Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

# Advances in Distributed Real-Time Sensor/Actuator Systems Operation

— Operating Systems, Communication, and Application Design Concepts —

Dissertation zur Erlangung des
naturwissenschaftlichen Doktorgrades
der Julius-Maximilians-Universität Würzburg

vorgelegt von


Marcel Baunach

aus
Distelhausen

Institut für Informatik
Lehrstuhl für Technische Informatik


Würzburg 2012

# Advances in Distributed Real-Time Sensor/Actuator Systems Operation

— Operating Systems, Communication, and Application Design Concepts —

Dissertation zur Erlangung des
naturwissenschaftlichen Doktorgrades
der Julius-Maximilians-Universität Würzburg

vorgelegt von

**Marcel Baunach**

aus
Distelhausen

Würzburg 2012

Eingereicht am: 18.06.2012
bei der Fakultät für Mathematik und Informatik

1. Gutachter: Prof. Dr. Reiner Kolla
2. Gutachter: Prof. Dr. Kay Römer
3. Gutachter: Prof. Dr. Sergio Montenegro

Tag der mündlichen Prüfung: 30.10.2012

# Lectori salutem

Der Einsatz digitaler Systeme sowie drahtloser Kommunikation innerhalb weitreichender und keineswegs auf die Erde beschränkter Netzwerke hat sich durch ihre explosionsartige Verbreitung während der letzten Jahrzehnte in den meisten von Menschen besiedelten Gebieten zu einem wesentlichen Bestandteil des täglichen Lebens entwickelt. Dies ist verglichen mit "früheren" Arten der Kommunikation nur verständlich und konsequent angesichts der enormen Reichweite, Geschwindigkeit und Kosteneffizienz. Und so tragen die vielfältigen Möglichkeiten ihrer teils bewussten und unmittelbaren, häufig aber auch kaum wahrgenommenen und sehr indirekten Nutzung inzwischen wesentlich zur Sicherung und (gefühlten) Steigerung unseres Lebensstandards bei. Ein Ende dieser Entwicklung ist aktuell nicht abzusehen, weshalb auch die Weiterentwicklung informationsverarbeitender Systeme sowie der Ausbau zugehöriger Infrastrukturen und Netze weiter voranschreiten wird, um immer mehr relevant gewordene, nützliche – aber auch weniger nützliche – Dienste zur Verfügung zu stellen. Stets und überall. Und um Information zu gewinnen. Daten, die dann – hoffentlich angemessen gefiltert und vorverarbeitet, in jedem Falle aber unverfälscht – den Nutzern wiederum zur Verfügung gestellt oder in deren Interesse eingesetzt werden. Ob dies in jedem Fall eine gesellschaftliche oder individuelle Bereicherung bedeutet bleibt kritisch abzuwarten und soll hier lediglich ins Bewusstsein des Lesers gerückt werden; aus wissenschaftlicher und technischer Sicht sind die in diesem Kontext zu behandelnden Fragestellungen jedoch ein schier unerschöpflicher Quell an Inspiration und Herausforderung.

Die vorliegende Arbeit wird sich mit einer ausgewählten Teildisziplin dieser digitalen Systeme und Kommunikationsnetze beschäftigen: Den "Drahtlosen Sensor/Aktuator Netzwerken" und ihren elementaren Bausteinen, den "Drahtlosen Sensor/Aktuator Knoten". Durchaus weitläufig und in mittleren bis großen Stückzahlen in einer zu observierenden Umgebung verteilt, werden diese eingebetteten Systeme eingesetzt, um verschiedenste Informationen aus der Umwelt zu extrahieren, zu verknüpfen, und entsprechende Reaktionen auf diverse Ereignisse einzuleiten bzw. selbst auszuführen. Ihre (angestrebte) Autonomie und Autarkie macht sie dabei sowohl zivil als auch militärisch hochgradig interessant. Nutzen, Sicherheit, Zuverlässigkeit und Performanz müssen also gewährleistet sein!

Hinsichtlich der Zuverlässigkeit und Performanz – und nur diese beiden Aspekte werden wir im Rahmen dieser Arbeit behandeln – verlangt dies nach einem tiefgreifenden technologischen Verständnis bei der Entwicklung von Hardware, Software, und Kommunikationsstrategien. Unter besonderer Berücksichtigung von Reaktivität und Ressourceneffizienz ist ein angemessenes Co-Design im Kontext der jeweiligen Anwendung unerlässlich, um trotz der verhältnismäßig geringen Performanz und der eingeschränkten Energievorräte typischer Knoten komplexe Verfahren lokal implementieren zu können und übergreifende Aufgaben kooperativ bzw. kollaborativ zu bewältigen – auch in hochgradig dynamischen Umgebungen!

Am Beispiel des im Rahmen dieser Arbeit entstandenen Lokalisationssystems SNoW Bat werden diesbezüglich einige neue Strategien, Methoden und Paradigmen vorgestellt: Gekapselt in das von Grund auf neu entwickelte Echtzeitbetriebssystem *Smart*OS, das zusammen mit diversen Erweiterungen als Schwerpunkt dieser Arbeit gelten darf, werden wir einigen auch nach Jahren der Forschung noch immer auftretenden Problemen beim Design kompositioneller Software mit neuartigen Konzepten etwa zur dynamischen Ressourcenverwaltung begegnen und ereignisgesteuerten Echtzeit-Applikationen im Bereich der drahtlosen Sensor/Aktuator Netze ein bisher unerreichtes Zeitbewusstsein ermöglichen.

Knotenübergreifend werden wir die Reaktivität und Leistungsfähigkeit dieses Systems anhand speziell aufeinander abgestimmten Algorithmen zur Distanzmessung und Positionsbestimmung sowie eines ebenfalls neu entwickelten Datenaggregationsprotokolls demonstrieren. Letzteres wird im Rahmen dieses Co-Designs auch zeigen, dass die semantische Nutzung implizit verfügbarer Information – sowohl aus der Umwelt als auch aus dem verteilten System selbst stammend – zur Optimierung des Datendurchsatzes und zur Erhöhung der Informationsdichte bei der Sensordatenfusion genutzt werden kann. Umfangreiche Tests aller vorgestellten Techniken, Konzepte und Paradigmen fanden unter realistischen Bedingungen auf SNoW[5] Sensorknoten statt.

# Acknowledgments

This thesis emerged from my work and research efforts at the Institute of Computer Science at the Julius-Maximilians-University of Würzburg, where I already studied computer science and physics, and received my diploma degree. Looking back, I consider the time in Würzburg as an enormous benefit. The countless discussions of theoretical and practical aspects of science aroused in me not only the enthusiasm in the search for solutions, but also the urge to seek new challenges.

I therefore would like to thank those people who have contributed with their patience and support to the success of this work:

My advisor *Prof. Dr. Reiner Kolla*, for the opportunity to work at the Chair for Computer Engineering V as research assistant and Ph.D. candidate.

My second and third reviewers *Prof. Dr. Kay Römer* and *Prof. Dr. Sergio Montenegro*, and *the members of board for the defense of the dissertation*, for the time and efforts in evaluating my work.

My parents *Josef Baunach* and *Ingrid Baunach*, for their great support throughout my entire life.

My proof-readers *Clemens Mühlberger* and *Christian Appold*, for plenty of constructive criticism, and for the applied time and care.

My research colleagues and comrades-in-battle *Clemens Mühlberger, Christian Appold, Jürgen Bregenzer*, and *Florian Mayer*, for many fruitful discussions, for sharing ideas, and for personal advice.

My dearest friends[1] *Alexander Eckert, Roland Mallok, Clemens Mühlberger*, and *Dieter Ziegler*, for their invaluable personal advice. And simply for being there – in particular during the exceptionally difficult times in 2011. Thank you once more.

This work would not have been possible without you!

[1] in alphabetical order

**Layout Information**
These tools have been used for the text:

Typesetting: LaTeX (MiKTeX 2.7)

Font: Fourier (based on Adobe Utopia)

Editing: TeXnicCenter 2.0

Graphics: Corel Draw X3

Graphs: gnuplot 4.4

**Technical Information**
This equipment has been used for all test beds:

Sensor node platform: SNoW[5]

Embedded C compiler: mspgcc 3.2.3

Timing measurements: Tektronix TDS3034B

## Declaration

I, Marcel Baunach, hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Distelhausen, June 18, 2012

# About this Work

«Per aspera ad astra.»

---

*(Seneca,*
*Roman philosopher)*

# Organization and Classification

## 1. Background and Evolution

Before heading to the actual work and its various theoretical and practical parts, I'll give a short retrospective on the evolution of the research area "Wireless Sensor/Actuator Networks" at the Chair for Computer Engineering V at the University of Würzburg.

The "SNoW Project" originated from a practical course in hardware design which I supervised during some exceptionally snowy winter months. Originally coming from the hardware synthesis and FPGA/VHDL area, we decided to take a short trip to off-the-shelf microcontroller systems. Especially interested in the digital circuit layout and in general embedded systems design, we asked the participating students to implement a simple radio communication stack for a newly developed wireless sensor system: The *Sensor Node of Würzburg* (SNoW[5]). Initially, the software for the first prototype of this platform had to be implemented using plain C, before the first version of our operating system YaOS – which later turned into *Smart*OS – introduced the "convenience" of preemptive multitasking. Virtually at the same time, the first successful radio transmission took place. Many ideas which emerged during the practical course made their way into further projects, and the results finally attracted considerable attention of students and other researchers within the institute. In fact, plenty of positive feedback and the increasing number of interested students encouraged us to start an entirely new research focus at our institute: *Wireless Sensor Networks* (WSN). Supported by my advisor, I and two Ph.D. colleagues at the research group intensified our work in this area. Since then, we supervised about 100 practical courses, student research projects, and diploma thesis. Needless to say, the intended short trip became a long journey – which has not even ended yet. In fact, we traveled a long and exciting way with many intermediate stops, ranging from e.g. hardware prototyping and operating systems design, over theoretical and practical issues in (wireless) communication and localization, to technical and physical problems in distributed real-world deployments. Finally,

> *"Wireless sensor networking is a decathlon of information sciences."*
>
> *(Reiner Kolla)*

proofed to be a quite precise characterization of this research area. It involves numerous well-known areas like hardware/software (co-)design, energy awareness, remote-management and security issues, networking and distributed algorithms, self-organization and scalability, information aggregation and processing, and real-time operation in dynamic environments – just to name a few. In addition, it touches countless application scenarios in long established areas like transportation (e.g. logistics and goods monitoring), industrial and home automation (e.g.

robotics and security systems), environmental surveillance (e.g. object detection, localization and tracking), healthcare (e.g. patient monitoring and treatment), and cyberphysical systems in general.

In the meantime we extended our focus to *Wireless Sensor/Actuator Networks* (WSAN), since we believe in the proactivity of this ubiquitous and still evolving technology. On the software side, YaOS was completely revised, and turned into the full grown operating system *Smart*OS for time-critical embedded systems. Besides the support for three entirely different CPU architectures[2], various novel kernel concepts have been developed and implemented. The 16 bit sensor node SNoW$^5$ reached its third revision, and several extensions for sensors, actuators, communication, and power supply units became available. Besides this energy efficient low performance node, the 32 bit SuperG platform [210] was developed as high performance infrastructure element and wireless communication gateway with six simultaneously operating radio transceivers. Today, more than sixty SNoW$^5$ nodes and eight SuperG platforms are available within our hardware pool. These, and the availability of an adequate remote-management system (SNoW Ghost), allows the realization of quite complex projects – like e.g. the indoor localization system SNoW Bat. As expected, their real-world implementation repeatedly showed the enormous diversity of the involved research areas, and demanded for numerous solutions for a multitude of different and often unpredictable problems. A selection of the most interesting challenges and approaches will be discussed within this work. As my research colleagues would certainly agree, the most central lesson learned can be summarized as:

*It is a challenge to make the complex appear simple.*

## 2. Research Approach and Methodology

Since the work on wireless sensor/actuator networks meant the beginning of an entirely new research area at our institute, appropriate basics had to be created first. With the SNoW Project a practical framework was initiated during this work, which would allow to study various aspects of embedded systems on specific hardware and under real-world conditions. Therefore, both the sensor node SNoW$^5$ (→ Chapter 2) and the operating system *Smart*OS (→ Chapter 4) were designed in the first place and implemented from scratch to gain a complete and accurate understanding of all hardware and software issues, and to amply comprehend the related processes in detail. The aspired goal was to get a solid, yet completely transparent basis for scientific experiments and the evaluation of novel concepts. In particular, the specific requirements of typical components within the WSAN domain (e.g. sensor nodes and infrastructure elements), as well as their interaction with the environment and with traditional computers systems and networks, could therefore always be completely taken into account. However, when making such demands, a scientist in a theoretic working environment will find himself exposed to two self-evident and inevitable questions:

[2] Atmel ATmega128 (8 Bit), Texas Instruments MSP430 (16 Bit, *Smart*OS reference implementation), Renesas SuperH SH-2A (32 Bit)

1. What is the advantage of putting efforts into concrete implementations, compared to limiting oneself to analytic approaches followed by simulative studies when indicated?

2. Is it reasonable to take the risk of re-inventing the wheel, and subsequently to conduct research concerning only a small part of the wagon built on it?

First, analytic considerations must always be undertaken – independent from the research approach. They are not only required for comparing related work, but also to ensure the quality, viability and sustainability of novel concepts by appropriate proofs. But how can we evaluate the impact of our research on existing problems, which we are expected to solve successively?

In this concern, the advantages of a practical, though potentially complex approach with several side effects are manifold and invaluable at the same time: Since the developed systems, and the techniques implemented therein, have to prove their capabilities and reliability under realistic conditions and arbitrary environmental influences, this will not only reveal the true effects of underestimated or even entirely hidden problems for improving the own work, but it is also an almost endless source of inspiration for new ideas. There is always something one would not have expected, and the attempt to transfer specific theories to the real-world will in general create a diversified insight into more or less related areas. In fact, many errors and preventable quality degradations in hardware and software systems arise from the fact, that researchers and developers (at times) do not know exactly what they are doing during the design and implementation stage, or at least do not assess the consequences sufficiently far-sighted. If the complexity *is* manageable, this problem can be attenuated significantly by the exclusive use of self-made components, since these are commonly better understood and much easier to adapt or extend by new features. For the development of compositional time and safety critical systems in particular, such an approach proved to be quite adequate, since errors and unexpected behavior emerging from the use of "black-boxes" at lower system levels (e.g. the hardware/software interfaces) are hard to trace, locate, solve or compensate for. Finally, successfully tested systems can more easily be transferred to target applications. A significant disadvantage of practical approaches, however, is their restricted flexibility regarding the intentional modification of environmental properties and system abstractions. Once implemented or installed, only few scenarios can be observed under hardly influenceable conditions. This complicates both the comparison to other systems and the reproduction of results, since system states and environmental conditions are difficult to configure and hardly conservable or restorable in detail.

Thus, simulation is often preferred as an alternative to real hardware for the implementation of techniques, methods and theories under examination. Besides, simulation offers some more advantages regarding the problems just described: Especially for systems with a large parameter and state space regarding the software, hardware, *and* the environment, simulation commonly offers much more flexibility for analyzing more configurations in significantly less time. Furthermore, one is not constrained by the given specifications of concrete hardware or environments, but it is much easier to apply certain abstractions to modify them and to "focus on the essential". A potential fallacy(!), since it is exactly this convenience which sometimes leads

to a central and well-known problem: Though abstraction may produce easy to handle models, it can rapidly lead to the masking of relevant facts and problems, and the relation to reality may deteriorate unnoticed. In addition, errors or imprecision in the models may not become apparent in the simulation results, and thus lead to false – albeit plausible – assumptions. This is especially true for the modeling of various physical and temporal phenomena as e.g. discussed for the SNoW Bat localization system in Part III of this work. Another difficulty are side effects within the hardware and software to be analyzed (e.g. CPU gate and instruction level behavior, or complex task interactions as considered in Part II). These are often ignored as they are simply not evident or cannot be simulated thoroughly. Finally, the effort for developing a highly precise simulator is often comparable to working with real systems[3].

Eventually, it should be noted that in addition to the due diligence in the implementation of real systems and simulation models, both approaches have their specific strengths and weaknesses. In particular, both cannot conduct a complete system state analysis, and thus cannot guarantee or validate the correctness of the system under test[4]. The final selection depends on the main objective: Though mature simulators for either pure low-level or pure high-level evaluations are available for the WSAN domain[5], they can hardly unite both demands to process complex models in great detail. Instead, and especially if temporal behavior or environmental interaction must be considered, the analysis should be done directly within a real test bed. Of course, in-situ probing within the systems under test might also affect their behavior, and must be considered adequately.

Even though the mentioned advantages of real-world implementations compared to simulation might still seem to be less attractive from a scientific point of view – i.e. due to the desire for analyzing more aspects and solution strategies under arbitrary system configurations – I selected the first approach to gain a more embracing understanding for embedded, time-critical, reactive, and networked systems. In addition, the domain of sensor networks – in spite of the demand for more practical work and a growing interest in real-world applications and deployments [108] within the community – is still mostly treated from a theoretical point of view. However, for some extremely complex algorithms, especially for those studies which required lots of intermediate data logging for debugging and the selection of an appropriate parameter set within a large configuration space, the severely resource constrained sensor nodes were often too weak regarding their CPU performance and memory footprint, and a hybrid approach was chosen: Real-world equipment was used to obtain original (sensor) information, which in turn was fed into a simulation environment for applying various algorithms. While this hardware-in-the-loop (HIL) approach meant a good trade-off, especially for higher level software like the localization algorithms in Chapter 13, it was omitted for low-level concepts related to the operating system or network layer.

[3] In fact, a simulator is usually neither perfect, nor is it based on the same architecture as the simulated system – otherwise the system itself could be used.

[4] Considering this, the application of model checking [264] is a quite novel and promising strategy for the formal verification of WSAN systems. Since the involved components (i.e. sensor nodes) are similar or even identical, exploiting certain symmetries can sometimes even attenuate the state space explosion problem and lead to a significant acceleration of the validation process [10, 11].

[5] Examples are OMNeT++ [285] and ns-2 or ns-3[136] for discrete event networks simulations, and MSPsim [273] for instruction level sensor node simulation.

# 3. Scientific Contribution

Throughout this work, we'll take a close look at several quite different research areas related to the design of networked embedded sensor/actuator systems. Figure 1 gives an overview on the major parts, and relates the various chapters to each other:

**Part I** provides a general introduction to wireless sensor/actuator networks. Besides introducing some definitions and the most central characteristics, we'll identify some motivations for researching this pervasive approach. We'll also constitute its conceptual embedding in the wide area of computer sciences, separate it from similar disciplines, and present some related work along with the state of the art in the context of this work.

**Part II** addresses the specific design aspects for embedded sensor/actuator systems in highly dynamic environments. We'll present the exclusively developed SNoW[5] sensor node for our real-world evaluations, and introduce the new preemptive operating system *Smart*OS along with its highly precise time management, collaborative resource sharing (DynamicHinting) and dynamic memory management (CoMem) concepts. These improve the reactivity and compositionality of concurrently running tasks within time-critical and resource constrained embedded systems in general. Considering the sometimes hard to access nodes, their quite large number and the spatial extensions of the evolving networks, we'll finally present the remote maintenance system SNoW Bat for over-the-air software updates based on the *Smart*Net communication protocol.

**Part III** presents a concrete WSAN application for indoor localization and object tracking: SNoW Bat. Based on the hardware and software concepts from Part II, we'll introduce some novel algorithms for ultrasound based distance measurement (Cut), wireless data aggregation (HashSlot), and position estimation (pVoted). Applied as a subsystem within a real-world installation for autonomous vehicle path control, the underlying approaches had to be robust against environmental influences, and technical imponderabilities. In particular, the measurement noise and timing imprecision as well as hardware and communication failures had to be considered carefully. Beyond, and despite of the severe resource constraints of typical sensor nodes, other goals were to achieve a high localization frequency and precision.

**Part IV** draws a conclusion about this work, discusses and evaluates its scientific contribution from a more distant point of view, and gives recommendations for future research directions.

**Part V** contains additional material and technical details for a more comprehensive understanding of various details within this work.

The variety of the topics illustrates the potential complexity of current sensor network applications; especially when enriched with actuators for proactivity and environmental interaction. Besides their conception, development, installation and long-term operation, we'll mainly focus on more "low-level" aspects: Compositional hardware and software design, task cooperation and collaboration, memory management, and real-time operation will be addressed from a

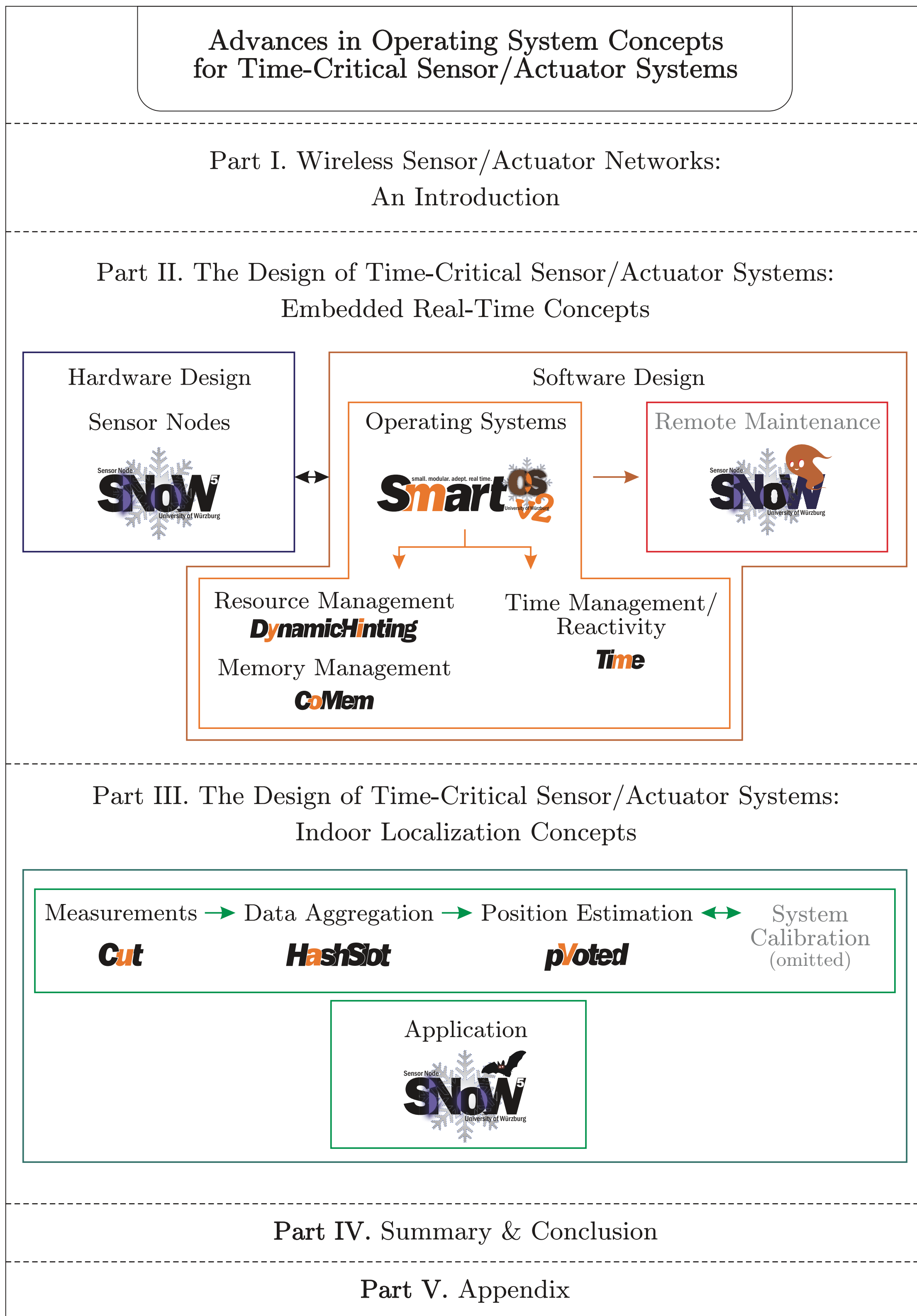



**Figure 1.:** This work at a glance

local node perspective. In contrast, inter-node synchronization, communication, as well as sensor data acquisition, aggregation, and fusion will be discussed from a rather global network view. The diversity in the concepts was intentionally accepted to finally facilitate the reliable implementation of truly complex systems. In particular, these should go beyond the usual "sense and transmit of sensor data", but show how powerful today's networked sensor/actuator systems can be despite of their low computational performance and constrained hardware:

*If their resources are only coordinated efficiently!*

## 3.1. Subject Summary

Before heading to the actual work and its various theoretical and practical parts, I'll briefly introduce each individual research contribution, and provide a *very quick* overview by summarizing the main results (→ Figure 1 for dependencies):

- SNoW$^5$ – a new sensor node and hardware platform for research and education as well as for the evaluation and analysis of novel approaches and theoretical considerations under real-world conditions. Each single concept, technique, and software implementation within this work has been tested and evaluated on SNoW$^5$ nodes.
  *Main contribution*: Modularity and expandability for rapid prototyping, and flexible wireless communication for highly configurable communication protocols.

- *Smart*OS – the software counterpart to the SNoW$^5$ hardware: A novel preemptive operating system and extensive software platform for embedded systems in general, and for all algorithmic approaches and applications within this work in particular.
  *Main contribution*: Compositional software design through reflective task synchronization with priority inheritance based collaboration concepts. Time-awareness through precise event timestamping and temporal semantics. Support for complex design patterns like inter-task synchronization and communication, centralized IRQ handling, and task-specific exception support as commonly known from full grown operating systems.

  - Timestamping – a sophisticated *Smart*OS concept for dealing with temporal imprecision resulting e.g. from software execution imponderabilities and the discretization of time within digital systems.
    *Main contribution*: Reliable capturing of event timestamps as well as the precise scheduling and invocation of (re)actions with – in average – perfectly symmetric error intervals around the true event occurrence or intended reaction time, respectively.

  - DynamicHinting – a novel runtime paradigm for the collaborative management of exclusively shared resources among concurrently running tasks in real-time systems.
    *Main contribution*: Combines task priority reflexion, the reduced duration of bounded and unbounded priority inversions, task starvation avoidance, and deadlock detection (all at runtime) to facilitate and simplify compositional software design significantly.

    - CoMem – a novel concept for dynamic heap memory management in resource constrained (embedded) real-time systems.
      *Main contribution*: On-demand heap reorganization in case of memory shortage, task priority reflexion, and guaranteed worst case allocation times (WCAT) for time-critical requests.

- SNoW Ghost – a remote-management system for over-the-air software updates.
  *Main contribution*: Provides software broadcast and flooding (push) as well as viral software updates (pull). SNoW Ghost will only be considered briefly; see [20] for details.

- SNoW Bat – The real-world indoor localization, tracking, and vehicle steering system will serve us as a representative for complex compositional software design using of the previously presented hardware/software techniques.
  *Main contribution*: Coordinates a distributed system of autonomously operating sensor nodes (46 in our real-world test bed installation) to achieve a localization frequency of up to 2.5 Hz *and* a typical position estimation accuracy of about 9 mm.

    - Cut – a simple but resource efficient DSP algorithm for ultrasound based distance measurements between a sender and several receives.
      *Main contribution*: Angle and distance independent error characteristics with an average measurement precision of about ±0.8 mm despite of low-cost off-the-shelf transducers.
    - HashSlot – a novel TDMA communication protocol for efficient data aggregation in ultrasound based localization systems.
      *Main contribution*: Provides a scheme for computing collision-free and tightly packed TDMA slots for optimal throughput and perfectly deterministic data aggregation time. Based on the semantic use of implicitly available information, it needs no central coordinator or explicit sender communication, but nevertheless supports the dynamic adjustment of the amount of acquired information to the receiver's varying QoS demands.
    - pVoted – a novel 3D position estimation algorithm based on progressive voting for potential locations to filter inaccurate distance measurements.
      *Main contribution*: Achieves a theoretical 3D accuracy below 3 mm, and provides a quality indicator for each estimation (can be used for e.g. path prediction and application optimization).

## 3.2. Underlying Publications

This work is based on the following publications with me as the main author (sorted by topic and year). Unfortunately not all of them found their way into this work as a separate section. Nevertheless they are deeply related to the main subject of this thesis, and might possibly serve as sources of further information. References are given in the text where appropriate.

### Journal Publications:

[29] Marcel Baunach:
**"CoMem: Collaborative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks"**
Springer Real-Time Systems Journal (**RTSJ**), Volume 48, Issue 1, January 2012.

[28] Marcel Baunach:
**"Dynamic Hinting: Collaborative Real-Time Resource Management for Reactive Embedded Systems"**
Elsevier Journal of Systems Architecture (**JSA**), Volume 57, Issue 9, October 2011.

### Sensor node (SNoW$^5$):

[34] Marcel Baunach, Reiner Kolla, Clemens Mühlberger:
**"SNoW5: A Modular Platform for Sophisticated Real-Time Wireless Sensor Networking"**
Technical Report No. 399, University of Würzburg, January 2007.

[31] Reiner Kolla, Marcel Baunach, Clemens Mühlberger:
**"SNoW5: A Versatile Ultra Low Power Modular Node for Wireless Ad Hoc Sensor Networking"**
5th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (**FGSN 2006**), Stuttgart, July 2006.

### Operating system (*Smart*OS):

[37] Marcel Baunach, Clemens Mühlberger, Christian Appold:
**"Enabling Real-Time in WSN Applications"**
6th European Conference on Wireless Sensor Networks (**EWSN 2009**), Cork, February 2009.

[36] Marcel Baunach, Reiner Kolla, Clemens Mühlberger:
**"Introduction to a Small Modular Adept Real-Time Operating System"**
6th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (**FGSN 2007**), Aachen, July 2007.

### Resource management (DynamicHinting):

[22] Marcel Baunach:
**"Dynamic Hinting: Real-Time Resource Management in Wireless Sensor/Actor Networks"**
15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (**RTCSA 2009**), Beijing, August 2009.

[23] Marcel Baunach:
**"Priority aware Resource Management for Real-Time Operation in Wireless Sensor/Actor Networks"**
8th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (**FGSN 2009**), Hamburg, August 2009.

### Dynamic memory management (CoMem):

[25] Marcel Baunach:
**"CoMem: Cooperative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks"**
18th International Conference on Real-Time and Network Systems (**RTNS 2010**), Toulouse, November 2010.

[26] Marcel Baunach:
**"Collaborative Memory Management for Reactive Sensor/Actor Systems"**
5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (**SenseApp 2010**), Denver, October 2010.

[24] Marcel Baunach:
**"Dynamic Memory Management for Resource Constrained Sensor/Actor Systems"**
9th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (**FGSN 2010**), Würzburg, September 2010.

### Remote maintenance (SNoW Ghost):

[20] Marcel Baunach:
**"Ghost: Software and Configuration Distribution for Wireless Sensor/Actor Networks"**
7th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (**FGSN 2008**), Berlin, September 2008.

### Wireless data aggregation (HashSlot):

[21] Marcel Baunach:
**"Speed, Reliability and Energy Efficiency of HashSlot Communication in WSN Based Localization Systems"**
5th European Conference on Wireless Sensor Networks (**EWSN 2008**), Bologna, January 2008.

[33] Marcel Baunach, Reiner Kolla, Clemens Mühlberger:
**"A Method for Self-Organizing Communication in WSN Based Localization Systems: HashSlot"**
2nd IEEE International Workshop on Practical Issues in Building Sensor Network Applications (**SenseApp 2007**), Dublin, October 2007.

### Position estimation (pVoted) and localization systems (SNoW Bat):

[27] Marcel Baunach:
**"pVoted: A Progressive On-Line Algorithm for Robust Real-Time Localization and Tracking in spite of Faulty Distance Information"**
1st International Conference on Indoor Positioning and Indoor Navigation (**IPIN 2010**), Zürich, September 2010.

[35] Marcel Baunach, Reiner Kolla, Clemens Mühlberger:
**"SNoW Bat: A High Precise WSN Based Location System"**
Technical Report No. 424, University of Würzburg, May 2007.

## Related Publications and Co-Authorship

In addition to the aforementioned contributions, I was co-author for the following publications:

[38] Marcel Baunach, Clemens Mühlberger, Christian Appold, Martin Schröder, Florian Füller:
**"Analysis of Radio Signal Parameters for Calibrating RSSI Localization Systems"**
Technical Report No. 455, University of Würzburg, March 2009.

[209] Clemens Mühlberger, Marcel Baunach:
**"Tab WoNS: Calibration Approach for WSN based Ultrasound Localization Systems"**
7th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (**FGSN 2008**), Berlin, September 2008.

[32] Clemens Mühlberger, Marcel Baunach, Reiner Kolla:
**"Beyond Theory: Deployment of a Real-World Localization Application as Low Power WSN"**
2nd IEEE International Workshop on Practical Issues in Building Sensor Network Applications (**SenseApp 2007**), Dublin, October 2007.

