WirelessHART- Implementation and Evaluation on Wireless Sensors

DAVID GUSTAFSSON

Masters' Degree Project
Stockholm, Sweden April 2009

XR-EE-RT 2009:003
Abstract

The process automation industry is adopting wireless technologies, but in such harsh environments technologies like Bluetooth and ZigBee are not reliable enough. An alternative to Bluetooth and ZigBee is WirelessHART that can cope with the noisy environment but instead imposes very high demands on the hardware. We show that it is possible to run WirelessHART on sensor nodes with as little as 10 kilobytes of RAM and a processor running at 2.46 MHz. This is the first major step towards an open alternative to the proprietary implementation of WirelessHART.
Acknowledgements

This thesis has been performed in the Networked Embedded Systems group at SICS. I would like to thank my main advisor Niclas Finne from SICS for his support and all the fruitful discussion we have had. I would also like to thank my advisors Thiemo Voigt from SICS, Mikael Johansson from KTH and Tomas Lennvall from ABB for their support. Additionally I would like to thank the whole NES group for their support and for making this a very enjoyable time.

I want to dedicate this thesis to my family and especially my mom and dad for their never-ending love and support.

This thesis has been performed within the SICS Center for Networked Systems funded by VINNOVA, SSF, KKS, ABB, Ericsson, Saab Systems, TeliaSonera and T2Data.
## Contents

1 Introduction ........................................... 1  
1.1 Problem Statement ................................. 1  
1.2 Method ............................................. 2  
1.3 Alternative Approaches ............................ 2  
1.4 Scientific Contributions ............................ 2  
1.5 Thesis Structure .................................. 2  

2 Background ........................................... 5  
2.1 Wireless Sensor Networks .......................... 5  
2.2 The Contiki Operating System ..................... 5  
2.3 Industrial Process Automation Protocols .......... 7  
2.3.1 HART ............................................ 7  
2.3.2 WirelessHART ..................................... 9  
2.3.3 ZigBee PRO ...................................... 11  
2.3.4 6LoWPAN ........................................ 12  
2.3.5 ISA100.11a ....................................... 13  

3 Design and Implementation ......................... 15  
3.1 Architecture ....................................... 15  
3.1.1 Events .......................................... 16  
3.1.2 Callback Functions ............................. 16  
3.1.3 Final Design .................................... 17  
3.2 Code Modules ...................................... 17  
3.2.1 Network Layer (whnet) ......................... 17  
3.2.2 Transport Layer (whtrans) ..................... 18  
3.2.3 Communication Table Storage (session) ......... 19  
3.2.4 Pool of Packets (packetpool) ................. 20  
3.3 Handling of Commands ............................. 20  
3.4 Communication Tables ............................. 20  
3.5 Joining a Network .................................. 22  
3.6 Routing Packets .................................... 22  
3.7 Packet Buffers ..................................... 23  
3.8 Remaining Work .................................... 23  

4 Evaluation ........................................... 25  
4.1 TMote Sky .......................................... 25  
4.2 Reliability ......................................... 25  
4.2.1 Simulated Experiments ......................... 26
4.2.2 Experiments on Real Hardware .............................. 27
4.3 Memory Footprint .............................................. 27
  4.3.1 Memory Consumption ................................... 28
  4.3.2 Code Size .................................................. 29
  4.3.3 Comparing Memory Footprint ......................... 30
4.4 Time Slotted Experiment .................................. 30
4.5 Future Experiments .......................................... 31

5 Related Work .................................................. 33

6 Conclusions and Future Work ................................. 35
  6.1 Conclusions ................................................. 35
  6.2 Future Work ................................................ 36

References ....................................................... 37
List of Figures

2.1 Typical architecture of a multihop wireless sensor network [34] . 6
2.2 Structure of the Rime stack, showing the more basic abstractions
at the bottom . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 HART OSI 7-Layer Model . . . . . . . . . . . . . . . . . . . . . 8
2.4 Channel hopping . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.5 An example of a WirelessHART network . . . . . . . . . . . . 10
2.6 Types of frequency hopping supported by ISA100.11a . . . . . . . 14

3.1 The core modules that make up the WirelessHART stack . . . 17
3.2 A subset of the communicating tables and the relationship
between them . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.1 Node overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Link qualities relation to transaction success rate in simulation . 26
4.3 Link qualities relation to transaction success rate running on real
hardware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.4 Node overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSMA</td>
<td>Carrier Sense Multiple Access</td>
</tr>
<tr>
<td>FSK</td>
<td>Frequency Shift Keying</td>
</tr>
<tr>
<td>HART</td>
<td>Highway Addressable Remote Transducer</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>ISM</td>
<td>Industrial, Scientific and Medical</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MIC</td>
<td>Message Integrity Code</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
</tr>
<tr>
<td>OSI Model</td>
<td>Open Systems Interconnection Basic Reference Model</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RSSI</td>
<td>Received Signal Strength Indication</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time Division Multiple Access</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Packet</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Wireless Sensor Networks (WSN) are used in many different application areas ranging from environmental monitoring to sniper localization [29, 35]. WSNs consist of sensor nodes equipped with sensors and radio transceivers. The nodes jointly monitor their immediate surroundings and send the results to the sink. This has resulted in an increased popularity among industry and especially companies interested in the area of industrial process automation. Several organizations [2, 4, 5] in this field have actively been endorsing the adoption of wireless technologies. A milestone came in September 2007 when the HART Communication Foundation released the first open wireless communication standard for process measurement and control, namely WirelessHART. Another breakthrough came one year later, in September 2008, when the International Electrotechnical Commission (IEC) approved the specification of WirelessHART as a Publicly Available Specification (IEC/PAS 62591 Ed. 1).

1.1 Problem Statement

Before the release of the WirelessHART specification the specifications of ZigBee and Bluetooth had already been publicly released. But these technologies were rejected by the process automation industry because they could not satisfy the high demands. In a comparison between ZigBee and WirelessHART Lennvall et al. show that the latter is more suitable for industrial applications [25].

The goal of this thesis is to provide an open implementation of WirelessHART. The major challenges for this project are the constraints imposed by both the protocol and the hardware. These constraints are large memory demand and the time critical nature of the protocol. The principal design goals of the implementation are low computational complexity and low memory consumption.

The software should be designed to run on wireless sensor nodes. The normal sensor node is small in size and has a relatively low price. The size and price are very important and enables the monitoring of not only the most expensive equipment but also the less expensive one. More powerful hardware consumes more energy and this is not desirable since the power source most often is limited, e.g. a battery. Changing batteries can be a tedious task especially if the nodes are deployed in remote and/or hostile environments. The
implementation should be done in the Contiki operating system. Contiki is an operating system developed at SICS and specially designed for memory-efficient networked embedded systems and wireless sensor networks.

The implementation should also be evaluated with respect to reliability as this is one of the main design goals of the protocol. Furthermore, memory usage and time slotting should be evaluated to see if it is feasible to run WirelessHART on sensor nodes.

1.2 Method

The method used in this thesis is experimental computer science. We start by studying the literature of WirelessHART and other protocols related to the topic and then continue to design the implementation. The protocol has been developed in iterations. Each iteration contained the following two steps: a) implementing a part of the protocol and b) check correctness of the implemented part by basic functionality testing. An evaluation has been performed to show that the stack can run on wireless sensors.

