having also the east port connected
inter node a node between first and last node 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

jitter time variation in a series of time intervals
JUnit testing Java unit testing framework

Leds.h general LED IO switch configuration file
line code method of coding data on a transmission line
little endian least significant byte stored at lowest address
LocalTime.h local time tracking configuration file

Make GNU make utility to maintain groups of programs
Manchester coding line code incorporating data and clock in one signal, also termed PE
ManchesterDecoding.h Manchester decoding configuration file
marshalling transformation of data to a transport-able format
master device a device sending commands to the network
Maven Java build manager
MOSFET metal-oxide-semiconductor field-effect transistor
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

offline algorithm that needs the whole data to process
online algorithm that processes piece-by-piece
origin node top most, left particle having at least one connection at the east port or south port
orphan node particle without any connection

package mangling package modification before relaying
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
Periphery.h non vital periphery configuration file
physical 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 by simulator when a particular location in the program is reached
programmer hardware to write the MCU flash
protocol stack set of protocol layers

RC circuit RC circuit
real time protocol a protocol that ensures responses within specific time constraints
receiver the node receiving a transmission
ReceptionPCI.h reception interrupt configuration file
RelayHeaderPackage automatically forwarded package to north and east port
ResetPackage package to reset a network node
return to zero signal of consecutive bits returns to zero, even on bits with same value
rooted tree rooted tree network is a tree with a specially designated root node
RS-232 standard for serial communication

SampleFiFoTypes.h FIFO buffer configuration file

Scheduler.h scheduler configuration file
session layer ensures reliability and automatic recovery, also termed layer 5
SetNetworkGeometryPackage package stating a new network geometry
Shape-Shifting Display a mechanical display that is able to approximate $3D$ surfaces
SimulAVR simulator for the Atmel AVR family http://www.nongnu.org/simulavr/
south port particle's bottom/south TX/RX connection wires
spanning tree a spanning sub graph of a graph
State pattern design pattern to implement behavior changes according to a context
Stdout.h $printf(...)$ configuration file
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
Synchronization.h synchronization types configuration file
SynchronizationTypesCtors.h synchronization types configuration file

tail node last particle of a chain
Time.h local time tracking configuration file
TimePackage synchronization package
TimerCounter0.h timer counter configuration file
TimerCounter1.h timer counter configuration file
TimerCounter.h timer counter configuration file
transport layer transforms packets to data, also termed layer 4
tree network a connected network that contains no cycle
TxRxTimer.h TX/RX configuration file

unicast sending to one network participant
unipolar a signal that uses two polarities according to a reference point
unipolar a signal that uses one polarity according to a reference point
unmarshalling transformation of transport-able format to data

Vectors.h ISR vectors configuration file

watch triggered by simulator when a particular location in the memory is modified

Acronyms

ACK acknowledgement
ARQ Automatic Repeat Request

Bi-ϕ-L Bi-Phase-Level

CSMA Carrier Sense Multiple Access
CTC clear timer on compare match mode

DOF degree of freedom

FIFO First In First Out
FSM Flying Spaghetti Monster
FSM Finite State Machine

GCC GNU Compiler Collection

ISR Interrupt Service Routine

JSON JavaScript Object Notation

LCD Liquid Crystal Display
LED Light Emitting Diode
LSB least significant bit or byte

MCU Microcontroller Unit
MLS Moving Least Squares
MTU Maximum Transfer Unit

NRZ Non Return to Zero
NRZ-L NRZ-Level

OOP Object Oriented Programming

OSCCAL internal RC oscillator calibration register
OSI Open System Interconnect
OSSRH Open Source Sonatype Repository Hosting

P2P Peer-to-Peer
PCB printed board circuit
PCI Pin Change ISR
PCIF Pin Change Interrupt Flag
PCM Pulse-Code Modulation
PDU Protocol Data Unit
PE Phase Encoding
PHY Physical Layer
PWM Pulse Width Modulation

RAM Random Access Memory
RC resistor-capacitor
ROV Raw Observation Value
RTC Real Time Clock
RX reception
RZ Return to Zero

SMA Shape Memory Alloy
SMAV Simple Moving Average
SPI Serial Programming Interface
SRAM static RAM

TCNT Timer/Counter
TP test point
TUI Tangible User Interface
TX transmission
TX/RX transmission/reception

WMA Weighted Moving Average

Bibliography

[1] M. K. Rasmussen, E. W. Pedersen, M. G. Petersen, and K. Hornbæk, "Shape-changing interfaces: A review of the design space and open research questions," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12, Austin, Texas, USA: ACM, 2012, pp. 735–744 (cit. on pp. 1, 101).

