LumiNet

An Organic Interactive Illumination Network

Diploma Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
René Bohne

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr.-Ing. Stefan Kowalewski

Registration date:	Oct 14th, 2008
Submission date:	May 20th, 2009

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, Mai 2009
René Bohne

Contents

List of Figures

List of Tables

Abstract

In this thesis paper we propose a distributed physical computing framework for low-cost microcontroller-based devices. The framework can be used to implement bio-inspired algorithms in a network of identical target nodes. In addition to the implementation of this framework, bio-inspired examples are provided with this thesis work.

We provide a brief survey of physical computing frameworks, including wearable computing, and analyze different network reprogramming protocols from wireless sensor networks. We also introduce examples from bio-inspired computing, like spiking neural networks, genetic algorithms, and cellular automata.

While we discussed to implement a new framework from scratch when we started this work, we ended up with a modified version of an existing, widely used physical computing framework. We extended it not only to support the LumiNet hardware, but also added other features like, e.g., Assembly language support.

None of the analyzed physical computing frameworks supports distributed computing or network reprogramming, so one main goal of this thesis is to add these features to the new framework.

Überblick

In dieser Diplomarbeit stellen wir ein verteiltes physical computing Framework für kostengünstige mikrocontroller-basierte Geräte vor. Das Framework kann dazu genutzt werden, um biologisch-inspirierte Algorithmen in einem Netzwerk von identischen Zielknoten zu implementieren. Zusätzlich zur Implementation des Frameworks werden biologisch-inspirierte Beispiele mit dieser Arbeit veröffentlicht.

Wir geben eine grobe Übersicht über physical computing Frameworks, wearable computing einbegriffen, und analysieren verschiedene network reprogramming Protokolle von drahtlosen Sensornetzwerken. Wir stellen ausserdem Beispiele aus dem Bereich bio-inspired computing vor, wie z.B. spiking neural networks, Genetische Algorithmen und Zelluläre Automaten.

Während wir zu Beginn noch darüber diskutiert haben, ein neues Framework von Null an neu zu entwerfen, endeten wir mit einer Modifikation eines bestehenden, weitverbreiteten physical computing Frameworks. Dieses haben wir nicht nur um die Fähigkeit erweitert, die LumiNet Hardware zu unterstützen, sondern es wurden auch neue Features hinzugefügt, wie z.B. Unterstützung der Assembler Programmiersprache.

Da keins der untersuchten Frameworks distributed computing oder die Codeverbreitung via network reprogramming unterstützt, wurde es zu einer Kernaufgabe, diese Elemente in das neue Framework einzubauen.

Acknowledgements

I want to thank Professor Dr. Jan Borchers for inviting me to work on his LumiNet project. All credits to the hardware design go to him and many concepts that I implemented in this work are based on his visions.

I want to thank my supervisor, Gero Herkenrath, for his support and feedback. In numerous discussions with Professor Borchers and Gero Herkenrath the design of the system evolved into what it is now.

Additionally, I want to thank Professor Dr. Stefan Kowalewski for beeing the second examiner.

I also want to thank Dr. Walter Unger at the Lehrstuhl für Informatik 1 at RWTH Aachen University for the inspiring talk about distributed algorithms, and Olaf Landsiedel at the Distributed Systems Group of RWTH Aachen University for the meeting about distributed systems and sensor networks.

I want to thank James "Laen Finehack" Neal from Portland for his explanations of the Bynase protocol and David A. Mellis of the Arduino development team for his information and advice about the Arduino core.

Special thanks go to Anja Müschen. She was the only person not affiliated to RWTH Aachen University, who reviewed this thesis.

Finally (and most importantly) I want to thank my parents. I would not have been able to pay the tuition fees without their financial support.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

SENSOR NETWORK:
A sensor network is a set of devices - distributed in a network - which use communication lines (wireless or wired) to exchange sensor information

Definition:
Sensor Network

Source code and implementation symbols are written in typewriter-style text.

```
uint8_t r = 0;
```

The whole thesis is written in American English.

The plural “we” will be used throughout this thesis instead of the singular “I”, even when referring to work that was primarily or solely done by the author.

Chapter 1

Introduction

Recently there has been a great interest in organic user interfaces (OUIs), physical computing frameworks and bio-inspired algorithms.

Classic user interfaces (UIs) mainly consist of output devices like displays and input devices like a mouse or a keyboard. Most OUIs use flexible displays or projectors as output devices and sensors like accelerators as input devices. Unlike classic UIs that use the desktop metaphor, OUIs mimic things of daily life. If a UI feels natural to the user then it can be considered to be an OUI. A principle for OUIs is that input equals output. Deformable objects make use of this principle as deformation can be input that a user puts into the system or it can be output that a user experiences. Current OUIs use the same hardware as desktop computers and in some cases this might be limiting. A central computing unit with a flexible display and sensors that measure physical variables can be sufficient for an OUI, but in nature many things don't have a central computing device. (Many ideas in this paragraph were taken from or were inspired by [Holman and Vertegaal, 2008])

Cells, neurons and synapses can be role models for distributed computing devices in OUIs.

Physical computing frameworks can be used for creating prototypes of OUIs. They include all the parts needed to create devices that can interact with the physical world. Most physical computing frameworks have a central computing unit with many sensors and actuators connected to it. A physical computing framework that supports distributed computing has the potential to bring the benefits of neural networks into the world of organic user interfaces.

Bio-inspired algorithms take principles from nature and use them to solve problems. Most bio-inspired algorithms don't look at a single individual, but on a population of individuals. For example, evolutionary algorithms consider a set of possible solutions and only the fittest of those solutions are accepted. Using bio-inspired algorithms in organic user interfaces can support the natural feel of the system.

In this work a distributed physical computing framework is introduced. It adds distributed computing into the world of physical computing frameworks.

While the following sections explain some basic terms of this paper, the following chapters present a new distibuted physical computing framework that supports low-cost microcontroller boards.

2—"Related work" In this chapter we give an overview of related work that influenced this thesis or presents similar ideas.

3—"Design" explains the design of the framework. Requirements of the implementation are defined and a brief description of early implementations is given in this chapter.

4—"Implementation" gives an overview about the final implementation of the distributed physical computing framework.

5—"Evaluation" analyzes the implementation and gives examples for applications that use the new framework.

6—"Summary and Future Work" summarizes this work and identifies its contributions. It also gives an outlook on future work.

1.1 (Wireless) Sensor Networks

Wireless sensor networks face the same problems that physical computing frameworks have. Sensors must be supported, incoming sensor data must be processed, and in some cases output variables must be changed. Unlike current physical computing frameworks, wireless sensor networks already support and use distributed computing.

SENSOR NETWORK:
A sensor network is a set of devices - distributed in a network - which use communication lines (wireless or wired) to exchange sensor information.

Definition:
sensor network

The main task of a (wireless) sensor network is to monitor physical values like temperature, force, sound or pressure. The devices operate autonomously in order to collect their local values, but the cooperation of many devices in a network allows data exchange and distributed calculations. Sensor networks do not need to be wireless, but in recent years, wireless sensor networks became more important than wired solutions. A typical sensor node is equipped with a microcontroller, some sort of communication interface, and on-board sensors or connectors for off-board sensors. Wireless sensor nodes contain a radio transceiver and a battery.

The wireless structure allows a larger set of possible spatial distributions of the sensor nodes compared to wired sensor networks. But there are still some interesting applications left where wired sensor networks have benefits. An example is wearable computing, where a wired sensor network could be distributed over the body and the wired architecture keeps the overall structure of the network simple and reduces costs, while sensor data like temperature or heartbeat-rate can easily be exchanged. Another example is home automation. It is easy to use a wired network for home automation when a new house is constructed. Installing wires in an already existing house is more difficult than installing a wireless sensor network, but it is not impossible and has some benefits like power supply or faster reaction times, no radiation, better security, and less disturbance caused by noise.

Wireless networks are often ad-hoc networks and they have special demands for communication protocols and routing strategies, derived from existing solutions used in the internet.

A wired network can be a prototype for a wireless network in some cases, allowing tests of some core algorithms and ideas.

1.1.1 Network Reprogramming

The nodes of a network can be reprogrammed one by one using in system programming (ISP). This has several disadvantages, e.g., the network must be disassembled and each node has to be programmed using a PC and special programming hardware. Or the ISP pin headers of each node must be accessible so that a mobile programmer can upload new program code to every single node.

An easier solution would be to leave the network as it is and to upload new program code only to a single node of the network. Then this node propagates the code to other nodes in the network until every node has received the new program code. This is called network reprogramming.

1.2 Wearable Computing

Wearable Computing is a new research field that adds small computers and electrical parts like LEDs to clothes. It can be used to motivate beginners for microcontrollers and electrical engineering. Even children can build their own electronic clothes by using a simple framework. This research field also deals with real-world problems like how to create circuits on fabric, how to attach devices that support body-movement or how to make sure that all parts are washable. Today wearable computing projects use low-cost parts and wires. It is possible to use more than one microcontroller in a wearable computing project, so that it can be considered a wired sensor network. Because the controllers are usually fixed on the clothing, reprogramming the network is a problem, especially if the devices are hard to reach because they are hidden under fabric. Some projects and examples for wearable computing can be found in chapter 2—"Related work".

1.3 Bio-inspired Artificial Intelligence

Bio-inspired or biologically-inspired means that an artificial system is inspired by nature. The term ″organic″ can have many meanings and is used by artists and scientists for different things. In this thesis, we use the word ″organic″ as a synonym for ″bio-inspired″.

There are several levels of abstraction at which a system can be inspired by nature:

- User Interface (UI)
- Algorithms
- Communication Protocols
- Firmware / Operating System
- Hardware

Bio-inspired **user interfaces** are sometimes called organic user interfaces, although there might exist organic user interfaces that are not inspired by biology at all.

Algorithms can be inspired, e.g., by the behavior of species (swarm intelligence) or by evolution (evolutionary algorithms). Cellular automata are an example for a bio-inspired structure that can be the foundation for many algorithms. Some bio-inspired algorithms have been implemented on LumiNet for this work. Chapters 5.2.3 and 5.2.4 show these examples.

Higher-level **Communication protocols** can be inspired by language or social behavior of a species. Lower-level communication can be inspired by analog - electrical or chemical - information exchange like the one between cells. While serial communication protocols are very popular in information technology today, cells of the body or neurons in the brain do not use serial communication.

The **firmware** level connects the higher levels with the hardware. On more sophisticated devices, an operating system replaces the firmware. An operating system could use bio-inspired filesystems and inter-process communication, extending the parallel and distributed capabilities of the operating system.

On the **hardware** level, the circuit design can be inspired by nature. Hardware can even be used to interface with biological cells. This allows communication with the human brain or other parts of the body on a physical level.

Chapter 2

Related work

In this chapter, we will provide an overview of research publications and commercial products in the fields of physical computing frameworks, wearable computing, bio-inspired computing, and wireless sensor networks.

Related work was searched by using the following terms: bio-inspired, organic, bootloader, sensor network, wearable computing, network reprogramming, smart LED.

2.1 Physical Computing Frameworks

2.1.1 Wiring

Wiring[1] is an open source physical computing framework. It consists of a graphical development environment and a hardware board. The project addresses projects in the field of (electronic) arts, teaching, electronic prototyping, and tangible media.

Currently, two hardware boards are available: the Wiring I/O Board and the Wiring Mini. They have 43 digital I/O pins and 8 analog input pins, powered by an Atmel AT-Mega128 microcontroller. This controller has two hardware

[1]http://www.wiring.org.co/

UARTs and supports the Inter-Integrated Circuit (I^2C) interface. One serial connection is connected to the USB port and the other one can be used for applications. The boards can be programmed via USB.

2.1.2 Arduino

The Arduino[2] physical computing platform consists of a microcontroller-based I/O board and cross-platform Java IDE. The Arduino Diecimila hardware board (see figure 2.1) has 14 digital I/O pins, 6 analog input pins, a 5V-regulator, a USB to serial bridge, and an Atmel ATmega168 mcu that operates at 16 MHz. While older boards used an ATmega8 mcu, newer boards now use an ATmega328 mcu. The Arduino board[3] is based on the Wiring I/O board. The programs that run on this hardware are called sketches. The main Arduino I/O board can be extended by so-called shields. A common pin header allows to connect shields to the Arduino board in only one direction. Shields can contain any electronic components, like accelerometers, light sensors, temperature sensors or other pin headers that allow to connect external peripherals. For example, there exists a shield that allows to control DC motors and a shield with a breadboard attached to it. With the Ethernet shield the Arduino gets connected to a local area network and possibly to the internet. For wireless applications an XBee shield can be connected to the Arduino. For almost all shields, an open-source library that allows to write sketches, which use the shield, is available. A typical Arduino hardware board costs about USD 30. [Igoe, 2007] gives many examples for everyday problems that can be solved with the Arduino. Sensors and communication are very well explained in this book.

The Arduino IDE looks similar to the Processing[4] IDE and originated from it. Sketches can be programmed in a programming language that is very similar to the Wiring programming language, which is a simplified version of C. Actually, the compiler, which is used, is the gnu gcc compiler,

[2]http://www.arduino.cc
[3]in this work, this refers to the Arduino Diecimila
[4] http://www.processing.org