1.3 Alternative Approaches

WirelessHART has been designed in a layered fashion. Each of the layers solves a different part of the communication problem.

In this thesis the different layers have been implemented using one process. The cross-layer communication between the layers in the stack is done by using function calls. This is similar to how Rime and uIP are implemented. The application layer registers a set of callback functions for communication between the stack and the application.

An alternative approach would be to implement the different layers in individual processes and using synchronous events for communication between the layers. Each process would be waiting for an event to occur; process it and then wait for the next event to occur. Processes in Contiki are not preemptable and thus each process must release control, by yielding, for the other processes to be able to execute.

1.4 Scientific Contributions

The contributions of this thesis are twofold. First we show that it is possible to implement and run WirelessHART on resource-constrained sensor nodes. Secondly, we evaluate the stack with focus on reliability, memory usage and time slotting. The WirelessHART specification specifies the statistical communication reliability but no scientific measurements have been published.

1.5 Thesis Structure

The remainder of this thesis has been structured as follows. In Chapter 2, we will introduce wireless sensor networks and some of the specific problems encountered in this domain. After this we describe the Contiki operating
1.5. *THESIS STRUCTURE*

The chapter finishes with introductions to some relevant industrial automation protocols. The design and implementation is considered in Chapter 3 and the evaluation in Chapter 4. Chapter 5 will discuss related work and in Chapter 6 the thesis is concluded.
Chapter 2

Background

This chapter introduces wireless sensor networks and communication protocols that are used in industrial process automation.

2.1 Wireless Sensor Networks

A wireless sensor network is a network consisting of sensor nodes, also known as motes. The nodes communicate by transmitting information over a wireless medium. In our case a sensor node is a small embedded device with little computational power, very limited memory, one or more sensing devices and a radio transceiver. There is a wide range of sensors available that are used to gather data about the physical conditions in their surrounding environment, e.g. temperature, humidity, movement, pressure and noise level.

Nodes are equipped with the minimum required hardware in order to keep the price and the power consumption as low as possible. More powerful hardware costs more and with a network consisting of up to thousands of nodes this higher price can make a significant difference. It also consumes more power from the power source, which often is limited e.g. a battery. This in turn can lead to a higher maintenance cost as the power source needs to be replaced more often. Sensor nodes might be deployed in hostile and remote environments. In these conditions changing batteries is a painstaking process or at worst not even possible. Traditional networks try to achieve high quality of service (QoS) but in sensor networks the focus lies on minimizing the power consumption.

Normally a WSN is considered to be a wireless ad-hoc network, meaning that each node in the network has the ability to route packets towards their final destination. The decision on whom to forward the packet to is made dynamically and is based on the information about physical connections between nodes. Figure 2.1 shows a typical architecture of a multi-hop wireless sensor network.

2.2 The Contiki Operating System

Contiki is an event-driven operating system for embedded devices [15]. The core components of Contiki include an event-driven kernel, communication stacks and protothreads [17], a stackless thread-like abstraction. On top of the kernel is where the applications are running. They can be dynamically loaded and
unloaded. This makes it possible to update the software remotely on devices still deployed in the field. The updated version can be sent compressed. At arrival it is decompressed and loaded. Sending compressed updates reduces the amount of data that has to be sent but it increases the processing required at arrival, i.e. decompression. Since radio communication is more power consuming than CPU processing [24] it can be more power efficient to send updates compressed, as shown by Tsiftes et al. [33].

As previously noted, the kernel is event-driven which is common for memory-constrained embedded devices. A multi-threaded system uses one stack per thread whilst in an event-driven system there is no need to fragment the memory which leads to lower memory requirements. For this reason Contiki [15], TinyOS [21] and SOS [20] are all event-driven systems.

Event-driven programs tend to grow into complex state machines. To simplify the development efforts Contiki offers a thread-like abstraction called Protothreads. Protothreads provide linear execution for event-driven systems. Context switching is done by stack rewinding. Stack rewinding is memory efficient since each thread does not need to have a separate stack. The overhead for running protothreads is between two and twelve bytes, depending on the architecture [16].

Contiki has three communication stacks and they are Rime [14], uIP [13] and uIPv6 [18].

uIP was designed to have a minimal set of features that is required for a full TCP/IP stack. It is compliant with the requirements specified in RFC 1122 [8]. The amount of memory used by uIP depends on the hardware configuration it is running on. It also sets the boundaries for the amount of traffic and the number of simultaneous connections. According to Dunkels it is possible to run the uIP implementation with only 200 bytes of RAM [12].

uIPv6 is the smallest IPv6 Ready [3] stack available and it is designed for low-cost networked devices (e.g. sensors and actuators). It does not depend on any particular MAC or data link layer. The interface between the uIPv6 and the lower layers enables easy integration with many different MAC and data link layer protocols. The interface to the application layer is the same as uIP offers.

Rime is a layered communication stack for sensor networks. The idea behind the layered design is to reduce implementation complexity for communication.

Figure 2.1: Typical architecture of a multihop wireless sensor network [34]
2.3. **INDUSTRIAL PROCESS AUTOMATION PROTOCOLS**

The layers are incremental in the sense that the more complex abstractions use the more basic abstractions, as shown in Figure 2.2. The bottom layer in the stack implements the anonymous best effort broadcast abstraction. As you advance in the stack the abstractions become more reliable.

![Figure 2.2: Structure of the Rime stack, showing the more basic abstractions at the bottom](image)

### 2.3 Industrial Process Automation Protocols

Process automation involves using computer technology to monitor performance values and quality of outputs. This is done with the help of sensors that collect data on temperatures, pressure, vibration and so on. The collected data is analyzed and the settings for the individual parts of the production line can be adjusted to optimize the production.

Communication protocols for industrial automation have high demands on reliability and robustness. Moving from wired communication to wireless communication involves many advantages but also some disadvantages. Installing cables to every sensor is a tedious and expensive task and this is one of the biggest reasons why companies are looking into wireless solutions. This becomes evident from the increasing amount of wireless standards developed for this application area (two of these standards will be covered in Sections 2.3.2 and Section 2.3.5 respectively).

#### 2.3.1 HART

Highway Addressable Remote Transducer (HART) is a protocol for bi-directional communication between a host application and intelligent field instruments. Applications include remote process variable interrogation, parameter setting, and diagnostics [7].

Communication protocols are generally divided into layers corresponding to the layers of the Open Systems Interconnection Basic Reference Model (OSI Model). Figure 2.3 shows both wired and wireless HART divided into these layers.
CHAPTER 2. BACKGROUND

Figure 2.3: HART OSI 7-Layer Model

Physical Layer
The communication in HART is traditionally done over 4-20mA analog instrumentation wiring. The Bell 202 Frequency Shift Keying (FSK) standard is used to superimpose the encoded digital signals on top of the analog control signal. The digital values 0 and 1 are encoded by assigning them to different frequencies. Each individual byte is converted and transmitted as an 11 bit serial character.

Data Link Layer
The HART protocol operates in a master-slave fashion. All communication is initiated by the master. A master can either be a control station or an operating device. There can be a maximum of two masters, primary master — normally the control system — and secondary master — a handheld terminal or a laptop.