[2] A. D. Kapadia, I. D. Walker, K. E. Green, J. C. Manganelli, H. Houayek, A. M. James, V. K. T. Kanuri, T. Mokhtar, I. Siles, and P. Yanik, "Rethinking the machines in which we live: a multidisciplinary course in architectural robotics," *IEEE Robotics and Automation Magazine*, vol. 21, no. 3, pp. 143–150, Aug. 2014 (cit. on p. 1).

[3] M. D. Gross and K. E. Green, "Architectural robotics, inevitably," *Interactions*, vol. 19, no. 1, pp. 28–33, Jan. 2012 (cit. on p. 1).

[4] 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 (cit. on pp. 1, 2, 6, 107).

[5] M. Lasagni and K. Römer, "Force-guiding particle chains for shape-shifting displays," *CoRR*, vol. abs/1402.2507, 2014 (cit. on pp. 2, 4, 107).

[6] 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 (cit. on p. 3).

[7] B. L. Kelly, "Beam shape control using shape memory alloys.," DTIC Document, Tech. Rep., 1998 (cit. on p. 3).

[8] Maxim, *Guidelines for reliable 1-wire networks*, Appnote148, Maxim Integrated Products, Nov. 2001. [Online]. Available: http://www.maxim-ic.com/an148 (cit. on p. 4).

[9] 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. 4).

[10] D. Comer, *Computernetzwerke und Internets: Mit Internet-Anwendungen*. München: Pearson Studium, 2002, ISBN: 382737023x (cit. on pp. 6, 9).

[11] B. Sklar, *Digital communications: Fundamentals and applications*. Upper Saddle River, N.J: Prentice-Hall PTR, 2001, ISBN: 0130847887 (cit. on pp. 9, 13, 14).

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

[13] R. Williams, *Computer systems architecture: A networking approach*. Harlow, England New York: Addison-Wesley, 2001, ISBN: 0201648598 (cit. on p. 13).

[14] G. Coulouris, *Distributed systems: Concepts and design*. Harlow, England New York: Addison-Wesley, 2005, ISBN: 0321263545 (cit. on p. 13).

[15] L. Peterson, *Computer networks: A systems approach*. Amsterdam Boston: Morgan Kaufmann Publishers, 2003, ISBN: 155860832x (cit. on pp. 14, 20).

[16] 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. 16, 33, 53, 55, 139).

[17] J. Reichardt, *Lehrbuch Digitaltechnik: Eine Einführung mit VHDL*. München: Oldenbourg, 2011, ISBN: 9783486706802 (cit. on p. 19).

[18] U. Hammerschall, *Verteilte Systeme und Anwendungen: Architekturkonzepte, Standards und Middleware-Technologien*. München Boston u.a.: Pearson Studium, 2005, ISBN: 3827370965 (cit. on p. 21).

[19] K. Schmaranz, *Softwareentwicklung in C++*. Berlin u.a.: Springer, 2003, ISBN: 3540443436 (cit. on p. 21).

[20] J. Kurose, *Computernetze: Ein Top-Down-Ansatz mit Schwerpunkt Internet*. München: Pearson Studium, 2002, ISBN: 3827370175 (cit. on p. 28).

[21] 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. 36).

[22] E. Gamma, *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. München Boston u.a.: Addison-Wesley, 2004, ISBN: 3827321999 (cit. on pp. 39, 46).

[23] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji, "M-tran: Self-reconfigurable modular robotic system," *IEEE/-ASME Transactions on Mechatronics*, vol. 7, no. 4, pp. 431–441, Dec. 2002 (cit. on pp. 43, 102).

[24] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 147–163, 2002 (cit. on p. 49).

[25] 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 (cit. on p. 57).

[26] 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 (cit. on p. 58).

[27] M. Jorgensen, E. Ostergaard, and H. Lund, "Modular atron: Modules for a self-reconfigurable robot," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 2, pp. 2068–2073, 2004 (cit. on p. 101).

[28] K. C. Cheung, E. D. Demaine, J. R. Bachrach, and S. Griffith, "Programmable assembly with universally foldable strings (moteins)," *IEEE Transactions on Robotics*, vol. 27, no. 4, pp. 718–729, Aug. 2011 (cit. on p. 101).

[29] A. N. Knaian, K. C. Cheung, M. B. Lobovsky, A. J. Oines, P. Schmidt-Neilsen, and N. 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, pp. 1447–1453, 2012 (cit. on p. 101).

[30] P. J. White, M. L. Posner, and M. Yim, "Strength analysis of miniature folded right angle tetrahedron chain programmable matter," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2010, pp. 2785–2790 (cit. on p. 101).

[31] H. Ishii, D. Lakatos, L. Bonanni, and J.-B. J. Labrune, "Radical atoms: Beyond tangible bits, toward transformable materials," vol. XIX, no. February, pp. 31–51, 2012 (cit. on p. 101).