[257] Armin Runge, Marcel Baunach:
**"Precise Self-Calibration of Ultrasound Based Indoor Localization Systems"**
2nd International Conference on Indoor Positioning and Indoor Navigation (**IPIN 2011**), Guimarães, September 2011.

# Contents


**Lectori salutem** i

**About this Work** vii

**Organization and Classification** vii

1 Background and Evolution . . . vii

2 Research Approach and Methodology . . . viii

3 Scientific Contribution . . . xi

3.1 Subject Summary . . . xiii

3.2 Underlying Publications . . . xiv

**I Wireless Sensor/Actuator Networks: An Introduction** 1

**1 Introduction** 3

1.1 Wireless Sensor/Actuator Networks: Definitions & Paradigms . . . 3

1.1.1 Typical Characteristics and the Environmental Context . . . 5

1.1.2 The Design Space and Related Disciplines in Computer Sciences . . . 7

1.1.3 Differentiation with Respect to other Wireless Networks . . . 8

1.2 Wireless Sensor/Actuator Networks: Past – Present – Future . . . 8

1.2.1 WSAN Research Directions . . . 13

1.3 Summary . . . 17

**II The Design of Time-Critical Sensor/Actuator Systems: Embedded Real-Time Concepts** 19

**2 Hardware Platforms** 21

2.1 Introduction . . . 21

2.2 SNoW$^5$ - The Sensor Node of Würzburg . . . 22

2.2.1 Requirements and Design Goals . . . 23

2.2.2 Hardware Components and Conceptual Properties . . . 24

**3 Operating Systems** 33

3.1 Introduction . . . 34

3.2 Operating System Objectives . . . 35

3.3 Operating Systems for Embedded Systems and Sensor/Actuator Networks . . . . 38
3.3.1 The Kernel Classification Framework . . . . . . . . . . . . . . . . . . . . . . 39
3.3.2 Related Work and State of the Art . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3.3 Open Problems and *Smart*OS . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**4 *Smart*OS: A Small modular adept real-time Operating System 47**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.2 Philosophy and Classification . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.3 Kernel Architecture and Central Concepts . . . . . . . . . . . . . . . . . . . . . 49
4.3.1 Basic Terminology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.3.2 Process Organization: Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.3.3 Operation Modes and Syscall Handing . . . . . . . . . . . . . . . . . . . . . 52
4.3.4 Time Management: The Timeline . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3.5 Interrupts and Interrupt Handlers . . . . . . . . . . . . . . . . . . . . . . . . 55
4.3.6 Synchronization: Mutexes and Events . . . . . . . . . . . . . . . . . . . . . . 56
4.3.7 Hardware Abstraction: Resources and Resource Chains . . . . . . . . . . . 57
4.3.8 Exception Handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.4 Evaluation and Benchmarking . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.4.1 Concurrent Task Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.4.2 Kernel and Application Benchmarking . . . . . . . . . . . . . . . . . . . . . 62
4.4.3 Timer Interrupt Overhead . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.4.4 Interrupt Processing Overhead . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.4.5 Further Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**5 Time and Reactivity in *Smart*OS 69**
5.1 Introduction on Information Attribution . . . . . . . . . . . . . . . . . . . . . . 70
5.2 Time in Digital Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.3 Time in *Smart*OS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.4 Test Bed: Node Self-Calibration and Pairwise Drift Calculation . . . . . . . . . . 79
5.4.1 Signal Emission and Self-Calibration . . . . . . . . . . . . . . . . . . . . . . 81
5.4.2 Pairwise Drift Calculation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4.3 Real-World Test Bed Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.4.4 Conclusion and Outlook . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

**6 Real-Time Resource Management in Networked Sensor/Actuator Systems 87**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
6.2 Motivation and Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.2.1 Terminology and the Problem Model . . . . . . . . . . . . . . . . . . . . . . 90
6.2.2 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
6.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
6.3.1 Resource Management in Preemptive Systems . . . . . . . . . . . . . . . . . 95
6.3.2 Resource Synchronization Protocols with Priority Inheritance . . . . . . . 97

6.4 Resources and Priority Inheritance under *Smart*OS . . . . . . . . . . . . . . . . . . 102
  6.4.1 Extended *Smart*OS Specifications . . . . . . . . . . . . . . . . . . . . . . . . 102
  6.4.2 Priority Inheritance and Deadlock Occurrence . . . . . . . . . . . . . . . . 105
6.5 The DynamicHinting Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
  6.5.1 Addressed Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
  6.5.2 Cooperation and Collaboration . . . . . . . . . . . . . . . . . . . . . . . . 109
  6.5.3 The Proposed Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
  6.5.4 Receiving Hints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
  6.5.5 Processing Hints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
  6.5.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.6 Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
  6.6.1 Resource Allocation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
  6.6.2 Resource Deallocation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
  6.6.3 Timeout Handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
  6.6.4 HintHandlers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
6.7 Real-World Applications and Test Beds . . . . . . . . . . . . . . . . . . . . . . 123
  6.7.1 DynamicHinting and the Priority Ceiling Protocol . . . . . . . . . . . . . . 124
  6.7.2 Test Bed I – Continuous Data Streaming . . . . . . . . . . . . . . . . . . . 125
  6.7.3 Test Bed II – The Dining Philosophers Stress Test . . . . . . . . . . . . . . 129
6.8 Conclusion and Outlook . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

**7 Collaborative Memory Management for Reactive Sensor/Actuator Systems 139**
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
7.2 Motivation and Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
  7.2.1 Existing Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
  7.2.2 Feature Requests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
7.3 Related Work and State of the Art . . . . . . . . . . . . . . . . . . . . . . . . . 147
  7.3.1 Dynamic Memory Management in general Embedded Systems . . . . . . . 148
  7.3.2 Dynamic Memory Management in Sensor/Actuator Systems . . . . . . . . 150
7.4 The CoMem Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
  7.4.1 Basic Design Criterions and Application Example . . . . . . . . . . . . . . 153
  7.4.2 Collaborative Memory Sharing . . . . . . . . . . . . . . . . . . . . . . . . . 156
  7.4.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
7.5 CoMem Implementation and Usage . . . . . . . . . . . . . . . . . . . . . . . . . 161
  7.5.1 Internal Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
  7.5.2 API Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
  7.5.3 Implementation Consequences . . . . . . . . . . . . . . . . . . . . . . . . . 170
  7.5.4 Hard Real-Time Heap Organization . . . . . . . . . . . . . . . . . . . . . . 173
7.6 CoMem Evaluation and Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 178
  7.6.1 Dynamic Memory Stress Test . . . . . . . . . . . . . . . . . . . . . . . . . . 180
  7.6.2 Real-World Application under Real-Time Conditions . . . . . . . . . . . . . 186
7.7 Summary and Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

**8 Selected *Smart*OS Software Design Examples 193**
8.1 The *Smart*Net Radio MAC Protocol . . . 194
8.2 The SNoW Ghost Remote Maintenance System . . . 198
8.2.1 Concept and Operation . . . 199
8.2.2 Resource Demands and Performance Evaluation . . . 201
8.3 Summary . . . 201

**III The Design of Time-Critical Sensor/Actuator Networks: Indoor Localization Concepts 205**

**9 Localization Systems 207**
9.1 Introduction . . . 207
9.2 Motivation, Requirements, and Evaluation Metrics . . . 211
9.2.1 Points of View . . . 212
9.2.2 Evaluation Metrics . . . 215
9.2.3 The Design Space . . . 218
9.3 Related Work . . . 219
9.3.1 Scope of this Part of the Work . . . 220

**10 The SNoW Bat Indoor Localization and Tracking System 223**
10.1 Operation Principles . . . 224
10.1.1 SNoW Bat Operation Stages . . . 225
10.2 System Design Considerations . . . 227
10.2.1 The Sensor Node Platform Configuration . . . 227
10.2.2 Anchor Infrastructure and Deployment . . . 230
10.2.3 The Impact of the Data Aggregation on the Localization Frequency . . . 232
10.2.4 The Impact of the Software Design on the Localization Frequency . . . 235
10.3 Summary . . . 239

**11 Ultrasound Distance Measurement: The Cut Algorithm 241**
11.1 Timing and Chirp Emission . . . 242
11.2 Digital Signal Processing . . . 244
11.3 Summary . . . 250

**12 Efficient Data Aggregation: The HashSlot Algorithm 251**
12.1 Introduction . . . 252
12.2 Motivation and Requirements . . . 253
12.2.1 Existing Problems regarding the Communication Cost . . . 253
12.2.2 Feature Requests regarding the Data Fusion Cost . . . 255
12.2.3 Feature Requests regarding the Network Maintenance . . . 256
12.2.4 The Protocol Design Space . . . 256
12.3 Related Work – Wireless Data Aggregation Protocols . . . 257
12.3.1 Potential Candidates for SNoW Bat . . . 259

12.3.2 Non-slotted Methods . . . 260
12.3.3 Slotted Methods . . . 261
12.4 The HashSlot Protocol . . . 263
12.4.1 Slot Calculation . . . 264
12.4.2 Transmission Time Calculation and Slot Boundary Compliance . . . 274
12.4.3 Summary . . . 275
12.5 A first Real-World Test Bed and Performance Analysis . . . 275
12.6 Further Improvements for Real-World and Application-Specific Requirements . . 279
12.7 The HashSlot+ Extension . . . 283
12.8 Summary . . . 289

**13 Progressive Position Estimation: The pVoted Algorithm 291**
13.1 Introduction . . . 292
13.2 The pVoted Position Estimation Algorithm . . . 292
13.3 Evaluation . . . 299
13.4 Summary . . . 304

**14 Real-World Evaluation: The complete SNoW Bat System Installation 305**
14.1 Temporal Performance . . . 306
14.2 Spatial Performance . . . 308

**IV Summary & Conclusion 311**

**15 Summary, Outlook and Conclusion 313**
15.1 Scientific Contributions . . . 313
15.2 Future Work . . . 316
15.3 Conclusion . . . 316

**V Appendix 319**

**A *Smart*OS API and Data Type Reference 321**

**B SNoW$^5$ Schematics and Layout 329**

**VI Lists and Indexes 333**

**Bibliography 335**

**List of Figures 361**

**List of Tables 365**

**List of Listings 367**

**Abbreviations** 369

# Part I.

# Wireless Sensor/Actuator Networks: An Introduction

"Curiosity is always first in a problem that waits to be solved."

*(Galileo Galilei, Italian philosopher)*

# 1. Introduction

The technological and technical advances during the past 150 years did not only introduce the directed radio communication[1] between two parties (denoted as *sender* and *receiver* or *source* and *sink*), but today even the operation of complex wireless networks comprising an enormous number of participants (denoted as *nodes*) became reality for military, industrial, scientific, and private use. Beginning with pulsed electromagnetic waves and the unidirectional transmission of Morse signals in the 1890s, the application of modulation techniques permitted the first analog voice communication in 1913. Since then, the technology was mainly used for radio and television broadcasts with few senders and many receivers within their transmission range. Bidirectional communication (especially in full-duplex mode) was expensive and hard to manage, and thus remained reserved for non private customers. The invention of integrated circuits introduced digital signal processing and digital radio communication. In the early 1990s the DECT (Digital European Cordless Telephony) and GSM (Global System for Mobile communications) standards for cordless and mobile phones established digital voice transmission and service provision in the private domain and with wide coverage. Today, data oriented transmissions are available for computer systems via WLAN (Wireless Local Area Network) and the 802.11 standard, as well as for mobile phones via e.g. UMTS (Universal Mobile Telecommunications System) or 4G standards. In parallel to the corresponding voice and data oriented networks – which slowly merge into each other since Voice over IP and similar techniques transform all traffic into compatible formats for the world wide web – yet another type of wireless network has emerged: The *wireless sensor network* (WSN) and its proactive extension, the *wireless sensor/actuator network* (WSAN). We also refer to these as *sensor network* (SN) or *sensor/actuator network* (SANet) in those cases where the explicitly wireless communication type is not available or simply not relevant.

## 1.1. Wireless Sensor/Actuator Networks: Definitions & Paradigms

The main purpose of a wireless sensor network is perfectly described by its name: Basically, it consists of a set of *sensor nodes*, also denoted as *motes*, which are deployed randomly or at strategically relevant locations within the environment or onto objects and creatures under surveillance to locally measure or sample certain application or objective specific information by appropriate hardware equipment (*sensor data acquisition*). Some popular examples are temperature, humidity, distances, pollutants, sound, signal strengths, and vibration. In the

[1] Electromagnetic waves were predicted by James Clerk Maxwell in 1864, and experimentally validated by Heinrich Hertz in 1888.

most basic vision, the acquired sensor information is transferred to a central sink (*sensor data aggregation*) for simple monitoring, centralized analysis, or further processing in general. This so called *remote metering* allows the systematic observation of dynamic processes by combining the received information from several nodes into a "big picture" of the overall system (*sensor data fusion*). Examples are position estimation and object tracking as well as the tracing of event patterns and their temporal and spatial propagation. By comparing the observed information to an expected state (concerning both the environment and the system itself), adequate measures can be undertaken by other (non-WSN) subsystems, e.g. at the sink, to finally reach, control, or recover the desired state. This leads us directly to the first definition and paradigm:

---

**Definition I.1: (Wireless) Sensor Networks: The Sense and Aggregate Paradigm**
A *sensor network* (SN) is a spatially and computationally distributed system of sensor nodes for the conjoint monitoring of physical or chemical [122] conditions. In *wireless sensor networks* (WSN), which are commonly deployed in exceptionally vast or hard to wire environments, the sensor nodes operate mostly autarkic[2], and communicate and interact over-the-air.

---

As an extension to this purely investigative approach, proactivity becomes inherent to these distributed systems when supplementing the sensors with actuators which would allow them to exert direct influence on the environment. Though an additional communication channel returning from the sink to the sensor/actuator nodes is consequently required to supply these with behavioral instructions, e.g. actuator commands, this extension introduces the option to trigger reactions and corrections nearby the corresponding metering points.

In more advanced systems, the nodes are almost entirely independent from a central sink which issues the actuator commands, but gain partial autonomy by integrating their own control systems. These operate according to more general objectives from one or more coordinators, or simply by static rules specified within their local application software. To further support this reactive approach, nodes commonly communicate with each other and coordinate their behavior for achieving collective success (emergent behavior):

---

**Definition I.2: (Wireless) Sensor/Actuator Networks: The Sense and React Paradigm**
A *sensor/actuator network* (SANet) is a hierarchical system of cooperating sensor/actuator nodes which incorporates an also hierarchical control system for the distributed accomplishment of complex and distributed objectives. In *wireless sensor/actuator networks* (WSAN), especially when deployed in vast or hard to wire environments, the sensor/actuator nodes are required to operate mostly autarkic and partially autonomic[3].

---

Figure 1.1 gives an example: Though the depicted sensor/actuator node contains its own control system for reacting on self-measured sensor values, it also accounts for external information received wirelessly from other (maybe equal) nodes to fulfill its current *local objective*.

[2] Autarky or self-sufficiency: The freedom from external, non-natural resources (e.g. energy).
[3] Autonomy or self-governance: The freedom from external decisions and rules (e.g. commands).

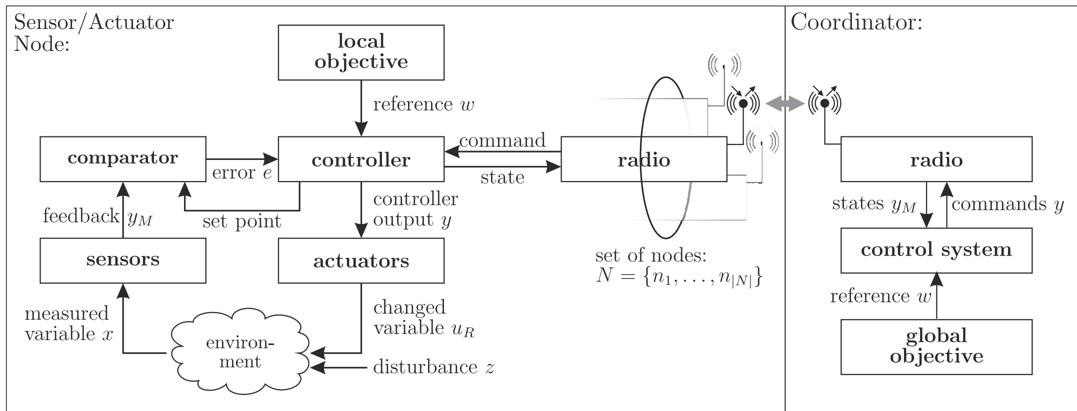



**Figure 1.1.:** Schematic of hierarchical control systems in wireless sensor/actuator networks

In addition, it also accepts instructions from a higher level coordinator which in turn might modify these local objectives based on its own variable but more comprehensive notion of the monitored dynamic system, the static application specification, and the *global objectives* derived therefrom.

Although we have just introduced distinctive definitions for WSN and WSAN, we'll use the notions of node, sensor node, sensor/actuator node, and mote as synonyms throughout this work, and always consider the particular device to incorporate both sensors *and* actuators. See Chapter 2.1 for a description of various node types and their design considerations.

### 1.1.1. Typical Characteristics and the Environmental Context

Besides their fundamental definition, wireless sensor/actuator networks can be described by a multitude of properties. Some of them are an inherent part of each deployed system, others depend on the actual application and the underlying objectives. Three very common notions refer to the terms *ubiquitous computing* [305], *pervasive computing* [201], and *ambient intelligence* [322]. Though these are often used interchangeably, they were originally introduced by Xerox (Mark Weiser, 1988), IBM (Lou Gerstner, 1994), and Philips (Simon Birrell, 1998), respectively, and express subtle differences in the system's interaction as well as in the user's perception of the technology. While [252] gives a comprehensive study on the evolution of these terms, we summarize their definitions as follows:

---

**Definition I.3: Ubiquity, Pervasiveness, and Ambient Intelligence**

*Ubiquity* denotes a technology's property of being invisibly omnipresent and deployed everywhere without making any statement about the interaction between the appendant devices (e.g. embedded computer systems in general). *Pervasiveness* denotes a ubiquitous technology's property of being permeate and inevitably influencing our everyday's live by providing services through interconnected and networked devices (e.g. mobile communication). *Ambient intelligence* denotes a pervasive technology's property of seamlessly bridging the gap between the real and the digital world by being practically invisible for the user (e.g. augmented reality).

---

According to Definition I.3, we will differentiate between a local view considering the nodes as (ubiquitous) embedded systems and a global view considering the entire networks as (pervasive) distributed systems. Figure 1.2 gives an extensive (yet possibly incomplete) summary of their various characteristics which can finally lead to the user's perception of an ambient intelligence for a multitude of services.

**The local node view.** While each node is per definition a wirelessly communicating and signal processing device, it is also designed for energy efficiency. However, the exact meaning of this demand is often only vaguely defined, and can in particular not be expressed universally by simply specifying the node's power consumption under certain operation modes. Instead, it depends on the duty cycle $D_m$ of each mode $m$ which is commonly influenced by the software system and the environmental situation. A detailed study concerning our SNoW[5] sensor node can be found in Section 2.2.2 and [93]. For this work, we define energy efficiency as follows:

---

**Definition I.4: Energy Efficiency**
A node is *energy efficient*, if it shows a favorable balance between energy consumption, reserves, and regeneration. That is, if it autarkically survives the intended network lifetime while reliably and sufficiently providing its individual services even in the worst case, e.g. during periods of maximal energy consumption and minimal harvesting which can be expected for the application it is used for. Of course this definition can easily be extended to the entire network by simply requesting energy efficiency for each single node.

---

In addition to the mentioned inherent properties, nodes are commonly considered to be designed as special purpose devices[4] with low performance but quite complex self-management features. For some systems, they are required to be mobile or even real-time capable for improved reactivity within highly dynamic environments. If the numbers and placement of nodes is sufficiently large and dense to provide a certain service with extensive coverage, they are considered to be ubiquitous.

**The global network view.** From a global network view, we see nodes interacting with the environment and with each other to cooperate or even collaborate[5] with respect to a common objective. Therefore, and for an increased deployment range, the direct node-to-node communication must inevitably be extended to multi-hop where intermediate nodes between the original sender and the final receiver forward the transmitted data. To eventually ensure a proper operation, nodes have to be integrated into the environment in a way to exert minimal undesired impact on the surrounding [193, 220] and to also receive minimal disturbance from it [122]. Optional characteristics in this context are their capability to establish their own (physical and logical) infrastructure, and to scale their operation with the number of nodes and communication links. Starting with a certain degree of complexity, self-organization [21, 82] becomes essential to maintain dynamic topologies, and to achieve reliable fault tolerance. In fact, this is the next step from the nodes' ubiquity to the overall system's pervasiveness.

[4] At least the sensor and actuator equipment is commonly application-specific.
[5] See Section 6.5.2[p109] for a detailed disambiguation of these terms.

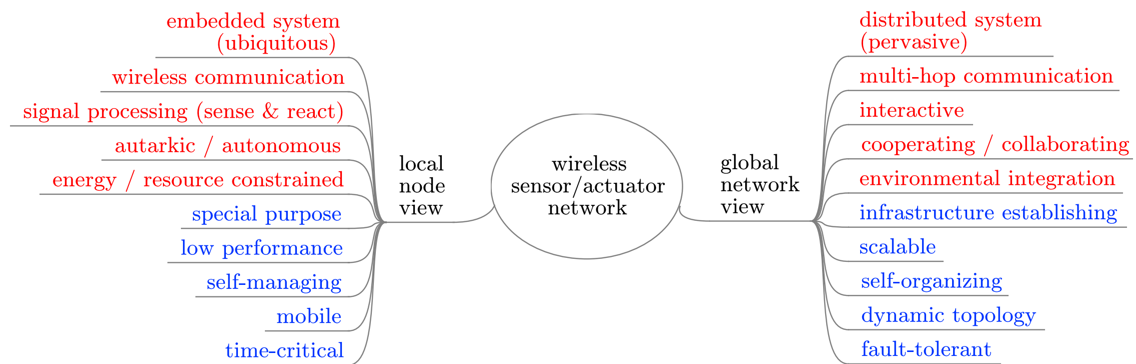



**Figure 1.2.:** Characterization of wireless sensor/actuator nodes and networks: Generally accepted inherent (red) and application-specific (blue) properties

**The environmental context.** When supported by an appropriate hardware/software/network design, the elementary characteristics of wireless sensor/actuator networks reveal three main advantages which finally allow them to integrate almost seamlessly into the context of common application scenarios, and to improve long established techniques which were initially based on wired devices (and centralized data processing):

1. spatially distributed operation through wireless installation
2. environmental interaction through decentralized data processing
3. service reliability and fault tolerance through self-organization

However, these properties come at sometimes huge system and application complexity – at least when considering the constrained performance and resources of today's typical sensor nodes.

### 1.1.2. The Design Space and Related Disciplines in Computer Sciences

The complexity of wireless sensor/actuator networks is reflected by a multidimensional design space. Though we won't discuss its various aspects in detail, but refer to later sections and appropriate literature like [4, 50, 99, 130, 249], Figure 1.3 gives an impression about its extent and shows a (still incomplete) property tree with four main branches. Besides application-specific requirements for the local nodes and the global networking aspect itself, it also covers the special challenges of highly dynamic and particularly sensitive or harsh environments in which these distributed systems are often expected to operate. Another criterion is the analyzability of the overall system. Indeed, any WSAN design introduces a multivariable optimization problem:

---

**Definition I.5: The WSAN Optimization Problem**
For a given objective, find a global and local system configuration (including hardware, software, and communication subsystems) to meet the specifications. At the same time, provide energy efficiency according to Definition I.4, and maximize the system utility and environmental compatibility under minimal costs and effort for development, deployment, and maintenance.

---

To solve this problem, flexible concepts are essential for the reliable composition of all involved subsystems. The customization of hardware (e.g. sensors, actuators, and MCUs), infrastructure

(e.g. communication and routing protocols), and software subsystems (e.g. operating systems and application tasks) provides the flexibility to tune minimal cost and energy consumption against maximal utility and lifetime of each individual node and the overall network.

As a consequence, wireless sensor/actuator networks are rooted in many areas of computer science. Figure 1.4 shows a structured overview on related research disciplines and the corresponding main requirements for the WSAN domain. More detailed discussions can be found in the corresponding chapters within this work.

### 1.1.3. Differentiation with Respect to other Wireless Networks

The ability to communicate wirelessly might raise the question why entirely novel techniques are required in coexistence with long established, widely accepted, and thoroughly approved standards for e.g. WLAN and cellular phones with permanently extending coverage, improving hardware, and increasing data rates. The reasons are mainly justified by application-specific requirements related to energy, infrastructure, real-time-behavior, data-flow direction, and the conflicting constraints imposed by most existing protocol standards focusing on large data volumes and service dissemination.

While today's wirelessly communicating computers and mobile phones are commonly data sinks receiving more information than sending, sensor nodes work in the opposite direction as they mainly collect or generate information and exchange it with others. Although the required data rates are rather low, special timing specifications must sometimes be met. These demand for the design of tailored radio protocols with deterministic transmission delays and guaranteed collision-freedom, even in the case of high priority message bursts and node mobility (→ Chapter 12). Nevertheless, these protocols must be energy efficient since communication is one of the most current consuming tasks of a sensor node (→ Section 2.2.2), and consequently deserve careful optimization[6] to avoid related power failures and the need for manual maintenance of hard-to-access and virtually autarkic devices[7]. In contrast, consumer handheld devices contain other, more energy consuming components like powerful CPUs and displays, but can easily be recharged when required. Finally, while some WSAN applications demand for pure ad hoc communication, others require the prior setup of special infrastructures with precisely defined geometric or technical properties (→ Chapter 13); both requirements are commonly not satisfied by the initially named standards.

## 1.2. Wireless Sensor/Actuator Networks: Past – Present – Future

Having introduced the special characteristics and strengths of wireless sensor/actuator networks, we still need to discuss which solutions or services this technology might provide and which

[6] A corresponding consideration according to Amdahl's law can be found in Section 10.2.4.

[7] Similar problems which often jeopardize a node's autarky and autonomy relate to dynamic resource management among preemptive tasks under real-time conditions (→ Chapter 6).

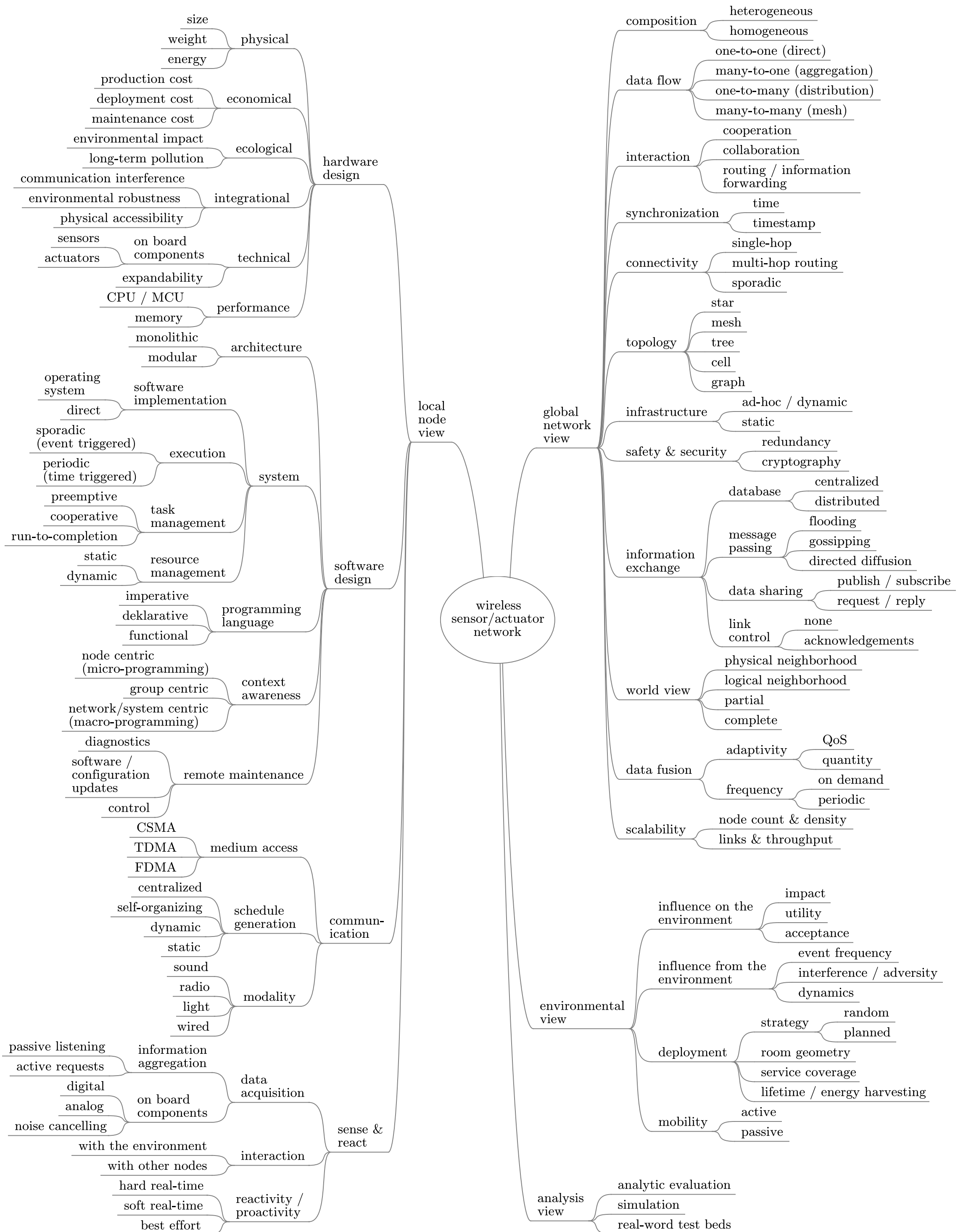



**Figure 1.3.:** The design space and classification of wireless sensor/actuator networks: A *few* selected properties and realization options

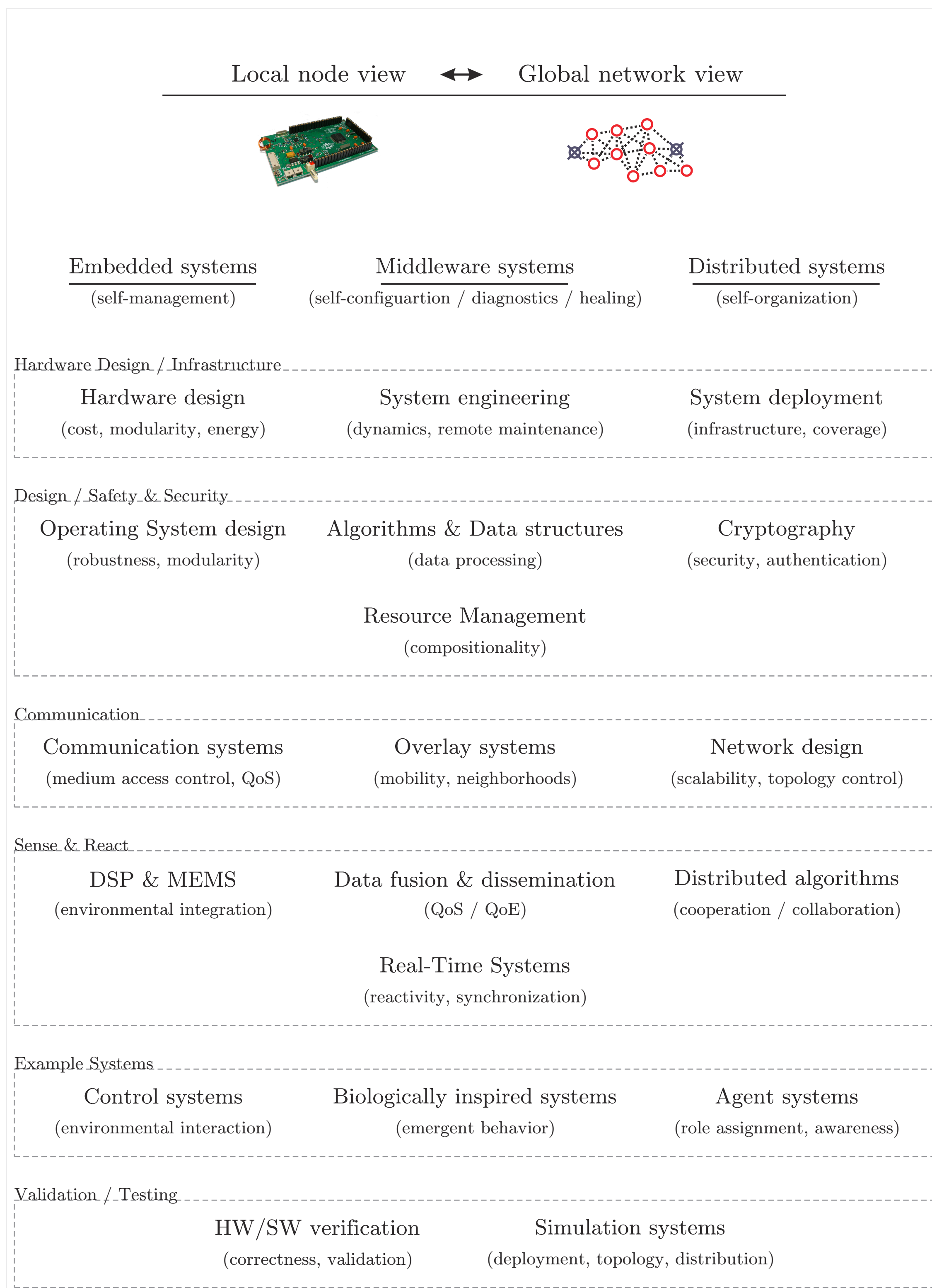
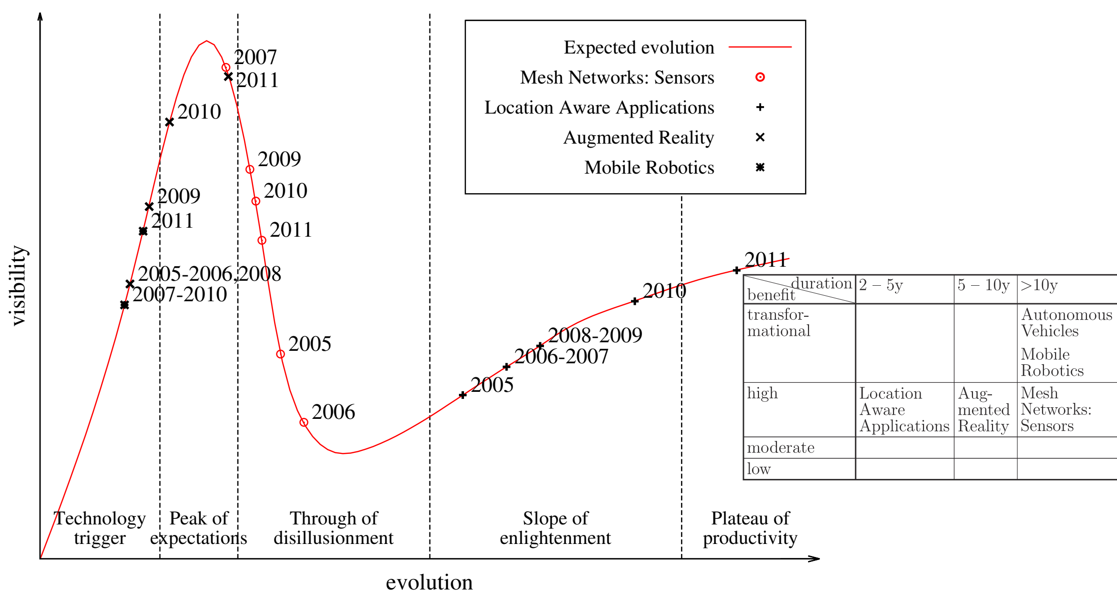



**Figure 1.4.:** A selection of relevant research disciplines for the WSAN domain (central example requirements in parentheses)




| benefit \ duration | 2 – 5y | 5 – 10y | >10y |
|---|---|---|---|
| transfor-mational | | | Autonomous Vehicles<br>Mobile Robotics |
| high | Location Aware Applications | Aug-mented Reality | Mesh Networks: Sensors |
| moderate | | | |
| low | | | |

**Figure 1.5.:** The Gartner Hype Cycle and priority matrix for selected emerging technologies

applications might significantly profit from its support to justify the efforts in solving the involved design and optimization problems.