Slaves (e.g. HART field device) respond only to command messages from a master. After the completion of a transaction the master will wait for a fixed time before sending another command. This enables the other master to break in so that two masters can take turns communicating with the field devices.

HART devices support both long (5 byte) or short (1 byte) addresses enabling the HART protocol to support point-to-point and multi-dropped communication with field devices. When using short addresses up to 64 slave devices may be multi-dropped on one communication link. When using long addresses the number of multi-dropped devices is more or less unlimited.

HART supports burst mode transfer. This is when a slave device regularly broadcast data without continuous polling by a master device. There can be one and only one field device in burst mode on a communication link at a time.

Application Layer
The application layer is the topmost layer in the HART protocol. It uses device commands, responses, data types and status reporting to interface to the
underlying layers and the responding application layer(s). The application layer is responsible for extracting and acting upon the information in the message and generating a response back. The response includes the status to be reported back to the requesting application layer.

### 2.3.2 WirelessHART

WirelessHART is an extension to the HART protocol that adds the wireless medium alongside the 4-20mA analog instrumentation wiring. WirelessHART is an open communication standard especially designed to address the requirements of the process industry. Simplicity, reliability and secure wireless communication are parts of the requirements that it addresses.

As previously shown in the OSI model in Figure 2.3, the protocol stack of WirelessHART contains a physical layer, a data link layer, a network layer, a transport layer and an application layer.

**Physical Layer**

WirelessHART use radios based on the IEEE 802.15.4-2006 standard. It operates at 2.4GHz which is part of the Industrial, Scientific and Medical (ISM) bands. These bands are typically intended for unlicensed use and are used by devices ranging from microwave ovens to wireless LANs and cordless phones. So the probability of interference is relatively high.

**Data Link Layer**

To successfully be able to coexist in the previously mentioned environment, WirelessHART uses several mechanisms to minimize interference from other wireless systems. Time Division Multiple Access (TDMA) and channel hopping are used to control access to the shared medium. TDMA is a protocol that provides collision free, deterministic communication by the use of time slots. Time slots are pre-scheduled and have a fixed length (10ms). Precise time synchronization is crucial to the operation of the network. Time slots are synchronized between devices. WirelessHART uses the notion of superframes to group a fixed number of slots together.

Each node synchronizes its internal clock with a selection of its neighbors. The network manager assigns each node with a set of neighbors to use as clock references and the network manager is responsible to keep the references’ clocks accurate. When a node receives a packet, its arrival time is recorded and compared to the ideal time at which the packet should have arrived. The difference between the two times are sent in every ACK destined for the source node. When a packet is received from a sync neighbor, the network time is adjusted either based on the arrival time of the packet or the time difference in the ACK, depending on which device initiates the communication.

To increase reliability, TDMA is used in combination with both channel hopping and Carrier Sense Multiple Access (CSMA). CSMA is used to verify the absence of other traffic before transmitting on the medium. The combination of TDMA and channel hopping enforces that the communicating devices have to rendezvous in time and frequency, as shown in Figure 2.4. It is possible to
ban channels which is useful when you know that certain channels have high interference. This is called channel blacklisting.

Figure 2.4: Channel hopping

Network Layer and Transport Layer

The basic building blocks of a WirelessHART network are: (a) Field Devices are devices attached to the plant process, (b) Handhelds are portable WirelessHART devices used for maintenance tasks i.e. running diagnostics, performing calibration and configuring devices, (c) Gateways connect field devices and host applications and (d) The network manager is responsible for configuring the network, managing of routing tables and scheduling communication between devices. Figure 2.5 shows an example of a WirelessHART network. WirelessHART forms mesh networks with redundant paths between nodes so that packets can be routed around obstacles (e.g. a dead link). All devices in the network must be capable of routing packets. The protocol defines two mechanisms for routing (graph routing and source routing) that must be supported by all devices.

Figure 2.5: An example of a WirelessHART network
A Graph Route is a list of paths that connect nodes in the network. The paths in each graph are configured by the Network Manager and stored at each device. A device routing a packet looks up the graph ID (stored in the packets network header) and forwards it to one of the neighbors listed.

A Source Route is one path from the source device through the network to the destination device. The device-by-device route is specified in the packet and if one of the intermediate devices fails, the packet is lost.

Application Layer

The application layer in WirelessHART is similar to the one in HART. They differ in the set of commands they accept. Wired and wireless HART include a set of common commands, a set of commands that are specific for the underlying physical medium and an optional set of commands that are specific to the application. The applications are designed to solve different tasks but the general flow of work is the same: receive command, act upon it and generate a response to send back to the requester.

Implementation Challenges

The limitation on computation and memory are two of the biggest challenges when developing software for embedded devices. Implementing WirelessHART on such devices is a very challenging task. The timing requirements specified in the WirelessHART standard are very strict. A time slot is divided into several slices and in each of these a node can either be running or be idle. A task can be very time consuming, e.g. when verifying the Message Integrity Code (MIC) of a received data link layer packet\(^1\).

The requirements concerning the memory are equally strict as the requirements on the computation time for the processor. Defined in the specification are a collection of communication tables that each network device should maintain. There is a minimum requirement of the number of entries that have to be stored. If the storage space in memory is not sufficient, the tables could be stored in external memory, e.g. in flash memory. External storage has longer access time compared to the internal memory which is one of the major drawbacks with this approach.

2.3.3 ZigBee PRO

ZigBee is a standard suite of high level communication protocols for wireless networks defined by the ZigBee Alliance. The mission of the organization is to define reliable, cost-effective, low-power, wirelessly networked, monitoring and control products based on an open global standard [6].

In October 2007 the ZigBee Alliance announced ZigBee PRO. ZigBee PRO extends the original set of features in ZigBee with better support for larger networks.

The protocol stack defines the ZigBee Network Layer and the ZigBee Application Layer. They build upon the physical and Medium Access Control (MAC) layers from the IEEE 802.15.4 standard. On top of the application layer ZigBee defines application profiles (public, private and published). The profiles

\(^1\)The implementation of the data link layer is not part of this thesis
CHAPTER 2. BACKGROUND

consist of rules and regulations which the developers must follow. The purpose of the profiles is to ensure interoperability between vendors (e.g. a light switch from vendor X can turn on or off a light from vendor Y) [19].

Network Layer

The responsibility of the network layer is to ensure correct use of the MAC sublayer and provide a suitable interface to the application layer. It is responsible for the configuration of new devices and to establish new networks. There are three types of devices: ZigBee end device, ZigBee router and ZigBee Coordinator. The nodes in a network can be arranged in a star, tree or mesh topology.

Application Layer

The application layer is the highest layer defined by the specification and it is the interface between the ZigBee system and the users. Its main components are:

Application framework. Provides an environment in which applications execute.

ZigBee device object. Responsible for defining the role of the device.

Application support sublayer. The interface between the network layer and application layer.

Improvements Introduced in ZigBee PRO

An example of an improvement in ZigBee PRO is the assignment of node addresses. ZigBee originally used a “cluster-tree” routing algorithm where a coordinator node acts as the root of the network and the address tree and nodes are assigned addresses depending on its position relative to the coordinator. This allows for relatively simple routing algorithms but can cause problems in several situations. In large networks this could lead to address exhaustion on long branches. Changes to the topology of the tree could lead to re-addressing of large parts of the network. ZigBee PRO implements Stochastic Addressing that allows nodes to randomly pick an address when joining the network and in case of an address conflict (two nodes with identical addresses) the unique MAC address will be used to tell the two nodes apart.