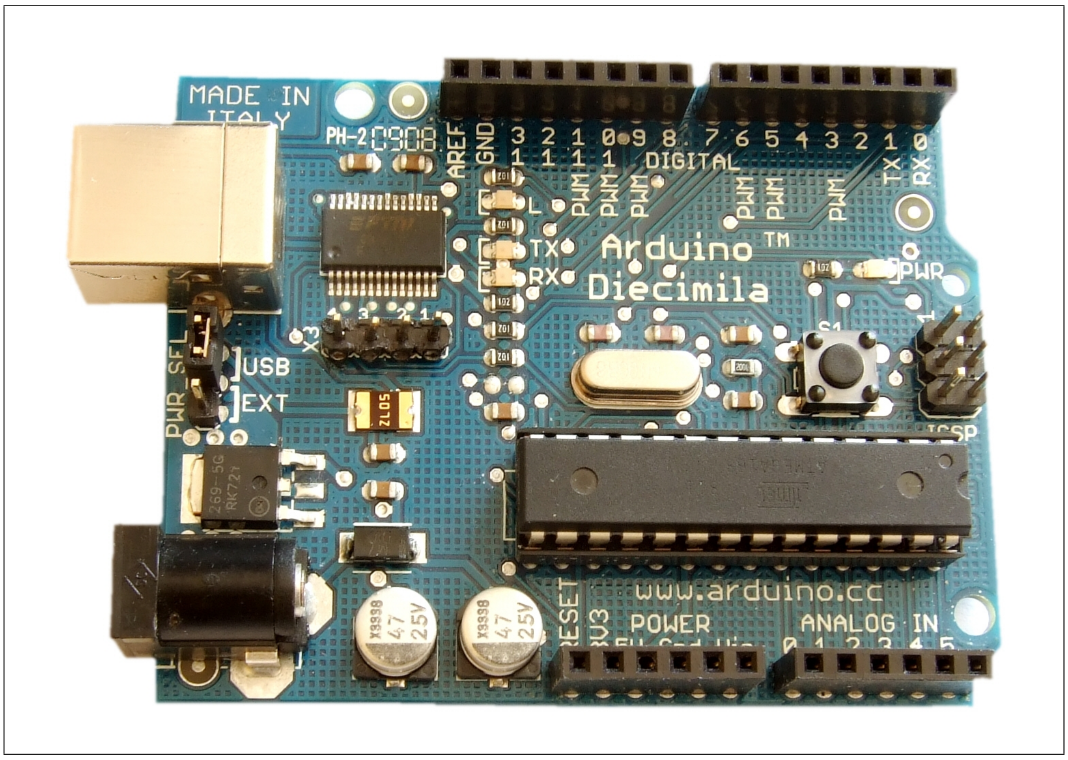
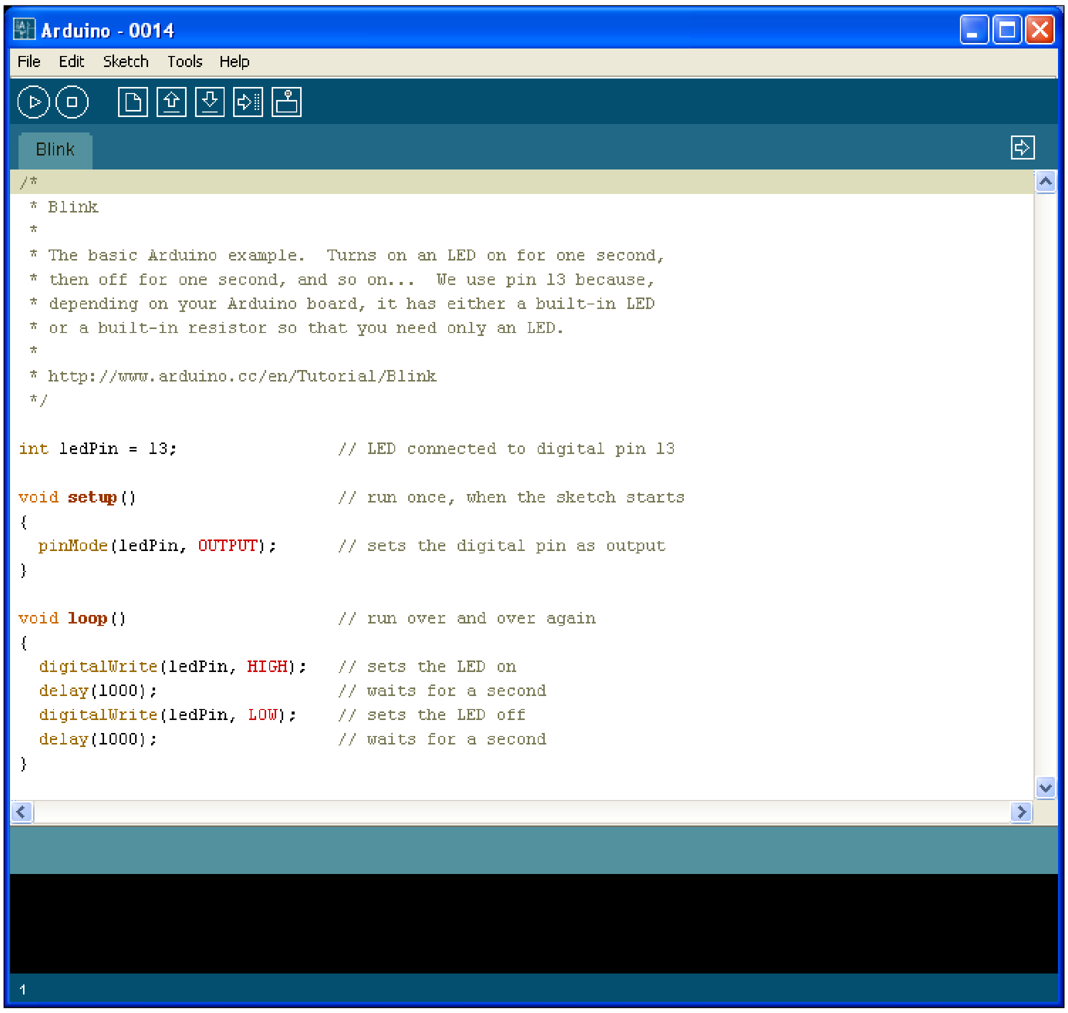


Figure 2.1: Arduino Diecimila hardware board

and the core libraries of the Arduino are written in C++ and C. That way sketches can be written in C or C++, but most users will use the simple Wiring programming language, which allows to write simple and readable code for the most common tasks (digital and analog I/O, serial communication, etc.). Arduino is an open-source project with a world-wide user community. More than 10000 people are registered in the forums and many projects are well documented, so that later projects can be built upon them. Figure 2.2 shows the Blink sketch edited in the Arduino IDE on a PC.



```
/*
 * Blink
 *
 * The basic Arduino example.  Turns on an LED on for one second,
 * then off for one second, and so on...  We use pin 13 because,
 * depending on your Arduino board, it has either a built-in LED
 * or a built-in resistor so that you need only an LED.
 *
 * http://www.arduino.cc/en/Tutorial/Blink
 */

int ledPin = 13;                // LED connected to digital pin 13

void setup()                    // run once, when the sketch starts
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()                     // run over and over again
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

Figure 2.2: Screenshot of the Arduino IDE

2.1.3 Wearable Computing - The LilyPad Arduino

Definition:
Wearable Computing

WEARABLE COMPUTING:
Wearable Computing is computing on clothes. Small microcontrollers can be used on clothes to do computations and they can be connected to electronic components like LEDs or sensors.

Clothes are a part of fashion, thus wearable computing has the potential to become a part of a new fashion. For example blinking LEDs attract people's interest and individual

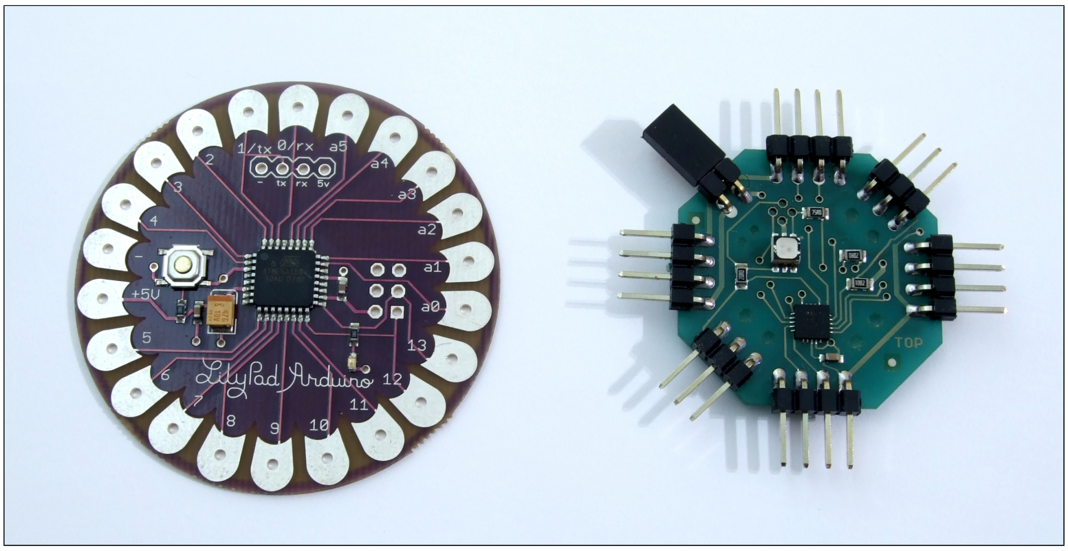


Figure 2.3: LilyPad and LumiNet

blink patterns add an individual note to clothing.

Electronic textile (e-textile) researchers look for new ways to add electronic components and micorcontrollers into clothes. [Buechley and Eisenberg, 2009] present three techniques for attaching electrical components to textiles. They also show problems that arise when e-textiles get washed.

The LilyPad Arduino is a fabric-based wearable computing framework that allows inexperienced programmers and people, who have never ever written a piece of software, to design and create their own e-textiles. Figure 2.3 shows a LilyPad Arduino next to a LumiNet node. [Buechley et al., 2008] say that technology (as automation or as entertainment) can improve the human condition and can "expand and democratize the range of human expression and creativity" (p. 423). The LilyPad Arduino is the successor of an e-textile construction kit that used an ATtiny26 microcontroller. This initial construction kit used Atmel programming tools and thus the user had to use a text editor for writing programs and command line tools for compiling and uploading. Another problem was that this kit needed special programming hardware and that the chip had to be removed from the clothes for reprogramming. This process was too complicated and thus the development team decided that the new LilyPad hardware should be compatible with the Arduino IDE. Since the Arduino project did not support the ATtiny chip, the new LilyPad was designed to use the ATmega168 microcontroller. The open-source Arduino software was modified to support the LilyPad Arduino and some libraries were developed that allow to use the LilyPad sensors and actuators. There were several reasons for not creating a new programming environment but instead use the Arduino IDE:

- The project team wanted to focus on the e-textile hardware, not on the software.
- Ideas and tools should be available to a wide audience. The Arduino community offers support and the LilyPad users extend this community.

Typical LilyPad setups consist of only one processing unit (LilyPad mainboard) and many devices like sensors or LEDs connected to it. The programming logic is located at this central point, no distributed computations are performed. LilyPad applications do not use wires but conductive thread to connect the mainboard and the peripherals. This has electrical disadvantages but allows sewing the components to the fabric. Silver thread corrodes over time, so that the resistance increases - especially if the clothes get washed. Many components can be connected to the LilyPad Arduino main board. Some popular components are:

- LilyPad Accelerometer: holds an ADXL330 three axis acceleration sensor. It outputs analog signals in the range from 0V to 3V for each axis. It is the most expensive sensor for the LilyPad Arduino and the most difficult one to understand
- LilyPad Bright White LED: a bright white LED with 250 mcd
- LilyPad Button Board: contains a momentary push button, thus it opens when it gets released and it closes when it gets pushed

- LilyPad Buzzer: allows to make sound with a buzzer at different frequencies
- LilyPad Light Sensor: a sensor for ambient light. At daylight it outputs 5 V, indoors about 1.5 V, and at night or when covered it outputs 0 V
- LilyPad Power Supply: has battery clips for an AAA battery. It takes input from 1.2V to 5V and transforms it to 5V with a maximum of 100mA. It is short circuit protected and has a power switch and an indicator LED
- LilyPad Tri-Color LED: not so bright rgb LED with a common anode
- LilyPad Vibe Board: a vibration motor that shakes when power is applied to it
- LilyPad Temperature Sensor: holds a MCP9700 temperature sensor for measuring ambient temperature conditions. It gives analog voltage values proportional to the temperature. For example, at 0 degrees Celsius the sensor will output 0.5 V, and at 25 degrees Celsius it will output 0.75 V

A project that uses LilyPad Arduinos is the EduWear[5] project, funded by the European Commission.

2.1.4 Blink Pattern Control System

[Hosomi et al., 2007] explain that it is difficult for a user of a wearable computing framework to program the microcontroller with a desired blink pattern that some LEDs should show. Thus they introduce a control system that allows to easily program blink patterns to a microcontroller. They identify three requirements for a blink pattern (called flicker pattern) control:

1. A programming method for a blink control pattern

[5] http://dimeb.informatik.uni-bremen.de/eduwear/

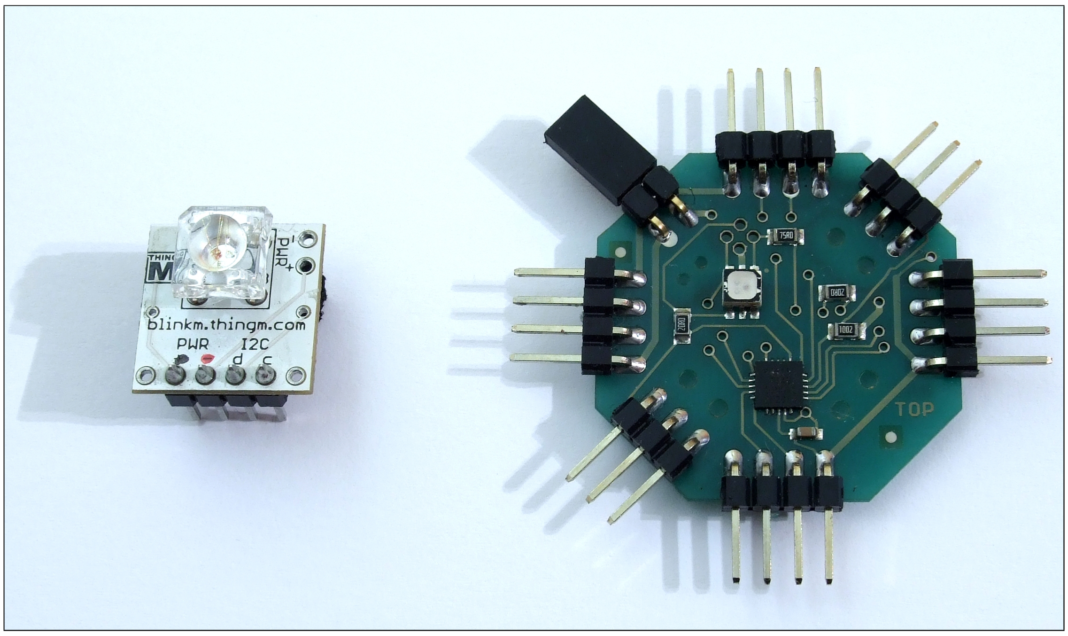


Figure 2.4: BlinkM and LumiNet

2. Compression methods for a blink pattern.

3. Application development environments: a graphical program editor runs on the PC and the hardware board communicates with the PC via serial communication. The same problem is addressed by the BlinkM device (more about this device follows in the next paragraph) that uses a Java GUI for simple creation of blink patterns. Both systems only allow to program pre-defined patterns that cannot react on user input or environmental events

2.1.5 BlinkM

BlinkM[6] is a smart LED made by a company called ThingM. It consists of a bright RGB-LED and an eight-bit microcontroller. A single device costs about USD 15. BlinkM devices communicate using the I^2C Bus. The BlinkM is the I^2C slave and can be controlled by any I^2C Master. BlinkM is limited to I^2C communication and does not provide more

[6]http://blinkm.thingm.com

than one communication port. The default usage is to connect the BlinkM to an Arduino Board that runs a special sketch called BlinkM Communicator and then start another piece of software, the BlinkM Sequencer, to upload blink sequences to the BlinkM device. A sequence is divided into different time-slots. Each time slot gets a color assigned. Then the whole sequence is uploaded to the BlinkM device and stored there. Now the device can play this sequence without an Arduino or any other I^2C master. It only needs 3V-5V power supply and plays the sequence. No other software is executed then, especially no communication between different devices takes place and no sensor data can be read in order to react to environmental events. A more powerful solution, called BlinkM MaxM, is available that has four limited analog input pins. The I^2C bus has some electrical disadvantages if it is used over long distances by many devices. Figure 2.4 shows a small BlinkM device next to a LumiNet node. The electronic parts, including the microcontroller are hidden on the back of the BlinkM device.

2.1.6 NeuoLED

NeuoLED[7] is a commercial product manufactured by a company called Cosinova. Any amount of NeuoLED devices can be connected to form a network. The unusual shape of the devices allows to create networks of almost any topology. Each device consists of three sets of three rgb LEDs, a light sensor, a push-button, and a Java-enabled microcontroller. Although it is possible to use a bus system and addresses, the nodes can also be used like artificial neurons that only propagate signals from one device to the other. A demo application, e.g., detects the shape of a person and renders its shadow on the LEDs of the NeuoLED network. New program code can be written in Java and can be uploaded to the network using a network reprogramming algorithm. Every node can store more than one application and the different applications can be started at runtime without reprogramming the network.

[7]http://www.NeuoLED.com

2.2 Networks

In this section, two examples for bio-inspired networks are given (Cybords and Spiking Neural Networks) and wireless sensor networks are explained.

2.2.1 Spiking Neural Networks

[Floreano et al., 2006] explain Spiking Neural Networks and introduce an implementation on an eight bit microcontroller. A Spiking Neural Network is a network that uses spikes for data exchange. Spikes are electrical pulses that biological neurons send to each other. Two properties of a spike are important:

- **Firing rate** of a neuron: average quantity of spikes emitted by a neuron within a long time window
- **Firing time**: the precise time when a single spike is emitted

2.2.2 Cybords

Bynase uses the firing rate

Cybords[8] are very simple microcontroller-based boards that concentrate on biologically inspired ideas. Cybords were invented by Ward Cunningham, the computer programmer who developed the first Wiki. They communicate with each other via a bio-inspired protocol called Bynase. Bynase uses the firing rate to indicate different values. On the electrical side, Cybords do not use pull-up or pull-down resistors on the communication pins. The sender simply puts pulses with a pulse frequency of about 15kHz on the line, regardless of the line's state. Collisions are allowed and there is no media access layer.

A value is encoded by a sequence of pulses. The pulses are proportionally HIGH or LOW within a certain time window and the distribution of pulses within this time window

[8]http://c2.com/cybords/

is random. In other words, the sender sends out noise, but with a certain probability for the signal to be HIGH. Thus a value of 60 means that 60 percent of the pulses within a certain time window are HIGH. This can be accomplished by the pseudo code seen in figure 2.5.