During the past decade, wireless sensor networks attracted an enormous attention within research and industry. Initially brought to life in the early 80s by military forces[8], eager for more flexible and efficient information retrieval through on-demand node deployments with ad hoc capabilities, the presentation of the civilian Smart Dust project [9, 148] in 1998 triggered an unprecedented hype around this still new and hardly recognized technology. The combination of miniaturized circuitry, wireless communication, and micro electromechanical systems (MEMS) created entirely new opportunities, and the vision of deploying several thousands of intelligent, yet small and cheap sensors in arbitrary places, rapidly stirred up the hope for revolutionary monitoring applications. The combined power of a so called *swarm* of motes and the invaluable advantage of wireless communication – finally eliminating the expensive and fault-prone cabling between sensors which has been used previously – inspired projects in medicine, meteorology, geophysics, biology, astronomy, and various other disciplines. Even an entirely new research community within the computer sciences emerged from the desire to solve the manifold upcoming problems, and consequently the number of related publications increased exponentially [134]. Regarding its impact, the Technology Review Journal identified the wireless sensor networks (WSN) in 2003 as one out of "10 emerging technologies that will change the world" [16].

For many years, however, only few real-world deployments brought notable benefits for industrial customers or end-users. In fact, the exaggerated early visions were not realizable at once, and thus researchers focused on theoretical work and the hype disappeared along with the industry's attention. Since 2005, the Gartner analysts for information technology evaluate

[8] See the DARPA Distributed Sensor Networks (DSN) program as an example.

the maturity of emerging technologies with reference to the so called *Hype Cycle*, which is an indicator for their visibility, acceptance and impact to the market[9]. Independent from the technology under consideration, the Hype Cycle is always presented as a graph as depicted in Figure 1.5: Its visualization avoids to put the absolute time on the x-axis but applies evolution steps instead. Beginning with the initial trigger, followed by a peak of extreme expectations and a valley of disillusionment, the sensibility for the technology's central aspects grows, and finally leads to the plateau of productivity which is robust but far from the early visions. The resulting sequence exposes a constant shape and allows to conveniently compare an arbitrary number of technologies within a single figure. During the annual market analysis the current visibility of each candidate is evaluated and marked on the shape. Since the individual evolution depends on highly dynamic factors and can even end abruptly, the markers might consequently jump back and forth or even disappear. However, most technologies make their way in the direction of solid acceptance. Figure 1.5 shows the evaluation for sensor networks (denoted as "Mesh Networks: Sensors") and some related technologies. While the latter grow steadily, the sensor networks jump between hype and disappointment; they even disappeared temporarily in 2008. Nevertheless, they are considered to inflict high benefits in over 10 years as depicted by the *Priority Matrix* which is always associated with the Hype Cycle. So, what is the reason for the lasting stagnation? Considering the market, there are only few concrete requests for sensor network solutions, leading to even more theoretical work or feasibility studies. Since these won't arouse public attention, the market won't grow and the requests remain absent. Indeed, the technical progress – especially in the hardware area – always resulted from technology pushes but not from market pulls.

So, how can we change the situation and escape from this vicious cycle? One long sought chance would be to finally find a *true* killer application making the benefit more visible to the consumer. Especially in the end-user market the usability in terms of deployment, configuration, and maintenance must be improved significantly to gain a broader acceptance. Other wireless technologies, like mobile communication and wireless LAN, are ahead by far in this regard. In fact, the vast network of cellular phones is already considered as the largest real-world "sensor network". Another option is to form a symbiosis with other potent and future proof, but still premature and thus susceptible technologies, and to derive concrete requirements from these overlay applications. Let's once more have a look at Figure 1.5: Location aware applications are already close to mature in the end-user market – which is mainly caused by satellite navigation (GNSS, → Part III) and mobile communication. Augmented reality and mobile robotics are at significantly earlier stage, and might still incorporate sensor network applications for improved perception quality and environmental interaction. Additionally, the first is expected to cause a high benefit in 5-10 years, and the second is even considered as "transformational" in about 10 years. Indeed, we can already see increased efforts in pushing into multimedia systems (MMS) and cyberphysical systems (CPS) [171], which will definitely require complex sensor systems as central parts of their control loops. Some other promising application areas for wireless sensor/actuator networks include logistics [225, 255] and traffic control

[9] The Gartner, Inc. Hype Cycle and Priority Matrix 2011: `http://www.gartner.com/technology/research/hype-cycles/`, last accessed in 01/2012.

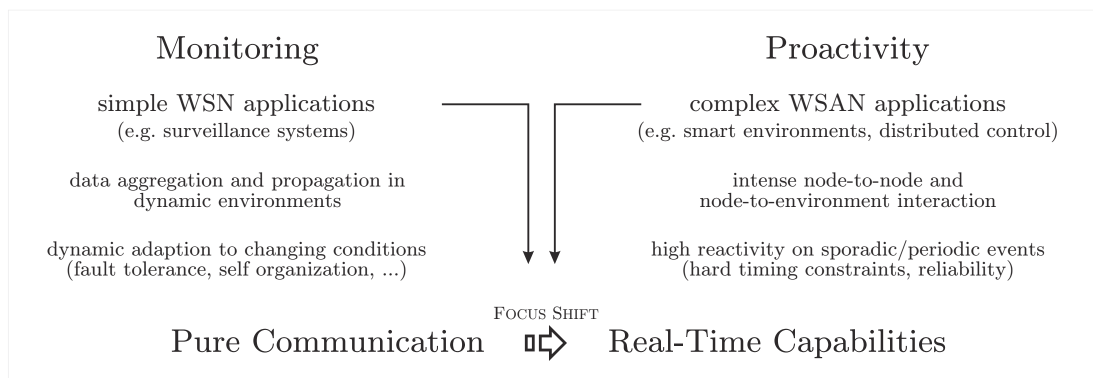



**Figure 1.6.:** Why real-time matters

[39], biomedicine and healthcare [78], facility and plant management [66, 219], environmental surveillance [227, 311, 313] and habitat monitoring [183, 193, 220], and – once more – military operations [118]. Yet, to succeed in such cooperations – and even within critical infrastructures –, powerful hardware/software concepts are required to facilitate the deployment and operation while fulfilling the high quality standards for reliability, security, and performance.

### 1.2.1. WSAN Research Directions

If we envision and comprehend wireless sensor/actuator networks with their flexible but resource constrained nodes as part of larger systems for physical interaction and environmental control, we'll discover some considerable shortcomings of many current sensor network approaches which mainly focus on the sensing and the networking aspect. As illustrated in Figure 1.6 we see and predict a clear focus shift from monitoring in classical WSN or SN applications towards proactivity in advanced WSAN and SANet control systems. Cyber physical systems (CPS) for example already integrate complex computational and physical processes [171]. Comparable to a sensor/actuator node's control loop in the small (→ Section 1.1), computation in those larger host systems also influences the surrounding "world" and vice versa. While many WSAN designs still consider node failures as tolerable evil, CPS commonly demand for extremely reliable operation where malfunctions would cause serious consequences in application areas like highly confident medical devices, intelligent materials, advanced automotive, avionic and naval control (X-by-wire), as well as for defense and warfare systems. The divergent philosophies of the underlying design concepts impose demanding challenges when composing such systems to merge physical dynamics and computation. We'll highlight them next, and address several aspects throughout the text – a summary can already be found in Figure 1.7.

**Time management and time-awareness.** Embedded systems are most commonly real-time or time-aware systems, where a considerable amount of operations involves time as limiting factor (deadlines and timeouts) or at least as measurement variable and scheduling criterion for the execution of processes (worst case execution time, WCET) and interactions (worst case response time, WCRT).

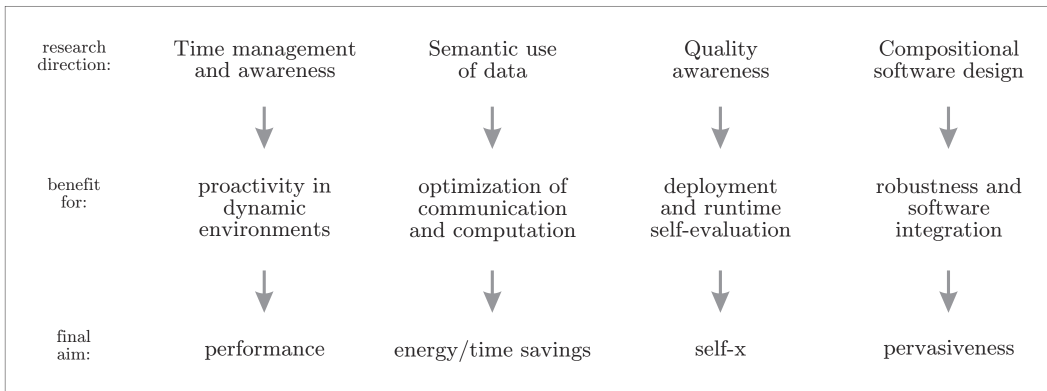



**Figure 1.7.:** WSN/WSAN research directions, benefits, and aims

However, the notion of time is missing in almost all WSAN operating system kernels. Though it is sometimes available at higher layers, e.g. as additional component for simple measurements and periodic triggers, it then suffers from imprecision caused by inter-layer communication and the simple fact that it is treated just like other components which are subject to unpredictable conditions: Conflicting tasks, variable execution delays, and resource-related or semantic limitations are just a few examples. Indeed, available OS concepts mainly focus on software abstractions and design patterns for modular applications and service-oriented programming; they do hardly cover the imponderabilities of dynamic environments but render the already complex activities even less manageable with regard to real-time requirements. While static WCET analysis is frequently applied for truly mission critical systems in order to optimize and guarantee their correct overall behavior, it is hard to accomplish on-line, and almost infeasible when considering the weak mote hardware compared to the complexity of the devices they control: Unknown system conditions and sporadic interrupts are just two common examples. Finally, any optimization effort is rapidly undone even by minor software and configuration updates or by modifications to the network's topology.

*For the future we need* to avoid non-determinism wherever possible. Finding new approaches or redesigning current ones to be more predictable will become inevitable. For many host applications, timing precision is relevant down to the smallest subsystem. Therefore, we should integrate time management at the lowest levels (within the kernel or even through appropriate hardware support), and must not let developers at higher levels try to compensate for the deficiencies at deeper levels. Of course, the kernel's time-awareness must be propagated up to the task level where the system API provides appropriate semantics for the user code.

**Compositional software design.** Complex systems do most commonly exhibit a modular structure, where various components are flexibly assembled, and, at least in the case of software, can even be exchanged at execution time as long as their interfaces are compatible.

While hardware prototyping is done at development time, mostly conforms to fixed interconnections, and can thus be statically tested for compatibility, software composition is much more dynamic. As one consequence, the latter introduces resource management problems which

result once more in timing violations, race conditions, priority inversions or even deadlock situations. These phenomena can also be observed in the SANet domain, where a multitude of (sometimes expandable) motes satisfy specific requirements. While the hardware operates quite stable, the software development still suffers from critical deficits, and embarrassingly the use of watchdog timers (WDT) for triggering hard resets in case of unsolvable failures is quite common. Although some operating systems provide very simple synchronization primitives in addition to inter-task-communication, it is still the developers who are responsible for their proper use and for surveying all potential interactions and execution flows. This does not only affect the directly visible task code, but also, and in particular, hidden device drivers, library functions, and communication protocols. In this context, task priorities – both explicitly defined by the developer or implicitly assigned by the OS – are yet another problem. Apart from the need for general configuration and functionality update mechanisms, this is one reason why reliable remote-management systems are required for managing the spatially distributed and sometimes hard-to-access nodes. Though such systems do exist for the partial or complete deployment of application software over-the-air, these do not cover the verification of the resulting or modified system. For partial updates in particular, the additional management effort (including e.g. dynamic address maps, jump tables, or code fragmentation) is still hardly researched.

*For the future we need* improved synchronization primitives for reflecting task priorities even under exceptional circumstances. Software modules should become aware of their influence on the system to react adequately and to resolve critical situations in a collaborative manner. After all, under real-world conditions there is no human intelligence around to restart an inevitably terminated task or even reset a node after a software failure. Finally, this problem will even increase along with the software integration density on each node, which is required to reduce hardware, deployment, and maintenance cost.

**Semantic use of data.** Distributed systems most commonly process information from various sources, whereby data dissemination, aggregation and fusion become central aspects. Closely related to reliability, energy and timing demands, these capabilities also account significantly for the overall WSAN service quality.

Wireless network and communication protocols represent one of the most popular research areas within the WSAN domain. Myriads of approaches consider static and dynamic strategies for information coding, collision prevention or avoidance, routing, and topology control. A special sub-problem is the explicit querying for information from the network, and the related reply strategy. For large systems with several sinks (e.g. robots in industrial environments) both the queries and the replies might interfere and annihilate each other sporadically, leading to reduced information availability and increased communication latency. Only little work is done for adjusting the number of replies implicitly, i.e. without querying the corresponding sensors directly and successively. Instead, embedded data such as locations, QoS demands or other context information within the query should be semantically exploited to select an appropriate subset of responding nodes, and to coordinate their transmissions for guaranteed success. Eventually, no more data than necessary should be transferred to reduce interference, energy consumption, and processing time at both the sender and the receiver side.

*For the future we need* to develop context-aware network protocols and communication schemes to maximize the fusion benefit, i.e. to obtain an adequate amount of information from a minimum of wirelessly transmitted data. Since communication cost is obviously not equal to fusion cost, appropriate techniques will definitely depend on the system state, and the current demand must be efficiently propagated to the nodes and respected by the network (see e.g. [47]).

**Quality awareness.** Physical environments are commonly highly dynamic and hard to assess over time, meaning that the technical systems therein must be robust against uncertainty and external disturbances. Affected areas include sensing and measurement, cooperation and collaboration, data fusion and decision making, deployment density and node failure, energy reserves and regeneration predictions – just to name a few.

Unexpected conditions and events are a worrying problem in general. To reliably capture environmental contexts despite of imprecise (though sometimes apparently consistent) and incomplete information, iterative processes of *control*, *perception* and *reasoning*[10] are often applied. However, this is a time-consuming approach during which the observed circumstances might not even remain constant. Where available, error characteristics are commonly included into the data processing step to filter outliers, speed up the computation, and to validate the (interim) results through plausibility tests. The quality of the acquired information is crucial for algorithms and self-x mechanisms with local relevance and global impact.

*For the future we need* to empower applications to be quality aware. In fact we have to provide dynamic indicators for the quality (e.g. reliability or precision) of the obtained data and information. This would facilitate the implementation of quality aware data fusion processes which are able to stop progressively improving iterations once a certain quality threshold (→ QoS) or timeout (→ WCRT) is reached.

**Synopsis.** The just postulated demands represent the author's opinion regarding the relevant research directions which already are and will still become even more essential for the success and further evolution of next-generation WSAN applications.

The integration of the wireless sensor/actuator network technology into other areas (related to e.g. the private, industrial, and military sectors) must still be extended, and will be a benefit for all parties involved. Furthermore, forming a symbiosis with still new and upcoming technologies (→ Figure 1.5) is an inevitable step: We definitely do offer the potential to introduce a new flexibility into existing processes, and conversely will receive invaluable input for discovering entirely new challenges and application areas. Of course the acceptance of a wireless technology which mainly relies on many but inherently weak and susceptible devices depends on the trust we can gain regarding the overall reliability and service quality of our hardware and software deployments. For sure, these demands can only be met through the clever and professional support for the points just discussed.

Figure 1.7 summarizes the necessary research directions, their benefits and goals on our way to spreading WSAN concepts. While various related points will be discussed, and novel concepts

[10] Here: The process of correlating events to contexts.

will be presented throughout the main chapters of this work, many other popular aspects like e.g. emerging behavior or biologically inspired computing are not within scope, and were entirely left for further research. See e.g. [68, 82, 88, 224, 275] for an overview of challenges.

## 1.3. Summary

In this section we introduced the most relevant definitions and paradigms for understanding the purpose and operation of wireless sensor/actuator networks in general. Regarding their characteristics and design space we identified related disciplines in computer sciences, but also separated these distributed systems from similar wireless networks. Finally the discussion of past, present and future goals illustrated the evolution of this still emerging technology, and also indicated the necessary research directions for future success – at least in the author's opinion.

# Part II.

# The Design of Time-Critical Sensor/Actuator Systems: Embedded Real-Time Concepts

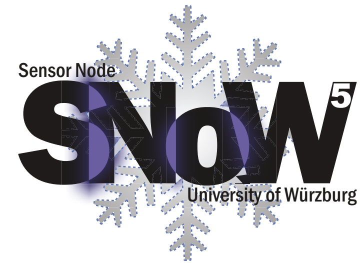
»Auch in Wissenschaften kann man eigentlich nichts wissen. Es will immer getan sein.«

---

*(Johann Wolfgang von Goethe, German poet)*

# 2. Hardware Platforms

## 2.1. Introduction




The miniaturization of powerful information processing technology is essential for its integration into daily life. Only small-sized and lightweight devices with little and almost unrecognizable impact on the environment in which they are deployed will be accepted by humans and other creatures within their operation range. Examples can be found in the area of location tracking (Active Badge [302]), in-situ habitat monitoring (Great Duck Island [193] and Skomer Island [220]), and animal behavior research [274]. It is quite similar for industrial applications, where tiny devices can be attached much easier and with less influence on the system under surveillance or control. Related examples tackle the monitoring of technical equipment within industrial plants and on board of ships [260] or aircrafts [309]. However, one must always take into concern, that physical requirements like form-factors and weight already lead to considerable limitations regarding these devices' resource variety, computational performance, and overall capabilities. In addition, economic, ecologic, and integration related constraints put further limitations on their design [55]. Table 2.1 gives an overview on relevant aspects.

Besides adhering to predefined specifications, the devices must still be able to operate autonomously, and to communicate and cooperate adequately for compensating their individually low performance by powerful distributed operations. Since the specific design obviously depends on application and environmental factors, the nodes might even be equipped or configured differently and form heterogeneous networks. In particular, we also distinguish between "*sensor nodes*" or "*motes*" for short, and "*infrastructure nodes*" or "*gateways*".

**Infrastructure nodes.** Since gateways are commonly installed within the infrastructure, they may be rather large and line powered, offer significantly better performance than ordinary motes,

| Constraint category | Design aspects |
|---|---|
| physical | size, weight, power consumption |
| economical | production, deployment and maintenance costs |
| ecological | environmental impact, pollution |
| integrative | communication interference, environmental robustness, accessibility |
| technical | on board components, expandability |
| performance | CPU/MCU architecture, memory |

**Table 2.1.:** Sensor node design constraints and design aspects

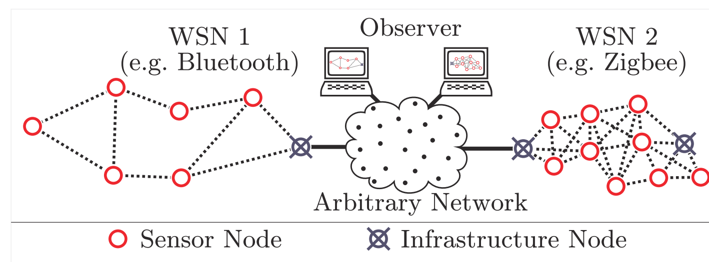



**Figure 2.1.:** Example for a network of heterogeneous nodes

and support various wireless and wired interfaces for multi-modality. If required, gateways are responsible for complex computations[1], the storage and forwarding of information between different networks (horizontal and vertical handover), and for the network observation and control through arbitrarily connected workstation computers. Figure 2.1 gives an example for a heterogeneous network comprising the mentioned node types.

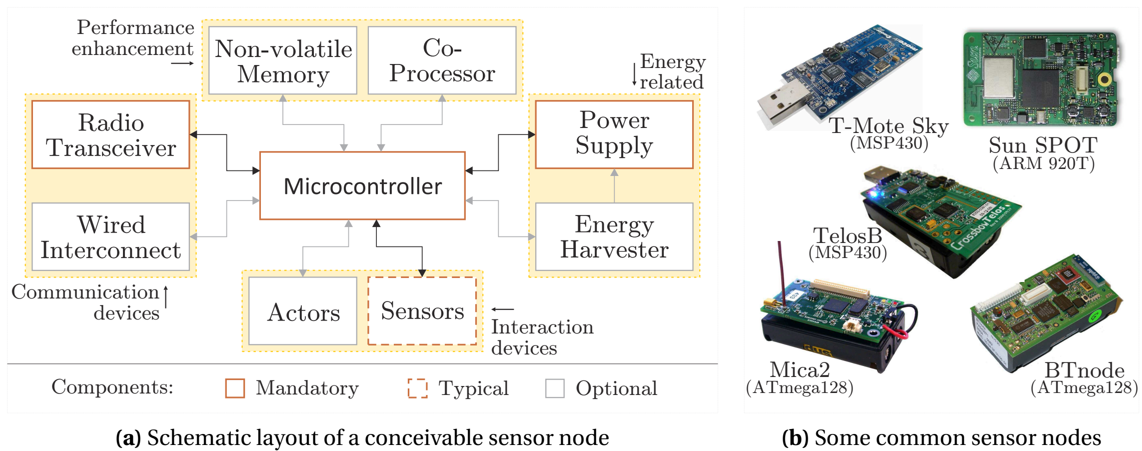
**Sensor nodes.** In contrast, the comparably smaller motes are often installed on objects or carried by creatures under surveillance. For flexible deployment within the environment and potential mobility, these devices are often battery powered or generate their required energy via sophisticated harvesting techniques (e.g. by transforming solar [93] or mechanical energy). Event though they commonly provide a low performance microcontroller (MCU) and just one wireless communication unit, it is their task to acquire information, perform some lightweight (pre-)processing, communicate with others if required, and initiate adequate reactions. Figure 2.2a shows the layout of a typical sensor node. Sensors and actuators – to which we refer as "*interaction devices*" – are selected according to the application scenario, and performance enhancing components (like DSPs or even FPGAs) can be added on-demand.

During the past years, myriads of sensor nodes have been developed [55, 99, 315]. Beginning with the Smart Dust platform [9, 148] in 1998, the nodes' usability was continuously improved, and until today several real-world installations were successfully deployed. While Figure 2.2b shows some common nodes, Table 2.1 gives a comparison regarding various design criterions. Among the most popular are the Telos [229] and Mica [74] platforms, the BTnode [45], EYES [297], Gumsense [198] and the ESB [105]. In general, motes can be characterized along a truly multidimensional configuration space. Again, Table 2.1 gives a comprehensive overview on the various design aspects. These also reflect the fundamental requirements of WSN applications – like e.g. energy and cost efficiency, networking, and scalability – as described in e.g. [4, 249] and the following sections.

## 2.2. SNoW[5] - The Sensor Node of Würzburg

Before heading to the main research topics *"reactive software design"* and *"indoor localization"* within this work, we'll introduce the design concepts and basic architecture of our specially

[1] which are most commonly provided as network services




**(a)** Schematic layout of a conceivable sensor node **(b)** Some common sensor nodes

**Figure 2.2.:** Sensor nodes: Schematic and examples

developed sensor node SNoW$^5$ (→ Figure 2.6). Designed as an energy efficient embedded system for wireless sensor networking, the SNoW$^5$ still reflects the current state of the art for typical sensor nodes. Yet, it offers more hardware flexibility while at the same time, it is not overloaded with speculatively required equipment. Instead, it employs carefully selected components featuring certain important advantages for our research attempts, and it is in line with our demand to build a totally understood system from scratch. In retrospective, the node's versatility turned it into a powerful tool for WSN research and educational applications. However, we'll just give a short conceptual overview here. See Appendix B and [31, 34] for a detailed description of the SNoW$^5$ platform, and Table 2.3 for an extensive comparison to other existing nodes.

### 2.2.1. Requirements and Design Goals

Since research and education are the main application areas of SNoW$^5$, we value flexibility, expandability, and convenient debugging higher than small dimensions and specialization. More specific designs for certain scenarios can still be derived from the existing one when required. On the other hand, overloading the device with a multitude of components like sensors and actors had to be avoided for cost and energy reasons. In addition, these would have made the device larger and take away its all-purpose character. In fact, we mainly focused on the following aspects when designing the SNoW$^5$ platform:

- compact design for reduced environmental impact within real-world deployments
- modularity for rapid prototyping and application-specific customization
- flexible wireless communication for efficient data exchange and node cooperation
- autonomous operation for robustness against temporary network and link failures
- convenient debugging and appropriate hardware support for remote maintenance
- energy efficiency for a long lifetime despite of weak power supplies

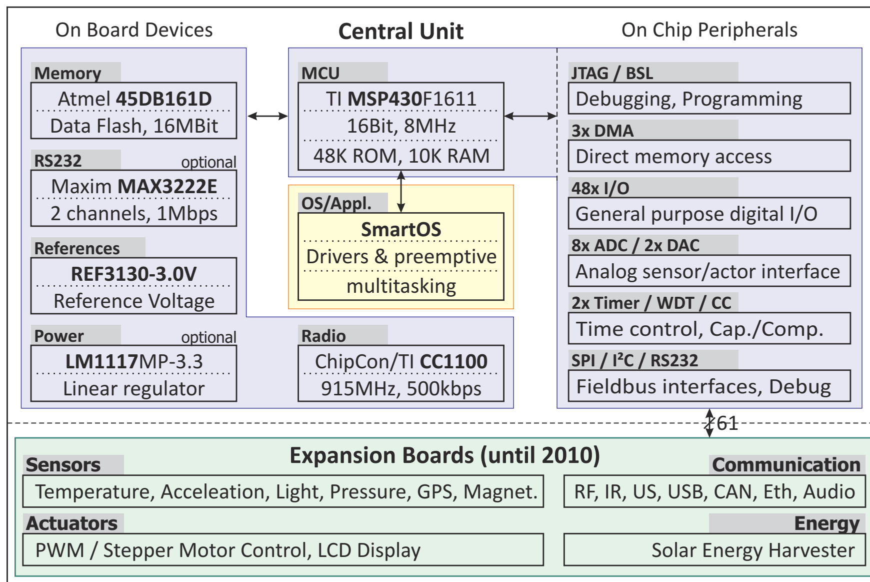



**Figure 2.3.:** The SNoW$^5$ platform overview

## 2.2.2. Hardware Components and Conceptual Properties

Having motivated the design goals for our versatile sensor node platform, we'll briefly review its most central hardware (→ Figure 2.3) to become familiar with the conceptual properties, and to establish the required understanding for the remaining work.

### General Design and Expandability

According to our demand for a flexible design, the SNoW$^5$ was designed as a mainboard carrying only the most important active components for stand-alone operation. Besides an MCU and a radio transceiver, we limit the equipment to few supplements which can be switched individually for energy aware operation: An RS232 bus driver, a reference voltage generator, and a power regulator. While we consider an extra flash memory as indispensable for autonomous long-term operation and over-the-air software updates (→ SNoW Ghost, Chapter 8.2), sensors and actors, in particular, were omitted entirely. Instead, we propagate a stackable design where such components can be easily attached through a complete set of header ports. This avoids the pre-assignment of valuable MCU pins to speculatively available devices, reduces costs[2] and energy consumption, simplifies maintenance, and preserves the option to mount interaction devices and energy harvesting related equipment in direct contact with the environment. Since most of the MCU's on-chip peripherals support external interfacing, these were completely made available for expansion boards. In addition to the 56 digital and analog signals, and the 2 power lines, three extra lines are freely available for sharing additional signals or power supplies across the expansions. The 2.54 mm raster facilitates rapid prototyping via standard bread boards, and simplifies signal observation for testing and debugging purposes. In particular, this

[2]Prototypes ranging around 50 EUR per node.

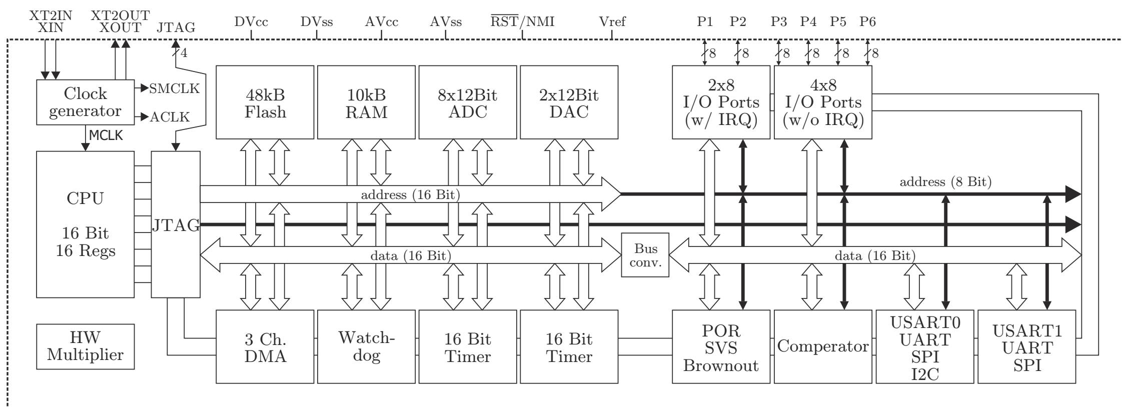



**Figure 2.4.:** Block diagram of the Texas Instruments MSP430F1611 MCU

can help significantly to analyze the timing behavior of reactive systems. Figures 2.7 and 2.8 show some examples of customized nodes with stacked expansion boards. See Appendix B for circuit details.