ZigBee PRO has a function called Frequency Agility which means that the application can move the whole network to a better channel. When the network is formed the channel with the least noise and traffic will be used. If the quality of the channel degrades over time, the host application can scan for a better channel. If one is found, the application can move the whole network to this channel.

2.3.4 6LoWPAN

6LoWPAN is an acronym that stands for IPv6 over Low power Wireless Personal Area Networks. 6LoWPAN is the name of a working group at Internet Engineering Task Force (IETF). The goal of the group was to define
how to carry IP-based communication over IEEE 802.15.4 based networks while conforming to existing standards. This ensures interoperability with other IP-based solutions. The headers of a User Datagram Packet (UDP) running over IPv6 is 48 bytes (IPv6 is 40 bytes and UDP 8 bytes) which makes up to almost 50% of the Maximum Transmission Unit (MTU) for the 802.15.4 standard. Needless to say, without compression IPv6 is not a feasible alternative for WSNs. The solution proposed by the working group is an adaption layer that deals with header compression.

Adaption Layer

Between the MAC layer and the network layer 6LoWPAN inserts an adaption layer that allows carrying IPv6 packets in 802.15.4 frames. The 6LowPAN header contains only the necessary information and will be compressed if possible. The three key features of the adaption layer are:

- **Header compression**: Fields in the IPv6 header that can be derived from the 802.15.4 frame or by simple assumptions are eliminated.

- **Fragmentation**: Fragmentation into multiple frames to be able to accommodate IPv6 packets that are larger than the 802.15.4 MTU (significantly smaller than the IPv6 MTU).

- **Layer-two forwarding**: Transparently forward packets in the mesh network (which might not be fully connected) to the end of an IP hop.

The IPv6 datagrams running over IEEE 802.15.4 are prefixed by a header stack. Each header in the stack contains the type of the header followed by zero or more header fields. The header includes, in order, Mesh Addressing Header, Broadcast Header, Fragmentation Header, IPv6 Header and UDP Header. In order to set apart the different possible headers a dispatch value is used. The dispatch value defines the type of headers that follow and their ordering.

2.3.5 ISA100.11a

The task of the ISA100.11a project group is to define all specifications (including security and management) for wireless devices. The draft of the specification was rejected by the ISA100 committee members and is, at the time of writing, being revised [22].

Physical Layer

ISA100.11a adopts the IEEE 802.15.4-2006 standard as the physical layer. The main reasons for choosing a physical layer that operates in the ISM bands was that: (a) a licensed radio frequency band require custom hardware and (b) to secure a single worldwide frequency would have been out of the scope of the committee.

Data Link Layer

The typical target environment for which the protocol has been designed is an environment with relatively high interference. The interference can render one
or more channel completely unusable. This is part of the reason why industrial protocols usually go hand in hand with frequency hopping. In ISA100.11a three methods for frequency hopping are supported by the data link layer. These are slotted channel hopping, slow channel hopping and slotted/slow channel hopping which is a hybrid of the two previously mentioned methods. Slotted channel hopping, which uses one slot per hop, equals using TDMA, while slow channel hopping, which uses several slots (typically 10-25) per hop, corresponds to CSMA. Figure 2.6 shows the different frequency hopping methods.

![Frequency Hopping Methods](image)

**Figure 2.6: Types of frequency hopping supported by ISA100.11a**

**Upper Layers**

The upper layers of ISA100.11a are still under development. The main functionality of each layer is:

- **Network Layer.** Addressing, routing, quality of service and management functions.
- **Transport Layer.** Flow control, end-to-end error recovery and management topics.
- **Application sub-Layer.** Support for wireless field devices, object-oriented modeling, protocol tunneling.
Chapter 3

Design and Implementation

A significant part of the work spent on this thesis has consisted of implementation of a WirelessHART stack in Contiki. In this chapter the design and implementation of the stack is to be described. First, an overview of the overall architecture is shown and then some specific challenges such as node joining and routing of packets are discussed.

Despite the fact that WirelessHART is an emerging standard, there are very few closed source and no open source implementations available. Song et al. [31] have developed a prototype implementation of the standard (described more in Section 5 Related Work).

Even though our implementation has not been designed for any particular hardware, the target hardware during implementation, testing and evaluation has been the Tmote Sky [10].

To be able to run WirelessHART on the type of constrained hardware that makes up a typical sensor node, the implementation has the following essential design goals:

**Low computational complexity.** WirelessHART has very strict timing requirements for the devices participating in the network. To be able to meet these requirements the computational complexity should be kept as low as possible.

**Low memory consumption.** Memory is a scarce resource and should be used efficiently. The WirelessHART stack maintains a set of communication tables for storing information about routing, end-to-end transactions and more. Also packet buffers are stored so in case a packet is lost it can be retransmitted. The stack should leave as much memory as possible for the running application.

### 3.1 Architecture

Implementing a protocol can be a very cumbersome task. To simplify this task the majority of protocols are designed in a layered fashion. Each of the layers solves a different part of the communication problem. This results in lowering the level of complexity per layer. Furthermore, each layer can be implemented independently from the others. Implementing layers completely isolated from
each other could cause a whole lot of communication overhead between the different layers and reduce performance [9]. To minimize these effects a cross-layer approach can be used. This means that certain information is shared between the layers so that they can benefit from the information in other layers.

In Section 3.1.1 and 3.1.2 we present two different design proposals and in Section 3.1.3 the final design is presented.

3.1.1 Events

Since Contiki supports multiple processes, a natural approach would be to implement each layer in a separate process and using events to communicate between the processes. Contiki supports two types of events: asynchronous and synchronous events. Asynchronous events are enqueued by the kernel and dispatched to the target process at a later time. A design using asynchronous events would not be feasible with the time constraints imposed by WirelessHART. An asynchronous event will be queued if the queue is not full and the time until it is dispatched to the target process is unknown. If the queue is full the event will be discarded.

Synchronous events on the other hand cause the target process to be scheduled immediately and when the event has finished processing, the control is returned to the posting process. Focusing on the timing requirements of the specification, synchronous events would be a possible solution in contrast to asynchronous events.

In this design, each process would be waiting for an event to occur, process it and then wait for the next event to occur. After processing an event, the control must be released back to the operating system so that other processes can execute. Since Contiki processes are not preemptable, this has to be done explicitly in each process by yielding. The event-based approach would be very flexible and each layer could easily be modified independently from the other. Additionally, it would also keep the clear distinction between the different layers.

The implementation of events in Contiki only supports one data argument. This constraint could be overcome by using structures in C to bundle the arguments together.

3.1.2 Callback Functions

An alternative solution to the one explained in the previous section would be to implement the layers in one process and using modules in C to separate the different layers, with the layers using function calls to communicate among them. Instead of forcing the application layer to name its receiving functions to predefined names, it will register a set of callback functions for communication between the stack and the application.

The drawback with this solution is that it is not very flexible. To replace a layer in the stack, the source code of the adjacent layers need to be edited and recompiled.