[32] C. Khoo and F. Salim, "Lumina: A soft kinetic material for morphing architectural skins and organic user interfaces," *Proceedings of the 2013 ACM international joint . . .*, pp. 53–62, 2013 (cit. on p. 101).

[33] N. Correll, Ç. D. Önal, H. Liang, E. Schoenfeld, and D. Rus, "Soft autonomous materials—using active elasticity and embedded distributed computation," in *Experimental Robotics: The 12th International Symposium on Experimental Robotics*, O. Khatib, V. Kumar, and G. Sukhatme, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 227–240 (cit. on p. 101).

[34] M. Weller, E. Do, and M. D. Gross, "Posey: Instrumenting a poseable hub and strut construction toy," in *Proceedings of the 2nd international conference on Tangible and Embedded Interaction*, ACM, 2008, pp. 39–46 (cit. on pp. 101, 103).

[35] M. P. Weller, M. D. Gross, and S. C. Goldstein, "Hyperform specification: Designing and interacting with self-reconfiguring materials," *Personal and Ubiquitous Computing*, vol. 15, no. 2, pp. 133–149, 2011 (cit. on p. 101).

[36] K. Gilpin, K. Kotay, and D. Rus, "Miche: Modular shape formation by self-dissasembly," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2007, pp. 2241–2247 (cit. on p. 102).

[37] S. Goldstein and T. Mowry, "Claytronics: A scalable basis for future robots," *Robosphere*, pp. 1–6, 2004 (cit. on p. 102).

[38] M. Yim, D. Duff, and K. Roufas, "Polybot: A modular reconfigurable robot," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 1, 2000, pp. 514–520 (cit. on p. 102).

[39] M. Jorgensen, E. Ostergaard, and H. Lund, "Modular atron: Modules for a self-reconfigurable robot," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 2, pp. 2068–2073, 2004 (cit. on p. 102).

[40] V. Zykov, E. Mytilinaios, M. Desnoyer, and H. Lipson, "Evolved and designed self-reproducing modular robotics," *IEEE Transactions on Robotics*, vol. 23, no. 2, pp. 308–319, 2007 (cit. on p. 102).

[41] B. Kirby, J. Campbell, B. Aksak, and P. Pillai, "Catoms: Moving robots without moving parts," in *Proceedings of the ...*, vol. 20, 2005, p. 1730 (cit. on p. 103).

[42] B. T. Kirby, B. Aksak, J. D. Campbell, J. F. Hoburg, T. C. Mowry, P. Pillai, and S. C. Goldstein, "A modular robotic system using magnetic force effectors," in *IEEE International Conference on Intelligent Robots and Systems*, 2007, pp. 2787–2793 (cit. on p. 103).

[43] M. Follador, M. Cianchetti, A. Arienti, and C. Laschi, "A general method for the design and fabrication of shape memory alloy active spring actuators," *Smart Materials and Structures*, vol. 21, no. 11, p. 115 029, 2012 (cit. on p. 103).

[44] G. Song and N. Ma, "Robust control of a shape memory alloy wire actuated flap," *Smart Materials and Structures*, vol. 16, no. 6, N51, 2007 (cit. on p. 103).

[45] P. J. White and M. Yim, "Scalable modular self-reconfigurable robots using external actuation," in *IEEE International Conference on Intelligent Robots and Systems*, 2007, pp. 2773–2778 (cit. on p. 103).

[46] 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*, vol. 107, no. 28, pp. 12 441–5, 2010 (cit. on p. 103).

[47] M. Coelho and J. Zigelbaum, "Shape-changing interfaces," *Personal and Ubiquitous Computing*, vol. 15, no. 2, pp. 161–173, Feb. 2011 (cit. on p. 103).

[48] J. Campbell and P. Pillai, "Collective actuation," *The International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 299–314, 2008 (cit. on p. 103).

[49] M. De Rosa, S. C. Goldstein, P. Lee, P. Pillai, and J. Campbell, "A tale of two planners: Modular robotic planning with ldp," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, Dec. 2009, pp. 5267–5274 (cit. on p. 104).

[50] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *IEEE International Conference on Intelligent Robots and Systems*, 2007, pp. 2794–2800 (cit. on p. 104).

[51] D. J. Christensen and H. H. Lund, "Metamodule control for the atron self-reconfigurable robotic system," in *Proceedings of the The 8th Conference on Intelligent Autonomous Systems*, 2004, pp. 685–692 (cit. on p. 104).

[52] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, "A language for large ensembles of independently executing nodes," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5649 LNCS, 2009, pp. 265–280 (cit. on p. 104).

[53] C. Unsal and P. K. Khosla, "A multi-layered planner for self-reconfiguration of a uniform group of i-cube modules," *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 1, 598–605 vol.1, 2001 (cit. on p. 104).

Index