### On Board Components

**The microcontroller.** The central unit of a sensor node is its microcontroller (MCU). The SNoW[5] is equipped with an 16 bit ultra low power MCU from Texas Instrument's MSP430 family [279]. The selected device is the MSP430F1611 [280] providing 48 kB of flash memory (ROM) and 10 kB of RAM[3]. It supports five operation modes with energy characteristics ranging from 0.2 µA current consumption in low power mode LPM4 up to 75 µA in LPM0/1[4], and 4 mA in active mode AM at $f = 8$ MHz CPU frequency and $V_{CC} = 3.0$ V supply voltage[5]. While the clock frequency is generally still software adjustable between 32 kHz and 8 MHz via a rather unstable DCO (Digitally Controlled Oscillator), an external 8 MHz quartz crystal provides increased stability at the highest permitted speed[6]. This did not only prove to be the right decision when running time-critical and preemptive task systems on low performance hardware, but it also allowed to use a simple frequency divider for deriving a 1 MHz clock as required for the internal *Smart*OS time management (→ Section 5.3). Another reason for choosing the MSP430 was its large variety of on-chip peripherals as depicted in Figure 2.4. Since we wanted to design a general purpose node for research and education, these provide high flexibility for arbitrary applications. In fact, the anchor node software within the SNoW Bat indoor localization system from Part III makes use of *each* single peripheral (except for the watchdog timer). Finally, the MSP430 still receives considerable support from the open source community around the mspgcc C compiler [292], and it is reasonably priced to setup even large WSAN deployments.

[3]During the SNoW[5] design stage, the MSP430F1611 offered the largest RAM within the MSP430 family.
[4]LPM1 is activated by *Smart*OS when in idle mode.
[5]Though energy could be saved in active mode when operating at lower supply voltages down to 1.8 V, at least 2.7 V are required for flashing the ROM memory. Since we provide over-the-air programming via SNoW Ghost, we decided to stick to 3.0 V, since this will also keep the RS232 level converter active.
[6]See Figure 5.3[p75] for the crystal's characteristics.

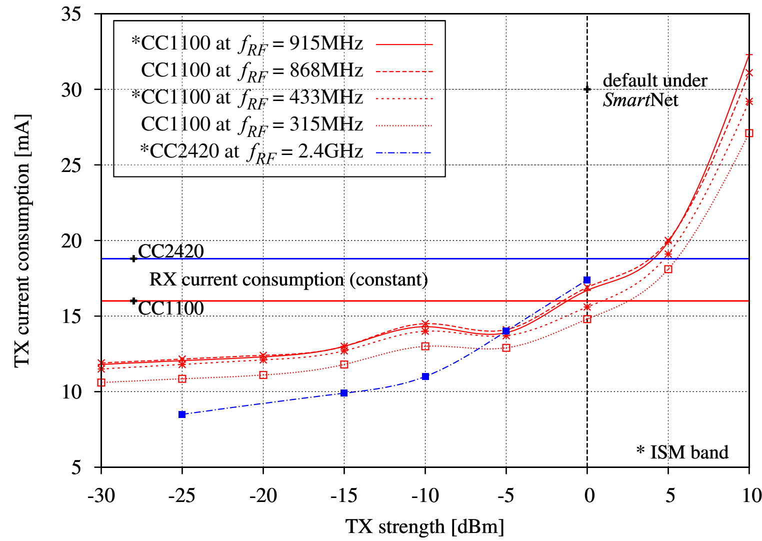



**Figure 2.5.:** The SNoW[5] radio transceiver current consumption at various base frequencies (→ [281, 283])

**The radio transceiver.** The second characteristic component of each sensor node is its wireless communication unit. Keeping the focus on research and education, we intentionally avoided a device featuring a specific transmission standard or MAC (Medium Access Control) protocol. Instead, we selected the Sub 1 GHz RF transceiver ChipCon/TI CC1100[7] [281], and preserved the option to freely develop novel approaches for the medium access and higher layers within the OSI (Open Systems Interconnection) model [140]. Compared to some other common RF units (e.g. the RFM TR1000 [243]), it provides a fully digital interface. Two 64 B rx and tx buffers allow for a packet and stream oriented communication with variable and unbounded packet lengths at data rates ranging from 1.2 – 500 kbit/s. Depending on the antenna circuitry, several base frequencies are supported; some within the ISM bands[8] for industrial, scientific and medical use. Figure 2.5 depicts the current consumption – which is of utmost importance for the node's lifetime and the network's reliability – in comparison to the widely used 802.15.4/ZigBee compatible transducer CC2420 [283] operating at 2.4 GHz. Besides various software selectable modulation formats, the CC1100 supports channel switching for flexible FDMA (Frequency Division Multiple Access), optional CCA (Clear Channel Assessment) for collision avoiding CSMA (Carrier Sense Multiple Access) protocols[9], and preamble-based low power listening. Its extensive configuration space and interrupt capabilities provide a convenient base for highly precise rx/tx timestamping and radio-based node (de-)synchronization [206] (→ Chapter 12). Considering the popular area of WSN based localization systems, RSSI (Received Signal Strength Indicator) measurements might even be used to estimate the distances between a sender and its receivers (→ Section 9.2, [192]). However, for the problems described in [259][10] we do neither use nor recommend this option.

[7] Texas Instruments Inc. acquired ChipCon AS in early 2006 and continued most of their low power RF solutions.
[8] The ISM band is license free since 1985.
[9] RX to TX switching time: ≈9.3 µs
[10] Diploma thesis conducted in conjunction with this work.

| Under test | Operation | MCU mode | Theoretical[1] [mA] | Measured[2] [mA] |
|---|---|---|---|---|
| CPU | active | AM | 4.3 | 5.2 |
| | idle | LPM1 | 0.3 | 1.0 |
| Radio | cont. rx | LPM1 | 16.2 | 17.4 |
| | cont. tx (0 dBm) | LPM1 | 16.9 | 18.1 |
| Data Flash | cont. read | AM | -[3] | 5.5 |
| | cont. write | AM | -[3] | 9.5 |
| SNoW Ghost[4] | radio rx / flash write | AM/LPM1 | -[3] | 19.3 |
| | flash read / ROM write | AM | -[3] | 23.3 |
| Ultrasound[5] | cont. rx | LPM1 | 2.5 | 3.4 |
| | cont. tx | LPM1 | 5.3 | 6.1 |

[1] Values taken from data sheets. Does only account for ICs, not for passive components
[2] Measured under *Smart*OS
[3] Unpredictable (depends on the duty cycle of the involved devices)
[4] The remote software update system as described in Section 8.2
[5] Involves the MICADUS extension board

**Table 2.2.:** Current consumption of various SNoW[5] sensor node components in mA (Node under test: ID 82, $V_{CC}$ = 3.0 V, $f$ = 8 MHz via ext. quartz crystal)

**The external flash memory.** To account for autonomous operation in case of transient communication failures, we added Atmel's 16 Mbit, non-volatile flash memory AT45DB161B [13] to the node. The device can help to avoid the loss of short-term information, and also allows long-term data storage, e.g. for logging and redundant saving of highly relevant information across several nodes. Related concepts can be found in [97][11]. Additionally, this component is required as firmware buffer for reliable over-the-air software updates (→ Section 8.2), since these images will commonly not fit into the node's RAM, and node resets during the update process – caused by e.g. malfunctions or power failures – would render the node inoperable. Further information about an appropriate embedded file system can be found in [190].

### Energy Consumption

Keeping the energy efficiency aspect from Definition I.4 in mind, the successful realization through our node design is at first hard to predict in absolute numbers and almost impossible to evaluate objectively for the general case. In fact, it depends on the system software [169], the individual system configuration, and various environmental conditions. However, considering the current consumption for some relevant operation modes, quantitative information can be given. Table 2.2 summarizes some measured values and compares them to the achievable best case as specified within the official data sheets of the involved components. Reflecting these values, two observations might catch the reader's eye: First, there is a fairly constant difference of ≈1 mA. This "loss" is caused by passive components (e.g. pull-up resistors), production related variations, and measurement imprecision. Second, the values might appear to be quite high compared to the specifications of some other nodes. How can this be explained, although many components are identical or at least comparable? In fact, this is a hardware/software co-design issue and caused by two facts: First, we obtained the values using the SNoW[5] default

[11] Diploma thesis conducted in conjunction with this work.

supply voltage of 3 V though 1.8 V would be sufficient for ordinary program execution. Second, *Smart*OS operates the MCU at its maximum CPU speed when in active mode ($I_{AM} \approx 4$ mA), and cannot go below low power mode 1 ($I_{LPM1} \approx 0.6$ mA compared to $I_{LPM4} \approx 0.2$ μA) when in idle mode to keep the sub-main clock alive. The reason is the continuously running *Smart*OS real-time clock (→ Section 4.3.4): With a resolution of 1 μs it is a quite power consuming but highly desirable feature as we will see in various sections within this work[12]. While other nodes claim to achieve significantly better energy values (e.g. 1.8 μA for the TelosB), they obviously apply lower clock frequencies, voltages, power modes, or duty cycles. Yet, we consider the higher power consumption legitimate for obtaining various important advantages like remote update capabilities[13], faster computations or reduced MCU duty cycles, and – most important for this work – higher timing accuracy for reactive and time-critical applications. Upgrading other nodes likewise would definitely also raise their energy consumption significantly.

A methodology for evaluating the power consumption of wireless sensor networks can be found in [266]; though based on the TelosB it can also be transferred to the SNoW$^5$ or similar nodes. Just to get an idea: For a simplified node model without energy harvester and an average current consumption[14] of $I \approx 1$mA, an initial charge capacity of $Q = 8700$mAh would last for

$$t = \frac{Q}{I} \approx \frac{8700\text{mAh}}{1\text{mA}} \approx 8700\text{h} \approx 1\text{y}. \tag{2.1}$$

### Summary

This introductory chapter described the most central design considerations along with some technical aspects and specifications of the SNoW$^5$ sensor node platform. Developed during the early stage of this work, these details gained considerable relevance for the real-world evaluation of the presented concepts within the remainder of this work since *all* test benches were carried out using SNoW$^5$ hardware as depicted in Figure 2.6.

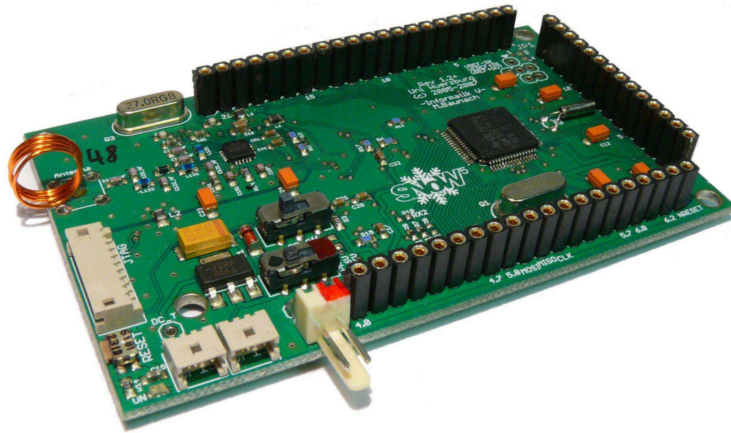
[12]Examples are precise event timestamping (→ Section 5.3), accurate ultrasound distance measurement (→ Chapter 11), reliably timed radio transmissions (→ Section 12.7), and representative benchmarking issues in general.

[13]Programming the flash ROM requires ≥ 2.7 V according to [280].

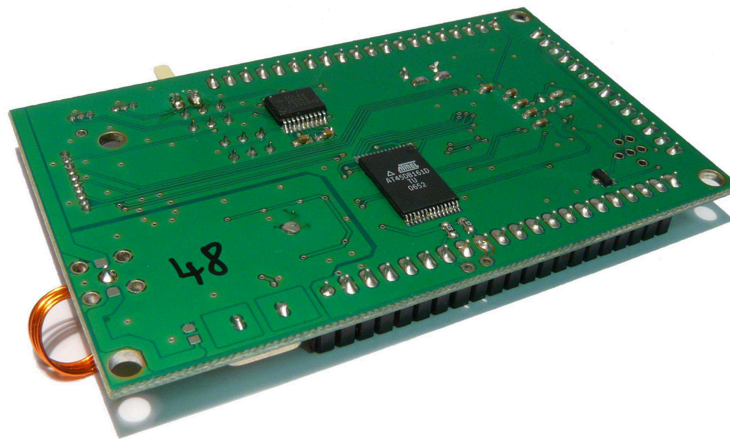
[14]Based on SNoW$^5$ values from Table 2.2 with duty cycles $D_{idle} = 0.9$, $D_{active} = 0.07$, and $D_{rx} = D_{tx} = 0.03$.

| Sensor Node | Mica2 | BTnode | ESB ScatterWeb | EYES | Telos | SNoW$^5$ |
|---|---|---|---|---|---|---|
| References | [74] | [45] | [105] | [297] | [229] | [31, 34] |
| Developer | Crossbow | ETH Zurich | FU Berlin | Univ. of Twente | UC Berkeley | Univ. of Würzburg |
| Price | 99 - 125 USD | 165 EUR | 98 EUR | ? | 99 USD | 60 EUR |
| *Microcontroller unit* | | | | | | |
| IC | ATMega128L | ATMega128L | MSP430F149 | MSP430F149 | MSP430F1611 | MSP430F1611 / F16xx |
| Speed (MHz) | 7.37 | 7.37 | ? | 5 | 0.4 - 8 | 0.4 - 8 |
| Architecture | 8 bit RISC | 8 bit RISC | 16 bit RISC | 16 bit RISC | 16 bit RISC | 16 bit RISC |
| Flash ROM (kB) | 128 | 128 | 60 | 60 | 48 | 48 |
| RAM (kB) | 4 | 4 | 2 | 2 | 10 | 10 |
| Power, active mode (mA) | 8 | 8 | 3.2 | 3.2 | 4 | 4 |
| Power, sleep mode ($\mu$A) | 15 | 15 | 1.6 | 1.6 | 2 | 2 |
| Wakeup time ($\mu$s) | 180 | 180 | 6 | 6 | 6 | 6 |
| *Onboard memory* | | | | | | |
| IC | AT45DB041B | 62S2048U | MC 24LC64 | ST M25P40 | ST M25P80 | AT45DB161B |
| Type | Flash | SRAM | EEPROM | Flash | Flash | Flash |
| Non-volatile | yes | no | yes | yes | yes | yes |
| Interface | SPI | ? | $I^2C$ | SPI | SPI | SPI |
| Size (kB) | 512 | 240 | 64 | 512 | 1024 | 2048 |
| Power, idle ($\mu$W) | 5 | ? | 0.03 | 150 | 150 | 5 |
| Power, read (mW) | 10 | ? | 0.15 | 12 | 12 | 10 |
| Power, write (mW) | 37.5 | ? | 0.3 | 45 | 45 | 37.5 |
| *Primary wireless communication* | | | | | | |
| IC | CC1000 | CC1000 | TR1001 | TR1001 | CC2420 | CC1100 |
| Interface | SPI | SPI | non-SPI | non-SPI | SPI | SPI |
| Data rate (kbit/s) | 38.4 | 38.4 | 19.2 | 57.6 | 250 | 500 |
| Modulation | FSK | FSK | OOK,ASK | OOK,ASK | O-QPSK | 2FSK,GFSK,ASK,OOK,MSK,QPSK |
| Frequency (MHz) | 433 | 433-915 | 868 | 868 | 2400 | 315, 433, 868, 915 |
| Hardware address check | no | no | no | no | yes | yes |
| Digital RSSI/LQI | no | no | no | no | yes | yes |
| Power, RX (mA) | 7.4 | 7.4 | 3.8 | 3.8 | 18.8 | 14.5 |
| Power, TX @ 0 dBm (mA) | 10.4 | 10.4 | 12 | 12 | 17.4 | 16.1 |
| Low power listen mode ($\mu$A) | 74 | 74 | 1800 | 1800 | – | 15 |
| Sleep mode ($\mu$A) | 0.2 | 0.2 | 0.7 | 0.7 | 1 | 0.4 |
| *Interfaces / Sensors / Misc* | | | | | | |
| PC Communication | RS232 | Bluetooth / JTAG | RS232 / JTAG | RS232 / JTAG | USB | RS232 / JTAG |
| Integrated sensors | no | no | yes | no | yes | no |
| Extension pins (incl. JTAG) | 51 | 55 | 24 | 14 | 16 | 67 |
| Accessible free Digital I/O | ? | 21 | 8 | 8 | 13 | 41 |
| Accessible free ADC ports | ? | 2 | 0 | 8 | 6 | 8 |
| Accessible free DAC ports | 0 | 0 | 0 | 0 | 2 | 2 |
| Accessible buses | SPI, $I^2C$ | SPI, $I^2C$ | SPI | – | SPI, $I^2C$ | SPI, $I^2C$ |
| Accessible DC ports | 1 | 1 | 1 | 1 | 1 | 1 + 2 (free for expansion) |
| Recommended OS | TinyOS | – (TinyOS) | – (TinyOS) | PEEROS | TinyOS | *Smart*OS (FreeRTOS, TinyOS) |
| *Overall DC / physical specifications* | | | | | | |
| Size (mm× mm) | 32 × 58 | 32 × 58 | ≈ 45 × 54 | ≈ 32 × 92 | 32 × 65 | 50 × 85 |
| Supported operation voltage (V) | 2.7 - 3.3 | 3.3 or 3.8 - 5 | 3 - 26 | 3 | 1.8 - 3.6 | 1.8 - 20 |
| Regulated supply | no | yes | yes | no | no | yes |
| Power, active mode (mA) | 30 | ≈ 33 | 12 | ? | 14 | 8 |

**Table 2.3.:** Comparison of some selected wireless sensor nodes

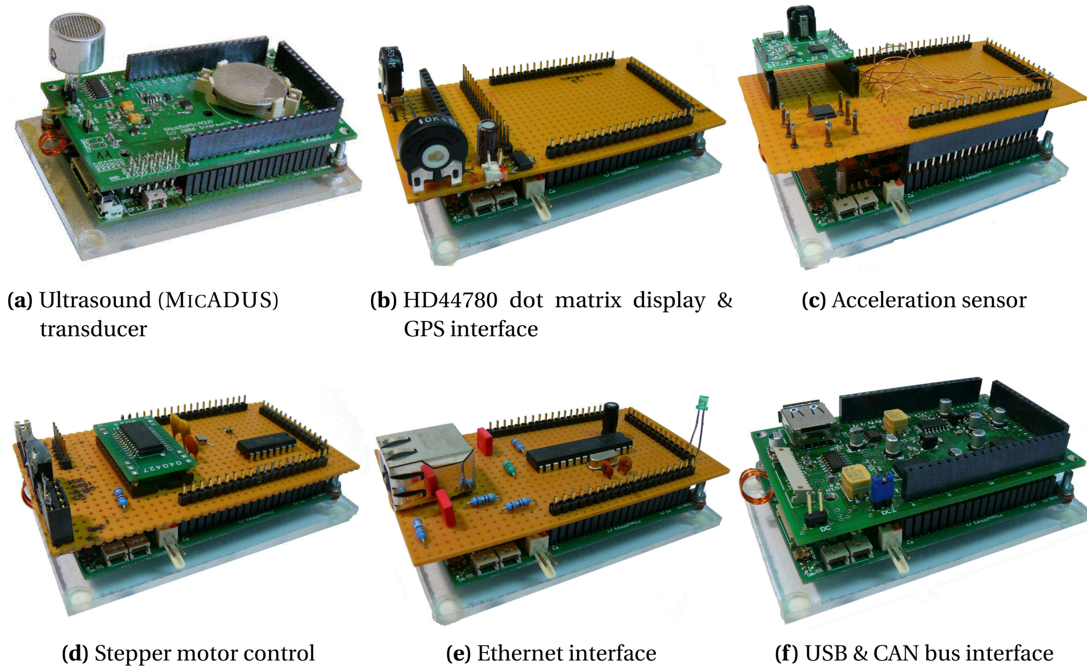


**(a)** Top view



**(b)** Bottom view

**Figure 2.6.:** The SNoW$^5$ mainboard (approximately in original size)



**(a)** Ultrasound (MICADUS) transducer

**(b)** HD44780 dot matrix display & GPS interface

**(c)** Acceleration sensor

**(d)** Stepper motor control

**(e)** Ethernet interface

**(f)** USB & CAN bus interface

**Figure 2.7.:** The SNoW$^5$ sensor node with various extension boards

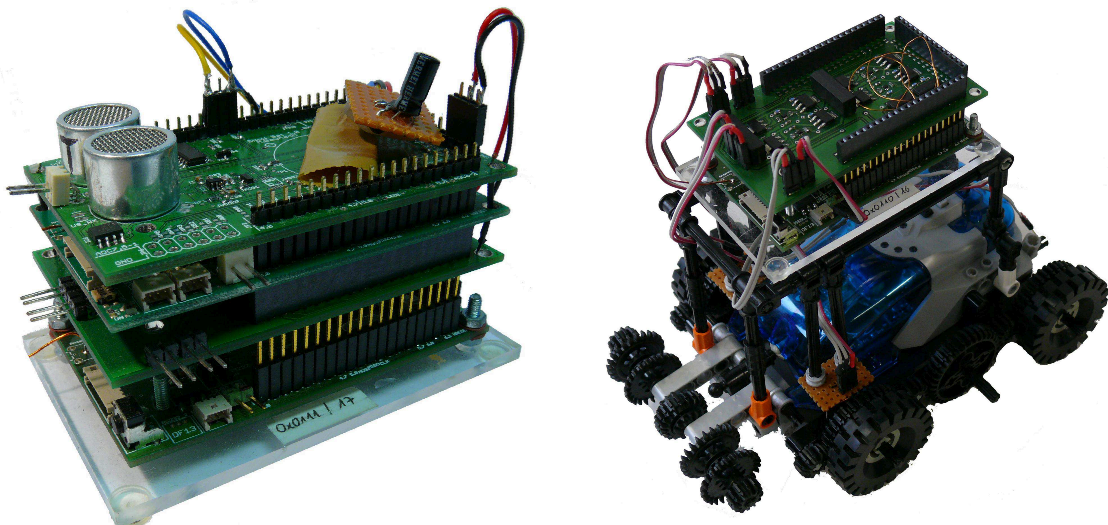


**Figure 2.8.:** The SNoW$^5$ sensor node equipped with a PWM motor controller and the MICADUS ultrasound extension (left), and mounted on a modified LEGO Spybotics vehicle (right)

# 3. Operating Systems

**Abstract**

Sensor/actuator networks often consist of a more or less large number of cooperating nodes handling even complex objectives. Even though these embedded systems are subject to low computational performance, tight memory, and severe energy constraints, it is nevertheless necessary for some applications to support the execution of "multifunctional" software and "time-critical" operations within e.g. control and feedback control systems.

Up to a certain complexity, an endlessly repeating (periodic or event-triggered) single-loop implementation which successively steps through the various operation stages might be sufficient for truly small systems. For more elaborate demands however, such an approach would hardly be comfortable since it is extremely inflexible and commonly results in extreme CPU load and energy waste – at least if it has to check and process various (external) conditions over and over at high frequency. Even worse, the response time of such single-loop systems is quite high, and the program code is hard to maintain, extend, or reuse.

Implementing several of those loops for "parallel" execution is one obvious solution to decompose the application logic into smaller parts which are less overloaded and more convenient to handle. This is where operating systems (OS) attempt to provide efficient and suitable solution concepts, and take significant influence on the performance of the overall device: At the lowest software layer – right on top of the hardware – they coordinate the application execution, the interaction between software components, and their synchronization regarding any access to operational resources. Apart from these "local" responsibilities further demands might arise from the distribution of applications over several nodes.

This chapter deals with the main challenges and objectives to be considered when designing operating systems for reactive embedded systems in general, and for sensor/actuator nodes in particular. We'll also present a classification framework for kernel architectures and present some well-established representatives before the next chapter finally introduces our novel solution: *Smart*OS.

## 3.1. Introduction

Once monolithic information processing systems reach a certain complexity and are not satisfactorily manageable any more to guarantee their reliability in any situation (a problem which can already be critical for small systems), they are commonly broken down into so called modules, which can be developed independently from each other. The motivation for this *decomposition* is exceptionally manifold, and so are the usually high expectations to this strategy and the resulting problems regarding their realization[1]. Though details will be addressed in later sections, it is quite obvious that modules, whatever they might do in detail, must finally be reassembled to a complete system – the so called application – operating within its specifications. While this *composition* of modules is usually planned and carefully checked for consistency and compatibility at development time, their coordination and interaction at runtime can be interpreted as a central objective of any operating system. The exact range of features required therefore, can hardly be defined in general: The various application types and scenarios impose specific demands for which not all operating systems might be equally qualified. In this respect, the so called "operating system architecture" is of central relevance. It defines the basic operation of its most central components – the so called kernel – and consequently introduces some kind of philosophy for the development of all modules based on it. Ensuring the correct behavior – both with each other and with respect to the environment – is then the responsibility of the operating system.

The literature describes various demands on operating systems which already differ in their most inherent components. While purists see only the most elementary low-level functions for process-scheduling, and maybe for inter-process communication (IPC) as well as for synchro-

[1] It should already be mentioned, that the principle of modularization will neither simplify the system design nor its implementation unreservedly. Usually only a shift or hiding of some problems occurs: There is nothing for free!

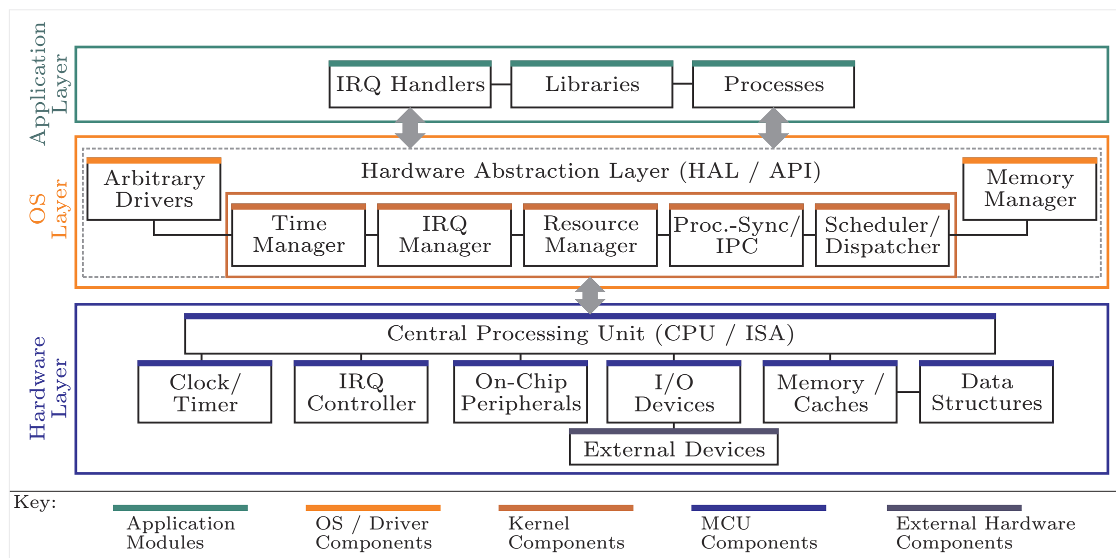



**Figure 3.1.:** Reference architecture for sensor node and embedded systems hardware and software layers

nization, others suggest file systems, networking capability, update functionality, and similarly complex mechanisms as native operating system components. Therefore, we first clarify the terminology and distinguish strictly between the terms *operating system* and *kernel*:

---

**Definition II.1: Kernel**
As *kernel* we denote exactly those mechanisms of the operating system, which are mandatory for the philosophy-compliant coordination of activities on each supported hardware. They provide the basis for any compatible application and are thus always available and identically defined.

---

Conversely, kernel-implied requirements should never be bypassed or ignored[2]. We intentionally just *demand* to satisfy the requirements; whether a violation can be detected or even prevented at runtime often depends on the available hardware (e.g. a memory management unit), and would consequently imply an excessive restriction on the portability.

---

**Definition II.2: Operating System**
As *operating system* (OS) we denote the complete functionality which the same provides for running applications. It integrates the kernel as central component, but may also comprise advanced mechanisms – if necessary even with varying and platform-specific characteristics and API.

---

An operating system may include methods for runtime analysis, energy management, cross-system communication, etc. Regarding the performance and memory requirements these are often available on-demand, and may even be replaced by application-specific variants. See Figure 3.1 for a reference architecture regarding hardware and software layers.

For the design of an operating system architecture, we distinguish three main perspectives on its core competencies: On the one hand, the OS should provide an abstraction of the hardware so that arbitrary application code runs without any special adaptation[3]. On the other hand, the OS should coordinate the coexistence and execution of modules so that they can interact while not affecting each other in an uncontrolled way. Ultimately, the OS should ensure a fair execution of these modules so that they get sufficient resources to meet their specifications. The related problems will be addressed next.

## 3.2. Operating System Objectives

**The operating system as hardware abstraction.** The hardware components within computer systems are extremely versatile both in their internal structure and function, and their interconnection and communication among each other. Although their interfaces[4] are specified

[2] Violating this philosophy by ignoring or circumventing the OS specifications, may rapidly lead to unmanageable side effects and invalidates the benefits of an operating system.
[3] apart from re-compilation
[4] e.g. communication protocols, instruction sets, timing behavior, etc.

by the manufacturer, an enormous effort would arise if the support for each component would have to be implemented from scratch for all applications.

If software modules access hardware components exclusively via OS functions (and encapsulated drivers), this hardware abstraction layer (HAL) offers further convenience and clarity through platform independence and standardization. In this case the OS represents some kind of virtual machine (VM) which is customized for the actual hardware and offers a unified view on the underlying systems and components. However, this demand involves significant problems for the OS architectures, and the solutions are commonly hard to reconcile. On the one hand, the provided functionality should be constant and independent from the OS port and the hardware equipment. On the other hand, even individual hardware features need to be respected and exploited for increased functionality and performance. Considering both, the system must remain correct while being small, fast, flexible, and simple to maintain and port. Just consider the communication between two computer systems: For the transfer of data to a certain destination, unified functions like `send(dst,data)` allow for compact programs, and also provide an enormous comfort. Apart from a standardized syntax for "important" functions (e.g. according to POSIX [133]), compatible programs show comparable behavior[5] and do their work invisibly for their caller. In the case of `send`, the application profits from an intuitive access on complex networking techniques. Success or failure is indicated by an also standardized return value.

In consequence, we need to question the importance of each potential feature when designing an operating system architecture: Which functionality should be integrated directly into the kernel, and in which concern will it affect the OS philosophy and the application development? Which extensions must always be available, and which ones can be added on-demand? Is it advisable to locate them in libraries and to provide them through drivers? Will components need to interact or depend on each other? In fact, the previously unconsidered question about when and how long modules may access these components must also be resolved. Thus, operating systems do most commonly also coordinate the (dynamic) access of concurrently operating modules to resources.

**The operating system as arbiter and resource manager.** Resources constitute the working basis for all programs which are executed by a computer system. Beginning with the central processing unit (CPU), which also executes the operating system and the resource manager[6] itself, we generally divide resources into four types:

- *Processors* for the execution of instruction sequences (→ Section 4.3.2),
- *Peripherals* like I/O devices, timers, data sources and sinks (→ Section 4.3.7),
- *Memory* and data structures therein (→ Chapter 7), and
- *Virtual resources* for process synchronization (→ Section 4.3.7).

[5] Since compatible systems are not necessarily identical, even slightest differences can lead to varying runtime results, in particular regarding the temporal behavior. Sometimes, the OS even has to compensate for deficits which are hidden in the instruction set of the used processor.

[6] Though the resource manager can also be implemented at application layer for highly customized requirements, we assume it to be an OS kernel component.

| Availability | Assignment | Sharing | Management | Access | Duration |
|---|---|---|---|---|---|
| physical ●● | shared ●● | temporal ●● | static | preemptive ● | short-term ●● |
| virtual ● | fixed | spatial | dynamic ●● | exclusive ● | long-term ● |
| | | | | cooperative ● / collaborative ● | |

Key: *Smart*OS specific: ● CPU ● other resources

**Figure 3.2.:** Resource classification framework: In general and under *Smart*OS

Independent from their type, resources are always distributed among the current set of concurrently operating processes, but may only be used when granted by the resource manager. This policy synchronizes the modules, and introduces mutually exclusive resource access (r/w) and reservation to avoid race conditions.

According to the classification framework from Figure 3.2, resources can be *physical*, meaning that they exist as real hardware components, or *virtual*, meaning that they are just emulated by the resource manager. We classify them as *shared* if their owner or user process may change at runtime, and as *fixed* (e.g. during development or system startup) otherwise. For shared resources we distinguish between *temporal sharing*, meaning that no more than one process can access the resource at the same time, and *spatial sharing*, meaning that several instances of a resource are available and can be used simultaneously. Additionally, the management of shared resources among the particular processes is either done in a *static* manner, i.e. by a predefined schedule, or *dynamically*, i.e. upon request by the processes at runtime. Then, the access is either *exclusive*, meaning that no other process can access the resource as long as it is allocated by its current owner, or *preemptive*, meaning that the resource manager can temporarily withdraw the resource and assign it to another process as long as the initial owner's operation is not compromised. While preemption is commonly done by saving a resource's state before the handover and restoring it afterwards, exclusive resources must be handed over voluntarily by the processes in a *cooperative* manner. The novel DynamicHinting technique for *collaborative* resource sharing is introduced in Chapter 6. Finally, we'll distinguish between *short-term* and *long-term* allocation. While the first means that a process does not suspend itself while holding the resource, the latter permits self-suspension. The distinction will become relevant when dealing with priority inversion and deadlocks (→ Section 6.3.1).