On the contrary of the event-based solution, this approach has no restrictions (in practice) on the number of arguments that can be used.
3.1.3 Final Design

The final design combines the benefits of the event-based solution and the callback-based solution. All the layers are implemented in the same process using function calls to communicate between the layers. The functions to call are defined by macros in a configuration file. This makes it easy to connect an application to any layer in the stack. The stack needs to be recompiled for the changes in the configuration file to take effect. The increased flexibility facilitates bypassing of layers and enables the application programmer to develop a custom layer in the stack. To transparently include the new layer in the stack, it is required that the API towards an old layer is compatible with the old API.

To enable the different layers to share information, we use packet buffers. They are derived from the buffers used in the Rime stack. The buffers can be seen as an abstraction layer towards the network interface. The network specific issues (such as byte order) are only handled when a packet is sent or received. This enables the layers to benefit from the information intended for other layers.

3.2 Code Modules

The stack is based on five core modules: whlink (data link layer), whnet (network layer), whtrans (transport layer), session (storage of communication tables) and packetpool (central storage of packets). The modules and their relation to each other are shown in Figure 3.1. Together the modules comprise of approximately 3500 lines of code. Each module will be covered more in detail below, except for the data link layer which is not part of this thesis.

![Diagram of core modules](image)

Figure 3.1: The core modules that make up the WirelessHART stack

3.2.1 Network Layer (whnet)

The whnet module implements the network layer in the WirelessHART stack. The basic functionality it performs is routing of packets and the process of joining nodes in a network. The joining of networks and the routing are covered more in Section 3.5 and Section 3.6 respectively. The most important functions are covered below.
CHAPTER 3. DESIGN AND IMPLEMENTATION

Send Packet
Before a packet is sent, the correct session has to be found so that routing information can be located and set in the packet (graph route and, if requested from the application, source route). Finally, the data link layer addresses are set. The sender address is set to the node’s address and the receiver address to the address of the first node in the source route or one of the nodes in the graph route (depending on if source route is available).

Report Status of Sent Packet
The status of a sent packet is forwarded from the data link layer to the network layer. If the packet was a routing packet then it will be released. Otherwise the confirmation will be forwarded to the transport layer.

Receive Packet
When a packet is received from the data link layer (except for advertisement packets) it is either for the device or for routing. If the packet is addressed to the node it is forwarded to the transport layer. Otherwise it is sent back to the data link layer for routing towards its final destination. The routing process will be covered more in detail in Section 3.6.

Receive Advertisement Packet
When an advertisement packet is received from the data link layer, it will be sent to the join handler. The join handler takes care of the network joining, this will be covered in more detail in Section 3.5.

Send Response Packet
When a response packet is received from the transport layer, the network layer addresses in the packet buffer are swapped so the sender becomes the receiver and vice versa. Additionally, the routing information for the node is found through the session and set in the packet. Finally the data link layer addresses are set, the sender address is set to the node’s address and the receiver address is set to one of the nodes included in the graph.

3.2.2 Transport Layer (whtrans)
The whtrans module implements the transport layer. The transport layer’s main functionality is to provide reliable end-to-end transactions.

Send Packet
Before sending a packet, it has to be inspected to see if the packet type is acknowledged or not. If it is not acknowledged, then a sequence number is added to the transport layer header and the packet is sent off to the network layer. Otherwise if the packet is acknowledged the correct entry in the transport table should be found. The sequence number in the transport table is incremented by one and set in the transport layer header and the packet is stored in the transport table so that it can be retransmitted if lost. A flag in the transport
table is set indicating that a transaction is in progress. Before sending the packet a timer is started and if a response arrives it is stopped. If the timer expires the packet is considered to be lost and it is retransmitted.

**Report Status of Sent Packet**

The status of a sent packet is forwarded from the network layer to the transport layer. If the status reported is an error then the packet will be retransmitted.

**Receive Packet**

When the network layer receives a packet addressed to the device it is forwarded to the transport layer.

If it is an acknowledged request packet the transport table entry for the packet must be found. If the entry has been previously unused, then the sequence number in the table entry should be set to one less than the sequence number in the packet. Otherwise the sequence number should be inspected and if the sequence numbers are equal then the last packet should be sent again. If the sequence number is not one more than the sequence number in the transport table then the packet is out of order and should be discarded. Otherwise a response packet is acquired from the pool of packets and the attributes from the request packet are copied to the newly acquired response packet. The data in the packet is extracted and the commands are sent to their intended receiver (either the application or the stack). After finishing handling the request packet, the response packet is sent off to the network layer, if it contains any data.

If the packet is an acknowledged response packet, then the transport table entry should be found and if the sequence number in the packet and the sequence number in the transport table entry are equal, then the packet completes the transaction and the packet stored for retransmitting can be released and the active bit in the transport table entry can be reset. The data in the packet is extracted and the commands are distributed to either the stack or the application.

**Send Response Packet**

When a response for a request packet should be sent, the correct transport table entry should be located. It is inspected so that the sequence number in the packet is correct, otherwise the packet is discarded. The packet’s sequence number is incremented by one, the transport type is switched from a request to a response type. The packet itself is stored in the transport table and then the packet is passed to the network layer. If a packet is lost, the packet in the transport table is used to retransmit the packet.

3.2.3 Communication Table Storage (session)

The *session* module implements storage of the communication tables, see Section 3.4. It has functions for reading from and writing to the different communication tables. The communication tables are shared among the layers and the module was designed in such a way that the internal logic and representation should be hidden from the rest of the stack. This enables the implementation to change transparently, e.g. the module can be updated to use
external memory for storing the tables and only keep the most recent or the
most frequently used, cached in memory.

3.2.4 Pool of Packets (packetpool)

The `packetpool` module implements a pool that allocates and handles all
packets. To get a packet, one has to acquire it from the `packetpool` and when
the packet is not needed anymore it should be released back to the pool. The
design used is similar to the one for the `session` module, namely to hide the
internals from the rest of the stack.

3.3 Handling of Commands

The data in HART and also in WirelessHART are sent as commands.
Commands are used for configuring the stack, collecting statistics from the
stack and sending application commands. The command header consists of a
command number and a length. The length is the number of bytes following
the header. In the implementation a command header is represented as a C
structure. A packet can contain multiple commands.

When a packet is received, it is inspected to ensure the packet is valid,
as previously explained. When the packet has been verified, the commands are
parsed one by one. A command is either intended for the stack or an application.
The commands are filtered based on the command number and forwarded to
the correct receiver. Commands intended for the stack are processed and if the
command requires a response, it is appended to the end of the response packet.
The commands not intended for the stack are forwarded to the application.

In the current implementation the filtering of the commands has been
statically defined in the stack and can not be changed dynamically. The
preferred way would be that each of the layers and applications register callback
functions that should be called when receiving a certain command. The cost of
such a system must be carefully considered to ensure if the extra functionality
is worth the extra memory usage.

3.4 Communication Tables

Each network device maintains a set of tables that control the communication
performed by the device. Figure 3.2 shows a subset of the tables and the
relationships between them. The information stored in the tables is used for
routing, secure transmissions and end-to-end acknowledged communication.
The session table is the central point in all communication and contains every
corresponding node.

In the implementation we represent the communication tables as arrays of
C structures. The size of the arrays can be defined in the configuration file and
if missing, a set of default sizes will be used. The minimum required sizes, as
stated by the specification, have been chosen as default.