```
int value = 123;
while(TRUE)
{
    if(value >= random())
    {
      digitalWrite(pin, HIGH);
    }
    else
    {
      digitalWrite(pin,LOW);
    }
}
```

Figure 2.5: Bynase: the sender sends out probabilistic HIGH pulses

On the other side, the receiver looks at the line at fixed intervals. It counts the pulses that are HIGH within a certain time window. This can be the same time window that the sender uses, but as the clocks are not synchronized, usually the receiver uses a different time window. For example, the receiver might take 100 samples as in figure 2.6. The value of the variable `result` says how many samples out of 100 were HIGH. This is the same probability for the pulse to be HIGH that the sender did send. A value of 0 means that the line is never HIGH and a value of 100 means that the line is always HIGH.

Bynase is not good for transmitting fast changing data, because the time windows should not be too small. But it is good for asynchronous communication where the exact time when a receiver looks on the line is not important.

While Spiking Neural Networks represent a scientific model of how neural networks can be translated into the artificial world of computers, Cybords represent a practical experiment that relies on the same basic bio-inspired ideas.

```
int result = 0;
for(y=0;y<100;y++)
{
  if(pin is HIGH)
  {
    result++;
  }
}
```

Figure 2.6: Bynase: the receiver counts the number of HIGH pulses

2.2.3 (Wireless) Sensor Networks

important requirements for sensor network applications

[Sugihara and Gupta, 2008] define four important requirements for sensor network applications:

- Energy-efficiency: sensor nodes need to be operating for months. To reduce power consumption, it is a good idea to avoid unnecessary wireless data transmission.
- Scalability: the network must allow to add and remove nodes dynamically. A problem is the overall communication bandwidth and again, reducing data transmission can be a solution.
- Failure-resilience: the network must remain functional even if some nodes are malfunctioning, unexpected failures occur or if communication becomes unreliable.
- Collaboration: network nodes must collaborate in two ways:
  - Data collection: in most applications, (processed) data must be transmitted to a central server. The nodes must help each other to transfer the data to the destination.
  - Collaborative information processing: readings from multiple sensors must be processed.

Wireless sensor networks (WSNs) demand operating systems that can handle events. Different operating systems and wireless sensor network platforms use different programming models and styles. [Mozumdar et al., 2009] compare the programming models of two free academic operating systems (MANTIS OS[9] and TinyOS[10]) and one proprietary network stack implamantation (Ember[11] ZigBee). Two main paradigms are noted:

1. Multi-threaded or multi-tasking programming (MANTIS OS)
2. Split-phase non-preemptive request-response programming (TinyOS and Ember ZigBee)

The authors identify a single code writing style that can be ported easily across these three platforms by creating an API abstraction layer for non-blocking OS calls and for sensors and actuators. This code writing style is like a finite state machine (FSM). They also compare the libraries that implement frequently used functions. The ZigBee implementation provides the most advanced and richest set of functions.

network reprogramming

If new program code must be installed on nodes of a wireless sensor network, it is a good idea to use network reprogramming algorithms that allow to leave the network unchanged, because some nodes might not be (physically) accessible or it would be too time consuming if each node has to be connected to programming hardware and then would be reinstalled at the target location. A data dissemination protocol must be used to propagate new program code in a WSN. In contrast to runtime data dissemination, protocols for software updates need very reliable data transport protocols. Most dissemination protocols consists of at least the following three basic steps:

- Advertisement of available software update
- Selection of a source

[9]http://mantis.cs.colorado.edu
[10]http://www.tinyos.net
[11]http://www.ember.com

- Reliable download to target

Among others, three protocols are well known in the field of WSNs: XNP, MNP and Deluge (see [Hui and Culler, 2004]).

XNP

single-hop protocol

XNP (Crossbow Network Programming) is a single-hop in-network program code dissemination protocol running on TinyOS. Single-hop means, that program data is transmitted from one node to another device that is within communication range. Nodes that received code do not become senders. A host PC loads a program image to a single node or to a single group of nodes within the radio range of the host. The transmission can use a unicast mode, where XNP checks the delivery of each packet via ACK/NACK, or a multicast mode, where all packets are transmitted without check can be used and after the full image is transmitted, receivers can request missing (or corrupted) packets. All received data is stored in external memory. The host can send a reboot command after program code transmission and then a special bootloader copies the program image from external memory into program memory. After this, the new program can be started.

MNP

multi-hop protocol

MNP (Multihop Network Programming) is a multi-hop network reprogramming protocol running on TinyOS that uses the XNP bootloader. In a multi-hop protocol, the target can become a source after successful software download. The MNP protocol operates in four phases:

1. Request/Advertisement: sources advertise new program code. Interested nodes make requests. Nodes listen to both, advertisements and requests and decide if they have to receive new program code or have to forward received code.

2. Download/Forward: The host sends the whole packet to receiving nodes (without ACK). The receivers store the program code in external memory.

3. Update/Query: nodes can request missing packets. Nodes that have received the full program image then become source nodes.

4. Reboot: after rebooting, the node copies the received image from external memory to program memory.

Deluge

Deluge also operates on TinyOS. It is a multi-hop program code dissemination protocol that uses incremental updates. It can propagate large data objects from one or more sources to one or many targets. The advertisement packet contains a version number and target nodes request single packets until they received the full program image. Nodes can advertise packets immediately after receiving; even if they have not yet received the full image. Requests can also be used to receive missing or corrupted packets since there is no ACK or NACK mechanism.

Current dissemination protocols use incremental updates for size reduction. They only exchange the pages of a program that changed from one version to another. This reduces data transfer and overall transmission time, but it is only suitable when new versions of the same application shall be transmitted.

A problem in WSNs is that attackers can easily reprogram a whole network. There exist solutions that add public-key based program code authentication, strong integrity verification or freshness checks (e.g. [Dutta et al., 2006]).

2.2.4 Distributed Particle Display System

[Sato, 2008] introduced a new display system called Distributed Particle Display System. Hundreds of wireless nodes with rgb LEDs are controlled by a PC with a camera. The distributed pixels can be installed on any object and the resolution of the display can be changed by adding or removing nodes or by changing the space between them. The camera is used to determine the location of the nodes. For this the LEDs are activated sequentially one after the other. Once all nodes are located, the PC can control them with a wireless communication interface. The project did not create new hardware but used the existing S-node[12] that has a microcontroller, a wireless communication unit and an rgb LED. The nodes can also be attached to clothes and the camera might then be able to track a person that wears some smart nodes.

The bio-inspired concepts of spiking neural networks and the Bynase protocol are used in this work for in-network communication between nodes.

The program code dissemination protocols used by wireless sensor networks solve some problems that do not exist in wired networks, but the concept of multi-hop reprogramming is used in this work for what we call programming by infection.

The Arduino project has the biggest impact on this work, because as we will see later, this work extends the Arduino framework to support LumiNet hardware. The same happened in the process of the LilyPad project.

2.3 LumiNet Hardware Board

The hardware board that was used in this paper is called LumiNet hardware board. The current hardware revision is version 3.6. Details about the board like schematics and the board layout can be found in B—"APPENDIX: Reference Of Hardware Design". The hardware board was developed by Professor Jan Borchers. It is a low-cost board that only uses components that are essentially necessary for a ATtiny-based microcontroller board. The only additional component is an rgb LED. Other additional components,

[12]by YMATIC Corp.

like an external crystal are missing. The small footprint and the additional holes allow to use the board for wearable computing projects. Because application development for this board was not easy, one of the main tasks of this thesis was to create a application development framework for this hardware.

2.4 Comparison

The following table compares the key properties of the related hardware boards mentioned in this section. Properties like distributed computing and network reprogramming can be implemented on most platforms, but only LumiNet (as a result of this thesis) and NeuoLED support these features by default. The prices are estimations that have been valid when this thesis was written.

Project	Bio-inspired	Distributed computing	Network reprogramming	Size	Price
Wiring	No	No	No	Medium	100 USD
Arduino	No	No	No	Medium	30 USD
LilyPad	No	No	No	Small	20 USD
BlinkM	No	No	No	Small	15 USD
NeuroLED	Yes	Yes	Yes	Big	Unkown
Cybords	Yes	No	No	Small	Low
LumiNet	Yes	Yes	Yes	Small	8 USD

Table 2.1: Related work: hardware boards

Chapter 3

Design

This chapter describes the software design of the distributed physical computing framework for the LumiNet hardware and presents early implementations of the framework as well as improvements to these implementations.

3.1 Network Topology

The network will consist of three classes of nodes:

three classes of nodes

- Normal nodes: most nodes are normal nodes. All of them run the same application.
- Vector nodes: these nodes infect the network with new program code.
- Sensor nodes: these special nodes run a special program for reading and interpreting sensor data. They send interpreted sensor values to the network. Sensor nodes do not progagate program code and they have to be programmed by hand since each sensor node has its own application that can only operate with the connected sensor device.

The nodes can be connected with at most one neighbor on every side. This allows to create two dimensional fully meshed networks. The framework must not limit the topology of the network.

3.1.1 Overall Design

The new distributed physical computing framework that is developed in this work should make use of the physical and electrical properties of the LumiNet hardware. A network can be created by using the four pins in each of the four cardinal directions. On the software side, a library must be implemented that allows to use the pins for communication. A LumiNet hardware board in such a LumiNet network is called node. The same application runs on every normal node and the framework is meant to be used for developing distributed applications for the whole network; although it can also be used for developing applications for a single node.

Because the same application runs on every normal node and the framework defines the communication between nodes, a multi-hop network reprogramming mechanism can be integrated into the framework. This allows to reprogram all nodes at once without disassembling the network. A bootloader that supports this mechanism must be implemented and stored in the flash memory of the nodes.

Because it is possible to control the I/O pins on the lowest level, in addition to the serial communication, an example for a bio-inspired communication protocol should be implemented for the LumiNet hardware. To make the platform even more organic, some bio-inspired example applications should be created as a proof of concept.

Communication is just one aspect of the framework. More functions should be included in the framework in order to make it as comfortable as possible. It should have a steep learning curve, so learning how to use it should be easy, although some basic programming skills are still necessary.

3.2 Early Implementations

The initial idea behind this work was to provide a software framework that allows to easily create applications for the LumiNet hardware. Bio-inspired ideas should be used where possible and a good usability should enable novice programmers to use the system. The framework must also allow to reprogram the nodes without modifying the network.

First implementations used C macros and functions to meet these general requirements, but the user had to use his own code editor and compile the sources with the command line tools. There was a demand for an even better usability.

Another problem was the reprogramming mechanism. Vector nodes were not able to receive new program code, but they had to be prepared with an ISP programmer. Special batch files modified the payload and merged it with the main application of the vector node. This was a very complicated process that should be hidden from the user.

Three main requirements can be extracted from the initial idea:

- Good usability
- Easy reprogramming
- Bio-inspired approaches

3.3 The Arduino Idea

Because the Arduino is the most popular open-source physical computing framework today, it was a big inspiration for the initial implementations of our framework. Since the Arduino software did not support the ATtiny family of micro-controllers, a new design goal was to modify the existing Arduino framework so that it can be used on the LumiNet hardware. This included two main parts: the so-called core for the micro-controller, and the Arduino IDE. So the usability was improved by this step, but the other requirements were not addressed by this decision. The Arduino framework was extended to meet the remaining requirements by supporting the Bynase communication protocol and the programming by infection mechanism that was developed for early implementations of our framework.

Porting the Arduino framework to the LumiNet hardware provides more benefits than just better usability:

- Usability: a JAVA IDE that allows coding, compiling and uploading of sketches all in one place
- Multi-platform: the IDE runs on many platforms including Windows, Mac OS X and Linux
- Community: more than 10000 registered users from all over the world share their projects and knowledge in the forums.
- Open source: everyone is allowed to modify the sources and bring in new ideas to the project

3.4 Requirements

three main requirements

In the end, three main requirements can be listed:

- M1: Improve the usability (port the Arduino framework to the LumiNet hardware)
- M2: Enable easy reprogramming (programming by infection must be included in every sketch, invisible to the user)
- M3: Use bio-inspired approaches (add Bynase and provide bio-inspired example algorithms)

six important requirements

In addition to the three main requirements, we defined six important requirements that should be met by the final implementation of the framework:

- R1: Support many sensors and actuators
- R2: Support all communication directions of the LumiNet hardware
- R3: Support distributed processing
- R4: No constant pattern generators - the network must be interactive
- R5: A wide audience should have access to the framework
- R6: Multi platform compatibility

In the following subsections, more details about the main requirements are provided.

3.4.1 Usability

Microcontrollers are often programmed in programming languages like C or Assembler. These languages are difficult for beginners and concepts like pointers, stacks, and interrupts can be too complicated for them. The LumiNet framework should motivate beginners to use microcontrollers and solve simple problems with them, e.g. blink an LED. A set of commonly used functions collected in a software library can help beginners to solve their problems without noting that they are using a C compiler in the background. They do not even know what a compiler is, but they want to see the blinking LED when they hit the Start button or power up the device. However, the software library cannot provide a function for every possible problem. So the user has to learn how to use the basic functions by investigating examples. The LumiNet framework should provide useful functions for common problems that can be solved by a microcontroller system and it should give examples that explain how to use these functions. Compiling and uploading of new program code should be invisible to the user and must be as comfortable as possible. As mentioned before, the step towards an Arduino-compatible framework increases the usability and almost all examples and most parts of the documentation of the Arduino project can be used by LumiNet users.

3.4.2 Network Reprogramming

Before the network can be infected by new program code, the local topology of the network must be determined. Each node has to identify its neighbors and if a vector node is present. No node knows the complete topology of the network, only the local topology defined by the neighborhood of the node can be detected by the topology scan algorithm.

The data transfer should use a fast serial communication protocol instead of the slow Bynase protocol to reduce programming time.

A multi-hop dissemination protocol must be used to reach all nodes.