**The operating system as scheduler.** The CPU takes a special role among the resources since it executes not only the application processes but also the operating system itself. According to our classification it is a shared resource. It is temporally shared if it provides just a single core, and even the support for preemptive processes yields no truly parallel execution then; kernel and processes have to alternate and interleave appropriately. In case of several cores, spatial sharing allows for true parallelism of processes and the kernel[7]. In spite of rapidly advancing research in the field of multi-core processors [222, 287], we'll focus on single core architectures and simply denote these as CPU or processor. The reason is their still persisting dominance in the field of embedded systems and sensor networks in particular. Besides, the demand for low

[7] Note, that each core is again temporally shared, and that the kernel may even use a dedicated core.

costs despite the large node quantity and energy efficiency for a long runtime, this decision is also encouraged by their generally lower complexity and smaller size (→ Section 2.1). Finally, when used efficiently, even sensor nodes with simple MCUs are sufficient for most current applications.

Apart from a solid software design, this very efficiency is largely the responsibility of the operating system and its applied strategy for interleaving processes on the CPU – the so called *scheduling*. The scheduler is a central part of the kernel, and ensures its own execution and the execution of the processes. Just like the selection of the application code to execute, the amount of CPU time which is provided for each process depends on the process set and on external events. In this concern, the interaction with the environment introduces significant dynamics, since many events have to be treated sporadically, and at unpredictable points in time. If reactions have to take place within a certain deadline, the sudden competition for resources (including the CPU) can lead to impairments of the desired temporal behavior. Depending on the operating system philosophy several approaches are available for this problem. While some ignore the problem entirely, others are optimized for fairness, or focus on real-time capability and fast response times. Many rely on prioritized processes, and also support the withdrawal of preemptive resources for the benefit of higher priority processes. Here, the CPU is treated significantly different from all other resources. While the latter must be allocated explicitly prior to use[8], the CPU is always implicitly made available – maybe due to a previously defined trigger. If no application process is ready to be executed, a special idle process is commonly selected.

## 3.3. Operating Systems for Embedded Systems and Sensor/Actuator Networks

Operating systems for WSAN and embedded systems differ significantly from conventional operating systems for desktops and servers: While *desktop operating systems* aim on large application diversity, multimedia, and convenient user interaction via graphical interfaces, *server operating systems* focus on high availability, security, and throughput for queries and data transfers. Both operate in direct contact and under maintaining surveillance of humans to detect and fix failures as these appear. In contrast, *embedded operating systems* operate without permanent human inspection. At the same time they often drive a ubiquitous technology which affects our daily life even in the most critical situations. Just like the underlying hardware, they are also characterized by resource-aware and maintenance-free operation along with specific (and sometimes verifiable or provable) guarantees for their correctness, reliability, failure-safety, reactivity, and real-time capabilities. As a subset, *networked embedded operating systems*, e.g. for the WSAN domain, face a specific problem: They have to coordinate the cooperation of initially independent, loosely coupled, and autonomously running node systems, and thus need to synchronize their common objectives via more or less reliable links to achieve pervasiveness. Additionally, they need to integrate several software subsystems to reduce the costs and effort for deploying and maintaining service specific hardware and infrastructures.

[8]Though this might not be visible within the application code: For example stack memory modifications for the declaration of local variables are commonly handled directly in the instruction set.

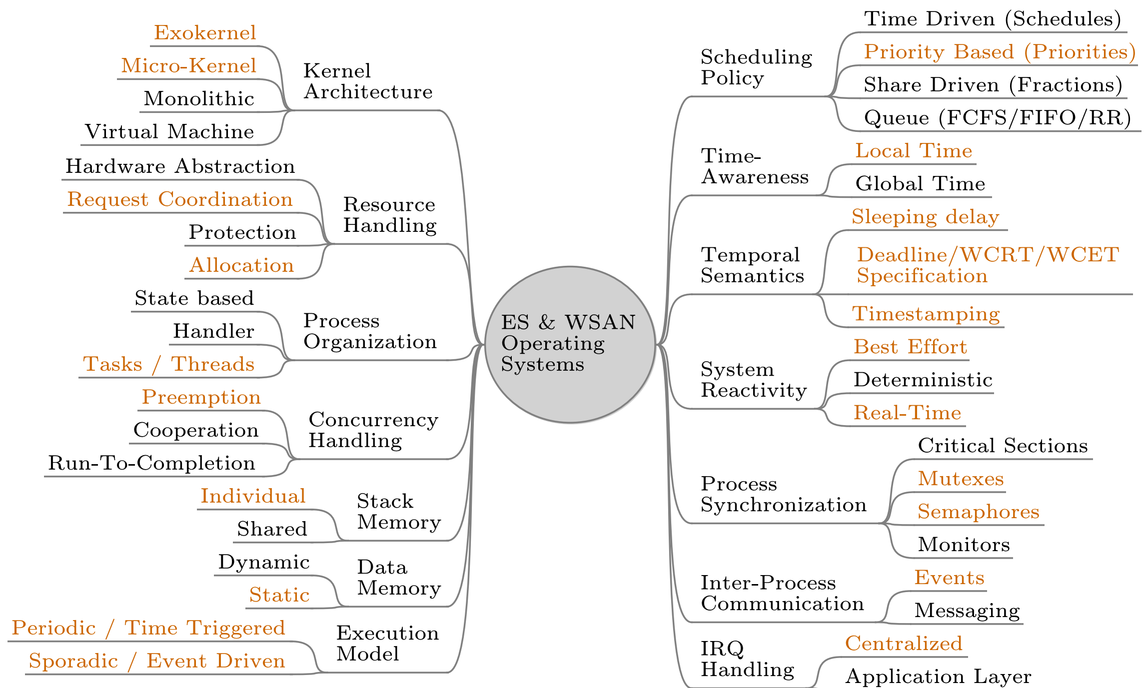



**Figure 3.3.:** Selected classification features of operating systems kernels for embedded systems and wireless sensor/actuator networks (*Smart*OS characteristics are highlighted).

Ultimately, the most demanding challenge when designing a (networked) embedded operating system for sensor nodes is to reliably cope with severe resource constraints [9] for the composition of concurrent software subsystems, and the operation within dynamic environments (under unreliable communication and information exchange).

## 3.3.1. The Kernel Classification Framework

To comprehensively understand the complexity of hardware and software systems, and to compare various operating system kernels in the context of wireless sensor/actuator networks, we'll first introduce an appropriate classification framework for the most central kernel features. Therefore, we focus on the reference architecture from Figure 3.1, but limit ourselves to some selected characteristics from Figure 3.3. Similar, though significantly coarser or more network-related classifications can also be found in [237, 275, 301, 314].

**Kernel architecture and hardware abstraction** constitute the basic operating system structure and philosophy:

*Exokernels* like Nemesis [294] strictly separate the protection of kernel-irrelevant resources from their management [94]. In particular, the resource-related coordination of concurrent processes is not supported by the kernel. While the latter provides only the resource allocation mechanisms, their operation and sharing is entirely managed at higher layers. Since the kernel

[9] e.g. CPU performance, memory, peripherals, energy, ...

consequently offers no resource-specific hardware abstraction (apart from the CPU management), this must also be accomplished by the application. As a benefit, this reduces inter-layer communication and makes even untrusted applications as powerful and efficient as the kernel itself. An even more strict variation of this concept are *nanokernels* like the Nano-kernel [17]. These outsource even the most basic functions, such as the timer control, to the application. In contrast, another less integrative variation are *resource kernels* [238] like Nano-RK [95]: Though these also separate resource protection from management, they commonly guarantee the allocation of requested resources within defined time limits.

*Microkernels* like SOS [120] directly incorporate abstractions for several hardware components, and provide related services to the higher layers. This includes scheduling and synchronization primitives like inter-process-communication (IPC) and resource sharing. Other features range from time and dynamic memory management to inter-system communication.

*Monolithic kernels* provide a complete hardware abstraction, or even execute all device drivers in kernel mode. In the WSAN domain such approaches are rarely used, since their complexity is often too demanding for low performance microcontrollers.

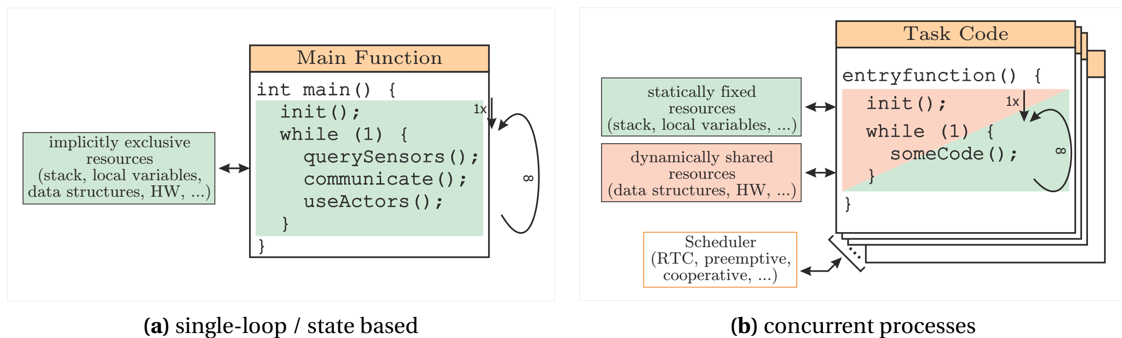
**Process organization, execution model, scheduling, and memory management** form the basis for the application design:

The *process organization* defines whether the application code is implemented as either a single state machine with externally triggered transitions, as a set of handlers which will be executed on-demand to react on external and peripheral events, or as a set of concurrent processes or threads which coordinate their execution by synchronization primitives. Since we consider strictly handler-based operation or even single-loop approaches as not satisfying for reactive platforms, we prefer individual tasks for each logical job, and put them together for the final application as depicted in Figure 3.4.

For *concurrency handling*, we distinguish between preemption, meaning that the scheduler may switch between processes at any time, cooperation, meaning that each process must release the CPU voluntarily, and run-to-completion, meaning that a process can neither be preempted nor preempt itself but runs until it terminates. For the presented options, the processes may either share a single *stack* area or possess their individual stack area each. The *data memory* can either be provided dynamically at runtime or assigned statically at compile or linking time.

The *execution model* is separated into periodic code execution (time triggered), which is most commonly scheduled and tested for feasibility at development time, and sporadic code execution (event-driven), where events trigger the execution. Event-driven programming is very popular for small embedded systems. It does not only save energy by limiting code execution to the event occurrence, but also gives the impression of compositional designs by supporting several simultaneously active event handlers (functions or processes) for distinct reactions. Handler functions are commonly atomic[10] and consequently profit from several advantages: Besides the option to save memory by sharing a single stack, the concurrent access on exclusive resources is precluded – at least if these return to a consistent state at the end of the handler –, and switching between handlers is rather simple since contexts need not be saved. Nevertheless,

[10]In some systems, special privileged parts of the kernel may still interleave the handler executions.




**(a)** single-loop / state based

**(b)** concurrent processes

**Figure 3.4.:** Comparison of process organization and implementation approaches

this method also imposes some drawbacks: If handlers are completely stateless, resources cannot remain allocated throughout several calls, and local variables must be saved globally to make the information persistent. Finally, the handlers must be serialized in an appropriate way: If their executions would overlap, unpredictable delays might occur. The last issue is addressed by the *scheduling policy*. Time driven [144] systems operate with respect to a schedule, priority-based systems account for specific priorities, and share driven systems provide a certain fraction of CPU time for each process and resource. Of course, the schedules, priorities, and fractions can be selected both statically (fixed) and dynamically (on-line), and combined with queues for serialization.

**Time, reactivity, process synchronization, and interrupt handling** are of central importance for time-critical systems:

*Time-awareness* is a crucial point when interacting with physical systems. While many applications need no explicit notion of time, but are simply required to execute "as fast as possible" on the particular hardware or possibly seek to meet certain timing constraints through careful implementation, others rely on time as inherent metric for handling system dynamics. These may even require some kind of *temporal semantics* for the specification of delays, deadlines and response or execution times, for the taking of timestamps, and for the scheduling of reactions. Therefore, real-time kernels commonly provide a system-wide local time which then yields a significantly higher precision than implementing a comparable service at higher layers (→ Section 5.3). For networked systems in particular, this may even be extended to inter-system synchronization and the provisioning of a network-wide global time for the consistent attribution of events and gathered information (→ Chapter 5). Considering the *system reactivity* on external and internal events, the temporal precision plays an important role. While best effort solutions need not care about time but react as soon as possible (ASAP), deterministic approaches can specify a precise interval for the delay. This is relevant for the design of hard/soft real-time systems where time frames must/should be met in order to assure a correct operation.

Besides the pure notion of time, accurately timed reactions can also depend on the exclusive access to shared code and resources, as well as on the proper cooperation of all involved processes: The *process synchronization* can be managed by the kernel by providing well-known

primitives like critical sections[11], mutexes, semaphores, and monitors [276]. Combining these with priority-based scheduling enables task collaboration as a novel concept for improved reactivity (→ Chapter 6).

While inter-process communication becomes feasible through event and messaging passing (e.g. pipes, publish/subscribe patterns, shared memory, etc.), external events are signaled through interrupts. These can either be processed by a centralized handler within the kernel, which operates the IRQ controller as a fixed resource and demultiplexes the incoming events by triggering the corresponding processes, or by specific handlers provided by the application.

**Further characterizations** may refer to energy management [169], error handling and self-diagnostics, verifiability and simulation, inter-system communication, portability, etc. However, these are out of scope for this work, and won't be addressed here. Remote system maintenance and software update strategies will be addressed in Section 8.2.

### 3.3.2. Related Work and State of the Art

Before we start over with *Smart*OS, we take a closer look at some existing embedded operating systems. Due to their enormous number, we limit our survey to the most popular ones in the sensor networking domain, and to those which support similar features than *Smart*OS: These comprise e.g. preemptive multitasking, extended timing capabilities, and dynamic resource management. Though a sharp classification according to our framework is hard because of very specific techniques and hybrid characteristics, a comparative overview of *Smart*OS and some other relevant operating systems can also be found in Table 3.1.

**TinyOS** [128, 129, 174, 175, 291]. Developed since 1999 TinyOS was the first operating system designed for the special demands of wireless sensor networks. Its central architecture supports a component based system design with each component being a computational entity comprising tasks for the actual computation, commands for triggering non-blocking operations, and events for signaling their completion. Applications are typically developed in nesC [109], a C-like language with some restrictions to the C standard on the one hand, and some extensions for component interactions on the other hand. The source code is finally compiled and linked into a static binary for direct or remote deployment. Though runtime modifications are rather hard to accomplish, some approaches are described in [215] and [197].

Three execution classes are supported for application code: asynchronous interrupts, synchronous tasks and preemptive TOSThreads. For the tasks, TinyOS uses an event based FIFO scheduling with run-to-completion (RTC) semantics. In fact tasks can be *posted*, i.e. appended to a queue of tasks waiting for execution, and, once started, run until their self-termination. This simplifies the implementation of the dispatcher, and also allows the use of a single shared stack for all tasks. Tasks contain synchronous code and can only be interleaved by asynchronous code (which is triggered externally) to post tasks for event handling; the handling delay, however, is

[11] While a *critical section* can only be executed by at most one task at a time but may be interrupted or interleaved by interrupt handlers or other tasks, an *atomic section* cannot be interrupted at all.

unknown and depends on the current queue state. This is exactly why TinyOS is hardly suitable for time-critical software [52], but needs severe application design efforts to achieve acceptable reactivity and fair task interleaving. An option is to apply the so called split-phase design: Long running computations must be split manually into smaller tasks for subsequent execution, to manually avoid missing deadlines or events such as incoming radio packets. Additionally, tasks can be enqueued only once at a time, which makes the maximum queue length known at compile time, but reduces flexibility, and complicates scheduling predictions. Though the scheduler is a component and can be replaced since TinyOS 2.x, only few alternatives exist. A proof of concept for EDF scheduling is proposed in TEP106[12], and preemptive scheduling is addressed in e.g. [83][13]. Since TinyOS 2.1, so called TOSThreads [158] allow cooperative multi-threading, but are simply put on top of the initial task model to stay backward compatible. TOSThreads use dedicated stacks, and can be preempted by each other and by ordinary tasks. As long as no such task is executed, a round robin scheduler with fixed time slices (default: 5 ms) alternates their execution. Apart from the missing resource allocation and sharing primitives, this hierarchy does not necessarily improve the system reactivity, but mainly reduces the splitting of computationally complex operations. To keep the kernel atomic all syscalls from within threads will post a special task which in turn will preempt the thread, execute the syscall, and return to sleep to continue the thread. This is done to ensure that only the kernel thread (which is a novel encapsulation compared to standard TinyOS) will execute kernel code.

Other features like dynamic memory or time-awareness are not supported within the kernel, but must be added at component layer when required. A comparison of TinyOS and *Smart*OS regarding their performance and reactivity under various conditions can be found in [52][14].

**Contiki** [85–87]. Initially developed in 2002 for the Commodore C64 Contiki is nowadays known as operating system for small embedded devices. Its design allows the modification of application code by runtime replacements, and the loading and unloading of modules. New code will be linked dynamically for native execution without any VM-like approach.

The kernel offers a lightweight scheduler for synchronous and asynchronous events, which, once triggered, will run-to-completion. In turn, these events trigger the execution of process code on a common shared stack. Processes can be considered as either applications or services, and include event handlers and poll handlers. While synchronous events will trigger the target process and preempt the invoking process immediately, asynchronous events will be enqueued and processed once the target process is scheduled anyway. Between asynchronous events, the kernel calls all registered poll handlers to e.g. support polling based software. Since events can be interleaved by interrupts (which are never disabled for the potential option to support real-time demands), these cannot trigger events[15], but may set flags to cause the immediate call of a polling handler right after the ISR. Interprocess communication is provided by the kernel through event posting. A special Contiki feature is its protothreads. These voluntarily

[12] TinyOS Enhancement Proposal 106: `http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html`

[13] Early advances in adding concurrency to TinyOS applications can also be found in TinyMOS [288], which tries to execute TinyOS within preemptive Mantis OS threads.

[14] Diploma thesis conducted in conjunction with this work.

[15] This is forbidden to avoid race conditions among handlers.

allow cooperative context switching by means of special yield directives (`yield`, `wait`) within the program code. Since these directives are visible to the developer, it becomes directly obvious if a certain resource needs to be locked explicitly across their call. Since the so called stack rewinding clears the stack at each yield directive, protothreads can easily share a common stack and do not need to save register contents; consequently do also not retain their states and local variables automatically[16]. Internally, the corresponding dispatcher makes use of a C switch statement, which, depending on the protothread's state variable, jumps to strategically placed case targets. These are injected after each yield point and render the storage of the program counter unnecessary. The scheduling of protothreads must finally be managed by the application software itself. In addition to processes and protothreads, a library for preemptive multi-threading is available for the event-driven kernel. While the first share a common stack region, threads require separate stacks and more sophisticated synchronization primitives.

Other features like dynamic memory are also supported by Contiki. However, it offers no inherent resource management, time-awareness, or temporal semantics.

**Mantis** [2, 46]. First published in 2003, the Mantis (MultimodAl NeTworks of In-situ Sensors) operating system was designed as a preemptive multi-threading kernel for improved real-time capabilities – at least in comparison to event-driven systems with run-to-completion semantics. Mantis supports five priority levels, and schedules fixed-priority threads according to the round robin policy (10 ms time slices) within each level. The timing is driven by clock interrupts, which are handled by the kernel and also allow the maintenance of a 32 bit timeline[17]. The temporal semantics is limited to the specification of sleep times, i.e. the limited self-suspensions of threads. For fast reaction on external events, other interrupts are handled by device drivers and trigger the execution of waiting threads by setting them to ready state. For its execution each thread uses an individual stack, which is dynamically allocated from the system heap upon thread loading. Though mutexes and semaphores are supported for thread synchronization, neither primitive considers priorities for coordinating pending requests.

**SOS** [64, 120]. Presented in 2005 the Simple Operating System pursues a module based strategy for simple software updates at runtime. The kernel provides components for dynamic memory management, message scheduling, and dynamic linking.

Modules implement handlers and functions for interactions. Functions can be called directly and are executed instantly (synchronous processing). Handlers, also denoted as tasks, are triggered by messages from interrupts or other tasks, enqueued into (three) priority queues, and finally called by the scheduler (asynchronous processing). Since handlers are scheduled cooperatively, they must release the CPU voluntarily, and consequently induce the potential problem of starving other handlers. Incorrect or unfair module operation is generally neither detectable nor avoidable since, apart from critical sections, no synchronization primitives exist. Considering time-awareness, a simple local time is maintained but no temporal semantics exists for task operations.

[16] The same is true for thread-blocking calls, which may switch the context.
[17] With a resolution of 1 ms an overflow occurs after $2^{32}$ ms $\approx$ 50 days.

**SenSmart** [69]. Under development since 2009, SenSmart combines a kernel runtime with binary translation to achieve preemptive multitasking for arbitrarily assembled tasks. Therefore, the task binaries – wherever these originate from, and whatever system software they might contain – are prepared on a so called base station: First, modifications to kernel relevant resources are caught by so called “trampoline functions” and emulated specifically. Second, scheduling does not rely on a system timer but is done by modifying branch instructions to invoke the kernel after each 256th branch. This so called “interrupt free preemption” is comparable to the t-Kernel [116, 117]: The scheduler applies a round robin scheme, and counts ticks of one of the MCU’s system timers to preempt the running task after its time slice is over. Third, memory access instructions are replaced by jumps to the memory manager, which supports logical address translation, memory protection and isolation, as well as on-demand stack relocation to reduce external fragmentation while increasing stack versatility.

Though SenSmart provides advanced memory sharing concepts, it lacks time-awareness and temporal semantics for its tasks. Real-time concepts, especially for the composition of complex systems, are also missing. In particular, it offers no synchronization primitives or inter-task-communication, but executes all tasks completely independent from each other.

**Nano-RK** [95]. Released in 2005, the Nano Resource Kernel supports fixed-priority preemptive multitasking where higher priority tasks may immediately suspend lower priority tasks. Apart from critical sections, inter-task-communication is provided via 32 signals or events. Yet, special focus is given to the resource reservation concept, which assigns resources according to (energy related) budgets and allocation counts as defined at development time: The associated semaphores support the priority ceiling protocol emulation (PCPE), and consequently prevent deadlocks (→ Section 6.3.1). For the CPU resource, each task defines its requirements, and is suspended immediately if it exceeds its budget. Similar to *Smart*OS, where a task $T$ can query the remaining time until a higher priority task would miss its deadline if itself ($T$) would not release a specific resource, Nano-RK tasks may query their remaining budget for self-adaptation. To reflect the timing demands of periodic sensor tasks, rate monotonic scheduling is also supported, and time-awareness is available through both a local time (1 ms resolution) and the temporally limited waiting for signals.

**Other operating systems.** Besides the just mentioned representatives, a multitude of further embedded operating system architectures for mote class devices is available. We won’t discuss them in detail, but refer to additional overviews which can be found in e.g. [89, 98, 240]. Where reasonable, specific details might yet be considered throughout later chapters.

### 3.3.3. Open Problems and *Smart*OS

As already requested in Section 1.2, time-awareness and compositional software design will be required for complex sensor/actuator systems under time-critical conditions. Thus, temporal semantics, priority-based scheduling, and sophisticated resource management should be combined for improved system reliability. Another problem is the dynamic handling of sporadic software failures or system imponderabilities through appropriate programming concepts, like e.g. exception handling. These issues will be addressed throughout the next chapter.

| Name | *Smart*OS | TinyOS | Contiki | MantisOS |
|---|---|---|---|---|
| References | [36] | [128, 129, 174, 175, 291] | [85–87] | [2, 46] |
| Project active | Yes | Yes | Yes | No |
| App. Lang. | C | nesC [109] | C | C |
| Kernel Arch. | exo / micro | monolithic | exo | micro |
| Res. Handling | coord. | HAL | - | protection |
| Process Org. | tasks | tasks[6] | handlers[7] | threads |
| Concurrency | preemptive | RTC | cooperative | preemptive |
| Stack Memory | individual | shared | shared | individual |
| Data Memory | static[3] | static[3] | dynamic | static[4] |
| Exec. Model | time/ev. driven | ev. driven | ev. driven | time/ev. driven |
| Scheduling[1] | prio (FCFS/RR) | queue (FIFO) | queue | prio (RR) |
| Time-Awareness[2] | local (1us) | - | local | local (1ms) |
| Temp. Semantics | sleep, DL, TS | - | - | sleep |
| Reactivity | BE / RT | BE | BE | BE |
| Process Sync. | mutex, sema.[9] | mutex | sema. | mutex, sema. |
| IPC | events | - | events, mess. | events |
| IRQ Handling | central / app[8] | app | app | app |

| Name | SOS | SenSmart | Nano-RK |
|---|---|---|---|
| References | [64, 120] | [69] | [95] |
| Project active | No | Yes | Yes |
| App. Lang. | C | any | C |
| Kernel Arch. | micro | exo | exo |
| Res. Handling | - | coord. for memory | coord. |
| Process Org. | handlers | tasks | tasks |
| Concurrency | cooperative | preemptive | preemptive |
| Stack Memory | shared | individual | individual |
| Data Memory | dynamic | static[5] | static |
| Exec. Model | ev. driven | - | time/ev. driven |
| Scheduling[1] | prio (3 FIFO queues) | queue (RR) | prio (RMS) |
| Time-Awareness[2] | local | - | local (1ms) |
| Temp. Semantics | - | - | sleep, DL for signals |
| Reactivity | BE | BE | BE |
| Process Sync. | CS | - | CS, sema.[10] |
| IPC | mess. | - | events (32) |
| IRQ Handling | central | central | central |

[1] the policy for queues and tasks with equal priorities is given in brackets
[2] resolution in brackets
[3] dynamic memory through libraries
[4] dynamic memory for the kernel only
[5] dynamic memory for stack placement only
[6] threads with individual stacks, cooperative concurrency, and round robin scheduling as library
[7] threads with individual stacks are available as library
[8] centralized acceptance, handling at application layer
[9] supported by PIP / PCP
[10] supported by PCPE

**Table 3.1.:** Comparison of WSAN operating system kernels (refers to Figure 3.3[p39])

# 4. *Smart*OS: A Small modular adept real-time Operating System


**Abstract**

This chapter introduces *Smart*OS, the **S**mall **m**odular **a**dept **r**eal-**t**ime **O**perating **S**ystem for sensor/actuator nodes.

Though several operating systems for sensor nodes are already available, our vision of an OS for such embedded real-time systems demanded for both inherent time-awareness to support reactive operation within highly dynamic environments, and sophisticated resource management to facilitate modular application design. Hence, we present a kernel to provide priority-based scheduling for fully preemptive tasks, integrated temporal semantics and highly precise time management. A unified interrupt and event handling concept, as well as dynamic resource management with support for modified priority inheritance protocols and on-demand task collaboration introduce an entirely novel programming paradigm into SANet software design. The support for exception-like failure handling completes the functionality.

Apart from presenting the basic concepts, some implementation details, and software examples, this chapter also provides the basis for the subsequent discussion of time-awareness (→ Chapter 5), dynamic resource management (→ Chapter 6), and dynamic memory management (→ Chapter 7) in particular. A complex real-world application based on *Smart*OS is presented in Part III of this work.

## 4.1. Introduction

*Smart*OS was developed for research and education in the area of time-critical embedded systems. Its reference implementation, to which we will refer throughout this and subsequent chapters, supports Texas Instrument's MSP430 [279] family of 16 Bit microcontrollers as found on the SNoW[5] sensor node described in Section 2.2. Additional ports are available for Atmel's ATmega128 8 Bit and Renesas' SuperH SH-2A 32 Bit architectures. Applications and their modules are typically developed in plain C, and linked together for creating monolithic binaries. All *Smart*OS-related examples and test benches within this work refer to code and binaries created using the GNU mspgcc toolchain[1] (version 3.2.3), and were executed on SNoW[5] sensor nodes (version 1.2+) with TI MSP430F1611 MCUs [280].

The main focus of *Smart*OS [36] lies on compositional software design through sophisticated time and resource management. While its specific characteristics were already summarized and highlighted in Figures 3.2[p37] and 3.3[p39], the benefits for embedded applications become visible, inter alia, within various test benches, and especially when discussing the design of the wireless communication MAC protocol *Smart*Net in Section 8.1, and the indoor localization system SNoW Bat in Part III of this work.

## 4.2. Philosophy and Classification

Regarding the classification framework from Section 3.3.1, *Smart*OS features a hybrid exo/micro kernel architecture and incorporates several advantages of both classes. While it supports arbitrary resource allocation (exo), it provides native abstractions only for a small set of system components (micro): Organized as a system of concurrently running tasks with individual stack areas, the preemptive scheduler supports dynamic user priorities in combination with synchronization primitives like mutexes to protect code sequences from interleaved execution by several tasks, and semaphores to coordinate the dynamic access to temporally shared but exclusive resources. The resource manager supports physical as well as virtual resources under both long-term and short-term allocation. It does not only assign them for exclusive allocation[2], but also coordinates pending allocation requests with respect to each involved task's priority. Therefore, it employs modified priority inheritance protocols (PIP and PCP) for collaborative resource sharing (→ Chapter 6). Inter-task-communication and environmental interaction is provided via events which can be invoked by tasks and interrupt handlers. While interrupt acceptance is always centralized, their actual handling is done at application layer.

Time-awareness is an inherent design concept in *Smart*OS. Apart from a local system time and highly precise IRQ timestamping (→ Section 5.3), a temporal semantic forwards the notion of time to the application tasks, and provides non-blocking versions of all kernel functions which

[1] The GCC toolchain for the Texas Instruments MSP430 MCUs [292]: `http://mspgcc.sourceforge.net`
[2] Protection is not supported due to missing hardware support on the considered microcontrollers.

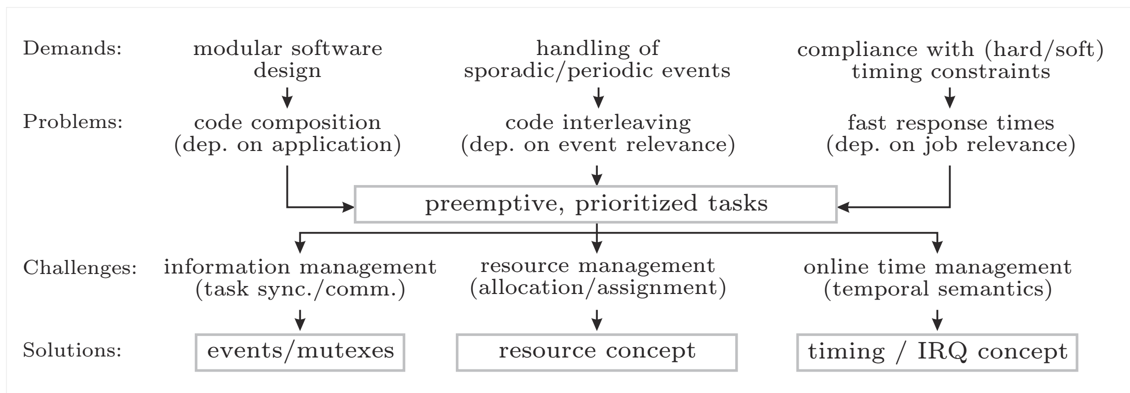



**Figure 4.1.:** From demands to solutions: The central concepts of the *Smart*OS philosophy

might not complete immediately: Sleeping is equally supported as the specification of deadlines for events and resource requests.

While each system's reactivity initially relies on best effort and always respects task priorities, additional real-time demands can be satisfied by limiting the worst case allocation time of resources via DynamicHinting. In summary, preemptive scheduling allows to combine the semantics of time and events to simultaneously support time triggered and event-driven execution models for both periodic and sporadic tasks. Another novel concept in the domain of SANet operating systems is task-specific exception handling capability for reacting on unforeseeable system conditions. Finally, though provided as a library on top of the collaboration concept, CoMem allows dynamic heap memory management for the collaboratively coordinated sharing of the typically sparse memory resource on most motes (→ Chapter 7).

## 4.3. Kernel Architecture and Central Concepts

In Chapter 3.1 we already demanded for a uniform hardware abstraction through the operating system. Besides a platform-independent philosophy, this also includes a constant set of kernel components and consistent signatures for all kernel functions, the so called kernel API (Application Programming Interface)[3]. In general, *Smart*OS is constructed according to the reference architecture from Figure 3.1[p34]. While it shares the CPU with the tasks, the (exo)kernel requires exclusive access to just one hardware timer and the IRQ controller – the allocation of these two resources is fixed throughout the entire system runtime. In this section, we'll introduce the seven central concepts of the *Smart*OS (micro-)kernel: tasks, time, mutexes, events, resources, IRQ management, and exception handling. Figure 4.1 summarizes the initial demands, the resulting problems, the self-induced challenges which arise from our decision to support preemptive and prioritized tasks, and our corresponding solutions for the development of reactive embedded applications. For a comprehensive understanding, we'll also present some code examples. How-

[3] By this, the application code has always the same view of the system, and both its development and porting should be simplified (at least in theory). For the same reason, libraries and drivers (which access software and hardware components, and most commonly require manual adaptation for each port), should also provide a uniform API.

ever, since the identifiers and function names therein should be self-explaining, we won't go into detail about each individual line, but refer to the complete API reference of kernel functions and data types in Appendix A.