The different tables and their main purposes are described below:

**Session table.** Security sessions ensure private, unmolested communication
between sender and receiver. Session table entries are used to establish a
3.4. COMMUNICATION TABLES

Figure 3.2: A subset of the communicating tables and the relationship between them.

- **Secure pipe between a master and a slave.**

- **Transport table.** Transport table entries are used for end-to-end acknowledged transactions and for the automatic retries of lost packets. A session typically contains two transport entries for each correspondent, one for unicast and one for broadcast.

- **Route table.** A security session is associated with the route table. It is used when selecting which graph to use for communication. Additionally, it also contains statistics, that is used by the network manager to optimize the communication.

- **Source Route table.** Contains the addresses for all the intermediate devices on the path to the final destination. The list of addresses can contain up to 8 entries.

- **Superframe table.** A superframe consists of a fixed number of slots. The table contains a list of links. A link refers to a neighbor that is allowed to communicate with the device.

- **Graph table.** Stores routing information that is used when sending a packet towards its final destination. A graph is a list of paths that connect two devices together. It is the network managers responsibility to configure the graphs correctly. A graph must contain an ID and a list of references to neighbors that are the next hop towards the destination.

- **Neighbor table.** A table of neighbors that is known by the device. It contains neighbors whose communication has been overheard and devices that share a link with this device. Entries include basic neighbor identity and statistics.
3.5 Joining a Network

A device joins a network first by obtaining access to the network and then becoming integrated into it.

Members of a network periodically send out advertisement packets so that a potential joining device can identify the network. The advertisement packets include sufficient information for the device to be able to synchronize with the network. The joining device and the network manager establish a communication channel. The encryption key used when joining a network is the so called Join key which is a predefined network specific key. The joining device presents its credentials to the network manager and if they are valid the device is allowed to join the network. The joining device receives a set of network security keys to use when communicating with the network manager. The network manager continues integrating the device in the network by providing it with superframes and links.

The network layer implementation of the join process is defined by a state machine that contains five states. In Contiki this was done using protothreads which are specially designed for simplifying programming state machines. The different states are:

Searching. At startup the sensor node will enter the searching state, waiting for reception of an advertise packet. Once an advertisement has been received, the network layer moves to the Got an Advertising Neighbor state.

Got an Advertising Neighbor. When entering the state, a timer is started and then the device continues to wait for more advertise packets. When the desired number of packets has been received or the timer expires, it moves to the Request Admission state.

Requesting Admission. Upon entering the state, a join request is sent to the network manager and a timer is started. A join request includes the device’s identity, its long tag, its join key and a list of neighbors detected by the device. If a packet containing network key, network manager session and nickname is received, then the device moves to the Loosely Coupled state. Otherwise, if the timer expires a new join request is sent. Join requests will be retransmitted a configurable amount of times and after that the join process will exit with an error.

Loosely Coupled. The device stay in the Loosely Coupled state until it has received a normal frame and links. At this point the device moves to the Operational state.

Operational. When in the Operational state, the device has successfully joined the network and is now fully integrated into it.

3.6 Routing Packets

When an incoming frame is received by the link layer the destination address in the header has to be either the broadcast address or the device’s address. If

\footnote{Our implementation exchange security keys but do not yet encrypt the messages.}
addressed to any other device, the packet is discarded. From the link layer the packet is sent to the network layer. The network layer routes the packets to the correct destination: the link layer, a joining device or the application layer.

If the device is the packet’s final destination then the transport layer is invoked. The transport layer parses the commands included in the payload. If the command was not intended for the application layer, then it will be processed by the stack. Otherwise the application layers command handler is called with the command.

If the destination address is the broadcast address then the same actions as if the device had been the packets final destination will be taken. In addition if the packet results in a broadcast response, it should be delayed by a random back-off time to minimize the flooding of responses.

A packet for forwarding must first be examined to see if the time to live has exceeded and if so the packet should be discarded. If the packet contains a proxy address that matches the address of the device, then the packet is for a joining node and it is the device’s responsibility to forward this message to the joining node.

If the destination address is a neighbor then the packet should be routed directly to the device.

Otherwise, the packet should be routed onward to its final destination. Either based on the graph ID or the source-route.

3.7 Packet Buffers

Packet buffers are used to store the intermediate information. When a packet arrives it is unpacked and stored in the packet buffer. The packet buffer stores all the information together with packet attributes. They are used to identify the stored information. Before sending a packet the attributes are converted to the network representation and sent off. This makes the internals of the protocol simpler, since it does not need to care about things like network byte order and packet structures.

The implementation has to handle multiple packet buffers for two reasons: a) The data link layer will schedule the packet transmission in a later time slot and b) buffers are stored in the transport layer for retransmission of lost packets. The packet buffers are stored in the packet pool. A packet needs to be acquired from the pool and then released back to the pool when no longer needed. The layer that acquires the packet buffer is responsible for releasing it. If the buffer is passed to another layer, the new layer will overtake the responsibility for releasing the packet.

3.8 Remaining Work

The current implementation of the WirelessHART stack is missing some parts before it is complete.

Integrating with the data link layer. During development two basic data link layers have been used, one with and one without time slotting. The link layers both have the same Application Programming Interface (API)
as the real data link layer in WirelessHART and should be fairly easy to integrate with the rest of the stack.

**Security layer.** The security layer ensures private communication by encrypting the messages. The security layer has not been implemented but this addition should be transparent to the rest of stack.

**Management services.** Allowing the application layer to configure the network layer and access the statistics that it has stored.
Chapter 4

Evaluation

In this chapter we first introduce the sensor nodes that were used during the experiments. Furthermore, the reliability, memory footprint and time slotting is evaluated.

4.1 TMote Sky

The TMote Sky [28], from Moteiv, is a low-power wireless sensor node. It features an MSP430 microcontroller from Texas Instruments with 10 kilobytes of RAM, 48 kilobytes of internal and 1 megabyte of external flash memory. This 16-bit RISC processor is a low-power unit that features extremely low current consumption both in active and sleep modes. This allows for extended battery life. For radio communication it uses the IEEE 802.15.4 compliant Chipcon CC2420 Wireless Transceiver. It has parts of the MAC layer functionality implemented in hardware (e.g. AES-128 encryption).

4.2 Reliability

The reliability experiments have been performed both in a simulated environment and in hardware. The purpose has been to show how different link qualities affect the communication.

In the experiments the data link in WirelessHART was replaced with a basic data link layer. The basic data link layer was not time-synchronized nor did it use frequency hopping. The purpose of the real data link layer in WirelessHART is to supply the upper layers with as little interference and packet collisions as possible. This functionality was not desired in our experiments since we wanted to study the behavior of the protocol when exposed to interference. When using channel hopping a new channel will most likely be used to retransmit the packet. In this experiment we demonstrate the case when all 16 channels are uniformly interfered and thus switching channels does not affect the link quality.

In the tests we exposed the implementation to different rates of packet loss. The probability $P$ that the originating node received an acknowledgment is shown in Equation 4.1.

$$P = P_x^{2^{(n-1)}}$$ (4.1)
Where $P_x$ is the probability that a packet was successfully received by node X and $n$ is the number of nodes that the packet pass through on the way including the sender and the receiver. The probability $P$ can be altered either by changing the number of nodes ($n$) or the probability that a node successfully receives a packet ($P_x$). For convenience we chose to fix the number of nodes and change the probability.