If reprogramming of a single node fails, the rest of the network should not be affected by this failure.

Data integrity must be provided: corrupted data has to be identified and only verified bytes can be programmed to the flash memory.

3.4.3 Bio-inspired Aspects

Bio-inspired ideas should be used where possible. The most obvious bio-inspired part of this work is the Bynase communication protocol, but the examples provided in the Evaluation chapter prove that the framework can be used for bio-inspired algorithms and concepts.

Chapter 4

Implementation

In this chapter we describe the final implementation of the distributed physical computing framework, explain how programming by infection and topology scan works, introduce the supported communication protocols, and give a short introduction on how to use the Arduino IDE for uploading sketches and bootloader code.

4.1 Multi-hop Bootloader

If new program code needs to be uploaded to the LumiNet nodes, it does not make sense to use an ISP programmer and flash the nodes one after the other, because this would take too much time and the network must be disassembled for this process. A LumiNet network can contain hundreds of nodes and they should be programmed all at once in a single step. The idea is to load the code to the network using a vector node. The vector node contains the new program code and will upload it to a node in the network. After receiving the new program code, each node will become a sender and give the progam code to its neighbors. In this way the new program code spreads in the network like a virus. This is called programming by infection.

programming by infection

4.1.1 Topology Scan and Vector Node Detection

Before the dissemination of program code can start, the topology of the network must be detected. Therefore, each LumiNet node performs the topology scan and vector node detection algorithm after power-on. After this, the node knows its direct neighbors, but there is no node that knows the whole network topology.

no node that knows the whole network topology

The topology scan algorithm for a normal node looks like this:

1. Activate inputs with pull-up resistors
2. All outputs to LOW
3. Read inputs: if an input is LOW, then a neighbor is detected. Until now this must be considered a normal node
4. All outputs to HIGH
5. Read connected inputs: only consider inputs where a neighbor was detected before. If the input is LOW, then this neighbor is a vector node. Set all other outputs to LOW

If more than one vector node was detected, only one will be selected. The priority sequence for this selection is: WEST, NORTH, SOUTH, EAST. This ensures that in a fully meshed network no deadlocks will occur, because the order of the vector paths is well-defined.

The topology scan algorithm for a vector node looks quite similar but step four is different:

1. Activate inputs with pull-up resistors
2. All outputs to LOW
3. Read inputs: if an input is LOW, then a neighbor is detected. Until now this must be considered a normal node

4. Keep all outputs at LOW
5. Read connected inputs: only consider inputs where a neighbor was detected before. If the input is LOW then this neighbor is a vector node. It is not recommended to install more than one vector node in a single LumiNet network.

Sensor nodes do not have this bootloader. They do not receive program code and they do not give it to their neighbors. Sensor nodes should be attached to the edge of the network. They perform the topology scan algorithm but they do not detect vector nodes:

1. Activate inputs with pull-up resistors
2. All outputs to LOW
3. Read inputs: if an input is LOW, then a neighbor is detected.
4. All outputs to HIGH

All nodes store the information about their detected neighbors in the EEPROM after the topology scan finished because it is needed by the bootloader or an application later.

4.1.2 Transmission Protocol

In the bootloader, data is transmitted using an RS232 protocol at 9600 baud. Normal nodes have a bootloader, located at flash byte address 0x1600. When a vector node is detected with the topology scan algorithm, the bootloader gets called. The connections information (previously written to EEPROM by the topology scan algorithm) is read and the program code is transferred to all connected nodes. The sequence in that the code is transferred, is: EAST, SOUTH, NORTH, WEST. The bootloader performs these steps:

1. Send 0xA0 (START)

2. For each payload byte 0xpp: send 0xB0 (DATA) followed by 0xpp and read the byte back from the line. If the received byte does not match 0xpp send: 0xB1 (RETRANSMIT) followed by 0xpp until it matches the original value. No escape characters are used, but it is important that no control character like 0xB0 or 0XB1 get lost or destroyed on the signal line. If this happens, the payload can be destroyed. Future versions of the transmission protocol should include checksums, but this slows the transmission down.
3. Send 0xC0 (STOP)

The rgb LED is used to indicate the state of a node. The following color codes are used:

- red: node is waiting for new program code
- green: node finished transmission to all of its neighbors
- blue: node is sending to a neighbor
- off: node is receiving data from a neighbor

When all nodes shine green or blue, the vector node must be detached and the network can be rebooted. Because the bootloader starts at byte address 0x1600, the remaining bytes 0x0000 -0x15FF can contain payload code. The flash is organized in pages with a pagesize of 64 bytes. Thus 88 pages (5632 bytes) are available for payload. After power up, the mcu starts to search for program code at address 0x000. If no valid program code was found, it continues to look for program code at higher addresses of the flash memory. This allows to put the bootloader for a normal node at address 0x1600 and clear all previous memory addresses. Thus, the bootloader get started on an otherwise empty normal node.

5632 bytes are available for payload

4.1.3 Dissemination Program for Vector Nodes

Vector nodes can be connected to a PC using an RS232 to TTL converter or a USB to serial bridge like the FTDI chip used on the Arduino hardware (see also 4.8.1—"Uploading Sketches to LumiNet"). The PC must be connected to the WEST port of the vector node. The network must be connected to the EAST port. If a jumper is present between ground and pin 1 (PB2), the PC can send the new program code to the vector node. After the vector node received the code it can send the payload to the connected LumiNet nodes. The vector node stores the payload in the flash memory at byte address 0x0800. If the jumper is not present, the vector node will not receive payload from the PC, but will send the payload that is stored in flash, starting at address 0x0800.

Hence the vector node can be preprogrammed by the PC and then the LumiNet network can be infected by its code even if no PC is available.

vector nodes do not have a bootloader

The vector nodes do not have a bootloader at byte address 0x1600. Instead of this, they have a regular program located at address 0x0000 that receives the payload from the PC or sends the payload to the network. Payload should not be bigger than 5632 bytes (88 pages, 64 bytes each) because that is the maximum size a normal node can store in the flash memory without overwriting the bootloader.

If for any reason corrupted payload is uploaded to a node, the node might not call the bootloader after the next reset. Invalid payload code can thus make a network inoperable and nodes with invalid payload must be reprogrammed using an ISP programmer.

4.2 Serial Communication

Two wires in each direction allow at least four modes of operation:

- Bidirectional (full duplex) asynchronous serial communication (like RS232).
- Unidirectional or bidirectional (half duplex) synchronous serial communication (like I^2C).

- Simple bang protocols: nodes wait for pin change on input pins and transmit bits at a special firing time (like spiking neural networks).
- Bio-inspired communication based on the firing rate in a spiking neural network (like Bynase).

LumiNet supports serial communication (RS232 at TTL level), but serial communication relies on synchronized clock speeds. LumiNet nodes do not have crystal oscillators and the internal clock source is not precise enough in all situations. So we do not recommend serial communication with LumiNet. The bootloader uses serial communication and therefore it should only be used with synchronized nodes. Clock calibration is important if the bootloader should be used. Most nodes perform very good if they are operated at 5V when the bootloader is executed. If the bootloader needs to be executed in a network that operates at 3.6V, the nodes have to be calibrated at this voltage level before the network is assembled.

the bootloader should only be used with synchronized nodes

The communication line uses the internal pull-up resistors of the receive pins. The line is HIGH when it is idle. The sender initializes the transmission of a byte by pulling the line low for one bit period - this indicates the start bit. The receiver detects the falling edge on that pin and will now receive 8 data bits, each one bit period wide. The least significant bit is transmitted first. The transmission stops with the stop bit, which is a HIGH level on the line for one bit period. The length of a bit period depends on the transmission speed, also called baud rate. In this case the baud rate is equal to the number of bits transmitted per second, including start bit and stop bit. So the bit period is shorter for higher baud rates and longer for smaller baud rates. The smaller the bit period becomes, the more sensitive the communication becomes to errors caused by noise. Lower baud rates allow less errors caused by noise, higher baud rates mean faster transmission speed.

Serial communication is implemented in assembler, based on the Atmel application note AVR305 ([Atmel, 2005]). The implementation of the software UART does not use any timers or external interrupts. The serial communication is fixed at 9600 baud, using one START bit, 8 data bits and 1

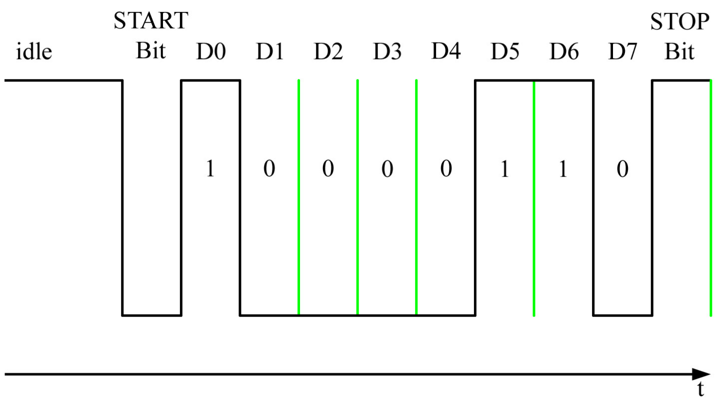


Figure 4.1: UART frame format. Showing 0x61 ('a') with one start bit and one stop bit

STOP bit. The frame format is shown in figure 4.1, where the character 'a' is transmitted.

Another error source is the clock of the microcontroller. Table 4-2 of the application note says that the software UART at 9600 baud on a 1 MHz microcontroller performs with an error of 2.7%. A lower baud rate of 4800 baud would result in an error of 0.3%. But transfer time is critical if a big network should be programmed. In order to switch from 9600 baud to 4800 baud, the b-value in uart.S must be changed from 14 to 31 according to the table from the application note. All nodes of the network must operate at the same baud rate. The baud rate can not be changed at runtime. If the microcontroller runs faster than 1 MHz (clock speed can be changed at runtime), the baud rate gets raised by the same factor. For example, at 2 MHz system clock, the baud rate of the serial interface rises from 9600 baud to about 19200 baud. If the baud rate should be changed for all nodes without modification of the system clock, then a modified bootloader must be flashed to every node.

4.3 Arduino Core for LumiNet

An Arduino core is the part of the Arduino framework that implements the main functions of the framework that are listed in the Arduino reference. This includes functions like digitalWrite() and delay(), but also analogRead() or Serial.begin(). The pin mapping is another important part of an Arduino core. Because the original cores only support official Arduino boards which use ATmega microcontrollers, Because it became a requirement to port the Arduino framework to the LumiNet hardware, a new core had to be written.

In this section we explain which changes are necessary to create a new core from the original Arduino core. The easiest way to create a new core for the Arduino framework is to copy an existing core and modify the files.

1. The folder called "arduino" from the hardware/cores directory must be copied and this copy must be renamed.

2. The pin mapping must be adjusted in the file "pins_arduino.c" and the corresponding definitions must be adjusted in "pins_arduino.h".

3. A new entry has to be added to the file "boards.txt" that is situated in the hardware directory. All lines starting with atmega168 (for example) must be copied and the name (in this case atmega168) is to be replaced by the name of the new core. Then the parameters must be modified and the most important change is the line newcorename.build.core. The value of this property is arduino and it has to be changed to the name of the new core.

4. The Arduino IDE should be started now and the new core must be selected in the Tools-Board menu. If the sketch compiles without warnings and errors, then everything is fine. If errors occur, the remaining files of the core must be modified until the sketch compiles. For new cores with an ATtiny controller, a lot must be modified, because the Arduino core is designed for ATmega microcontrollers. In this case it is better to start with a copy of our LumiNet core and compare the LumiNet core with the arduino core to understand the changes in detail.

Some official Arduino hardware libraries do not compile with our LumiNet core: Ethernet, Firmata, Stepper. The main problem with these libraries is that they use definitions that only work for the ATmega microcontrollers.

The Bynase protocol that is used in the LumiNet core is also available as a standalone library for the Arduino project. This allows bio-inspired communication between LumiNet hardware and Arduino hardware.

4.4 Arduino IDE for LumiNet

The Arduino software from the Arduino project cannot be used with LumiNet nodes. Instead, a customized package must be used. Although LumiNet nodes are compatible to Arduino sketches, it is not an official Arduino platform. Differences between the original Arduino distribution and the LumiNet version are presented in this section.

The Arduino software that was ported had the version number 0012 and later the versions 0013 and 0014 were used.

The file **boards.txt** contains a list of supported hardware boards that can be used with the IDE. The LumiNet version of the IDE contains all platforms that are supported by the original distribution and additionally it has three new entries:

- **LumiNet Vector Node**: this entry must be selected to prepare the IDE for a vector node. Payload can be uploaded to the vector node by pressing the Upload button, or if the bootloader should be uploaded to the node, the program image for the vector node is used.

- **LumiNet Normal Node**: this entry should only be selected if the user wants to upload the bootloader to a normal node. Payload can not be uploaded to a normal node directly; a vector node is needed to infect a LumiNet network.

- **LumiNet Sensor Node**: this entry uses the specified ISP hardware programmer (default: stk500) to flash a single LumiNet node using ISP. A different topology scan algorithm will be executed: vector nodes will not be detected, because sensor nodes cannot disseminate payload.

A new entry was added to **programmers.txt** in order to support the Atmel stk500 programmer with firmware version 2.

Because there has not yet been an official Arduino based on an ATtiny micocontroller, the official IDE is limited to atmega chips. In order to support the LumiNet hardware, changes to **ArduinoUploader.java** had to be made. The LuminetUploader was integrated into this class as well.

Programming by infection is not supported by the official Arduino IDE. Hence, a new uploader class was implemented: **LuminetUploader.java**. This file implements the serial communication protocol for the bootloader. It uses 9600 baud.

The official Arduino IDE does not support assembler files. Inline assembler is supported, but dedicated .S files with assembler code cannot be used neither in a core, nor in a library, nor in a sketch. Because the LumiNet core uses assembler code for the serial communication, modifications to Compiler.java were made.

4.5 Bio-inspired Communication - Bynase

While the bootloader needs a classic serial communication protocol to transfer the programm code, runtime communication is performed by a protocol called Bynase. Serial protocols have a problem that is critical with LumiNet hardware: the receiver has to be listening at the time when the sender is transmitting data. Some serial protocols also need precise crystal oscillators in order to keep the right timing. Another problem with serial protocols is that noise can destroy data. For example a single bit can flip, changing the byte 00100101 into 01100101 as a result of noise on the data line. A desirable communication protocol does not rely on good clocks on the receivers (and senders) and is robust against noise. An analog signal like a PWM signal could be a solution. But if the clock skew between the receiver and the sender is too big and constant, the receiver will always read the same fraction of the pwm signal. This might result in wrong readings. It should not be predictable when the pin is HIGH or when it is LOW. Thus PWM with a random distribution of HIGH and LOW signals would be a perfect solution to the problems. This is the concept that is used in the Bynase communication protocol. Bynase puts noise on the signal line and the data is statistically encoded. With a certain probability the line is HIGH. If the value (the data that should be send) is lower than a random number then the signal line should be HIGH, otherwise it should be LOW.

a desirable communication protocol

4.5.1 Random Number Generator

It is not important that the numbers are perfectly random. But it is important that the random number generator generates sequences of alternating low numbers and high numbers. The implementation must be very fast because it is called by an Interrupt Service Routine (ISR). The random() function of the Arduino framework takes too long. Ward Cunningham proposed an "inverted counter" for his Bynase implementation. This can be described by the following example:

inverted counter