### 4.3.1. Basic Terminology

Before dealing with the central concepts of the *Smart*OS kernel architecture, we'll first introduce the most basic terminology:

**Control block, OS object.** The term *control block* denotes the data structure for storing the management and state information of an OS object. *OS objects* comprise tasks, resources, events, mutexes, and IRQ handlers. An overview can be found in Appendix A.

**Context, priority, context switch, scheduler, dispatcher.** The term *context* denotes the environment in which a task is executed. It comprises its task state, specific register contents, stack memory, and various management information like priorities. *Priorities* are task-specific numeric values between 0 (lowest) and 255 (highest), which indicate a task's relevance in relation to others. If a task's execution is interleaved with another one this is called a *context switch*. While the *scheduler* selects the task to be executed next, the context switch itself is conducted by the *dispatcher*.

**Syscall, syscall wrapper, invoking task.** The term *syscall* denotes an atomic kernel function, which is executed in kernel mode, and finally ends up by calling the scheduler. Since syscalls must not be called from within task context, switching to the kernel mode is done by so called *syscall wrappers* (→ Figure A.2[p327]). A task which triggered the execution of a syscall is called *invoking task*.

### 4.3.2. Process Organization: Tasks

*Tasks* represent the most basic logical unit of any *Smart*OS-based application. While operations within each task are executed in sequential order, tasks are always preemptive, and can be interleaved for quasi-parallel execution[4]. Atomic sections are not supported. Although *Smart*OS supports an arbitrary number of tasks[5], the set $T$ of tasks is fixed after linking[6]. Each task contains a never returning entry function, and executes its job as an infinitely looped code sequence. During its declaration, each application task $t \in T$ receives a unique identifier, a fixed stack size, and an initial *base priority* $P_t$ between 2 and 254. While the base priority may be changed at runtime by application code, the stack is fixed and must be dimensioned properly at development time. A source code profiler is available to compute an upper bound for a task's stack requirement at compile time [154][7]. An example task and the system main function which

[4]Truly parallel execution cannot be supported unless more than one CPU or core is available.

[5]Yet, limitations arise from the data type for the task ID within the task control block (→ Appendix A).

[6]The reason is, that for higher performance through pointer arithmetic, the linker places the task control blocks successively aligned in the data memory (→ Figure A.1[p324]): Of course, some free slots could be reserved, or used slots could be replaced to gain more dynamics.

[7]Diploma thesis conducted as part of this work: Since such analysis is a hard problem, and sometimes even unfeasible, adequate annotations within specially prepared C comments can be added to the code to refine the estimation. For example the use of recursive functions can rapidly lead to the demand for an infinite stack size. Providing the maximum recursion depth by manual annotation can solve this problem.

```
OS_MAIN {
  os_run(); // Init system
            // and OS objects.
            // os_run() starts
            // the scheduler
            // and will never
            // return.
}
```

**(a)** The *Smart*OS `main()` function.

```
OS_DECLARE_EVENT(evButton);
OS_DECLARE_IRQ_HANDLER(
  OS_IRQ_PIN10, setEvent, &evButton);

OS_DECLARE_TASK(tDP, 50, 128);
OS_TASKENTRY(tDP) {
  initIRQPin(OS_IRQ_PIN10);
  while (1) {
    clearEvent(&evButton); // ignore button
                           // during reaction
    waitEvent(&evButton);
    react();               // any code
  }
}
```

stack size base priority

**(b)** Example task for IRQ handling under *Smart*OS

**Listing 4.1:** *Smart*OS `main()` function and interrupt handling example

initializes all OS objects before starting the scheduler is shown in Listing 4.1.

From the kernel perspective, tasks are initially independent from each other, but may develop dynamic dependencies at runtime (e.g. through resource sharing or inter-task-communication). As we will see in Chapter 6, the resource manager therefore also maintains a dynamic *active priority* $p(t)$ for each task $t \in T$ with respect to its current resource dependencies, and consequently the priority-based scheduler always considers the active priorities[8]. Basically, tasks transit between three *task states*: *running*, *waiting*, and *ready*, with ready being the initial state (→ Figure 6.4[p103]). As Figure 4.2 shows, the kernel maintains a system-wide *timeout queue* for tasks waiting for deadlines, and specific *event queues* for tasks waiting for the very event. Ready tasks are kept in the system-wide *ready queue*, with the head being the current *running task*. While the ready and event queues are sorted by decreasing priorities to serve higher priority tasks first, the timeout queue is sorted by increasing absolute deadlines to simplify the timer configuration for the resumption of waiting tasks as described in Section 4.3.4.

Apart from regular application tasks, *Smart*OS schedules an architecture-specific *system idle task* if no other task is ready. Thus, it has lowest priority (0), will never transit to waiting state, and does not incur dependencies to other tasks. It is used for the most elementary energy management, and controls the architecture-specific low power modes as long as they do not affect the application flow[9]. For each application, one specific *user idle task* (with base priority 1) can be implemented for more sophisticated energy savings and housekeeping actions in relation to the current system state; if it fails or enters waiting state, the system idle task is still available as fallback.

[8] If no resource-related priority adaptation is required, the active priority equals base priority (i.e. $p(t) = P_t$) throughout the entire system runtime.

[9] As already shown in Table 2.2[p27], the idle task for the MSP430 architecture will simply switch off the CPU (LPM1) but leaves all other peripherals untouched (especially the system timer).

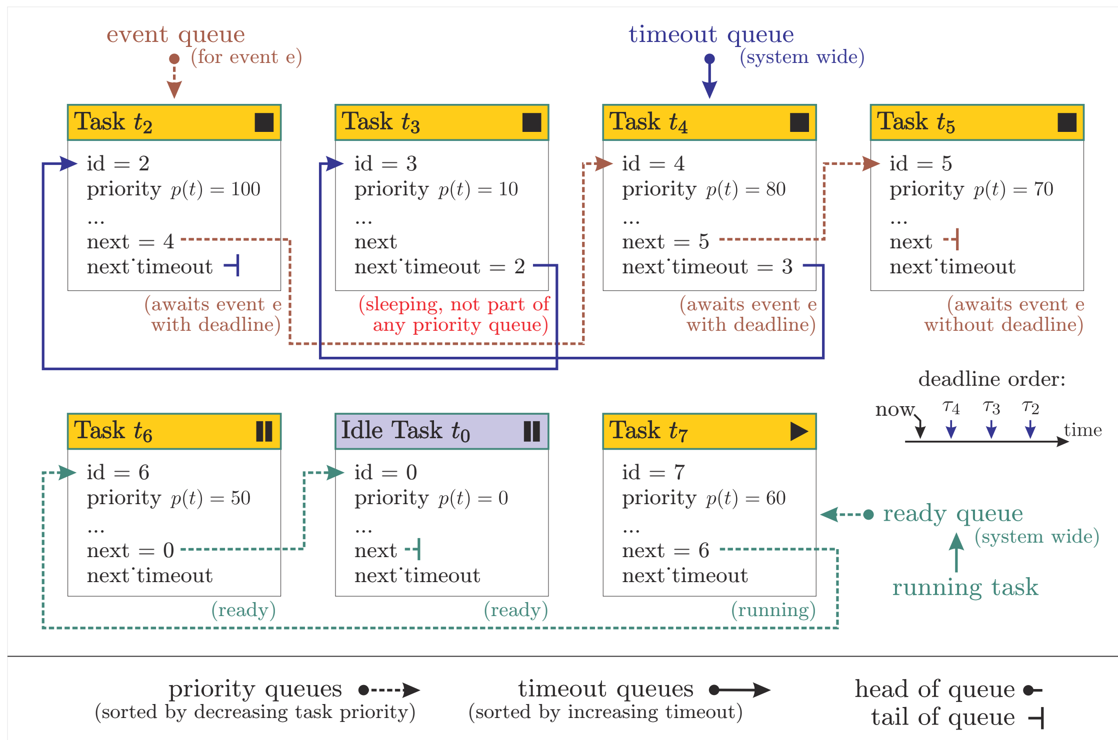



**Figure 4.2.:** Example for priority and timeout queues in *Smart*OS

### 4.3.3. Operation Modes and Syscall Handing

Since *Smart*OS shares the CPU with the tasks it manages, and thus has to protect its internal data structures from simultaneous access by concurrently running tasks and asynchronous IRQ handlers, it distinguishes between two operation modes: task mode and kernel mode. Tasks are executed in *task mode* where specific stacks are used and interrupts are always enabled. In contrast, all interrupt service routines – the so called *kernel ISRs* – as well as the syscalls are always executed in atomic *kernel mode* where the kernel stack is used and interrupts are disabled. If further interrupts occur while the kernel mode is active, their acceptance is deferred until the kernel returned. To yet maintain reactivity in spite of a non-preemptive kernel, *Smart*OS executes as little atomic code as possible. However, since all kernel components with potential write access to internal kernel data structures are encapsulated in either ISRs (time management) or syscalls (anything else), these are implicitly protected from race conditions. In fact, as Figure 4.5 illustrates, syscalls and interrupts are even comparable regarding their execution flow which is divided into three parts: While the *kernel entry* is responsible for stack preparation and context saving, the *kernel body* handles the actual request, and the *kernel exit* contains the scheduler and the dispatcher for task selection and context switching.

Since handling syscalls is quite tricky, we'll take a closer look at the underlying concept. As an example, Listing 4.2b shows the syscall for the `CAS` (compare and swap) kernel func-

```
Object_t *head = NULL; // any list head

void updateHead() {
  Object_t *oldHead, *newHead;
  do {
    oldHead = head; // current head
    newHead = ...   // new head
  } while
    (!CAS((Mutex_t*)head,
          (int)oldHead,
          (int)newHead));
}
```

**(a)** Lock-free modification of a list head

```
void OS_SYSCALL __syscall_CAS(
     Mutex_t *addr,
     int expVal, int newVal) {
  if (*addr == expVal) {  // success
    *addr = newVal;
    running_task->reg_context[12] = 1;
  } else {                // failure
    running_task->reg_context[12] = 0;
  }
}

int CAS(Mutex_t *a, int e, int n) {
  return syscall_CAS(a, e, n); }
```

**(b)** Atomic CAS emulation through a syscall

**Listing 4.2:** Mutex usage under *Smart*OS

tion[10]. In particular, though `CAS(...)` is defined to return an integer result, the underlying `__syscall_CAS(...)` is defined to be void. The missing adaptation is done by the associated wrapper `syscall_CAS(...)` as shown in Figure 4.5: In fact, all syscalls are initially void; since they are treated like ISRs, and the kernel mode always returns by `reti`[11] instead of `ret`[12], they cannot return a result as usual. Instead, return values must be passed directly via the invoking task's context, i.e. the currently running task as depicted in Listing 4.2b. By writing the result into the task's saved context according to the C ABI (Application Binary Interface) for register usage, the dispatcher will implicitly restore the return values when resuming the task, and the initially called `CAS` function will provide the integer result as expected. If a syscall is invoked from within kernel mode (e.g. by an IRQ handler), it simply returns by an ordinary `RET` instruction since we won't leave the kernel mode then. Return values are passed as described before.

### 4.3.4. Time Management: The Timeline

Unlike most available sensor network operating systems (→ Table 3.1), *time* is an inherent design concept of the *Smart*OS kernel, and the central foundation of any application software. The *Smart*OS time management maintains a local system time – the so called *timeline* – with a standardized resolution of 1 µs, and a counter width of 64 bit. Counting starts from 0 µs with the scheduler start, is entirely independent from application code, and overflows after $2^{64}$ µs ≈ 584942 years[13]. Besides its natural progression, the system time does neither warp in any direction, nor can it be set directly. However, since it is driven by a periodic timer component of the underlying hardware, it reflects all clock drifts and frequency instabilities (→ Chapter 5). No compensation or even synchronization with other systems will be provided by the kernel (but can be implemented at higher levels).

[10] While `CAS`-like functions are useful for implementing lock-free operations of any kind, no native support is available within the ISA of many MCUs, and we have to emulate an appropriate substitute in software.
[11] RETurn from Interrupt
[12] RETurn (from subroutine)
[13] Which should be sufficient compared the expected lifetime of continuously powered hardware devices.

```
// Task stack size: 10W, priority: 128
OS_DECLARE_TASK(tPeriodic, 10, 128);
OS_TASKENTRY(tPeriodic) {


  while (1) {
    periodicAction(); // any code

    sleep(1000000);   // relative
  }
}
```

**(a)** Relative timeout. Start of $n$-th iteration: $t(n) = t(0) + \sum_{i=1}^{n} (\lambda + \epsilon_i)$

```
// Task stack size: 10W, priority: 128
OS_DECLARE_TASK(tPeriodic, 10, 128);
OS_TASKENTRY(tPeriodic) {
  Time_t nextTime;
  getCurrentTime(&nextTime);
  while (1) {
    periodicAction();      // any code
    nextTime += 1000000;
    sleepUntil(&nextTime); // absolute
  }
}
```

**(b)** Absolute deadline. Start of $n$-th iteration: $t(n) = t(0) + \sum_{i=1}^{n} \lambda + \epsilon_n$

**Listing 4.3:** Periodic tasks under *Smart*OS (Period $\lambda = 1$ s, task resumption imprecision $\epsilon_i$ for the $i$-th resumption)

From an application's point of view, all time-involving API functions refer to the timeline. They provide a temporal semantic for the tasks, and forward the notion of time to support non-blocking versions of all kernel functions which might not complete immediately. The specification of absolute deadlines or relative timeouts allows the temporally limited waiting for events and resources (→ Sections 4.3.6 and 4.3.7). Based on this, arbitrary sleeping and periodic task executions can also be realized. As Figure A.2[p327] depicts for the call hierarchy of the syscall `__syscall_waitEventUntil`, sleeping is emulated by infinite waiting for the special NULL-event, which does not even exist and thus will never occur. Listing 4.3 gives an example for periodic tasks: Note, that while using relative timeouts results in accumulated errors for the begin of each iteration, absolute deadlines yield improved period stability.

Additionally, the *Smart*OS API allows to query the timestamps and relative order of external events, and, in consequence, both the attribution of collected information and the measurement of delays. In this context, the term *timestamp* denotes the timeline snapshot at the instant of an IRQ occurrence[14]. Based on the timeline, the kernel automatically captures a timestamp $t_{\text{IRQ}}$ for each interrupt, and can achieve a symmetric error interval $[-\frac{1}{2}$ µs, $\frac{1}{2}$ µs) around the true occurrence time $t'_{\text{IRQ}}$[15]. Though this timestamp will be overwritten by the next interrupt, it stays valid throughout its corresponding ISR and can be saved for further use. In fact, a timestamp is never missed, unless the interrupt itself was missed[16]. While these features are relevant for sensor systems, the specification of real-time requirements and delays for scheduling time-dependent (re)actions is indispensable for reactive sensor/actuator systems. In this context, a self-adaptive technique for scheduling tasks in time and to achieve the precise execution of reactions is presented in Section 5.4. Additionally, time-utility-functions (TUF) [176] can be implemented for self-reflective task adaptations as described in Chapter 6.

From the kernel's point of view, the scheduler relies on the timeline and the timer to resume

[14] Not IRQ acceptance, though.

[15] For MCU architectures without hardware supported timestamping (e.g. the TI MSP430), a CPU load-dependent imprecision may appear if an interrupt occurs while their acceptance is disabled.

[16] This will not occur as long as interrupts are buffered in hardware. Yet, their processing order might change then.

waiting tasks in case they have reached their absolute deadlines. As depicted in Figure A.2[p327] the call hierarchy always maps relative timeouts to absolute deadlines. In contrast, other operating systems, like SenSmart [69] and the t-Kernel [116, 117], avoid using a timer for triggering the kernel mode, but extend certain CPU instructions by stepping counters, and activate the kernel code as a certain threshold is reached (→ Section 3.3.2). Their motivation is that interrupts might not only get disabled by system-wide `dint` instructions, but they also cause significant CPU load while the underlying timer consumes lots of valuable energy. However, the tasks' temporal specifications will suffer significantly from such approaches since the next kernel invocation depends on the currently executed code. While real-time operating systems often sacrifice security for the benefit of reactivity, we also spend some CPU time and energy for increasing the scheduler precision. In fact, true resource protection cannot be provided anyway on typical WSAN MCUs, unless all memory access instructions are checked either at compile time or at runtime[17]. The induced CPU load of ≈0.063% can be neglected, but will nevertheless be discussed in Section 4.4.3.

Further details about the use of time in digital systems, its integration into *Smart*OS, and the highly precise interrupt timestamping will be discussed in Chapter 5.

### 4.3.5. Interrupts and Interrupt Handlers

As indicated in Section 4.3.3 and Figure 4.5, the acceptance and initial handling of interrupts is always done by standardized kernel ISRs (one for each interrupt source), but transfered to application-specific IRQ handlers which will be executed within the kernel body. Listing 5.1[p79] shows an example. Executing these handlers in kernel mode highlights their independence from tasks, simplifies task stack dimensioning, and provides them with an even higher (implicit) priority than any application task. Since disabling interrupts in task context is illegal[18], the kernel ISRs can interleave task executions immediately, and, unless the system already operates in kernel mode, the IRQ handlers are executed with some constant delay $\tau_{\text{entry}}$ after the interrupt occurrence. Figures 4.3 and 4.4 show examples with $\tau_{\text{entry}} = 15.84\,\mu s$. Since IRQ handlers are neither preemptive nor related to tasks, they must never suspend themselves or invoke functions which cause or require the ownership of a resource; these cannot be assigned to IRQ handlers[19]. The IRQ related kernel API is summarized in Appendix A[20].

For most MCU architectures it is common practice to share interrupts among several sources. This is a problem when developing modular software components independently from each other, as a common IRQ handler must still be adopted to meet the requirements of all involved tasks. Therefore, the *Smart*OS interrupt concept supports soft IRQ handlers for automatic demultiplexing[21]. While Figure 4.3 shows the execution of `setEvent(...)` as IRQ handler, Figure 4.4 corresponds to the module code from Listing 4.1b where an event is set by using

[17] Beware of using DMA (direct memory access) controllers as long as their destination addresses cannot be checked properly!
[18] Though it cannot be prevented on most MCU architectures, we assume that tasks will never do so.
[19] This would not be compatible with the resource management concept from Chapter 6.
[20] Illegal calls to prohibited functions will be detected and result in a kernel panic (shutdown or reset).
[21] e.g. for multiplexed port pins, DMA channels, DAC channels, etc.

| | Mutexes | Events | Resources |
|---|---|---|---|
| specific owner task | – | – | ✓[1] |
| access control | compare and swap[2] | wait for event[3] | request resource[3] |
| counting | –[4] | – | ✓ |
| arbitrary access order / cascading | ✓ | ✓ | ✓ |
| deadlock proof | – | – | – (PIP) / ✓(PCP) |
| affects task priorities | – | – | via PIP/PCP |
| access within IRQ handlers | read/write | test/set | test |
| access within tasks | read/write | test/set/wait | test/release/get |
| control block data type | `Mutex_t` | `Event_t` | `Resource_t` |
| RAM requirement | 1 W | 1 W | 6 W |
| description in section | 4.3.6 | 4.3.6 | 4.3.7, 6 |

[1] supports allocation and resource request coordination according to Figure 3.3[p39]
[2] non-blocking operation (spinning at application layer)
[3] supports task self-suspension for temporally limited waiting
[4] counting can be implemented at application layer

**Table 4.1.:** Comparison between mutexes, events, and resources under *Smart*OS

`setEvent(&evButton)` as soft IRQ handler for pin P1.0. Demultiplexing the shared port interrupt is done by some other module, which is not discussed within this work.

In general, each `void` function with exactly one parameter of type `int` can be registered as (soft) IRQ handler. This allows to specify the same function for various interrupt sources (e.g. `setEvent(...)`), but declare different parameters (e.g. events) for a case specific handling.

### 4.3.6. Synchronization: Mutexes and Events

Since the *Smart*OS scheduler supports preemptive task concurrency, appropriate synchronization primitives must be supported. A summary is given in Table 4.1:

*Mutexes* are the most basic approach, and realized through atomic compare and swap (CAS) functions for non-blocking access to integer variables or pointers of architecture word width. Since Mutexes have no owner tasks, full access is also granted for IRQ handlers. While Listing 4.2a shows an example for the lock-free modification of a list head, we'll also find the underlying idea applied in the context of dynamic memory allocation in Section 7.4.2.

*Events* provide a more sophisticated synchronization mechanism, which can be used for inter-task-communication (ITC) and the reaction on interrupts (→ Listing 4.1b). In comparison to mutexes, events support only two states for non-counting operation: set and unset. While changing or querying the state is non-blocking and may be done at any time from within tasks or IRQ handlers, only tasks may suspend themselves to wait for the occurrence of an event. The example in Listing 4.4 demonstrates their use for the producer/consumer design pattern. As already described in Section 4.3.2 and Figure 4.2, the kernel maintains an individual event queue for waiting tasks. According to the API's temporal semantic, waiting can either be limited by putting the task into the timeout queue, or suspend a task forever (→ Figure A.2[p327]). If an event is set by a task or IRQ handler, it either causes the task with highest priority within the specific event queue to leave waiting state, or it remains set for later consumption or explicit clearing.

```
OS_DECLARE_EVENT(evFull); //unset

OS_DECLARE_TASK(tProd, 50, 128);
OS_TASKENTRY(tProd) {
  while (1) {
    waitEvent(&evEmpty);
    fillBuffer();         // any code
    setEvent(&evFull);
  }
}
```

**(a)** Producer and initially unset full indicator

```
OS_DECLARE_ACTIVE_EVENT(evEmpty); //set

OS_DECLARE_TASK(tCons, 50, 128);
OS_TASKENTRY(tCons) {
  while (1) {
    waitEvent(&evFull);
    processBuffer();    // any code
    setEvent(&evEmpty);
  }
}
```

**(b)** Consumer and initially set empty indicator

**Listing 4.4:** The producer/consumer problem in *Smart*OS: Buffer handling

### 4.3.7. Hardware Abstraction: Resources and Resource Chains

*Resources* coordinate the access of tasks to physical hardware components, like peripherals or buses, and to virtual abstract entities, like data structures or application code. According to the classification framework from Figure 3.2[p37], *Smart*OS supports the dynamic management of temporally shared resources under both long-term and short-term allocation. While the CPU and the tasks remain preemptive in general, all other resources are always assigned exclusively. Recall, that (apart from the CPU, one timer and the IRQ controller) *Smart*OS has no notion about the components or entities it manages. In particular, it offers no resource protection. Instead, it only coordinates the access, and leaves their operation for application code at higher layers.

Comparable to waiting for events, tasks may suspend themselves to wait for a currently unavailable resource. In fact, the corresponding `Resource_t` control blocks contain one `Event_t` OS object each, to count the number of allocations by the current owner task[22] and to maintain a priority queue of concurrently persisting requests by other tasks. While this implicitly serializes tasks regarding their allocation requests, the proper use of the corresponding operational units is the responsibility of the application developers. Though *Smart*OS offers some assistance, the kernel will neither detect nor block illegal access attempts to not properly allocated units. In particular, it will *never* withdraw resources.

The most significant difference between resources and events is, that events are simply consumed by tasks, while resources stay assigned to tasks until these release them voluntarily and explicitly. For this reason, IRQ handlers may never allocate (or release) resources. Related problems for concurrent task systems (e.g. task starvation, priority inversion, and allocation deadlocks), will be deeply discussed in Chapters 6 and 7.

A novel concept in the area or WSN operating systems are the so called *resource chains* for the automatic allocation of dependent resources: In Listing 4.5, a hypothetical radio protocol task `tRadio` allocates and releases an externally defined resource `rRADIO` for transmitting a data packet (Lines L60 – L63). Within the radio chip driver the `radioSend` function first asserts if the resource belongs to the caller (L47), i.e. the running task[23]. However, when looking at the

[22] Resources are counting, though events are not. Thus, resources can be allocated several times by the same task, and must be released as often.

[23] Assertions can be removed from the binary code by a compiler switch. Failing raises a kernel panic.

```
1  OS_IMPORT_RESOURCE(rSPI); // SPI bus
2
3  /* init hardware interconnection */
4  int rRADIO_fInit(void) {
5    if (radioChipInit() == 0) return 0;
6    return 1;   // indicate success
7  }
8
9  /* initialize radio for use */
10 int rRADIO_fGet(Time_t* deadline) {
11
12   // create res. chain: rRADIO->rSPI
13   if (getResourceUntil(&rSPI,
14     deadline) == 0) return 0;
15
16   // configure radio (e.g. channel)
17   if (radioConfig(radioCfg) == 0) {
18     releaseResource(&rSPI);
19     return 0;
20   }
21
22   return 1;   // indicate success
23 }
24
25 /* de-initialize radio */
26 int rRADIO_fRelease(void) {
27
28   // set to defined state
29   radioStrobe(RADIOSTOBE_IDLE_MODE);
30
31   // dissolve res. chain rRADIO->rSPI
32   releaseResource(&rSPI);
33
34   return 1;   // indicate success
35 }
```

**(a)** Outline of a potential radio chip driver

```
36 /* create the resource */
37 OS_DECLARE_RESOURCE_EXT(
38   rRADIO,       &rRADIO_fInit,
39   &rRADIO_fGet, &rRADIO_fRelease);
40
41 /* transmission function */
42 int radioSend(int dst,
43               char *buf, int len) {
44
45   /* check for legal access */
46   QASSERT(
47     testResource(&rRADIO) == 1);
48
49   /* code for sending */
50   ...
51 }
```

```
52 OS_IMPORT_RESOURCE(rRADIO);
53
54 OS_DECLARE_TASK(tRadio, 50, 250);
55 OS_TASKENTRY(tRadio) {
56   ...
57   while (1) {
58     ...
59     preparePacket();
60     if (!getResourceFor(&rRADIO,
61       10000)) continue();
62     radioSend(0xFFFF, buffer, 32);
63     releaseResource(&rRADIO);
64     ...
65   }
66 }
```

**(b)** Usage of the radio resource

**Listing 4.5:** Resource usage under *Smart*OS

extended resource declaration (L37), we can see the specification of three handler functions. The `fInit` function (L4) will be called once prior to the scheduler start to initialize persisting configurations (L5). The `fGet` function (L10) will be called each time the resource is successfully allocated. It subsequently allocates depending resources (L13), and initializes fragile configurations (L17). Though resource allocations are managed by a syscall, `fGet` is executed in task mode, to support the complete set of API functions. Additionally, the initially specified relative allocation timeout (10 ms, L61) is converted and passed as absolute deadline to support the temporally limited execution of `fGet`. Here, the temporal limitation is used for bounding the allocation of the SPI bus resource (L13) which is required for accessing the radio chip. Finally, the `fRelease` function (L26) will be called in task mode after the last deallocation. It de-initializes the resource to gain a safe and defined state (L29), and dissolves the resource chain by releasing the dependent resources (L32). While `fGet` and `fRelease` operate transparent to the requesting task, they signal their success to the kernel: In case `fGet` fails, the resource is not allocated and `getResourceUntil(...)` returns an appropriate failure indicator to the requester.

### 4.3.8. Exception Handling

The occurrence of dynamic software failures or sporadic system imponderabilities must not be neglected throughout the application design stage. Since the corresponding runtime errors can not only occur at almost any code position, but also asynchronously, they must often be returned from the point of their detection through the entire but variable call hierarchy to the point of their handling. While function return values are often exploited for their propagation, this misuse of "operation results" is error-prone, hard to maintain, and finally causes various restrictions on the affected API definitions. To compensate this weakness, exception handling is a common mechanism in higher level programming languages (e.g. in Java [112], ADA [139], C++ [141]) and system architectures. It allows to conveniently separate the program logic from error handling: Developers are advised to encapsulate code which might not complete successfully in so called "try" blocks. On the first occurrence of any failure therein an exception is thrown by the application code. As a consequence, the try block is left immediately, and the failure itself is handled within subsequent "catch" blocks. Listing 4.6a gives a Java compliant example which throws and handles an exception of type `Exception_t` to indicate randomly generated error states. Since exceptions are commonly dynamic objects[24], polymorphism even allows to throw arbitrary exception types; the information therein can be defined specifically to support an adequate failure recovery. In case of an unhandled exception, it will simply be forwarded to the surrounding try block, or lead to program termination if there is no such block.

In contrast, lower-level languages like plain C do neither support object-oriented programming paradigms nor exceptions. Nevertheless a similar concept can be implemented with few limitations. Using the `setjmp`/`longjmp` functions from the C standard library [104] allows to emulate the typical try/catch structure: While `setjmp(buf)` puts the current execution context into an architecture-specific data buffer and returns 0 after its first call, `longjmp(buf, value)` restores the previously saved context (including the PC) and adjusts the `setjmp`'s former return value as specified. This way `longjmp` emulates the "throw" directive, and `setjmp` is used to decide between executing the try block (0) or the catch block ($\neq 0$). Under *Smart*OS, throwing an exception "object" is emulated by storing the specific identifier of type `unsigned char` into an automatic `Exception_t` structure on the stack[25], and by forwarding its specific information into the catch block. Though provided by *Smart*OS for consistency reasons, this procedure does not involve any kernel interaction, but is encapsulated in convenient C preprocessor macros which are entirely executed in task mode. By their naming (`TRY, CATCH, THROW`) even the look and feel of high-level languages is provided for *Smart*OS applications[26]. In order to support exception nesting and their independent handling within each task and IRQ handlers, the execution contexts will also be stored on the tasks' and the kernel's stacks respectively[27]. Listing 4.6b shows the mentioned application example with exception nesting and forwarding under *Smart*OS in comparison to the already discussed Java version. Throwing an exception outside of

[24] Note that dynamically allocated objects are created on the heap. This leads to fragmentation and may even cause out-of-memory conditions which are also hard to handle on small embedded systems (→ Chapter 7).

[25] See Section 7.2 for a disambiguation between automatic and dynamic variables or objects.

[26] The macro implementation is based on the `cexcept` library by Costello and Truta (`http://sourceforge.net/projects/cexcept/`).

[27] For the MSP430 MCUs each try block requires 11 words of stack.

```
1  Object obj;
2
3  try {
4    try {
5      obj = new Object(); // dyn. object
6      throw new  // type: any exception
7        Exception_t(rand() % 3); //[0,2]
8        // unreachable code
9    } catch (Exception_t e) {
10     // automatic destruction of obj
11     switch (e.id) {
12       case 0: /* complete handling */
13               break;
14       case 1: /* partial handling */
15               throw e;  //forward
16       default: throw e; //forward only
17     }
18   }
19 } catch (Exception_t e) {
20   /* handle e.id == 1 and e.id == 2 */
21 }
```

**(a)** Java compliant

```
1  MCB_t mem = NULL;
2  Exception_t e;  // type: unsigned char
3  TRY {
4    TRY {
5      mem = malloc(64); //dyn. alloc.
6      THROW
7        (rand() % 3); // [0,2]
8        // unreachable code
9    } CATCH (e) {
10     if (!mem) free(mem); //cond. free
11     switch (e) {
12       case 0:  /* complete handling */
13                break;
14       case 1:  /* partial handling */
15                THROW e; //forward
16       default: THROW e; //forward only
17     }
18   }
19 } CATCH (e) {
20   /* handle e == 1 and e == 2 */
21 }
```

**(b)** *Smart*OS (C) compliant

**Listing 4.6:** Exception handling examples: A comparison

any try block will immediately raise a kernel panic and stop the entire system.

While the emulation of the well-known exception concept adds significant convenience to the application design we'll see its particular benefit for resolving unpredictable and even asynchronously emerging resource conflicts under real-time conditions in Chapters 6 and 7. Nevertheless, compared to full grown and language inherent exception handling, where a large amount of additional information is managed, and where the compiler conducts additional static code checks, our emulation involves two major drawbacks: First, jumping out of or into try blocks is forbidden[28], but cannot be prevented reliably at runtime. Second, objects which were already created dynamically within a try block will not be destroyed automatically if an exception is thrown. Instead, the catch block is responsible for deleting these objects. The same is true for e.g. releasing resources which were allocated before the exception was thrown within the try block[29]. Lines 5 and 10 of Listing 4.6b give an example for the conditional but explicit release of the dynamically allocated memory block `mem`[30]. In contrast, the dynamically created object `obj` is destroyed automatically within the Java code from Listing 4.6a.

## 4.4. Evaluation and Benchmarking

In Chapter 4.3 we introduced the uniform hardware abstraction and platform-independent kernel API of the *Smart*OS operating system; unfortunately, a uniform syntax and philosophy

[28] Appropriate stack maintenance would be missing then.

[29] Adding a "finally" block which would always be executed (i.e. independent from the exception) is possible but still remains to be done.