During the tests we had three nodes N1, N2 and N3 connected as depicted by Figure 4.1. Node N1 was the master, node N3 the slave and node N2 an intermediate node routing the packets from N1 to N3 or vice versa. When node N1 received a response from node N3 then the transaction was said to be successful. If node N1 did not receive a packet within 30 seconds the last packet was retransmitted. A packet was retransmitted up to 5 times and after that the packet was considered to be lost.

\[ N_1 \rightarrow N_2 \rightarrow N_3 \]

Figure 4.1: Node overview

4.2.1 Simulated Experiments

The simulated experiments were performed in the Cooja (COnRiki Os JAvA) simulator [26, 27]. The simulation is a controlled environment where everything from packet loss to transmission ranges can be configured.

In the experiments we changed the link quality in Cooja. Twenty different link qualities were used and for each link quality 1100 packets were sent and the number of received packets were counted. Figure 4.2 show the results received during the experiments.

\[ \text{Transaction success rate} \]

\[ \text{Link quality} \]

Figure 4.2: Link qualities relation to transaction success rate in simulation.

The results shows that the implementation is very reliable. It also shows that when the link quality is low changes in quality have big impact on the rate of successful transactions. This shows the importance of blacklisting known bad
4.3 MEMORY FOOTPRINT

channels. This is especially important in environments where a channel can be rendered unusable because of interference (e.g. industrial environments).

Additionally the reliability has been measured to see how the protocol behaves with different link qualities. The results shows that the implementation is very reliable. When the link quality is low changes in quality have big impact on the rate of successful transactions. This indicates the importance to blacklist channels that are known to have high interference.

4.2.2 Experiments on Real Hardware

The hardware experiments were performed with three Tmote Skys. Received Signal Strength Indication (RSSI) was used to measure the link quality. RSSI is the standard way measuring the power present in a received radio signal. The drawback of using RSSI is that it does not take the noise level into account. To get round this problem we carried out the experiments in an empty office and the noise level can be assumed to be constant.

During the measurements node N1 and N3 where placed at a fixed position and varying the link quality by changing the distance of the intermediate node N2. At each distance 100 packets were sent and the number of received packets were counted. Figure 4.3 show the results of the measurements.

![Figure 4.3: Link qualities relation to transaction success rate running on real hardware](image)

The figure shows that a small difference in link quality has a major impact on the transaction success rate. Comparing the results from the simulated experiments and the hardware experiments indicates a similar behavior. Therefore the simulated experiments can be considered correct.

4.3 Memory Footprint

Due to the memory-constrained nature of the typical hardware running the stack, it is relevant to measure the memory footprint. The two important metrics are the consumed memory usage and the code size. The implementation was compiled with `msp430-gcc 3.2.3`. 
The memory footprint measurements were done with \texttt{msp430-size 2.16}. For the sake of completeness the memory footprint of the data link layer has been included in the results.

### 4.3.1 Memory Consumption

The memory usage is measured by summing up the data and bss$^1$ segments. First the memory usage is evaluated with a stack configured to follow the requirements in the specification. After that different non-compliant configurations are evaluated to see how much memory can be saved if the memory allocation is optimized for the network.

#### Minimum Memory Requirement

The amount of memory consumed by the communication tables and the number of packets are configurable at compile time. During the measurements the memory was allocated to the minimum required by the specification. Table 4.1 show the memory consumption of the different components of the stack.

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data link layer</td>
<td>86</td>
</tr>
<tr>
<td>Network layer</td>
<td>40</td>
</tr>
<tr>
<td>Transport layer</td>
<td>2</td>
</tr>
<tr>
<td>Session</td>
<td>2792</td>
</tr>
<tr>
<td>Packet pool</td>
<td>3574</td>
</tr>
<tr>
<td><strong>Total size</strong></td>
<td><strong>6494</strong></td>
</tr>
</tbody>
</table>

From the results we can see that the packet pool and the session collectively consume almost all of the used memory. This is natural since the packet pool and the session stores all the packets and communication tables.

The total memory used is 8649 bytes. This includes the Contiki OS, drivers and the WirelessHART stack. The consumed memory is less then the 10 kilobytes which is what is available on the Tmote Sky. Only about 14\% of the total memory are left to the application to use. An operating system, including communication stacks and drivers, should be memory efficient and leave as much memory as possible for the application. In this case there is little to do about the memory consumption if the information should be kept in memory. One alternative would be to use external storage to store the parts of the communication tables and only use the memory to cache communication table entries. In the following section an alternative is discussed that discards the minimum requirements from the specification to decrease the memory consumption.

#### Not WirelessHART Compliant

The allocated memory can be changed at compile time, as described above. This way the stack can be tailored for the network or the memory usage

\footnote{Data segment containing static variables}
4.3 MEMORY FOOTPRINT

can be reduced to fit it on an even more memory constrained hardware. The communication tables and the number of packets are the two biggest contributors to the amount of consumed memory. If running a smaller network than the, by the specification, minimum required network the memory usage can be reduced. This is done by tailoring the communication tables and the number of packet available in the pool. Table 4.2 show examples of how much memory that can be saved.

Table 4.2: Total memory used for different network configurations, in bytes

<table>
<thead>
<tr>
<th>Sessions</th>
<th>Routes</th>
<th>Graphs</th>
<th>Neighbors</th>
<th>Packets</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>10</td>
<td>6573</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>24</td>
<td>24</td>
<td>12</td>
<td>7265</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>28</td>
<td>28</td>
<td>14</td>
<td>7957</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>8649</td>
</tr>
</tbody>
</table>

Typically each device has at least four sessions (one unicast and one broadcast with the network manager and the same with the gateway) set up when joining the network. The measurements have been carried out with five to eight sessions (minimum requirement) while keeping the proportions between the tables from the specification.

As seen by the results a significant amount of memory can be saved by optimizing the stack for the network. In best case the available memory for the application can be increased from 1.4 kilobytes to 3.4 kilobytes. From the memories perspective this means that even more memory constrained devices can run the communication stack. But it will not be compliant with the specification.

4.3.2 Code Size

The code size reported is the text segment of the object file that contains executable instructions. Table 4.3 show the code size of the different components of the stack.

Table 4.3: Code size of different components in the stack, in bytes

<table>
<thead>
<tr>
<th>Component</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data link layer</td>
<td>2474</td>
</tr>
<tr>
<td>Network layer</td>
<td>3424</td>
</tr>
<tr>
<td>Transport layer</td>
<td>3868</td>
</tr>
<tr>
<td>Session</td>
<td>1926</td>
</tr>
<tr>
<td>Packet pool</td>
<td>184</td>
</tr>
<tr>
<td><strong>Total size</strong></td>
<td><strong>11876</strong></td>
</tr>
</tbody>
</table>

The code size for the different parts of the stack is fairly equal except for the packet pool, as shown by the results. A complete running system consumes 34 kilobytes. This includes Contiki, drivers for the Tmote Sky and the WirelessHART stack. This leaves 14 kilobytes for the application programmer, which is approximately 29% of the internal flash.
4.3.3 Comparing Memory Footprint

We use uIPv6 for comparing the memory footprint with our implementation. Table 4.4 show the memory footprint for the two communication stacks.