```
0 0 0
1 0 0
0 1 0
1 1 0
0 0 1
1 0 1
0 1 1
1 1 1
```

The slowest changing bit is the least significant bit (lsb). An implementation in C that is used by LumiNet at the moment comes from James Neal[1] :

```
uint8_t byrand ()
{
 uint8_t v = counter++;
 uint8_t r = 0,q = -1;
 while(q) {
 r = (r << 1) | (v & 1);
 v >>= 1;
 q >>= 1;

 return r;
}
```

It uses a normal counter and then inverses the bit sequence of the number.

4.5.2 Details About the Bynase Implementation

LumiNet uses an implementation of the Bynase protocol that can operate in all four directions. It uses one line for incoming data and one line for outgoing data. It would even be possible to use one single line for incoming and outgoing data, but as the hardware design of LumiNet has two pins in each direction, we decided to keep the protocol simple and use one dedicated line for each data direction. Two functions handle the Bynase communication: byin() and byout(). byin() and byout() are called within an ISR when timer1 overflows. This should happen every 55 clock cycles but the speed can be set as a parameter of the bynase_init() function. For each direction exists a byval_in value and a byval_out value (e.g. byval_in_south or byval_out_east). The value of these variables must be between 0 and 100, indicating the probability in percent that the line will be HIGH.

In an experiment (see Table 4.1) we changed the byval value for the PB2 pin from 100 to 0 and measured the resulting Voltages on pin PB2 with a voltmeter (Digitek DT-4000ZC).

[1]http://www.laen.org/

As a byval of 100 results in a voltage of 4.94V we set the other values in relation to 4.94V and even though the byval value and this percentage do not exactly match, the result is sufficient for the receiver to distinguish between ten different levels using the byin() function.

byval	Volt	Percent of 4.94V
100	4.94	100
90	4.59	92.9
80	4.21	85.2
70	3.82	77.3
60	3.44	69.6
50	2.90	58.7
40	2.32	47.0
30	1.74	35.2
20	1.16	23.5
10	0.58	11.7
0	0.00	0

Table 4.1: Voltage measured when different byval values were send out from a LumiNet node operated at 5V

4.5.3 Bynase for LEDs

The three LEDs (red, green, blue) have their own byval values (byval_red, byval_green, byval_blue) and are controlled by byout() just like any other output pin. This allows to dim the LEDs, because different byval values result in different voltages on the pins connected to the LEDs. So the LEDs on the LumiNet boards are not dimmed by PWM but by Bynase. The LEDs have a higher output priority and they are updated twice as often as a normal output pin. This reduces flicker.

LEDs on LumiNet boards are dimmed by Bynase

4.5.4 Byte Communication on top of Bynase

Usually, users want to transmit bytes over a data line. With Bynase it is only possible to put a value on the line. The resolution of Bynase is not good enough to distinguish between 256 values. So we added a layer on top of Bynase that can transmit bytes. A data value is split into four parts. Each part consists of two bits.

A Bynase value of 100 indicates the start of a new byte. This value is held on the line for t_{packet}, as all other values are. Then the value 0 is on the signal line. The four possible messages of a data part are 00, 01, 10 or 11, or as decimal values: 0,1,2 or 3. Instead of sending the decimal value directly, it is first incremented by one and then multiplied by ten, so values 10,20,30 or 40 are send as data parts. Between two data parts, a control message is added. It has the value 0, so the line is low for the time period t_{packet}. If the byte x has the bit representation aabbccdd, then the following values will be on the Bynase signal line:

1. 100
2. 0
3. (aa +1) * 10
4. 0
5. (bb +1) * 10
6. 0
7. (cc +1) * 10
8. 0
9. (dd +1) * 10
10. 0 (line idle again)

Each single value is hold on the line for a certain amount of time t_{packet} (e.g. 50 ms). So a byte takes nine times the time t_{packet} (e.g. 450 ms) which makes byte communication very slow. Shorter values for t_{packet} are possible but then it can occur that the receiver can not take enough samples for a good average of the statistically encoded data that it received.

4.6 LED Extensions

The LED is the most important device for any LumiNet node. For an illumination network, handy functions and macros that allow to control the LED easily, improve the usability. The following functions and macros are available:

- RED_ON, RED_OFF
- GREEN_ON, GREEN_OFF
- BLUE_ON, BLUE_OFF
- YELLOW_ON, YELLOW_OFF
- WHITE_ON, WHITE_OFF
- SetRGB(red, green, blue)

While macros like RED_ON or BLUE_OFF allow to switch an LED on or off, the function SetRGB allows to mix all possible colors.

SetRGB() has three parameters, each in the range from 0 to 255. Any RGB color can be mixed by this function. Example: SetRGB(10,60,23)

4.7 Sensor Nodes

Sensor nodes send data collected by sensors into the network. Examples for the following sensors exist:

- Analog values (ADC) - Potentiometers
- Brightness
- IR: infrared communication

In addition that that, the Arduino community shows many examples for different sensors. Most examples can be used with LumiNet, but it is important to check if delay values must be changed, because the Arduino board runs at 16 MHz while LumiNet nodes operate at 1 MHz.

4.8 How to use the Arduino IDE

With the IDE, the user can edit sketches, compile them and upload them to the hardware boards. This section concentrates on the upload process and explains the different kinds of upload that can be used with LumiNet nodes.

4.8.1 Uploading Sketches to LumiNet

When the user hits the Upload button, the current sketch gets compiled and will be uploaded to the hardware. The Arduino board has a bootloader that is compatible to the stk500 procotol. Thus the IDE can use a tool called avrdude to upload the compiled sketch to the Arduino board using the USB connection or any serial port.

LumiNet nodes do not support this upload process. They have a different bootloader and are not supported by avrdude. An uploader called LumiNetUploader was implemented and integrated into the IDE. To the user, the upload process seems to be the same as the one used with the Arduino hardware; the Upload button has the same functionality.

Because LumiNet nodes do not have a USB port, the connection to the PC must be established by one of the following possibilities:

- Use an RS232 to TTL converter and connect it to a serial port of the PC.
- Use an RS232 to TTL converter and an RS232 to USB bridge and connect it to a USB port of the PC.

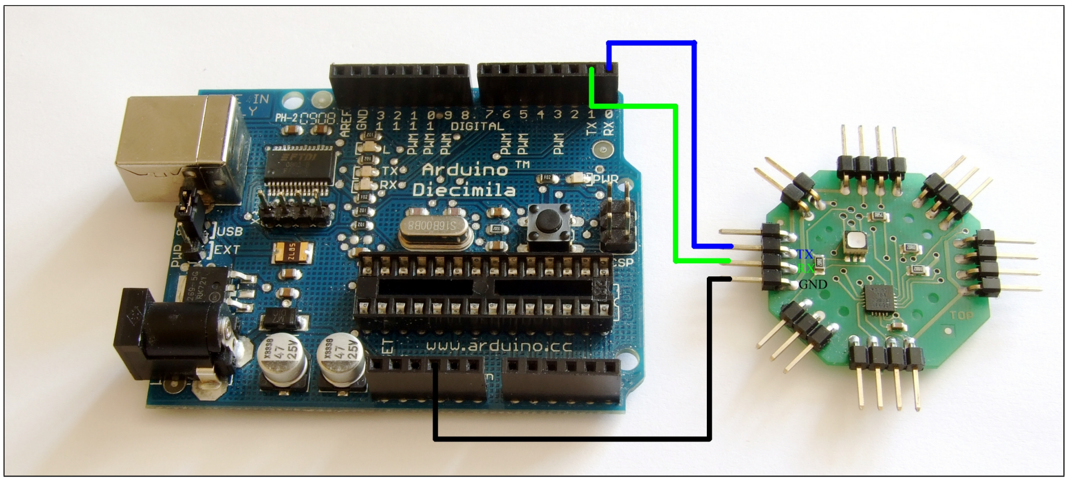


Figure 4.2: Use the Arduino hardware board as a USB to serial converter for programming by infection

- Use a TTL to USB converter like the one on the Arduino board (see figure 4.2) and connect it to a USB port of the PC.

In any case, the TTL side must be connected to the WEST port of a vector node. Then three simple steps must be performed to upload the sketch to a vector node:

1. In the Tools menu of the Arduino IDE the Board LumiNet Normal Node must be selected
2. In the Tools menu of the Arduino IDE the correct Serial Port must be selected
3. Upload sketch to board (hit the Upload button)

It is possible to use the Arduino board as a USB to serial converter. The safest way is to remove the microcontroller from the Arduino board and connect GND of the Arduino board to GND of the LumiNet node. Also the pins rx and tx (pin 0 and pin 1 on the Arduino board) must be connected to the left rx and tx pins of the LumiNet node (pins 6 and 7), as illustrated in figure 4.2.

Sensor nodes do not support programming by infection. Uploading sketches to a sensor node can only be performed by using an ISP programmer. The serial port of the ISP programmer must be selected from the serial port menu and in the Board menu, the LumiNet sensor node must be chosen. Programming sensor nodes is an advanced topic, the standard user writes code for normal nodes.

4.8.2 Uploading the Bootloader to LumiNet

The LumiNet nodes are preprogrammed with a bootloader. Normal nodes, sensor nodes, and vector nodes have different bootloaders. If for any reason a node lost its bootloader, it can be reprogrammed using the Arduino IDE. The bootloader cannot be uploaded via programming by infection. An ISP programmer is mandatory for this task. For example, the Arduino hardware board could be modified to become an ISP programmer for the LumiNet hardware (see figure 4.3).

Three steps have to be performed to upload the bootloader to a vector node:

1. In the Tools menu of the Arduino IDE, the board *LumiNet vector node* must be selected
2. In the Tools menu, the correct serial port of the ISP programmer must be selected
3. In the file menu, select Upload bootloader to board

For normal nodes and sensor nodes, the corresponding board must be selected in the first step.

The LumiNet edition of the Arduino IDE remembers the ISP programmer that was used and it has a keyboard shortcut for uploading the bootloader to the target node: CTRL+U. Because the original Arduino IDE does not have these two features, uploading bootloaders to many target boards is less comfortable at the moment.

If the user does not have an ISP programmer, the Arduino board can be modified to act as an ISP programmer. Four

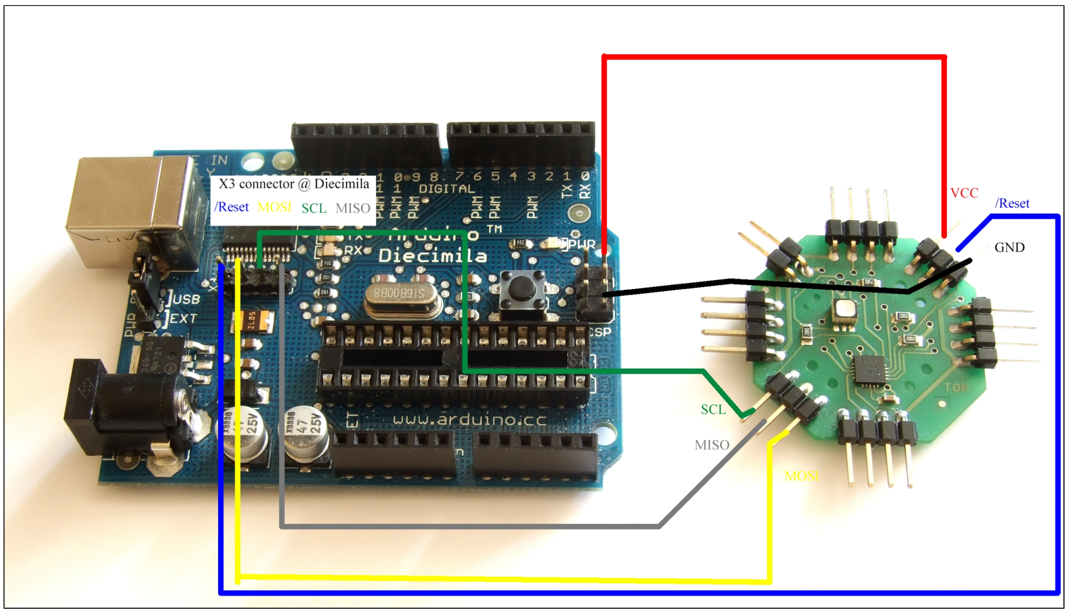


Figure 4.3: FTDI BitBang: Arduino Diecimila as an ISP for LumiNet

pins have to be soldered to the X3 port of the Arduino port. Then wires from these four pins and from GND and VCC must be assembled to a six-pin connector as illustrated in figure 4.3.

The Arduino ISP Programmer and the serial port that is connected to the USB connection of the Arduino board have to be selected in the menu. This feature is not available with the original Arduino IDE and is currently only supported for Windows and Mac OS X.

4.9 Problems Encountered and Their Solutions

In this section we present unexpected problems and solutions that should help to prevent those problems in the future.

4.9.1 Create BIN file from HEX file

Early implementations needed to create a .bin file with raw binary data from a .hex file that has intel hex ASCII encoding. Even the final implementation creates such a .bin file, but this is done in the background in a step of the LuminetUploader class of the IDE. The command that can convert hex to bin is called avr-objcopy and the tool can create the bin file with the following parameters:

```
avr-objcopy -I ihex -O binary main.hex main.bin
```

4.9.2 Fuses

Every Atmel microcontroller has fuse bits. They configure the microcontroller. LumiNet uses the attiny84 microcontroller and the following fuses must be programmed to a node:

```
efuse 0xFE
hfuse 0xDF
lfuse 0x62
```

The IDE programs these fuses of the node when a bootloader is uploaded, so the user does not have to take care of this.

4.9.3 Calling the Bootloader

As the bootloader is stored in flash memory at a fixed byte address, the normal program code that starts at flash address 0x0000 must call the bootloader if a vector node is present. A common practice for calling code that is located at a fixed memory address in ANSI C is to define a function pointer and then execute this function:

```
void startBootloader(void) = (void *) 0x1600;
```

```
main()
{
  ...
  startBootloader();
}
```

This works well for simple and small programs. But in more complex programs that use more memory, this notation does not work for the current avr-gcc compiler and the ATtiny84. The resulting assembler code might look like this:

```
lds r30, 0x0062
lds r31, 0x0063
icall
```

This means that the values stored at the sram addresses 0x0062 and 0x0063, are loaded into the so called Z register of the microcontroller, and then icall makes the controller run the code that is located at the memory address that was encoded in the sram cells. It seems like the linker does not store the correct values in the sram memory or that these values get overwritten at runtime. As a result, the node cannot call its bootloader anymore and this means that the network cannot be reprogrammed but must be disassembled and each node must be flashed using an ISP programmer.

The solution to this problem is to use the following assembler routine to call the bootloader:

```
ldi r30, 0x00
ldi r30, 0xb0
icall
```

This loads the word address 0x0b00[2] into the Z register (see C—"Glossary") and then calls the (bootloader) function that is located at this address.

[2]The flash memory uses 16bit words, so 0x1600 divided by 2 is the word address: 0x0b00

4.9.4 Use Assembler Code in an ANSI C Environment

The serial communication routines are written in assembler for performance reasons. Calling these assembler routines in an ANSI C context needs assembler code in a special format. The functions must be introduced by the .global modifier, followed by the .func modifier:

```
.global MyFunctionnameASM
.func MyFunctionnameASM
MyFunctionnameASM:
;* DO SOMETHING
    ret
.endfunc
```

If the C context wants to pass parameters to a function, the parameters are stored in registers r25, r24, etc., depending on the data type. If an assembler function has a return value, the same registers are used for it.

The assembler function must be declared extern in a .h header file.

4.9.5 Use Assembler Code in the Arduino C++ Context

The Arduino core is coded in C++. It was not possible to call the same assembler routines that were decribed in the previous section out of a C++ class. Actually the linker could not resolve the .global function names of the assembler routines. Instead, a C wrapper for the assembler code had to be written. Then the functions of this C wrapper can be called by C++ classes.

As a result, the same serial communication routines that are used in the bootloader can be used in any sketch for the LumiNet nodes if the application cares about deadlocks and the nodes have calibrated clocks.

4.9.6 Serial Communication and the delay() Function

Interrupts must be disabled while serial communication takes place. Otherwise wrong bits could be read or bits can be missed. The putc() and getc() functions disable all interrupts and reactivate them after they performed their task. The delay() function uses timer overflow interrupts and as a consequence, this function should not be used when serial communication is used. Instead, the _delay_loop_2() function should be used.

4.9.7 Clock Calibration

The microcontrollers come with a calibrated internal RC clock. This clock runs 8 MHz at 5 V and 25 degree Celsius. LumiNet should not be operated at 5 V because the LEDs are designed for 3.6 V. The battery pack supplies 3.6V. LumiNet runs with the CKDIV8 fuse programmed, so the clock of LumiNet runs at 8 MHz divided by 8 = 1 MHz. We measured the clocks of some nodes at 3.6V and instead of 1MHz they run at 1.017 to 1.024 MHz, which is too fast.

The clock signal can be debugged on pin PB2 if the low fuse of the device is programmed to 0x22. The signal on PB2 should read 1 MHz. If the clock is off, it must be calibrated. Atmel application note AVR053 describes how the clock can be calibrated.

Because PB2 is used for communication, the low fuse must be programmed to 0x62 after calibration is finished. The bootloader must be flashed at 0x1600 after calibration.

In an experiment we calibrated a node at a constant room temperature. Three important values can be found for this node in table 4.2. Different nodes have different values, so the table is not a reference for all LumiNet nodes but it shows the values measured for a certain node.

The ATtiny84 microcontroller on the sample node came with the value 0x74 pre-programmed which is a good value

OSCCAL	MHz at 5V	MHz at 3.6V
0x72	0.961	1.000
0x73	0.990	1.006
0x74	0.999	1.017
0x75	1.005	1.024

Table 4.2: Different values stored in OSCCAL register at 5V and 3.6V result in different clock speeds for a sample node.

for this node at 5 V. If the node is operated at 3.6 V (with this calibration value) it runs at 1.017 MHz which is too fast for the software UART implementation.

If a node cannot be calibrated close enough to 1 MHz it cannot be used with the software UART and thus cannot be used with the bootloader. Our discovery is that badly calibrated nodes should not be used in the LumiNet because this might destroy the payload.

Chapter 5

Evaluation

In this chapter we evaluate the implementation. Sensors and actuators connected to the LumiNet hardware will be used with the distributed physical computing framework. Examples for bio-inspired algorithms and applications show that the framework can be used for bio-inspired ideas and might motivate and inspire users to implement their own bio-inspired work.

5.1 Sensors and Actuators

Many sensors can be attached to a LumiNet node. Nodes that read and process sensor data should use a different program code than normal nodes. Because almost every Arduino sketch runs on LumiNet nodes, the sketches for reading sensors can easily be used on LumiNet sensor nodes. The Arduino community offers sketches for almost every kind of sensor, this includes potentiometers, photo sensors, temperature sensors, touch sensors, bending sensors, GPS, and DCF77 clocks.

Actuators are very similar to sensors. LumiNet does not define an actuator node class. Instead nodes with actuators are also called sensor nodes. Again, examples from the Arduino community can be used to control, for example, motors, speakers, relays, servos, etc.

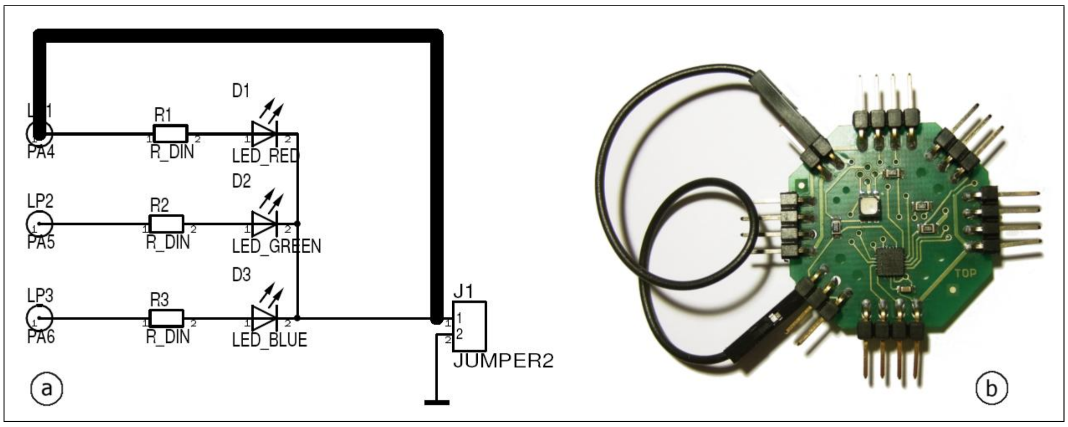


Figure 5.1: Lightsensor

It is even possible to use the on-board LED as a sensor as explained in the next paragraphs.

5.1.1 LED Light Sensor

[Dietz et al., 2003] explain how to use an LED as a photo-sensor. This concept needs two microcontroller I/O pins and an LED with a resistor between them.

Two steps must be taken to measure brightness with an LED:

1. The LED must be reverse-biased: the anode must be connected to a pin that is driven LOW and the cathode must be connected to a pin that is driven HIGH. The LED charges.

2. The HIGH pin must be turned to an input pin. The LED discharges. The time until the level falls below the digital input threshold can be measured. The longer this takes, the darker it is. The faster this is, the brighter it is.

As each LumiNet node has an rgb LED, a LumiNet node can be used to measure the brightness of ambient light.

The source code for the LED Light Sensor example can be found on the DVD that is provided with this thesis.

5.1.2 Touch Sensor

Under constant light conditions the light sensor that was introduced in the last subsection can be used as a touch sensor. When the LED is covered by a finger, the discharging time of the LED is much longer than when the LED is not covered.

Under changing light conditions this method does not produce reliable results. [Hudson, 2004] presents a way to solve this problem: two measurements must be taken:

1. Non-illuminated measurement phase: the brightness is measured by an LED without any other LED illuminated.

2. Illuminated measurement phase: while another LED is activated, the sensor LED measures the brightness.

Then the amount of reflected light can be determined by subtracting the brightness value of the illuminated measurement phase by the value of the non-illuminated measurement phase.

We tried to use the onboard rgb LED for this kind of sensor, but it is not possible to use one of them for illumination while another one is used for measuring the brightness, because they all share the same cathode. So this kind of sensor needs external LEDs, or simply use the light sensor from the previous section in ambient light.

It is also possible to touch the wire that connects the jumper pin with the rgb pin. The value of the measurement gets influenced by touching the wire. Thus the light sensor code can also be used as a touch sensor, but the user has to touch the wire, not the board or the LED.

5.2 Example Algorithms and Applications

In this section we present some examples for bio-inspired algorithms and applications that can be implemented on LumiNet. The examples use the distributed physical computing framework that we implemented for this work. Users can build upon these examples and create their own bio-inspired ideas with LumiNet. All examples can be found on the DVD that is supplied with this work. The software is also available for download at the LumiNet[1] project homepage.

5.2.1 Fading LED

This example is very easy: it uses a random number that determines the next color and then fades from the current color to the next color. All colors are displayed on the rgb LED that is controlled by Bynase for LEDs. No events are processed. This is a simple demo that demonstrates the capabilities of the LEDs. Users can extend this demo and add interaction by processing sensor readings or communication.

5.2.2 LightRing

The LightRing is one of the first applications that were developed on an early framework and was then ported to the new Arduino framework. A node samples all four input pins and stores the samples in a buffer. When the buffer is full, the samples are put on the output pins. This is similar to the idea of spiking neural networks. The node behaves like a neuron and the signals propagate through the network with an artificial delay that makes the signals interesting for human visual perception. Although it would be easy to use the Bynase functions for this application, we decided to keep the original algorithm to show that bio-inspired ideas can be implemented on LumiNet on the lowest I/O level.

[1]http://www.luminet.cc or http://hci.rwth-aachen.de/luminet

5.2.3 Cellular Automata on LumiNet

A cellular automaton consists of a grid of cells with a finite set of states. Each cell has a neighborhood of cells, that surrounds it. In a one-dimensional cellular automaton, each cell can have up to two neighbors, so three cells form a neighborhood. In a two-dimensional cellular automaton, the four direct neighbors (that are orthogonal) build the Von Neumann neighborhood and the set of all eight surrounding cells is called Moore neighborhood. All cells have the same rules for updating and all cells update at the same time. This leads to a new generation of cells. A Moore neighborhood can build 512 different patterns, a Von Neumann neighborhood can build 32 patterns. For each pattern, an update rule defines if the cell in the center of the neighborhood will change its state on the next update. One-dimensional cellular automata have 256 different update rules. The rules can be defined by two rows: the first row represents the current state of the three cells of the neighborhood and the second row contains the state of the cell in the center of that neighborhood after the next update.

neighborhood

If a cellular automaton is simulated on a finite grid (instead of an infinite plane), a problem is how neighborhoods at the edges should be handled. The simplest solution to this is to make their state constant. In this case, they are not affected by update rules. Another solution would be to define different neighborhoods with new update rules for cells located at edges. It is also possible to connect cells on the right side of the grid with cells on the left side of the grid, and cells on the bottom can be connected with cells on the top of the grid. This solution simulates an infinite tiling. The cells are no longer on a grid or plane but on a torus.

The one-dimensional cellular automata with a neighborhood of three cells have 8 update rules. The bit sequence of the center cells after the update process can be used to encode a binary number. The automata can be indexed by using this number. For example, the rule set shown in table 5.1 defines the cellular automaton with index number 110 (binary: 01101110).

current state	111	110	101	100	011	010	010	000
next center cell	0	1	1	0	1	1	0	0

Table 5.1: Cellular auomaton with rule set 110

This is a very interesting rule set because the behavior of it is neither completely random nor completely repetitive. Rule 110 has been the basis over which some of the smallest universal Turing machines have been built.

Table 5.2 defines the cellular automaton with index number 30 (binary: 00011110). This rule set can be used as a pseudorandom number generator with its center cell as output.

current state	111	110	101	100	011	010	010	000
next center cell	0	0	0	1	1	1	1	0

Table 5.2: Cellular auomaton with rule set 30

one-dimensional cellular automaton

All 256 elementary one-dimensional cellular automata can be simulated on LumiNet. A node is considered a cell of the automaton. The cells are arranged in a horizontal row, from the left to the right. Each cell stores the current state of the cells in its neighborhood. All nodes have the same eight update rules. A cell calculates its next state by the update rules and after an update, it changes its state and tells all neighbors about it.

two-dimensional cellular automaton

A grid of LumiNet nodes can be used to simulate a two-dimensional cellular automaton. Each cell has only four neighbors: one to the left, one to the right, one to the bottom, and one to the top. This allows communication within a Von Neumann neighborhood. As a consequence, it is not natural to run Conway's Game of Life (see [Gardner, 1970]) on LumiNet, because this uses the Moore neighborhood. Each cell updates its state using the same update rules. After an update, the cell sends the new state information to all four neighbors.

Cellular automata are considered to run synchronously. Without a central controller, at least two possible synchronization mechanisms are possible:

- A dedicated node behaves as a clock source for the network. This node sends a special (PERFORM_UPDATE) message at a constant update period.
- Another approach is to use a token. Only the node that holds the token is allowed to perform updates and then gives the token to the next node.

But it is also possible to let the cells update without this synchronization message. This will result in an asynchronous cellular automaton. An example for this class of automata would be a probabilistic cellular automaton. An example for a probabilistic cellular automaton is a forest fire simulation, where each cell can have one of these three states: 1. empty, 2. tree, 3. burning. With a certain probability, a cell with a tree will catch fire if at least one of its neighbors is burning. After some time, the tree burns down and the cell becomes empty. Empty cells cannot catch fire. Instead, on an empty cell, a new tree will grow.

example: forest fire

5.2.4 Genetic Algorithms on LumiNet

In 1859, Charles Darwin published his book "On the Origin of Species" (Darwin [1859]) in which he explains concepts like natural selection and survival of the fittest, which are the foundation of biological evolution. Evolutionary algorithms abstract from this biological process, especially by the following aspects:

1. **Selection and reproduction**: the fittest individuals will survive and produce offspring more probably.
2. **Crossover**: no two individuals are identical. Their genetic code is different and when two individuals produce offspring, parts from both parents' genes are mixed.
3. **Mutation**: random changes in genetic representation can help to adapt to a given environment.

Evolutionary Algorithms are designed for a purpose, for example to solve an optimization problem. So they do not try to simulate evolution but instead they use the concepts for solving problems.

Definition: *Genotype*

GENOTYPE:
The genotype is the genetic material of an individual. It contains distinguishable information units called genes. The genotype encodes the phenotypical properties of an individual. (see [Floreano and Mattiussi, 2008] p. 5)

An individual in an evolutionary algorithm is a possible solution to the problem that the algorithm tries to solve. The genotype of this artificial individual can be the binary representation of a floating-point number or any other data type. All possible genotypes form the search space.

Definition: *Phenotype*

PHENOTYPE:
The phenotype of an individual is its observable appearance, properties and characteristics. It is the manifestation of the individual. (see [Floreano and Mattiussi, 2008] p. 5)

Evolutionary algorithms often use a mapping between genotype and phenotype. In many cases this is a 1:1 mapping, but more sophisticated mappings are possible. The solution candidates of an evolutionary algorithm are elements of the problem space.

Definition: *Evolutionary Algorithm*

EVOLUTIONARY ALGORITHM:
An evolutionary algorithm is an (optimization) algorithm that uses mechanisms inspired by evolution, like natural selection, crossover and mutation. The fittest (surviving) individuals are possible candidates for a solution of a given problem.

The advantage of evolutionary algorithms compared to other optimization methods is that they only make few assumptions about the problem they should solve.

genetic algorithms

Genetic algorithms (see [Holland, 1992]) are a subclass of evolutionary algorithms operating on binary representations of the individuals' genotypes. The implementation of a genetic algorithm on LumiNet will be described in the following paragraphs.

In a row of LumiNet nodes, where all nodes are connected WEST to EAST, each node represents an individual of a population. All nodes use the same so-called "fitness function" to determine their own fitness. This function can be as simple as equation 5.1 and the higher the result of this function is, the higher is the fitness of the individual and thus the higher is the probability that the individual will survive.

Figure 5.2 shows the basic steps of a genetic algorithm. The following items explain how initialization, selection, crossover, and mutation are implemented by the provided example sketch for the LumiNet hardware:

- Initialization: Each LumiNet node creates an array of random binary values, the genotype.
- Selection: if a left neighbor node is present, it sends the genotype with the highest fitness value determined so far. If the fitness value of this gene is higher than the fitness value of the individual's own gene, than the gene is stored in a second array.
- Crossover: if the received gene is fitter than the current gene, then some bits of the own gene might be exchanged by bits of the better gene (received from the left neighbor). This happens with a high probability but not every time.
- Mutation: with a low probability it is possible that few bits of the gene can flip.

The (modified) own gene of the individual and the genome that was the best candidate to the node's left neighbor are compared and the fitter gene is sent to the right neighbor node. If a node has no right neighbor (the right-most node) then after some delay, the node sends the fitter value to its

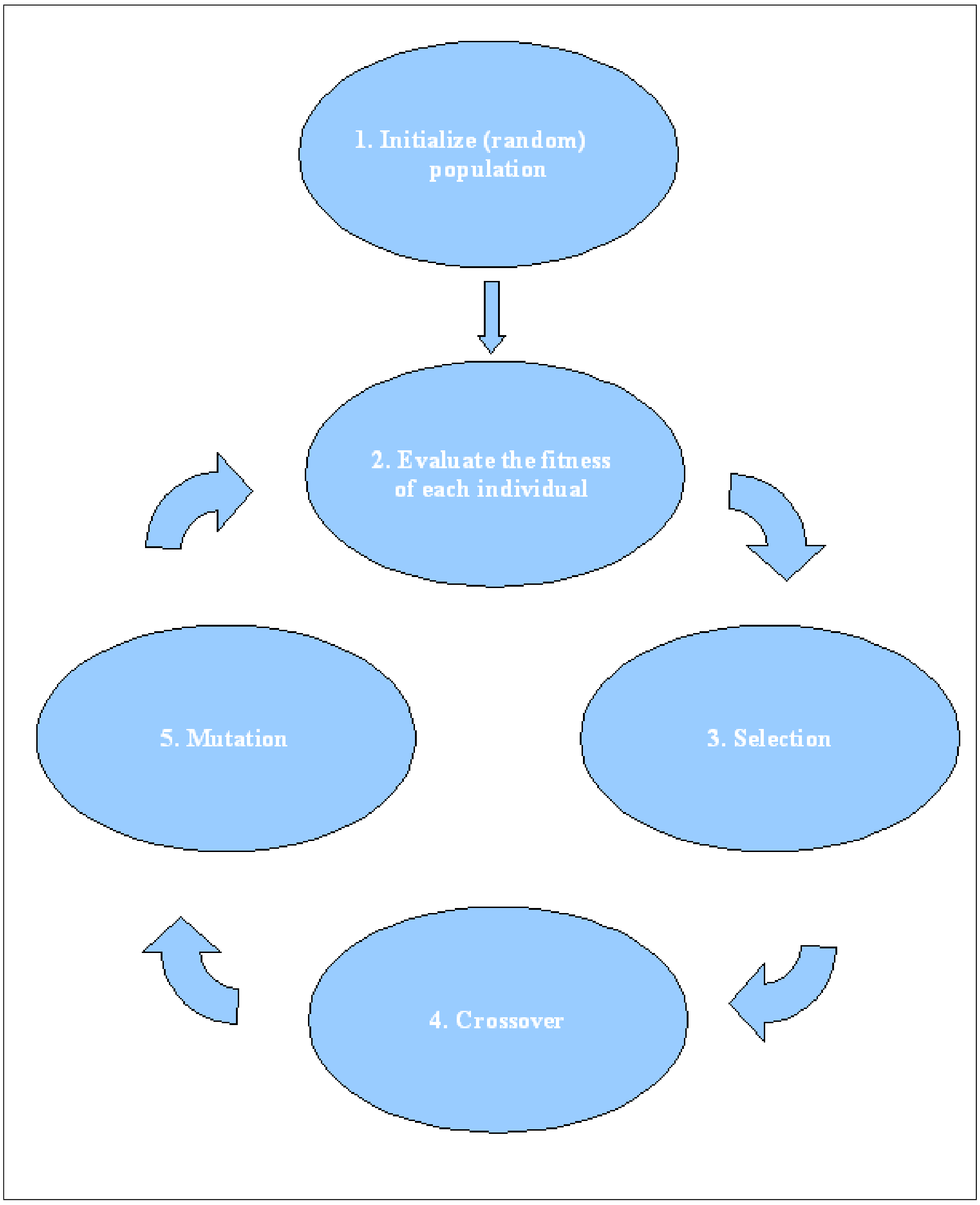

1. Initialize (random) population
2. Evaluate the fitness of each individual
3. Selection
4. Crossover
5. Mutation

Figure 5.2: Genetic Algorithm

left neighbor. Candidates that come from the right neighbor are only compared (selection) but no crossover or mutation happens after this selection. After a defined number of steps the algorithm terminates and the nodes output their genes by blinking bit after bit. Blue means: bit is set, red means: bit is not set (bit is clear). White indicates the start of the byte (gene).

The example GeneticAlgorithm determines possible solutions to this problem: Find the minimum of the equation 5.1:

$$f(x) = x^2 - 4x + 5 \tag{5.1}$$

Of course, this is a simple problem that can be solved using simple mathematics, but it shows that the concept of a genetic algorithm can be formulated and used on LumiNet.

5.2.5 Langton's Ant

A simple form of artificial life is called Langton's Ant. An aritifical ant moves over a two-dimensional grid, according to a simple rule set:

- If the cell is active, turn 90 degrees to the right, deactivate the current cell, and move forward to the next cell in the new direction
- If the cell is inactive, turn 90 degrees to the left, activate the current cell, and move forward to the next cell in the new direction

Langton's ant can be described as a cellular automaton, but we implemented it using direct commands and logic.

The implementation on LumiNet makes use of the rgb LED and all communication pins. The internal pull-up resistors of all nodes are activated for all input pins. All output pins are HIGH. When a falling edge on an input pin is detected the node indicates this by activating the red LED. This means that the ant is arriving on that node now. The node updates the blue LED according to the two update rules presented before. Then the node pulls the output line according to the update rule to LOW for a certain amount of time and then the line must be set HIGH again. The ant moves to the next node.

5.3 Power Consumption

LumiNet is designed for mobile applications that use a battery pack. Thus, electrical energy is a limited ressource. The microcontroller consumes less power if it operates at a lower clock speed. Therefore, 1 MHz is the default clock speed for a luminet node. The clock speed can be changed at runtime from 1 MHz up to 8 MHz by this code fragment:

```
cli();
CLKPR = (1 << CLKPCE);//INITIALIZE CLOCK CHANGE
CLKPR = 0x00;//NO PRESCALER
```

Sleep mode can be activated using this code:

```
#include <avr/sleep.h>
void setup()
{
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
  sleep_mode();
}
```

We measured the power consumption of a LumiNet node at 5V and at 3.6V that is

1. doing nothing (all pins INPUT),
2. doing nothing and pins 2,3,4,5,6,7 are HIGH (OUTPUT) and
3. doing nothing and in sleep mode.

The same sketches were tested on an Arduino Diecimila and a LilyPad Arduino, both operated at 5V. Table 5.3 shows the result of the measurement.

#	Diecimila 5V 16 MHz	LilyPad 5V 16 MHz	LumiNet 5V 1 MHz	LumiNet 5V 8 MHz	LumiNet 3.6V 1 MHz	LumiNet 3.6V 8 MHz
1	26.5	8.15	1.39	6.14	0.99	4.29
2	25.8	7.94	1.43	6.39	1.02	4.33
3	8.69	0.30	0.30	0.30	0.24	0.24

Table 5.3: Power consumption of an Arduino Diecimila, a LilyPad Arduino, and a LumiNet node, performing three different tasks at different voltage levels and system clocks

5.4 Requirements Analysis

In this section we analyze if all requirements are met by the implementation.

The three main requirements are met:

- M1 (improve usability): The Arduino framework works well with the LumiNet hardware and it is easy to write applications with it. It is easy to learn the core functions of the framework and the user community offers a lot of help and examples for a quick start.
- M2 (reprogramming): The user can use the modified Arduino IDE to prepare a vector node by uploading payload to it. The vector node can then infect an assembled network without the help of a PC.
- M3 (bio-inspired): A bio-inspired communication protocol was integrated into the core of the new Arduino framework, and on top of it bio-inspired applications can be implemented. Examples like cellular automata, genetic algorithms, etc. show that LumiNet can be used for bio-inspired applications.

The six requirements R1-R6 are also met:

- R1: Many kinds of sensors can be connected to a LumiNet hardware board, because the Arduino community offers a lot of examples. This thesis also shows how the on-board LED can be used as a light-sensor and a simple wire can be used as a touch-sensor.

- R2: All ports are supported by Bynase, RS232 and programming by infection.
- R3: The mcu of every node can be used for computations and data can be exchanged between nodes. This allows distributed processing. The genetic algorithm example uses distributed computations to determine the solution of an equation.
- R4: Sensors can be used for interaction or sketches can use random numbers to avoid constant pattern generation if no interaction is possible.
- R5: The framework was introduced to the Arduino community. Because the modified Arduino IDE is open-source, the members of the community can improve it in the future and can distribute and use it everywhere they want.
- R6: The Arduino framework is a multi-platform development system and the same multi-platform technologies like, e.g., Java are used by the new framework that is introduced by this thesis.

Chapter 6

Summary and Future Work

6.1 Summary and Contributions

In this work, a distributed physical computing framework was implemented for a low-cost microcontroller-based hardware board. Instead of creating a new framework from scratch, the popular Arduino framework was extended to support the LumiNet hardware. This allows a wide audience to use LumiNet and in return LumiNet users become a part of the big Arduino open source community.

The main contribution of this work is that it shows that bio-inspired concepts and distributed computing can be used in a physical computing framework. The Arduino framework also benefits from this work because the provided patches allow to use assembler files in the Arduino core and sketches can run on ATtiny microcontrollers. It is even possible to connect Arduino hardware and LumiNet hardware and use the bio-inspired Bynase communication protocol on both of them, since a Bynase library for Arduino is part of this work.

Inspired by the network reprogramming mechanisms of wireless sensor networks, the programming by infection strategy was implemented for LumiNet. It allows to reprogram LumiNet at places where no PC is available and it is not necessary to disassemble the network for this task. The programs that are stored on the vector nodes have been uploaded by the easy to use, modified Arduino IDE. This IDE also allows to upload the bootloader to empty LumiNet nodes and it makes software development easy, even for people who are new to programming.

6.2 Future Work

A few bio-inspired algorithms were implemented in this work, but there is no framework yet for creating such algorithms. Future versions of the IDE can support the development of bio-inspired algorithms by offering code skeletons or wizards.

A user study can help to identify if users have problems with the current framework. An interesting question would be if the users realize any differences between developing sketches for the LumiNet hardware and developing sketches for the original Arduino hardware. Another user study could compare the LilyPad Arduino and LumiNet in wearable computing applications.

The remaining libraries must be ported to the LumiNet framework, making LumiNet almost completely compatible to the Arduino. Advanced parts like interrupts and timers should be optimized and tested in many applications, although the current implementation seems to have no serious problems.

One of the main application areas of LumiNet is wearable computing. Other application areas must be detected and published to the community.

hardware redesign

A hardware redesign can improve various limitations of the current platform. For example, an external crystal could provide a better clock signal, although this change would occupy two pins. This modification would solve problems with the serial communication and allows easier synchronization of nodes.

The current hardware board connects only two colors of the rgb LED to hardware PWM pins. A redesign should connect all three colors of the LED to hardware PWM pins. The LED fading can then be implemented using PWM timers. Thus the Bynase implementation can be limited to communication and the saved resources of the mcu can then be used for custom code.

The jumper must disappear. A user has to remove the jumper for code upload via ISP programmer. This is hard to understand and a common source for errors.

A redesign must offer more labels and other marks that help the user to identify the directions. Maybe another shape of the boards can also support a quicker identification of the directions. The quite rectangular shape of the boards might be changed into a more triangular shape like the one used by the NeuroLED.

The pin layout of the connectors must be changed, because the current version is too dangerous. If the wrong ports are connected, this can result in a short circuit. We propose that VCC should be next to GND and that no other pins should lay between VCC and GND. The power supply connection must be more stable and safer, especially for wearable computing projects.

Whenever the Arduino software updates to a new release, the LumiNet software should also include the same improvements and provide compatibility. A full integration of the LumiNet project into the Arduino project is desirable.

Appendix A

APPENDIX: Installation and Setup

This chapter describes how to connect nodes to a LumiNet network and how to get started using the framework.

A.1 Hardware Installation

This sections explains how to connect nodes in a network. Special care must be taken when a power supply gets connected to a LumiNet network.

A.1.1 Connecting Nodes

Every node can have up to four direct neighbors. They are connected to each other using four wires in each direction. One of the wires offers the supply voltage (VCC) and one wire is the common ground terminal (GND). The other two pins can be used for communication. The wire that provides VCC is blue while all other wires are black. It is important that only blue[1] wires get connected to VCC.

[1]Blue is the default. If an other color, e.g. red, is used to mark VCC, then this must be used consequently.

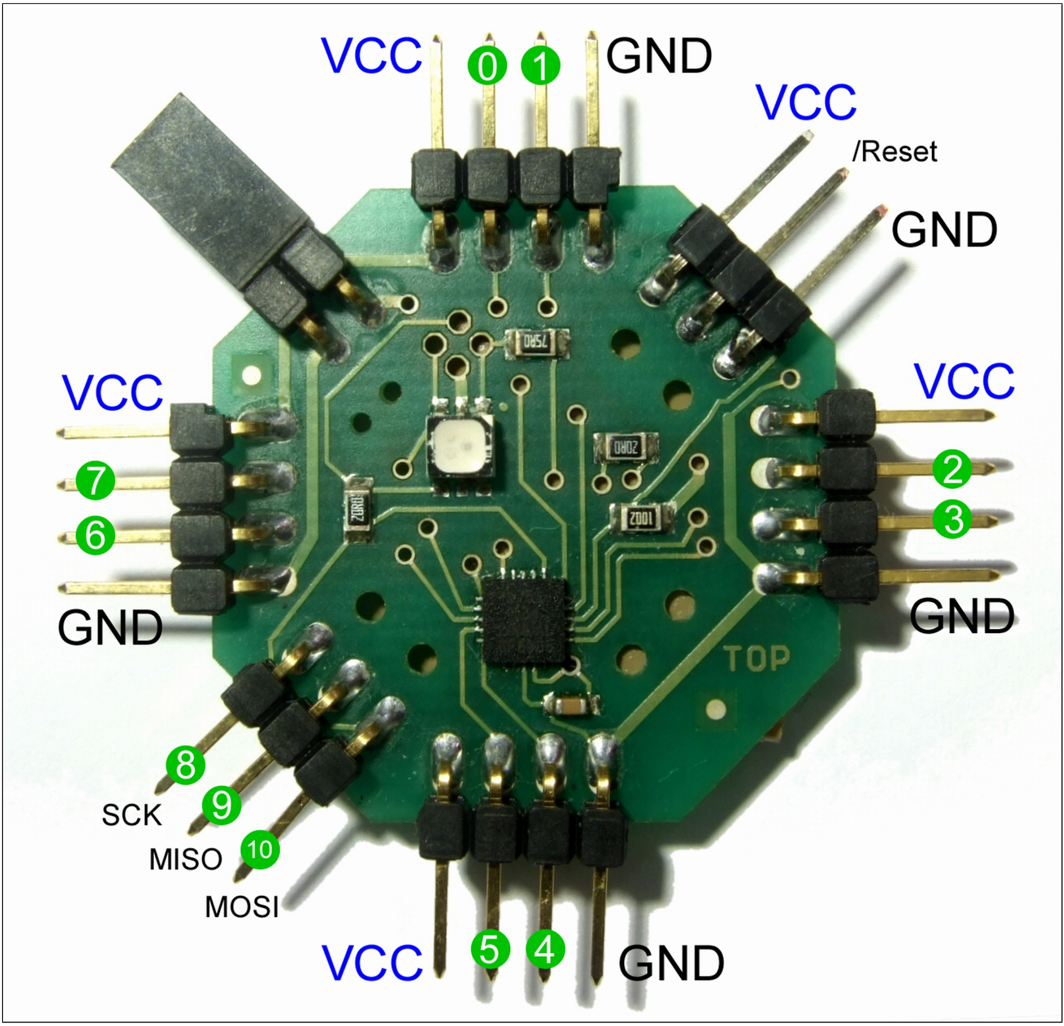


Figure A.1: Pins of the LumiNet hardware board

Figure A.1 shows all pins of a LumiNet hardware board. The numbers are the pin numbers that can be used by commands like pinMode(), digitalWrite(), etc.

Pins 8,9, and 10 have special functions, as they are not only used for ISP programming if the jumper is removed, but they are connected to the three colors of the rgb LED if the jumper is set. The mapping of these three pins is shown in table A.1

Two nodes can be connected to each other either horizontally aligned or vertically aligned.

pin	LED color	mcu pin	ISP pin
8	red	PA4	SCK
9	green	PA5	MISO
10	blue	PA6	MOSI

Table A.1: Pin mappings for pins 8, 9, and 10

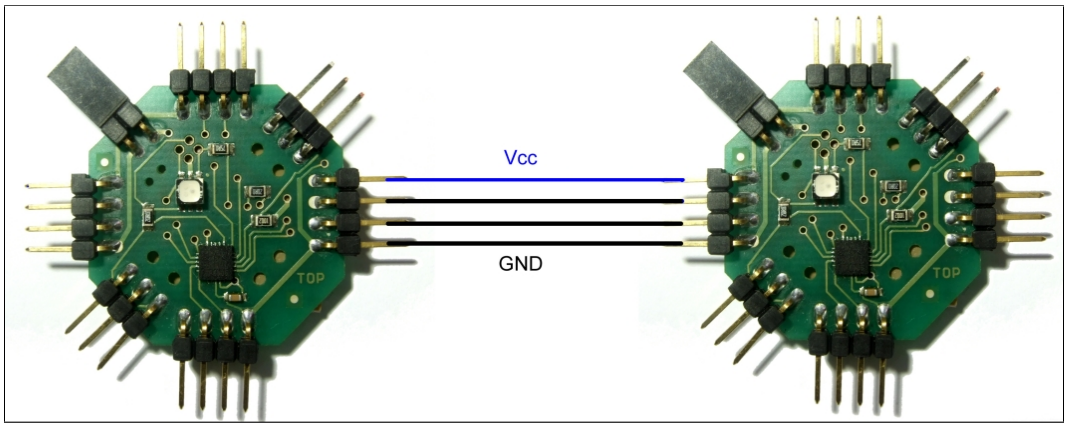


Figure A.2: Two LumiNet nodes - horizontal

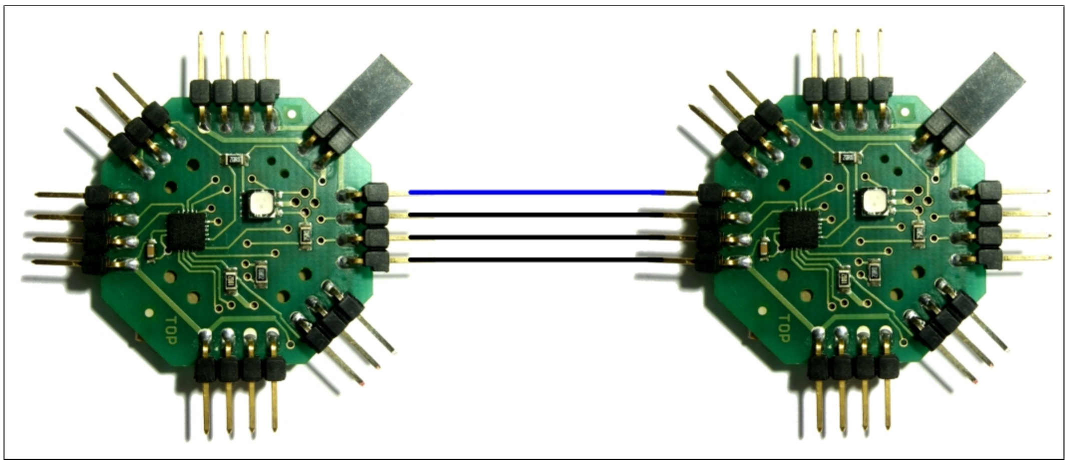


Figure A.3: Two LumiNet nodes - vertical

Figure A.2 shows how to connect two LumiNet nodes horizontally. The blue wire marks the supply voltage pin VCC.

Figure A.3 shows how to connect two LumiNet nodes vertically. In contrast to Figure A.2 , the nodes are rotated by 90 degrees. It is important that only a SOUTH port of a node can be connected to a NORTH port of another node. It is not allowed to connect a EAST, WEST or NORTH port to a NORTH port. Only a WEST port can be connected to an EAST port, etc.

A.1.2 Power Supply

The power supply can be attached to any pair of VCC and GND pins. It is recommended to use the two pins of the ISP pin header, because this way the power supply and four neighbors can be connected to the node at the same time.

reversed voltage levels can destroy the hardware

The supply voltage must be smaller than 5 V and since there is no protection against wrong polarity, reversed voltage levels can destroy the hardware. Additional components should be added for safety, especially in wearable computing projects. The power supply must provide enough current for all nodes of the network.

A.1.3 Connect an ISP programmer

The six pins that must be connected to an ISP programmer can be seen in figure A.1: on the right side (diagonal) VCC, /RESET, and GND, and on the left side: SCK, MISO, and MOSI. The jumper must be open, otherwise the node cannot be programmed via ISP.

A.2 Software Installation

On the DVD that is distributed with this thesis, a folder called "Software" contains the compiled binary files of the framework that can be used on Windows and Mac OS X. The software is also available for download at the LumiNet[2] project homepage.

[2]http://www.luminet.cc or http://hci.rwth-aachen.de/luminet

The corresponding subdirectory that fits the used operating system must be copied to the hard drive.

The directory ”LumiNet_sketches” should also be copied to the hard drive. All examples that are discussed in this paper are included in this directory.

A.2.1 Run the IDE

On Windows: double-click the run.bat file.

On Mac OS X: double-click the Arduino icon.

It is also possible to run the IDE from the command line. On Windows, the run.bat can be called from the cmd shell. On Mac OS X, a terminal should be opened and in the directory where the Arduino icon is located, the following command starts the IDE:

```
./Arduino.app/Contents/MacOS/Arduino
```

While run.bat on Windows automatically opens a shell in the background, there is no shell visible after double-clicking the Arduino icon on Mac OS X. It is recommended to open the shell because useful status and dabugging information are only provided in the shell and not in the IDE.

A.2.2 Open an Existing Sketch

In the ”File” menu the item ”Open” opens a file browser dialog. In the directory called ”Luminet_sketches” the corresponding .pde file must be selected. The sketch Blink/Blink.pde is a good start.

Every example includes a description that explains how to wire components for this example. It also gives a brief explanation of what the example does. The description can be found in the header of each .pde file.

A.2.3 Compile the Sketch

After the correct board (”LumiNet Vector Node”) was selected in the ”Board” menu, the sketch will be compiled after the ”Play” button is clicked. The sketch will not be automatically uploaded to the board by this step.

A.2.4 Upload a Sketch to the Network

Three steps are required in order to upload a sketch to a LumiNet network:

First, the sketch must be uploaded to a vector node:

1. Select ”LumiNet Vector Node” in the ”Board” menu.
2. Select the serial port that connects to the vector node.
3. Connect serial interface to the WEST port of the vector node
4. Add a jumper to the vector node
5. Click the ”Upload to Board” button.

Then the vector node must be disconnected from the PC and the jumper must be removed from the vector node.

In the next step, the vector node must be connected to the LumiNet network. When the network is powered up the next time, the programming by infection mechanism starts. After all nodes are reprogrammed, the power supply must be removed and then the vector node must be removed from the LumiNet network.

Another option is to connect the network to the EAST port of the vector node and leave the network connected while the vector node receives new program code from the PC. In this case, the network is automatically reprogrammed after the vector node received the new program code.

Appendix B

APPENDIX: Reference Of Hardware Design

This chapter shows the schematics and the board layout of the LumiNet hardware boards revision 3.6. The figures were created by Professor Jan Borchers.

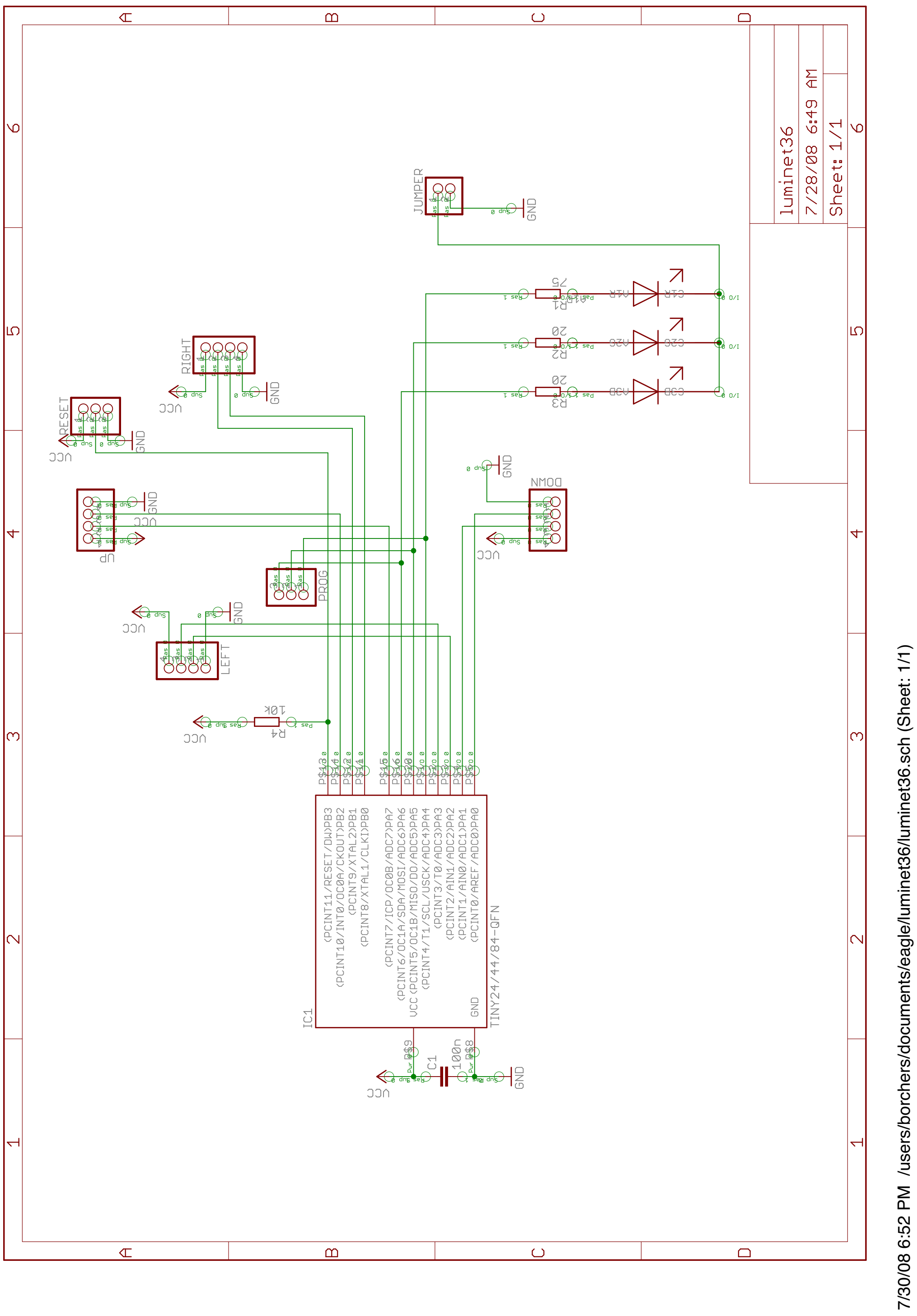


Figure B.1: LumiNet hardware board revision 3.6 schematics

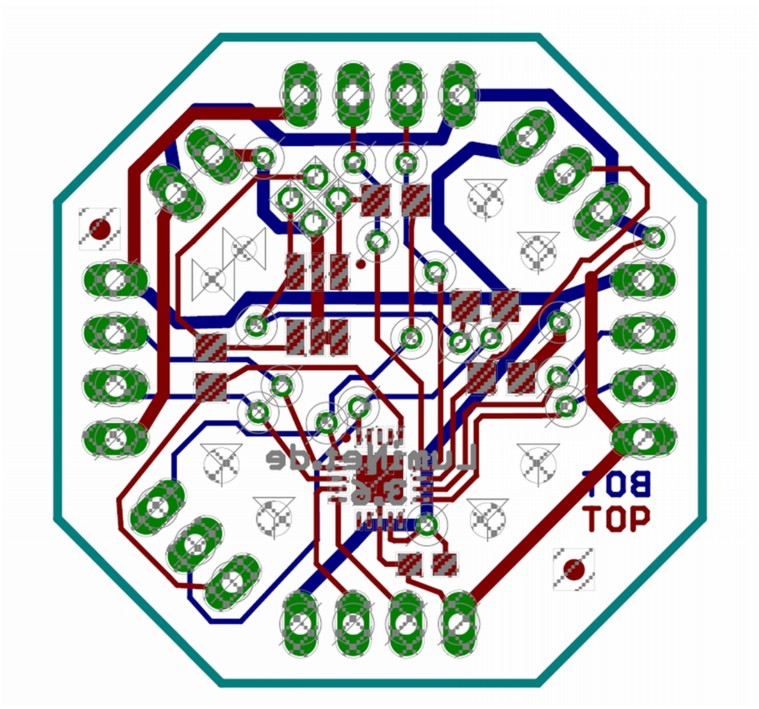


Figure B.2: LumiNet hardware board revision 3.6 layout

Appendix C

Glossary

Bootloader: A bootloader is a piece of software that can load program code to the flash. Usually, this program runs after boot-up and is stored in a dedicated region of the flash.

DCF77: is a radio station that broadcasts time information signals. The callsign of this station stands for D=Germany, C=long wave signal, F=Frankfurt, 77=77.5 kHz (frequency).

EEPROM: Electrically Erasable Programmable Read-Only Memory is a non-volatile memory. A microcontroller can use this memory to store data or read data like configuration bytes.

Flash memory: is a specific type of EEPROM that has to be erased, programmed, and read in blocks or pages. Microcontrollers often store their program code in this non-volatile memory.

I^2C: Inter-Integrated Circuit is a serial computer bus that uses two wires. It was invented by Philips. A popular variant used by some microcontrolelr vendors is called TWI (two wire interface).

GPS: the Global Positioning System uses satellites to allow navigation on Earth. Some GPS receivers can be connected to a Microcontroller using serial communication. The microcontroller can then read the current position from the GPS receiver hardware.

ICSP programmer: In Circuit Serial Programming allows to reprogramm microcontrollers in the circuit. Thus the microcontroller must not be removed from the circuit. This allows to reprogramm the mcu in a system, the mcu must not be programmed before it gets installed in the system.

ISP programmer: In-System Programming: see ICSP programmer

ISR: An Interrupt Service Routine, also known as Interrupt Handler, is a software callback routine that gets triggered by occurrence of an interrupt. This can be a hardware interrupt such as a timer overflow interrupt, or a software interrupt.

lsb: the least significant bit is the bit of an integer number that defines if the number value is odd or even.

mcd: stands for ″millicandela″ or 1/1000 of a candela. A candela is the the SI base unit of luminous intensity

Microcontroller: a simple microprocessor with additional peripherals like memory or timers on a single integrated circuit (IC).

msb: the most significant bit is the bit of an integer number is the bit that determines the signess of a signed number value or the bit with the biggest value of an unsigned number value.

RS232: Recommended Standard 232 is a standard for serial binary data signals connecting two devices.

SRAM: Static Random Access Memory is a kind of volatile memory that must not be periodically refreshed. Microcontrollers can use this memory to store dynamic data at runtime.

TTL: Transistor–transistor logic is a class of electronic devices that operate at a voltage between 0 V and 5 V. A TTL signal is defined as ″LOW″ when it is between 0 V and 0.8

V, and it is defined as "HIGH" when it is between 2.2 V and 5 V. Most microcontrollers operate within this range.

TWI: see I^2C

UART: a Universal asynchronous receiver/transmitter is a piece of hardware that is used for serial data communication. If a microcontroller includes a UART, then the main processor of the microcontroller can do other tasks while the UART handles the serial communication.

XBee: the XBee modules use the physical layer and the media access layer of ZigBee and allow Microcontrollers to use it via a serial command set.

ZigBee: the The ZigBee Alliance specified this suite of radio communication protocols. It supports small, low-power digital radios based on the IEEE 802.15.4 standard that specifies the physical layer and media access control for low-rate wireless personal area networks.

Z register: a special register of the Atmel ATtiny (and ATMega) mcus. It is a 16-bit register containing the register pair R30, R31. It is a pointer register that is able to point to a 16-bit SRAM address or to a location of the program memory (e.g. a word address in flash memory). The higher byte of the Z register (R31) is called ZH and the lower byte (R30) is called ZL.

Bibliography

Atmel. Avr305: Half duplex compact software uart. "http://www.atmel.com/dyn/resources/prod_documents/doc0952.pdf", 2005.

Leah Buechley and Michael Eisenberg. Fabric pcbs, electronic sequins, and socket buttons: techniques for e-textile craft. *Personal Ubiquitous Comput.*, 13(2):133–150, 2009. ISSN 1617-4909. doi: http://dx.doi.org/10.1007/s00779-007-0181-0.

Leah Buechley, Mike Eisenberg, Jaime Catchen, and Ali Crockett. The lilypad arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 423–432, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: http://doi.acm.org/10.1145/1357054.1357123. URL http://portal.acm.org/ft_gateway.cfm?id=1357123&type=pdf&coll=GUIDE&dl=GUIDE&CFID=24765571&CFTOKEN=91296219.

C. Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, London, 1859.

Paul H. Dietz, William S. Yerazunis, and Darren Leigh. Very low-cost sensing and communication using bidirectional leds. In *Ubicomp*, pages 175–191, 2003.

Prabal K. Dutta, Jonathan W. Hui, David C. Chu, and David E. Culler. Securing the deluge network programming system. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 326–333, New York, NY, USA, 2006. ACM.

ISBN 1-59593-334-4. doi: http://doi.acm.org/10.1145/1127777.1127826.

Dario Floreano and Claudio Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press, 2008. ISBN 0262062712, 9780262062718.

Dario Floreano, Yann Epars, Jean-Christophe Zufferey, and Claudio Mattiussi. Evolution of spiking neural circuits in autonomous mobile robots: Research articles. *Int. J. Intell. Syst.*, 21(9):1005–1024, 2006. ISSN 0884-8173. doi: http://dx.doi.org/10.1002/int.v21:9.

M. Gardner. Mathematical games: the fantastic contributions of john conway's new solitaire game "life.". *Scientific American*, October 1970:120–123, 1970.

John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-58111-6.

David Holman and Roel Vertegaal. Organic user interfaces: designing computers in any way, shape, or form. *Commun. ACM*, 51(6):48–55, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1349026.1349037.

Shinichi Hosomi, Masahiko Tsukamoto, and Shojiro Nishio. A system for controlling led blink in wearable fashion. In *IWCMC '07: Proceedings of the 2007 international conference on Wireless communications and mobile computing*, pages 665–670, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-695-0. doi: http://doi.acm.org/10.1145/1280940.1281081.

Scott E. Hudson. Using light emitting diode arrays as touch-sensitive input and output devices. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 287–290, New York, NY, USA, 2004. ACM. ISBN 1-58113-957-8. doi: http://doi.acm.org/10.1145/1029632.1029681.

Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM. ISBN 1-58113-879-2. doi: http://doi.acm.org/10.1145/1031495.1031506.

Tom Igoe. *Making things talk.* O'Reilly, 2007. ISBN 9780596510510.

Mohammad Mostafizur Rahman Mozumdar, Luciano Lavagno, and Laura Vanzago. A comparison of software platforms for wireless sensor networks: Mantis, tinyos, and zigbee. *Trans. on Embedded Computing Sys.*, 8(2):1–23, 2009. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/1457255.1457264.

Munehiko Sato. Particle display system: a real world display with physically distributable pixels. In *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, pages 3771–3776, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-012-X. doi: http://doi.acm.org/10.1145/1358628.1358928.

Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008. ISSN 1550-4859. doi: http://doi.acm.org/10.1145/1340771.1340774.

Index

Typeset May 18, 2009