[30] See Chapter 7 for details on our collaborative memory management approach CoMem.

| metric | under test | duration | Figure |
|---|---|---:|---:|
| $m_1$ | task preemption | $\tau_{entry}$ = 12.52 µs | 4.6 |
| $m_2$ | interrupt latency | $\tau_{entry}$ = 18.74 µs | 4.3 |
| $m_3$ | event invocation | $l_{event}$ = 111.70 µs | 4.3 |
| $m_4$ | context switching | $\tau_{system}$ = 43.36 µs | 4.6 |
| $m_5$ | getResource | $\tau_{system}$ = 52.70 µs | 4.5 |
| | releaseResource | $\tau_{system}$ = 52.10 µs | 4.5 |
| $m_6$ | setEvent | $\tau_{system}$ = 44.20 µs | 4.5 |
| | waitEvent | $\tau_{system}$ = 46.40 µs | 4.5 |
| – | getCurrentTime | $\tau_{system}$ = 5.00 µs | – |
| – | TRY block initiation | 9.0 µs | – |
| – | THROW → CATCH transition | 10.0 µs | – |

**Table 4.2.:** *Smart*OS Rhealstone ($m_i$) and other benchmark results

does not necessarily guarantee exactly identical semantics. Even if a comparison of application code at instruction level would certify equivalent effects on the system states[31], its temporal behavior may still vary. While changing CPU frequencies and architectures can result in significant variations on the hardware layer, temporal imponderabilities may already occur due to environmental dynamics on the same hardware. Since this leads to serious problems for time-critical systems, it is the operating system's responsibility, to keep the runtime overhead low, and to facilitate (or even guarantee) a reliable application execution even at high system load [314].

### 4.4.1. Concurrent Task Scheduling

Compositional software design is a hard challenge since the combination of independently developed tasks must still result in an operational system. While the operating system should make this composition transparent for the developer, it is hard to decide whether a certain task combination finally assembles to correct overall system.

Essentially, the related problems can be traced back to resource conflicts, and must either be addressed statically during development or dynamically at runtime. In this respect *Smart*OS relies on purely dynamic operation, and assigns resources (including the CPU) at best effort under consideration of user defined priorities. Since prior analysis of the overall system demands and inter-task dependencies is omitted entirely, the kernel is not capable of hard real-time operation. Additionally, and comparable to the Unix kernel [194], the *Smart*OS kernel itself is non-preemptive. Nevertheless, soft real-time systems can be implemented by using our novel collaboration based resource management concept for improving these weaknesses significantly.

Regarding the CPU, the *Smart*OS kernel supports deterministic task scheduling only for a single task with maximum priority, since resuming this task means to simply remove it as the

[31] Not *identical* effects, though! Even if the operating system remains consistent, the underlying architectures change, and incompatibilities must often be compensated by entirely different implementations of the same functionality. A common source of such discrepancies is for example the handling of interrupts, and the mostly varying extend of the instruction sets (→ CAS emulation in Listing 4.2b). Anyway, such a comparison will remain a theoretical consideration due to [290].

timeout queue's head, and insert it as new head into the ready queue. Obviously, both operations are in $O(1)$ and the total runtime is bounded to $\tau_{\text{system}} = 43.36$ μs according to Figures 4.5 and 4.6. Although it is obviously not possible to predict the behavior of a dynamic *Smart*OS system exactly, we will at least review some benchmark results to gain a deeper understanding of its temporal behavior on the MSP430 architecture. According to the Rhealstone suite [150], which proposes a set of metrics for benchmarking real-time operating systems, Table 4.2 summarizes the minimal execution times $\tau_{\text{system}}$ of some kernel functions (i.e. if the invoking task does not interfere with other tasks), some timings from the kernel mode execution as shown in Figure 4.5, and some timing benchmarks for exception handling.

### 4.4.2. Kernel and Application Benchmarking

The performance evaluation of (embedded) software systems is a difficult but necessary task, especially when integrating and analyzing novel techniques. While fine-grained benchmarks consider the execution time of individual functions, application-specific tests analyze the behavior of tasks or the overall system. For real-world test beds, so called "hardware monitors" profit from not influencing the system behavior during the observation. However, they are expensive as they demand for specific measurement equipment, and commonly require access to various MCU internal signals: For off-the-shelf MCUs, these signals must be made visible by injecting appropriate code as displayed in Listing 4.7a. In contrast, "software monitors" are much simpler to implement since they rely only on the system functionality itself. As a drawback, the system and the monitor might influence each other, and lead to both a modified behavior of the system under test, and corrupted or imprecise measurements.

For time-related benchmarks within this work, we applied *Smart*OS itself as software monitor, and used the timeline for measuring durations and execution times (→ Listing 4.7). Apart from the convenience compared to attaching e.g. oscilloscopes as hardware monitors in real-world scenarios, this would also allow us to conduct task-specific evaluations and long-term measurements. To verify the reliability of this approach, we initially measured the execution time of various functions internally and externally[32]: Comparing the results showed an almost constant imprecision of 5.00 μs, which is exactly the execution time of the `getCurrentTime` function as shown in Table 4.2.

### 4.4.3. Timer Interrupt Overhead

In Section 4.3.4 we already motivated the use of a hardware timer for the local time management. Regarding the induced CPU load we still have to discuss the 64 bit timeline maintenance strategy as depicted in Figure 4.5: Since hardware timers commonly provide widths below 64 bit, an interrupt signals an overflow and the kernel advances the timeline in software by calling `update_timer()`. Though cascading several timers would increase the overall length and support overflow handling in hardware, we limit the resource requirements of the *Smart*OS kernel to a single timer, and leave others for the actual application. As Figures 4.8 and 4.9 show for an unloaded system with only the idle task running, timeline updates occur at a minimal

[32]Measurement equipment: Tektronix Oscilloscope TDS3034B (`http://www.tek.com`)

```
Time_t start, stop, delay;

getCurrentTime(&start); // int. meas.
togglePin();            // ext. meas.
functionUnderTest();
togglePin();
getCurrentTime(&stop);

delay = stop - start;
```

**(a)** Execution time measurement (internal, external)

```
OS_DECLARE_TASK(tObserver, 100, 255);
OS_TASKENTRY(tObserver) {
  while (1) {
    doConfig();
    sleep(TEST_DURATION); // free CPU
    doStatistics();
    OS_HALT();
  }
}
```

**(b)** Observer task with maximal priority 255

**Listing 4.7:** Benchmarking under *Smart*OS

frequency $f_{\text{timer}} = \frac{1}{61440\mu s} \approx 16.28$ Hz and require 38.47 µs each[33]. This results in a duty cycle of $f_{\text{timer}} \cdot 38.47\mu s \approx 0.063\%$, which yields a good trade-off between energy or CPU overhead and the benefits which come with this approach. If the head of the `timeout_queue` changes, i.e. if timeouts are reached or if a task or interrupt handler invokes any corresponding syscall, the timeline update will occur immediately when leaving the kernel mode. Thus, the update frequency may increase, but on the other hand, the temporal overhead decreases since we need no extra switch to the kernel mode, but do the update while still being there. If the next timeout is less than 61440 µs away, the overflow value will be adapted adequately to allow the new head task's resumption in time.

### 4.4.4. Interrupt Processing Overhead

Regarding the CPU overhead for processing additional (non-timer) interrupts, we evaluate the performance loss for the application tasks according to [194]:

Let $\tau(t, f_{\text{IRQ}})$ be the time a task $t$ requires for the execution of a defined code sequence if it is interrupted by $f_{\text{IRQ}}$ interrupts per second. Then, $t$'s minimal fractional time loss $I(t, f_{\text{IRQ}})$ which results from the invocation of these interrupts calculates as

$$I(t, f_{\text{IRQ}}) = \frac{\tau(t, f_{\text{IRQ}}) - \tau(t, 0)}{\tau(t, 0)}. \tag{4.1}$$

For a representative evaluation of the inherent OS overhead caused by the standardized interrupt processing from Figure 4.5, the ISR itself (as supplied by the application!) should be minimal and contain as few instructions as possible. For our test we registered an empty function as IRQ handler for processing the interrupts of the second system timer (handler execution time: 46.07 µs), and configured it for various frequencies $f_{\text{IRQ}}$. As shown in Figure 4.7, the execution time of the test task $t$ doubles for $f_{\text{IRQ}} \approx 10.6$ kHz. In turn, when taking the system timer into account, the maximal interrupt frequency $f_{\text{IRQ,max}}$ a *Smart*OS system can handle on our particular SNoW[5] hardware from Section 2.2 is

$$f_{\text{IRQ,max}} = \frac{1\,\text{s} - 1\,\text{s} \cdot f_{\text{timer}} \cdot 38.47\,\mu\text{s}}{46.07\,\mu\text{s}} \cdot 1\,\text{Hz} \approx 21692.5\,\text{Hz}. \tag{4.2}$$

[33] The reason for using 0xF000 (61440 µs) instead of 0xFFFF as overflow value for the MSP430's 16 bit timer is a technical detail, and won't be discussed here.

| | ROM [B] | RAM [B] |
|---|---|---|
| *Smart*OS kernel[1] | 5299 | 28 |
| *Smart*OS debugging library | 4154 | 0 |
| CC1100 radio driver | 1466 | 42 |
| *Smart*Net radio protocol | 3142 | 564 |
| CoMem dynamic memory | 2291 | 16 |
| SNoW Ghost remote maintenance | 1535 | 226 |
| UART driver | 162 | 1 |
| SPI driver | 437 | 12 |

[1] Includes the priority inheritance protocol, DynamicHinting, runtime stack checking, and some helper functions

**Table 4.3.:** Maximal code and data sizes for the *Smart*OS kernel and various modules

Consequently, some on-chip peripherals, e.g. for processing data streams over standardized buses (like RS232, SPI, or I2C), can therefore not be handled via interrupts when operating at maximal speed. However, context switching makes the centralized interrupt handling indispensable, and the advantage of true multitasking and automatic interrupt timestamping compensates for this weakness as we will show in various chapters within this work.

### 4.4.5. Further Evaluation Metrics

Apart from the time-related metrics considered so far, code size and energy consumption are very popular numbers to look at, but always depend significantly on the application code and environmental interactions. While the energy consumption was already discussed in Section 2.2.2 and summarized in Table 2.2, the code size is hard to measure. Table 4.3 shows the maximal code size for the kernel and some selected modules. However, these modules contain a lot of rarely used functions, and, since the linker performs function garbage collection, finally shrink to significantly reduced application sizes[34]. E.g. the minimal application from Listing 4.1a consists of just the kernel and the idle task, and requires 4 kB of ROM and 96 B of RAM[35].

[34]The central kernel functionality will not be reduced, though.

[35]Compiled for the SNoW[5] sensor node: 28 B *Smart*OS kernel, 60 B idle task TCB, 8 B idle task stack

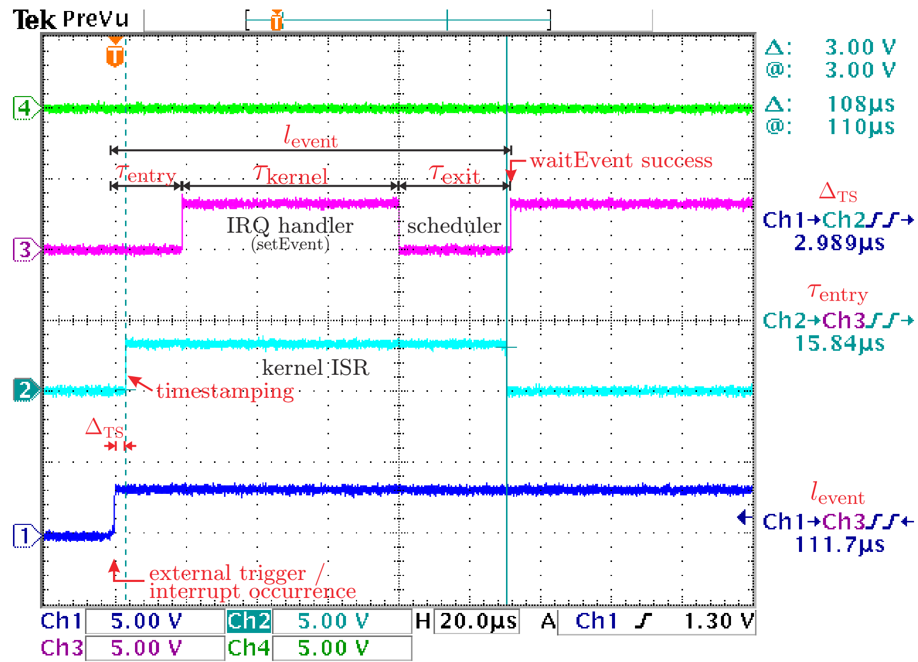



**Figure 4.3.:** Execution of the `setEvent` function when called as IRQ handler

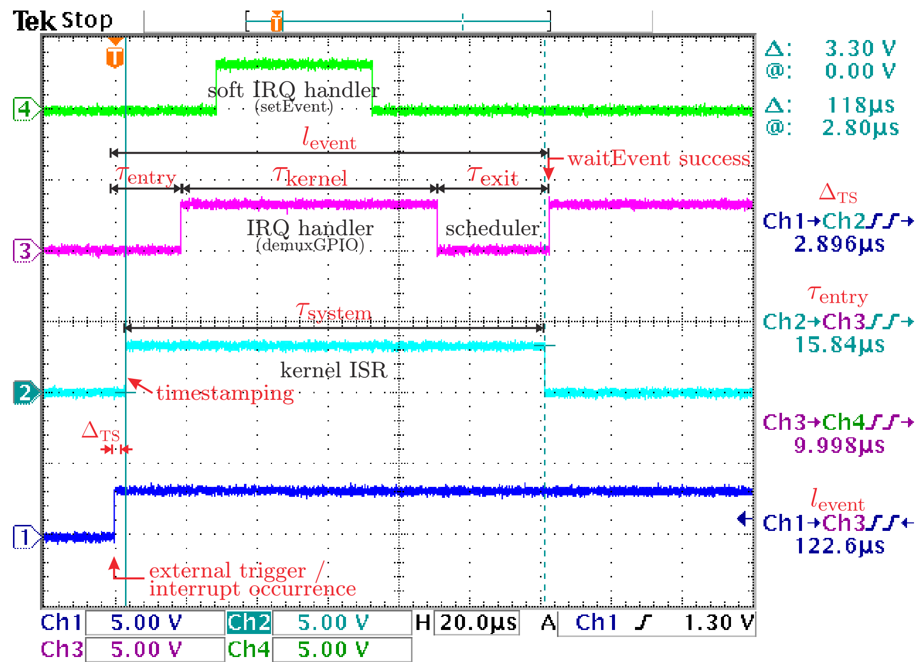



**Figure 4.4.:** Execution of an IRQ handler for I/O port demultiplexing, and a soft IRQ handler for the corresponding pin.

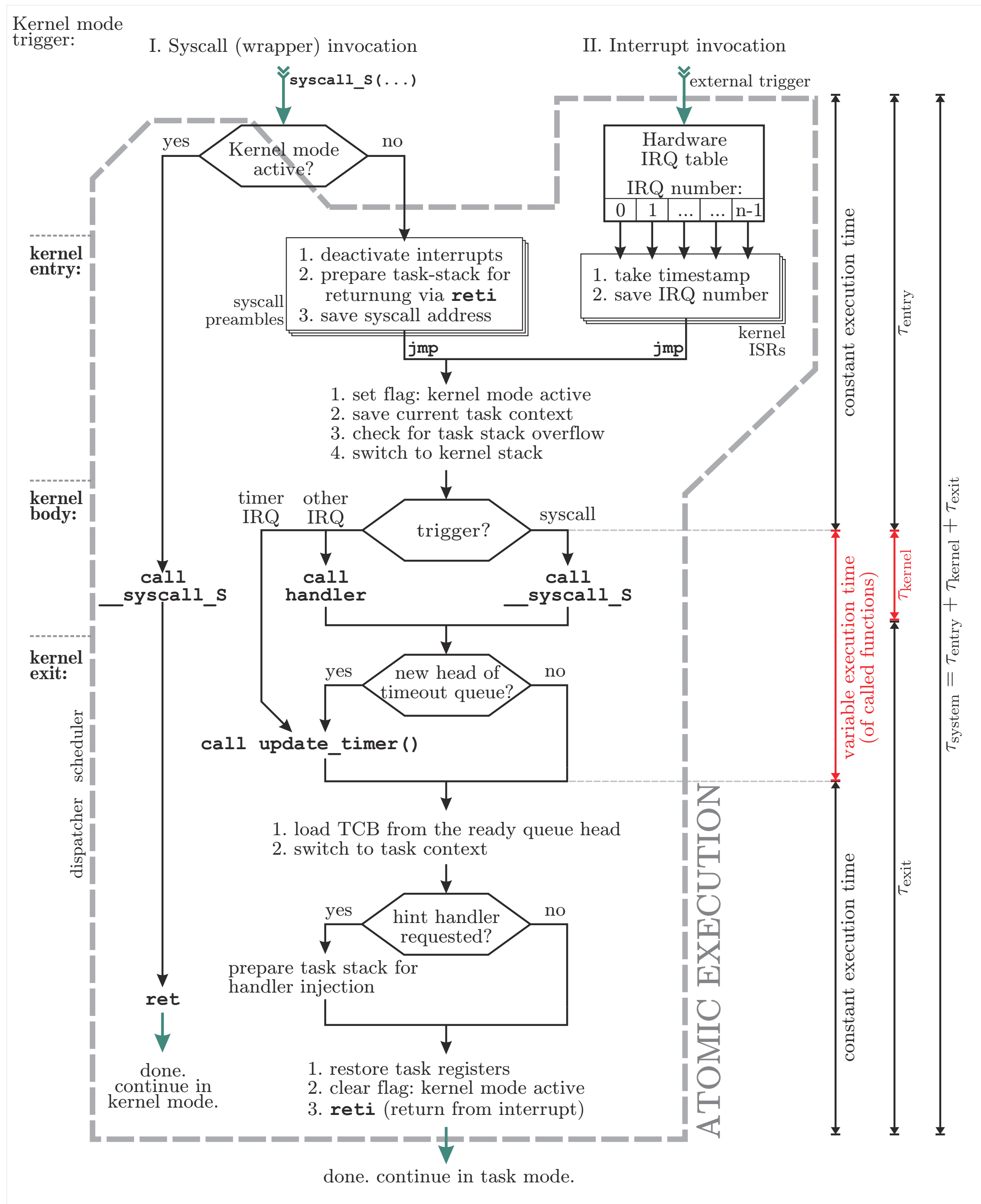



**Figure 4.5.:** The *Smart*OS kernel mode execution flow depending on its trigger: Syscall or interrupt invocation (→ Figure 6.10[p122] for the stack evolution)

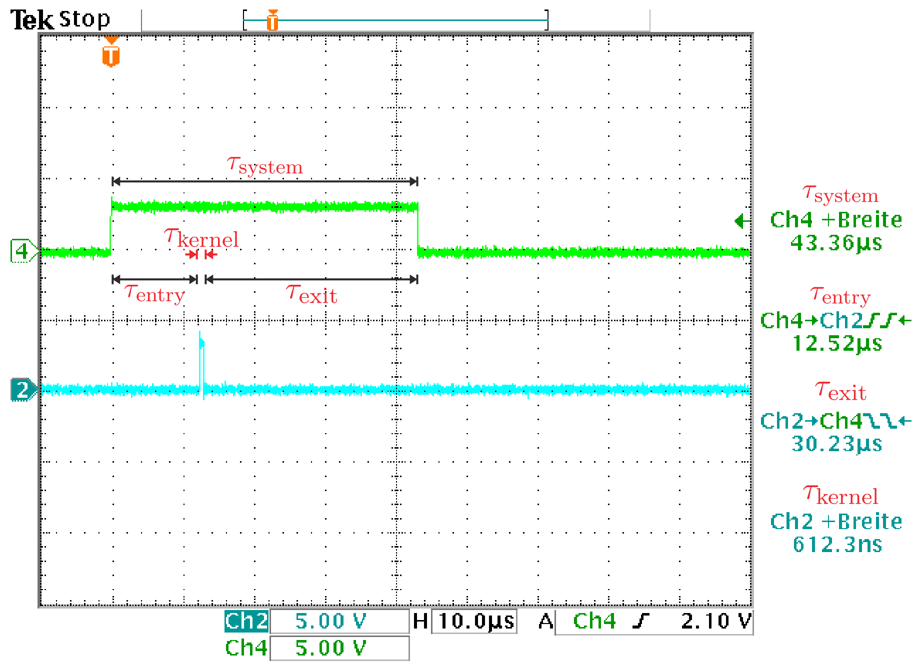



**Figure 4.6.:** Execution time of the `yield` function when called from within a task

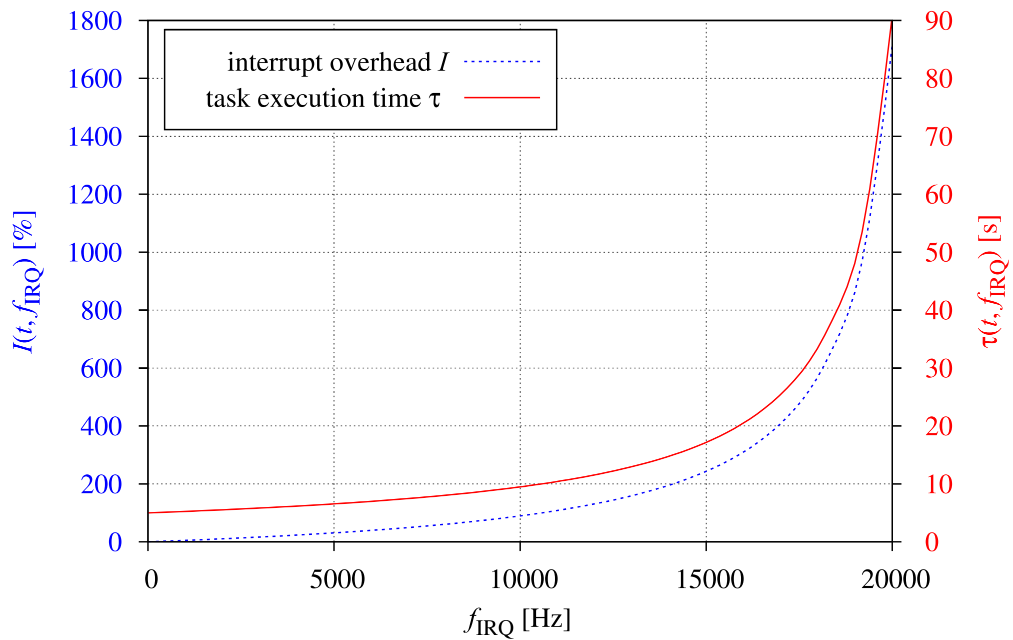



**Figure 4.7.:** Interrupt processing overhead for various interrupt frequencies $f_{IRQ}$ of a test IRQ (in addition to the hardware timer IRQ for the timeline management)

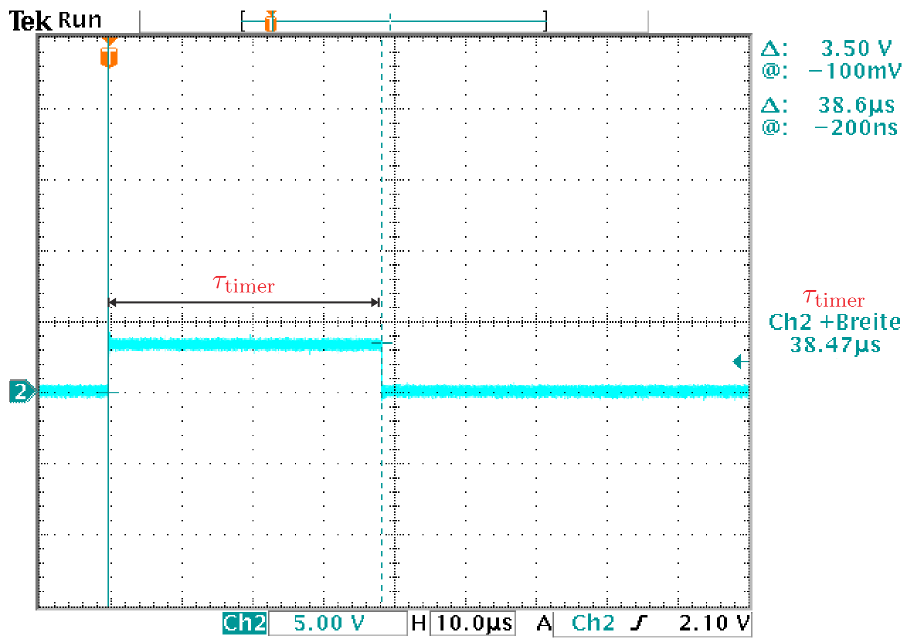



**Figure 4.8.:** Execution time of the system timer ISR
(unloaded system, idle task only)

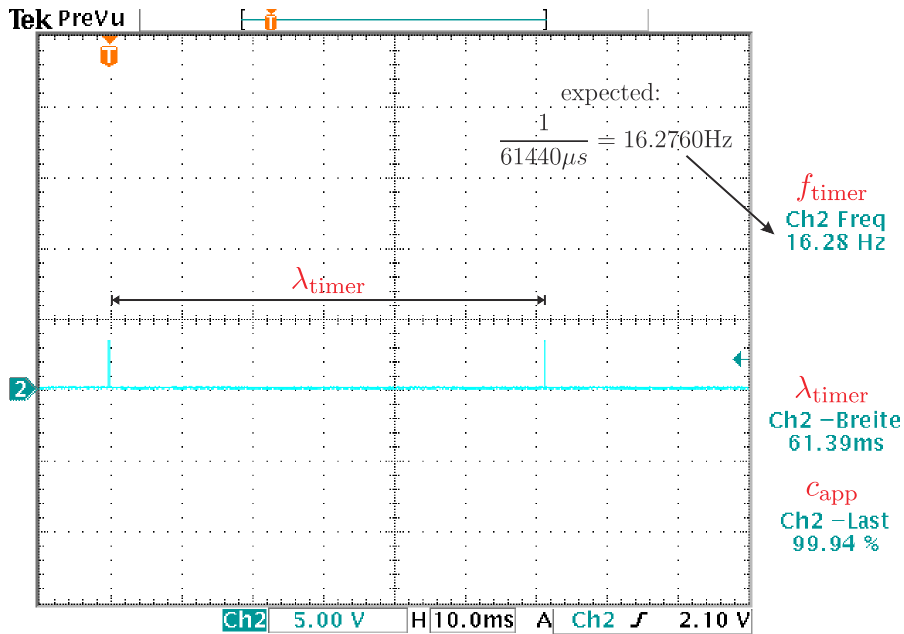



**Figure 4.9.:** Frequency and duty cycle of the timer IRQ
(unloaded system, idle task only)

# 5. Time and Reactivity in *Smart*OS


**Abstract**

In Chapter 4 we discussed the notion of time for bounding the execution of *Smart*OS kernel functions. The proposed temporal semantics of the kernel API allows us to specify timeouts and deadlines for waiting on events and resources without influencing other tasks by spin blocking or active loops. Up to now, this notion of time is individual for each task, and does neither induce temporal dependencies among them (since they have no information about the deadlines of others), nor does it allow to relate time with environmental information and (re-)actions.

While the first problem is relevant for coordinating concurrent task systems and will be discussed in Chapter 6, we will now address the latter problem which is of utmost importance for the precise attribution of environmental events and measurements (→ Def. I.1: The Sense and Aggregate Paradigm), as well as for the precise scheduling and execution of corresponding reactions (→ Def. I.2: The Sense and React Paradigm) in sensor/actuator systems.

In this chapter we'll initially summarize the related problems which result from the discrete time measurement in digital systems. Subsequently, we'll present a novel technique for the automatic creation of highly precise timestamps for external events, as well as for the scheduling of related (re-)actions and processes. Managed by the operating system kernel at the lowest possible software level, we achieve a symmetric error interval for the (true) timestamps and the scheduled reaction times – both with an average error close to 0 µs. Based on this symmetry, we'll subsequently introduce a dynamic self-calibration technique for managing the compliance with these times, i.e. to achieve the temporally exact execution of the corresponding actions. An application example will show that the integration of these techniques into *Smart*OS allows to determine the clock drift between two (or more) independently running embedded systems without exchanging any explicit information, except for the mutual triggering of periodic interrupts. In fact, a real-world test bed achieved a precision of ±3 µs in the worst case for the drift computation, and provides a basis for further applications and services like e.g. time synchronization and self-organization techniques from Chapters 11 and 12.

## 5.1. Introduction on Information Attribution

Temporal and spatial information are the two most fundamental measures for the "*attribution*" or "*tagging*" of states and events (i.e. state transitions) within any observed environment. In this context, the states describe a set of physical and logical conditions at a given *position* $\boldsymbol{x}$ and at a certain *time* $t$, and they are specified by one or more continuous or discrete *state values* $s_i$. Though position and time can obviously also be considered as state values, we'll attach a special meaning to them, and pay special attention to their generation:

While temporal information can initially be obtained through purely local measurements, obtaining spatial information is a more complex task in general, and commonly requires the fusion of various previously collected data and information (e.g. angles or distances towards well-known reference points). Thus we'll defer the latter problem to Part III of this work, but already introduce both attributes at once, since they are inherently linked to each other. In fact, measured or otherwise obtained environmental information is sometimes of no use unless it is associated with temporal and spatial information[1].

Recorded over a certain period of time, variations in the state values allow the detection and analysis of events and event patterns [248, 308] within the environment. These variations do not only indicate the events' spatial extension, propagation speed, and influence on the environment, but most commonly they also allow the prediction of future states for both the observing system and its surrounding. In this regard, the interaction with the environment, which we already proclaimed in Chapter 1 to be the most central objective in sensor/actuator systems, typically requires the precise knowledge of time and space to be associated with self-captured and foreign values from other nodes in order to properly correlate the contained information, and to trigger adequate reactions.

While the position information is commonly considered as a three dimensional vector $\boldsymbol{x} = (x, y, z)^T$ which can change freely over time and in any of its components, the time $t$ is always a one dimensional scalar which advances continuously at a fixed rate.

From the perspective of a perfect external observer, the state values $s_i$ are available for any given position $\boldsymbol{x} \in \mathbb{R}^3$ and time $t \in \mathbb{R}$. In particular they are always free from any error. Throughout this work we'll denote correct values as $s'_i, \boldsymbol{x}'$, and $t'$, respectively, and represent the environment's *true state* $S'(\boldsymbol{x}, t)$ at any queried position $\boldsymbol{x}$ and time $t$ as

$$S'(\boldsymbol{x}, t) := (\boldsymbol{x}', t', s'_1, \ldots, s'_m)^T \quad \text{with} \quad \boldsymbol{x}' = \boldsymbol{x} \quad \text{and} \quad t' = t. \tag{5.1}$$

From the perspective of an autonomous sensor system with a restricted view on the environment, the local and remote state values $s_i$ can only be obtained for some positions $\boldsymbol{x} \in \mathbb{R}^3$ and times $t \in \mathbb{R}$. While some of these values are directly available (e.g. from local sensor readings),

[1] Since many applications do not request for explicit location awareness, its integration into an operating system kernel is commonly omitted entirely. Although today's mobile communication devices (e.g. Smartphones) increasingly often integrate so called *location based services* (LBS), and thus need to know the user's position, the location information is commonly determined at higher levels.

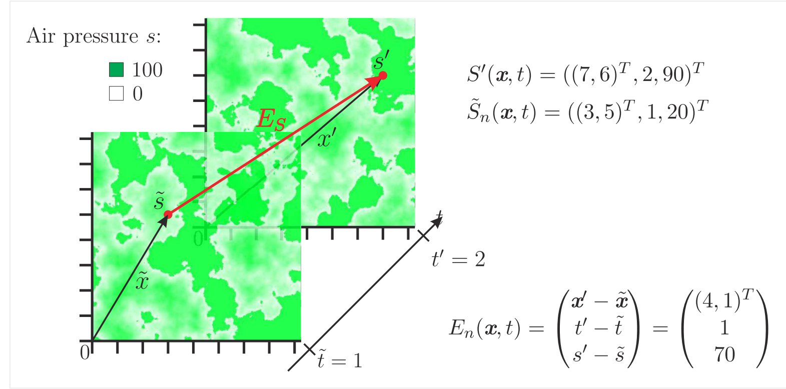



**Figure 5.1.:** Error in the state vector of a node $n$ regarding a 2D position $\boldsymbol{x}$ and time $t$

others must be derived algorithmically first (e.g. via sensor data fusion [118]). However, since these values are also based on prior measurements, they are never perfect but always exhibit some error. This is also and especially true for the associated time and position information. Henceforth, we'll denote imprecise values as $\tilde{\boldsymbol{x}}$, $\tilde{t}$, and $\tilde{s}_i$, respectively, and define the *observed state* $\tilde{S}_n(\boldsymbol{x}, t)$ of a node $n \in N$ for a queried position $\boldsymbol{x}$ at time $t$ as

$$\tilde{S}_n(\boldsymbol{x}, t) := (\tilde{\boldsymbol{x}}, \tilde{t}, \tilde{s}_1, \ldots, \tilde{s}_m)^T. \tag{5.2}$$

For the real system both $\boldsymbol{x}$ and $t$ need not necessarily match $\tilde{\boldsymbol{x}}$ and $\tilde{t}$, respectively, since even if the system expects its own values to relate to $\boldsymbol{x}$ and $t$ they might relate to $\tilde{\boldsymbol{x}}$ and $\tilde{t}$ instead. Based on the true and the observed values we define the error $E$ in the position $\boldsymbol{x}$, the time $t$, and each state value $s_i$ as

$$E_{\boldsymbol{x}} := \boldsymbol{x}' - \tilde{\boldsymbol{x}}, \qquad E_t := t' - \tilde{t}, \qquad \text{and} \qquad E_{s_i} := s_i' - \tilde{s}_i. \tag{5.3}$$

As exemplified in Figure 5.1 we also define the multidimensional and system specific error vector

$$E_n(\boldsymbol{x}, t) := S'(\boldsymbol{x}, t) - \tilde{S}_n(\boldsymbol{x}, t). \tag{5.4}$$

A detailed example for evaluating position estimation algorithms will be given in Chapter 13.