Table 4.4: Memory usage and code size of the WirelessHART and uIPv6 [18], in bytes

<table>
<thead>
<tr>
<th>Communication stack</th>
<th>Memory usage</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>WirelessHART</td>
<td>6494</td>
<td>11876</td>
</tr>
<tr>
<td>Non-compliant WirelessHART</td>
<td>4418</td>
<td>11876</td>
</tr>
<tr>
<td>uIPv6</td>
<td>1748</td>
<td>11488</td>
</tr>
</tbody>
</table>

As shown by the results the difference in code size for the stacks are insignificant. The memory on the other hand is significantly higher in WirelessHART than in uIPv6. The memory usage for a stack with the, by the specification, minimum required set of communication tables and packet buffers are almost a factor of four. If not complying with the specification the memory usage decrease and is only a factor of three more than uIPv6. Not being compliant with the specification reduces the memory usage by 25%.

4.4 Time Slotted Experiment

The time slotted experiment used a data link layer with time slots. The times only synchronized at initiation of the network but not during execution. This experiment was not run on hardware since we expect the clocks to eventually become out of sync. This is because of the clock drift in the hardware. Running the experiments in COOJA would not suffer from the same drawback since COOJA has no clock drift.

The experiment consists of five nodes as depicted by Figure 4.4. In our experiment we sent a packet from N1 to N5 and then a response back from N5 to N1. The time elapsed from when the packet was sent by node N1 until the response arrives back at node N1 was recorded. The data link layer does not yet use predefined schedules instead when receiving a packet it schedules the response, if any, in the next available slot. In our case, without packet loss, this would give us a schedule that would resemble one where the nodes where scheduled in numerical sequence from N1 to N5 and then back to N1.

In Figure 4.4 we can see that the distance from N1 to N5 and then back to N1 is 8 hops. Each time slot is 10ms long. Our measurements show that sending a packet takes 79ms which is equivalent to 8 time slots. This also shows that the implementation is able to schedule a packet in the next time slot after reception. A packet received in time slot X can be scheduled for sending in
time slot $X + 1$. This indicates that running WirelessHART on a CPU running at 2.46 MHz is feasible.

4.5 Future Experiments

In this thesis two different data link layers have been used. One very basic link layer without time slots and frequency hopping and one time slotted data link layer without support for clock synchronization and frequency hopping. With a fully implemented data link layer there are more experiments that can be done.

One interesting experiment to be done is to see how blacklisting of channels affects the overall network performance. How noisy does an environment have to be before gaining anything from blacklisting a channel? Is it better for the network manager to send out new communication schedules that avoid using the interfered channel and use the better channels more? Can the network manager do blacklisting in a smart way so that bad channels can be blacklisted temporarily?

Experiments on throughput, energy estimation for sending and receiving packets have not been carried out in this thesis. This is because of those experiments are very dependent on the schedule that are used in the data link layer. Instead of testing the performance of the stack they test the performance of the schedule used.
Chapter 5

Related Work

This chapter introduces related work performed in the area of WirelessHART. There are very little work directly relating to the development of a WirelessHART stack. The prototype implementation by Song et al. is the only work that is known. We also introduce work done on protocol development in Contiki.

The central building blocks of the WirelessHART standard has been described by Kim et al. [23] including parts covering the MAC and network layers. Furthermore the authors have carried out a literature study to identify the existing algorithms and methods for solving typical problems that are common when implementing the standard. The authors also suggest changes that can be implemented within the existing scope of WirelessHART to improve the network performance.

A prototype implementation of a WirelessHART stack has been developed by Song et al. [31]. The authors base their implementation on a simple IEEE 802.15.4 physical layer library in ANSI C that was provided by Freescale. They also discuss some implementation issues that appeared during the development and possible solutions. The implementation is demonstrated using the Freescale 1321xEvk toolkit in a network running one gateway and two devices. In contrast to their work my thesis includes an implementation of the stack running on top of Contiki. Additionally my thesis includes an evaluation of the implemented stack with regards to reliability, memory usage and time slotting.

Soldati et al. [30] have developed a mathematical programming framework for joint routing and link scheduling of deadline-constrained transmission and data evacuation in WirelessHART networks. They demonstrate that the real-time scheduling problem in WirelessHART is different from transmission scheduling for multi-hop wireless networks and therefore existing results are not applicable. A scheduling algorithm is proposed together with proof that it minimizes evacuation time with the least number of channels.

ZigBee and WirelessHART have been compared with regards to robustness, co-existence, power consumption and security by Lennvall, Svensson and Hekland [25]. The authors participated as working group members in the WirelessHART standard committee on behalf of ABB and hold extensive experience with ZigBee.

Suarez et al. [32] have implemented a ZigBee stack for Contiki, based on the Open-ZB stack [11]. Unlike traditional implementations the authors have
replaced the default ZigBee MAC protocol with the power-saving MAC protocol X-MAC. The results show that using X-MAC reduces the power-consumption by 90%, leading to increased battery life by a factor of ten.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

WirelessHART is growing in popularity. The protocol is very CPU and memory intensive. In this thesis we have implemented and evaluated the protocol with wireless sensors as our target platform. Wireless sensors are usually very memory constrained and run on as low CPU frequency as possible to be able to preserve power from the often limited power source.

Currently there is only one real implementation of the specification available that was developed at Dust Networks [1]. Our implementation is a first major step towards presenting a competitive alternative to the official stack.

Memory footprint is something that is very important when running software on the type of hardware used in this thesis. One has to be very careful with how and where the memory is consumed. It is especially important if the software is to be included in an operating system or used as a library cause then the software is not the primary application and just there for enabling an application to manage its work. In this thesis we show that the WirelessHART stack is able to run with less than 10 kilobyte of memory (including the Contiki OS). In the case of WirelessHART the memory is more important since the specification requires the minimum amount of information to be stored is quite high.

The WirelessHART specification specifies that the length of a time slot should be 10ms. This puts very high demands on the implementation of the whole communication stack and the running hardware. We show that our implementation is able to run with 10ms time slots with nodes running the CPU at 2.46 MHz.

Additionally the reliability has been measured to see how the protocol behaves with different link qualities. The results shows that the implementation is very reliable. When the link quality is low changes in quality have big impact on the rate of successful transactions. This indicates the importance to blacklist channels that are known to have high interference.
6.2 Future Work

In this thesis we have focused on the implementation of a functional WirelessHART stack. Work that is out of the scope of this thesis is the implementation of a network manager. The network manager forms the network, sets the communication schedules, establishes communication paths and monitors the network.

Another thing to consider is the use of flash memory to save the communication tables. The difficulty with this approach is the low performance of a read operation compared to the time slots in the protocol. This problem could be overcome by using a cache with the most recently or most frequently used entries.

Also to be fully compliant with the WirelessHART specification the missing security layer needs to be implemented. Some radio chips (e.g. Chipcon CC2420) support encryption and decryption in hardware. This could be utilized in the implementation of a security layer. We do not expect the implementation of the security layer to have any major impact on the memory consumption. The session module already allocates the communication tables needed by the security layer.
References