## 5.2. Time in Digital Systems

In contrast to the specific position vectors of e.g. sensor nodes (which can change sporadically, independently from each other, and arbitrarily in any of their components), time is a common property. It is system independent, and advances continuously with a globally constant rate of change[2]. If the sensor nodes manage to establish a network-wide and consistent notion of time, this information provides a natural base for their joint interaction among each other and with the environment. Since processors in synchronous digital systems like sensor nodes are always

[2]At least we expect this to be true, and simply ignore the theory of relativity when considering individual systems and networks. Furthermore, it is easy for corresponding sensor networks to get over location specific variations in time, since today's sensor nodes cannot be synchronized sufficiently precise anyway to observe this phenomenon, or to be considerably affected by it.

driven by a clock generator $C$ with frequency $f_C$ and period $\lambda_C = \frac{1}{f_C}$, time and time intervals can easily and independently be measured[3] – at least in theory: If it is possible to count the number of elapsed clock periods since system start, each captured event $e$ – e.g. indicated by an interrupt as described in Section 4.3.5 – can be attributed with the current counter value $c_e$. Consequently, the event's absolute local system time $\tilde{t}_e$ can easily be recovered by

$$\tilde{t}_e := c_e \cdot \lambda_C, \tag{5.5}$$

and the time difference (i.e. the delay) $\tilde{\Delta}_{e_1,e_2}$ between two events $e_1, e_2$ computes as

$$\tilde{\Delta}_{e_1,e_2} := \tilde{t}_{e_2} - \tilde{t}_{e_1} = (c_{e_2} - c_{e_1}) \cdot \lambda_C. \tag{5.6}$$

Obviously, both the time $\tilde{t}_e$ and the delay $\tilde{\Delta}_{e_1,e_2}$ already involve a certain imprecision due to the discretized counters $c_e \in \mathbb{N}$. In addition, we silently assumed for Eq. (5.5) and Eq. (5.6) that $\lambda_C$ is perfectly known and constant. Neither is true under real-world conditions.

Finally, as often requested for interactive systems, a reaction $r$ can be scheduled for a captured event $e$. Its intended execution time $t'_r \in \mathbb{R}$ is commonly related to any $t'_e \in \mathbb{R}$ by the specification of a corresponding delay $\Delta'_{e,r} \in \mathbb{R}$:

$$t'_r = t'_e + \Delta'_{e,r} \tag{5.7}$$

However, the reaction will in the best case, i.e. if the scheduler permits the timely switch to the responding task's context, be triggered upon reaching the corresponding counter value $c_r \in \mathbb{N}$ and the corresponding system time $\tilde{t}_r$:

$$c_r = c_e + \left\lfloor \frac{\Delta'_{e,r}}{\lambda_C} \right\rfloor \qquad \tilde{t}_r = \tilde{t}_e + \left\lfloor \frac{\Delta'_{e,r}}{\lambda_C} \right\rfloor \cdot \lambda_C \tag{5.8}$$

Although the described imprecision is quite intuitive and well-known, it is commonly simply accepted or ignored. Nevertheless it introduces certain "hidden" implementation problems in real systems; in particular since the temporal error is neither constant nor predictable. In the following we'll indicate and discuss the causes and effects of these problems, and present an approach to reliably compensate the related imprecision in the average case.

**Problem P1: Discretization of time.** The difference between the true global time and the individual system time has already become visible in Eq. (5.5) and Eq. (5.8). While the first progresses continuously, the use of a digital counter leads to a discretization of the latter, and imposes a resolution which is strictly proportional to the counter's clock frequency $f_C$. As illustrated in Figure 5.2, this may lead to serious systematic errors for the time measurement and the subsequent scheduling of reactions:

The simple *capturing of timestamps t* for external events – the so called *timestamping* – is immediately affected by the inevitable rounding, and suffers from a measurement error $E_t \in I_1$ with $|I_1| = \lambda_C$. For the naïve and adverse reading of the counter in Figure 5.2a, rounding down

[3]See Figure 2.4[p25] for a concrete CPU example regarding the SNoW[5] platform: $f_{\text{MCLK}} = 8\,\text{MHz}$.

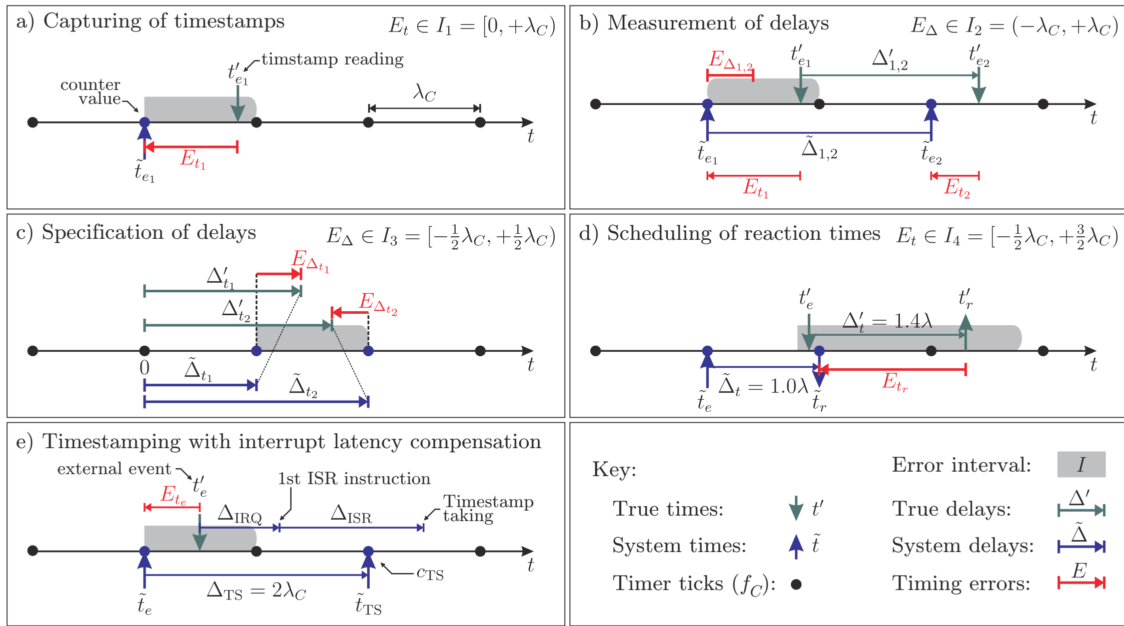



**Figure 5.2.:** Emergence of timing errors for the naïve discretization of time

results in $I_1 = [0, \lambda_C)$, and induces a symmetry around the average measurement error $E_t = \frac{1}{2}\lambda_C$. Depending on the use of such timestamps, the emerging errors might accumulate during the system runtime. According to Figure 5.2c, the explicit *specification of delays* $\Delta'_t$ in software is also subject to rounding errors $E_\Delta$. However, since we can round half up[4] manually this time, $E_\Delta \in I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$. Thus $I_3$ is at least symmetric around 0, and the average error is 0 µs.

Based on these two fundamental error intervals $I_1$ and $I_3$, others can be derived, and consequently exhibit an imprecision, too: For the *measurement of delays* $\Delta_E$, as depicted in Figure 5.2b, we see the implicit compensation of the asymmetry in $I_1$: $E_\Delta \in I_2 = I_1 - I_1 = (-\lambda_C, +\lambda_C)$. In contrast, the *scheduling of reaction times* $t$ on external events inherits the asymmetry in $I_1$: $E_t \in I_4 = I_1 + I_3 = [-\frac{1}{2}\lambda_C, +\frac{3}{2}\lambda_C)$. As illustrated in Figure 5.2d, system reactions will consequently suffer from an average systematic lateness of $\frac{1}{2}\lambda_C$. The resulting effects, and our proposed solution to compensate this asymmetry, will be discussed later.

Table 5.1 summarizes the errors and their intervals which must be expected for the naïve capturing of timestamps by simply reading the timer register (e.g. within an IRQ handler).

**Problem P2: Capturing of timestamps.** The creation of reactive systems demands for the precise assignment of timestamps for external events. Reaching a threshold within an analog-digital-converter (ADC) or detecting a signal edge at an I/O pin are just two simple examples. However, almost all observable changes within the environment have in common, that they are indicated to the CPU at runtime by so called *interrupt requests* (IRQs), and should be handled as soon and fast as possible by the corresponding *interrupt service routines* (ISRs) (→ Section 4.3.5 and Figure 4.5[p66]). Since ISRs are commonly more privileged than regular application code,

[4] e.g. according to DIN 1333

| error type | derived from | error interval (naïve discretization) | symmetry (naïve discretization) | error interval (compensated discretization) | symmetry (compensated discretization) |
|---|---|---|---|---|---|
| capturing of timestamps | fundamental | $I_1 = [0, \lambda_C)$ | $\frac{1}{2}\lambda_C$ | $I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$ | 0 |
| measurement of delays | $I_1 - I_1$ | $I_2 = (-\lambda_C, +\lambda_C)$ | 0 | $I_2 = (-\lambda_C, +\lambda_C)$ | 0 |
| specification of delays | fundamental | $I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$ | 0 | $I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$ | 0 |
| scheduling of reaction times | $I_1 + I_3$ | $I_4 = [-\frac{1}{2}\lambda_C, +\frac{3}{2}\lambda_C)$ | $\frac{1}{2}\lambda_C$ | $I_4 = [-\lambda_C, +\lambda_C)$ | 0 |

**Table 5.1.:** Error intervals for different discretization techniques (system time resolution: $\lambda_C$)

and will preempt it for their own execution, they seem to be perfectly suitable for capturing the timestamp for any emerging event. However, as depicted in Figure 5.2e, even the first instruction within each ISR is not executed before some additional delay, which is also known as *interrupt latency* $\Delta_{\mathrm{IRQ}}$. If the timer value $c_{\mathrm{TS}}$ for the timestamp itself is copied after another delay $\Delta_{\mathrm{ISR}}$ within the ISR, then we can compute the discrete timestamp $\tilde{t}_e$ for the captured event $e$ as follows:

$$\tilde{t}_e = c_{\mathrm{TS}} \cdot \lambda_C - (\Delta_{\mathrm{IRQ}} + \Delta_{\mathrm{ISR}}) = \tilde{t}_{\mathrm{TS}} - \Delta_{\mathrm{TS}} \tag{5.9}$$

Hence, a prerequisite for reliable time tracking via Eq. (5.9) is, that the correction value $\Delta_{\mathrm{TS}}$ is constant and free from rounding errors with respect to the discrete system time period.

**Problem P3: Simultaneity and scheduling reliability.** Although the perfectly simultaneous transition of two states can never occur in real systems[5], the surjective discretization of time can easily lead to the assignment of exactly the same system time for multiple events or scheduled actions. Since resource conflicts prevent the truly parallel processing of events as well as the simultaneous execution of (re)actions, and usually lead to an implicit serialization, their prioritization depends on the task scheduler or the task internal order. Since there is most commonly only a single IRQ controller, this is already true for the generation of timestamps. In fact, the maximum degree of parallelism is always limited by the number of available functional units[6]. A safe scheduling – e.g. to meet hard real-time demands – must be achieved by either static techniques at development time or dynamic methods at runtime. In this context, we'll present the DynamicHinting technique for dynamic resource management under real-time conditions in Chapter 6.

**Problem P4: Imprecision in the timer frequency.** Time measurement in digital systems is usually accomplished by using a pulse generator with specified frequency $f_0$. Internally, this component uses an oscillator – most commonly a quartz crystal – to generate a periodic clock

[5] The time measurement resolution must simply be chosen fine enough to increase the improbability for observing simultaneity!

[6] Functional units refer to e.g. processors and their cores, or autonomous peripheral components.

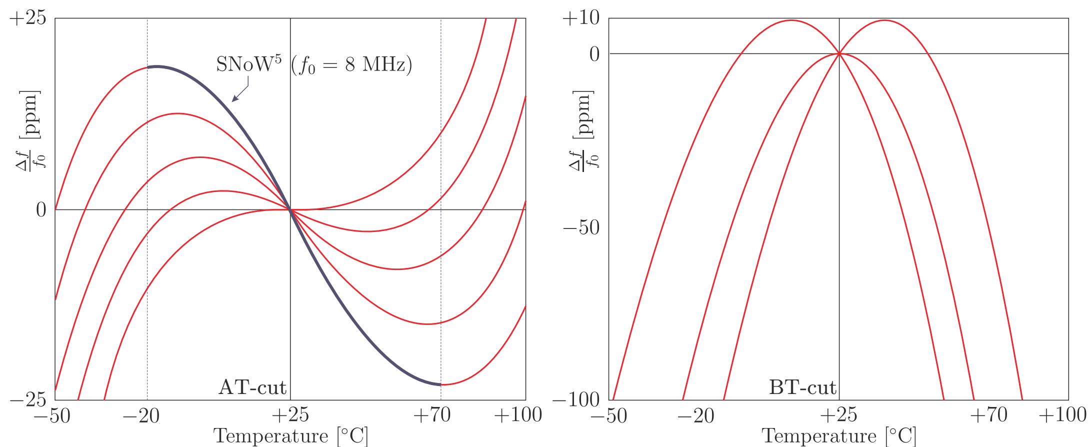



**Figure 5.3.:** Temperature sensitivity of typical HC49 quartz crystals (for various cutting angles)

signal. Since the characteristics and stability of such oscillators depend significantly on their manufacturing parameters, their age, and on various environmental conditions [125], a varying frequency drift $\Delta f$ must always be expected. The relative error $\frac{\Delta f}{f_0}$ is commonly expressed in units of ppm (parts per million). Figure 5.3 gives an example for the temperature sensitivity[7] of a typical HC49 quartz, as applied on the SNoW[5] sensor node from Section 2.2. For simple low-cost quartzes, and within the typical temperature ranges of WSAN applications, this can already result in deviations of ± 20 ppm[8].

Variations in the clock precision are especially critical in distributed applications. Since time measurement is initially individual for each involved system and therefore can drift apart (→ Figure 5.6), this may quickly generate inconsistent data, and must be compensated by adequate synchronization measures. The SNoW Bat indoor localization system as described in Part III of this work will illustrate this in several ways.

**Problem P5: Global time base and synchronization with other systems.** When does time measurement actually start, i.e. when is or was time $t_0 = 0$? If we consider an independent system which uses time only for its internal operation, e.g. to capture events and to schedule actions by a partial order[9], the use of a pure local time with arbitrary begin is absolutely sufficient – e.g. time $t_0 = 0$ may simply indicate the system start[10]. However, as soon as time is of global relevance, e.g. if actions have to take place synchronized on different systems, a common time base is often indispensable. This immediately raises the question about which time or system is used as a reference. In any case, the provider should be highly available and exhibit a high clock stability and precision. Several methods exist for the actual synchronization: These are either based on (regular) time checks or on the measurement of the pairwise drift between

[7] ... which in turn depends on the quartz material's temperature coefficient and its cutting angle ...

[8] Other variations of $f_0$ are caused by fluctuations in the applied voltage, the angle dependent influence of gravity and motion, and electromagnetic fields. However, these errors are small and negligible in typical sensor network applications. Further information can be found in [125].

[9] partial since the discretization of time may lead to simultaneity

[10] On the other hand, a completely independent system is also completely useless.

the involved systems. While dedicated reference systems allow synchronization based on centrally triggered events[11] (e.g. radio broadcasts as specified in the DCF77 protocol), distributed methods are available for multi-hop systems to successively achieve a common time base (e.g. via Desynchronization [207]).

## 5.3. Time in *Smart*OS

Considering the aforementioned problems, which originated from the integration of time-awareness into digital systems, P1-P3 directly affect the environmental interaction and can be addressed by each system individually. In contrast, P4 and P5 require some information exchange with other systems, which are however not necessarily available at all times. For this reason, P1-P3 are treated directly within the *Smart*OS kernel, while P4 and P5 must be addressed at higher layers.

As already described in Section 4.3.4, *Smart*OS relies on a periodic timer component to drive its local timeline. Based on this the kernel automatically captures a timestamp $\tilde{t}_e$ for each interrupt $e$, and compensates the error's asymmetry about 0 which would result from using the naïve approach with $I_1 = [0, \lambda_C)$ as explained in Section 5.2. Therefore, the centralized interrupt processing by standardized kernel ISRs is exploited to introduce a constant and carefully dimensioned delay $\Delta_{\mathrm{ISR}}$ for capturing the timer's counter value. According to Eq. (5.9) we have to apply an adequate correction value $\Delta_{\mathrm{IRQComp}}$ for the total delay $\Delta_{\mathrm{TS}} = \Delta_{\mathrm{IRQ}} + \Delta_{\mathrm{ISR}}$. Selected properly, this correction finally results in the symmetry about 0 for $I_1 = [-\frac{1}{2}\lambda, \frac{1}{2}\lambda)$, and in turn reduces the average timestamp error from initially $\frac{1}{2}\lambda$ down to 0 µs (while $E_{t_e}$ will still be equally distributed over $I_1$). At the same time, the propagation and amplification of systematic errors for time-dependent reactions, will also be kept low and symmetric about 0 µs, i.e. $I_4 = I_1 + I_3 = [-\lambda_C, +\lambda_C)$. Table 5.1 compares the error intervals of our compensation approach with the naïve technique.

In order to deal with the related problems P1 and P2, we propose a concept based on two synchronized clocks with interdependent frequency. Thereby, we assume the CPU frequency to be higher than the timer frequency, while conversely, the system time is derived from the quartz-stabilized CPU clock by an even integer divider. As already demanded in Chapter 2, both requests do not impose an unreasonable restriction on the hardware/software design: In fact they are already satisfied in many systems, since usually only a single central oscillator is used as base for all other system clocks. While the CPU is commonly directly driven by this main clock, other components apply power-of-two dividers to derive their individual frequencies. Finally, and for constrained embedded systems in particular, driving a local time with the maximum resolution would cause unnecessary CPU load[12].

Besides the following formal description of our approach, we also refer to the example in

[11]. . . which must be directly receivable for all participants since forwarding induces additional problems . . .

[12]The system time must be accumulated in software at every timer overflow. Especially for timers with small word widths, this can quickly lead to a huge performance penalty. Thus, as shown in Table 3.1[p46], most WSAN operating systems with an integrated system time support a resolution of just 1 ms (which is too imprecise for many application scenarios as we will see in Chapter 11 on ultrasound distance measurement).

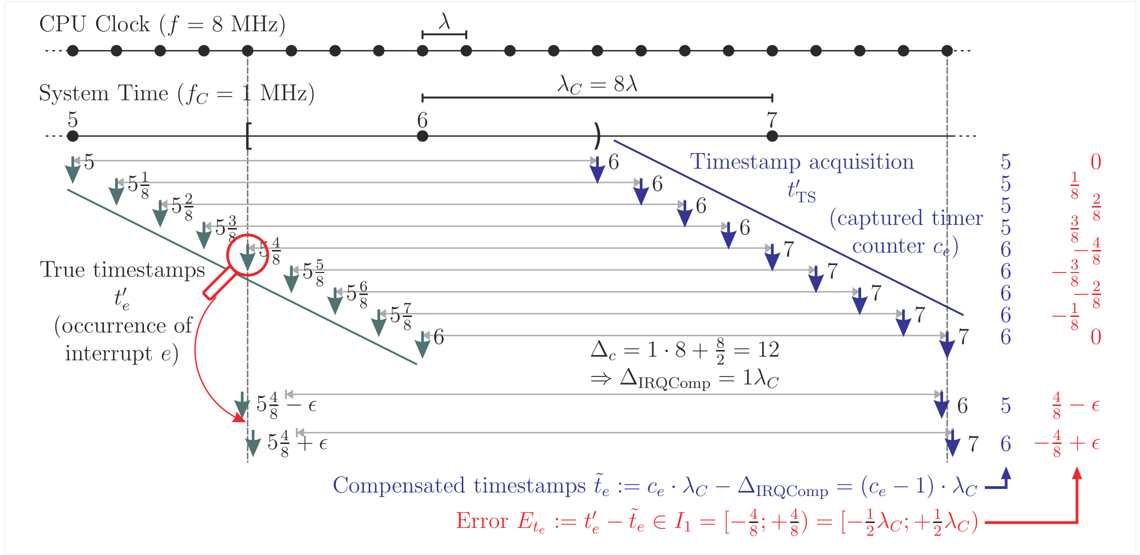



**Figure 5.4.:** IRQ timestamp acquisition under *Smart*OS

Figure 5.4 for a comprehensive understanding. Initially, we denote the CPU clock frequency and period as $f$ and $\lambda$, and the system time frequency and period as $f_C$ and $\lambda_C$, respectively. In addition, we demand for

$$f_C := \frac{1}{\alpha} \cdot f \quad \text{and} \quad \lambda_C := \alpha \cdot \lambda \quad \text{with} \quad \alpha \in \mathbb{N}^{\geq 2}, \alpha \text{ even.} \tag{5.10}$$

If an interrupt $e$ occurs at time $t'_e$, the corresponding timer counter $c_e$ will not be copied before some system inherent delay $\Delta_{\mathrm{TS}}$ has passed. Specifically, we request this delay to comprise exactly $\Delta_c$ CPU cycles as follows:

$$\Delta_c := n \cdot \alpha + \frac{1}{2} \cdot \alpha \quad \text{with} \quad n \in \mathbb{N}_0^+ \tag{5.11}$$

Thus, the delayed acquisition of the timestamp takes place at time

$$t'_{\mathrm{TS}} = t'_e + \Delta_{\mathrm{TS}} = t'_e + \Delta_c \cdot \frac{1}{f} = t'_e + \left( n \cdot \alpha + \frac{1}{2} \cdot \alpha \right) \cdot \frac{1}{f}. \tag{5.12}$$

To compensate for this delay, and to force the timestamp error interval $I_1$ to become symmetric around the true event occurrence time while also exhibiting an average error close to 0 µs, we select the correction value as

$$\Delta_{\mathrm{IRQComp}} := (n \cdot \alpha) \cdot \frac{1}{f} = n \cdot \lambda_C. \tag{5.13}$$

Finally, we simply have to subtract $n$ from the copied timer value $c_e$ to compute the timestamp

$\tilde{t}_e$ for the interrupt $e$:

$$\tilde{t}_e = \left\lfloor \frac{t'_{\mathrm{TS}}}{\lambda_C} \right\rfloor \cdot \lambda_C - \Delta_{\mathrm{IRQComp}} = c_e \cdot \lambda_C - n \cdot \lambda_C = (c_e - n) \cdot \lambda_C \tag{5.14}$$

Since $c_e$ (timer driven) and $n$ (constant) are integers of architecture word width, their subtraction is easily accomplished. Besides, the result's resolution equals the resolution of the system time, i.e. 1 µs under *Smart*OS. However, we still have to prove the symmetry about 0 µs for the error intervals in Table 5.1; in particular we have to show that $\tilde{t}_e \in I_1 = [-\frac{1}{2}\lambda_C, \frac{1}{2}\lambda_C)$.

**Lemma II.1.** *The error intervals $I_1, I_2, I_3$, and $I_4$ for taking timestamps, for measuring and specifying delays, as well as for computing reaction times are symmetric about 0.*

*Proof.* While $I_3$ is not affected by our novel approach, the demanded symmetry of $I_1, I_2$, and $I_4$ in particular can easily be proofed by some interval arithmetic. The expected error $E_{t_e}$ of the timestamp $\tilde{t}_e$ computes as

$$\begin{aligned}
E_{t_e} = t'_e - \tilde{t}_e \quad &\overset{(5.12),(5.14)}{=} \quad \left[ t'_{\mathrm{TS}} - \left( n \cdot \alpha + \frac{1}{2} \cdot \alpha \right) \cdot \frac{1}{f} \right] - \left[ \left\lfloor \frac{t'_{\mathrm{TS}}}{\lambda_C} \right\rfloor \cdot \lambda_C - n \cdot \lambda_C \right] \\
&= \quad t'_{\mathrm{TS}} - \left( n \cdot \alpha + \frac{1}{2} \cdot \alpha \right) \cdot \frac{\lambda_C}{\alpha} - \left\lfloor \frac{t'_{\mathrm{TS}}}{\lambda_C} \right\rfloor \cdot \lambda_C + n \cdot \lambda_C \\
&= \quad \underbrace{t'_{\mathrm{TS}} - \left\lfloor \frac{t'_{\mathrm{TS}}}{\lambda_C} \right\rfloor \cdot \lambda_C}_{\in [0, \lambda_C)} - \frac{1}{2} \cdot \lambda_C \quad \in \quad \left[ -\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C \right)
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \quad & I_1 = \left[ -\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C \right) \\
\Rightarrow \quad & I_2 = I_1 - I_1 = \left[ -\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C \right) - \left[ -\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C \right) = (-\lambda_C, +\lambda_C) \\
\Rightarrow \quad & I_4 = I_1 + I_3 = \left[ -\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C \right) + \left[ -\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C \right) = [-\lambda_C, +\lambda_C)
\end{aligned}$$

□

As concrete example, we'll take a look at the reference implementation of *Smart*OS for the MSP430F1611 [280] MCU and the SNoW[5] sensor nodes. While the main clock drives the CPU at $f = 8\,\mathrm{MHz}$, the divider $\alpha = 8$ derives the frequency $f_C = 1\,\mathrm{MHz}$ for the system time. According to Eq. (5.11), adequate delays between each interrupt occurrence and the acquisition of its timestamp are

$$\Delta_c := n \cdot \alpha + \frac{\alpha}{2} = n \cdot 8 + 4 \quad \text{with} \quad n \in \mathbb{N}_0^+. \tag{5.15}$$

```
1  ; Kernel ISR for IRQ number e
2  __hwirq_e:
3    ;---------------------------- TIMESTAMPING ------------------------------;
4    ; hardware IRQ latency          ; +6 CPU cycles from t'_e          Δ_IRQ = 6λ
5    nop                            ; +1 CPU cycle                \
6    nop                            ; +1 CPU cycle                 > Δ_ISR = 6λ
7    mov &TIMER_COUNTER, &__hwirq_TS ; +4 CPU cycles for latch in /
8    ; Total delay of captured timestamp:                           Δ_TS = 12λ = 1.5μs
9
10   ;--------------------- PREPARE AND ENTER KERNEL MODE ----------------------;
11   mov #e, &__hwirq_number    ; save IRQ number for processing in kernel body
12   jmp __kernel_entry         ; jump to kernel mode
```

**Listing 5.1:** Timestamping within the kernel ISR for an IRQ $e$

```
inline void getIRQTime(Time_t *time) {
  *time = (timeline + __hwirq_TS) - __hwirq_n; // t̃_e = (c_e − n) · λ_C  (→ Eq.(5.14))
}
```

**Listing 5.2:** Delayed timestamp calculation for the last interrupt

Listing 5.1 shows the kernel ISR for the interrupt $e$: Since the CPU inherently delays the acceptance of an interrupt by $\Delta_{\text{IRQ}} = 6$ CPU cycles, we already have to select $n \geq 1$. In fact, we did select $n = 1$ and thus have to wait for an additional number of $\Delta_{\text{ISR}} = 6$ CPU cycles within the ISR ($1 \cdot 8 + 4 = 6 + 6$). According to the specification of the `mov` instruction, which is used for saving the timer value `__hwirq_TS` in Line 7, it takes 4 CPU cycles until the value is read from the special function register `TIMER_COUNTER`. The remaining two cycles are filled up by `nop` instructions. After the acquisition of the counter value, the specific IRQ number is saved and the kernel mode is entered for the actual event handling (→ Figure 4.5[p66]).

To save CPU time the ISR will initially only save the current 16 Bit timer value which indicates the delay since the last timeline update. The computation of the final absolute timestamp is avoided, and delayed until the IRQ handler requests this information via the `getIRQTime(...)` function from Listing 5.2. According to Eq. (5.13), $n$ can simply be subtracted from $c_e$, which in turn is the sum of the timeline and the just captured timer value. The result can directly be interpreted as absolute system time $\tilde{t}_e$ in μs. Note that the applied computation is always correct, since IRQ handlers are always executed in kernel mode where further interrupts are disabled and neither the `timeline` nor `__hwirq_TS` will change concurrently (→ Figure 4.5[p66]).

## 5.4. Test Bed: Node Self-Calibration and Pairwise Drift Calculation

The test bed for demonstrating the benefit of our timestamping approach consists of pairs of nodes $A, B$ playing some sort of Ping Pong game as depicted in Figure 5.5: By a wired or wireless connection, one node, WLOG $B$, triggers an IRQ signal $e_0$ which is received and timestamped ($\tilde{t}_0$) by the other node $A$ through the just presented *Smart*OS timestamping technique. After some fixed delay $\Delta_{\text{delay}}$ the signal will be returned by $A$, and in turn the other node $B$ catches

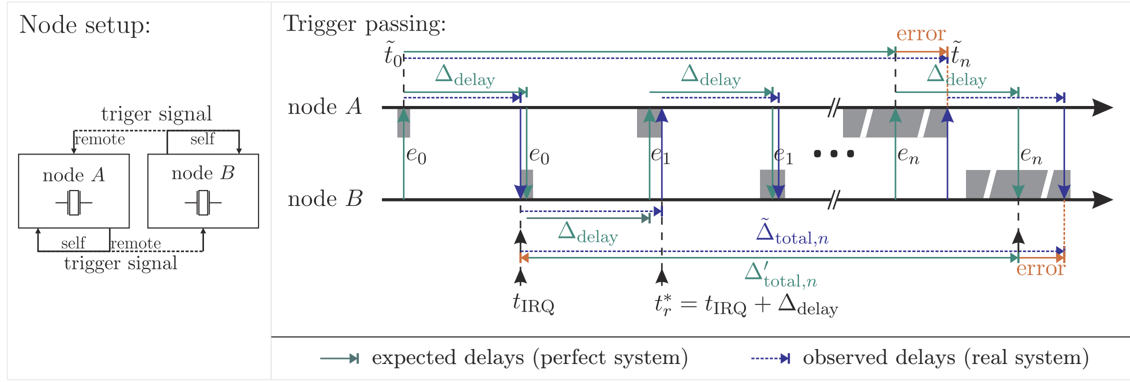



**Figure 5.5.:** The Ping Pong test bed for evaluating the timing precision of *Smart*OS

and returns the signal after the same delay $\Delta_{\text{delay}}$. Having received the last trigger $e_n$ with local timestamp $\tilde{t}_n$ in a *perfect system* (green arrows), the observed delay $\tilde{\Delta}_{\text{total},n}$ between each node's captured first and last signal timestamp should obviously equal the theoretical delay $\Delta'_{\text{total},n}$:

$$\tilde{\Delta}_{\text{total},n} := \tilde{t}_n - \tilde{t}_0 \stackrel{!}{=} 2n \cdot \Delta_{\text{delay}} =: \Delta'_{\text{total},n} \tag{5.16}$$

However, this equality will commonly not be observable in *real systems* (blue arrows). In fact each involved device will suffer from its own and the other device's imprecision:

*First*, the nodes apply independent clocks, drift apart, and thus will finally *not* defer their responses by exactly the same delay $\Delta_{\text{delay}}$. The blue arrows in Figure 5.5 indicate the difference from the view of an external observer. Though our nodes' CPUs are driven by quartzes from the same lot, the clock drifts vary depending on the selected node pair and the environmental influences described before. While the gray areas in Figure 5.5 illustrate the continuously growing maximum temporal extend of this phenomenon for each iteration, Figure 5.6 shows significantly different drifts $d_{A,B}(t)$ for three node pairs[13] measured over some time $t$.

*Second*, the responses must be scheduled and initiated by the responsible task on each node. Therefore, these tasks compute their intended local response time $t_r^*$ from each previously captured signal timestamp, and then sleep to release the CPU for other tasks. However, waking up sufficiently early to emit the signal in time is not that easy since some load-dependent and variable system overhead must always be taken into account.

*Third*, the base for each delay computation is never perfect since each captured timestamp $\tilde{t}_c$ exhibits an inherent error $E_{\tilde{t}_c} \in I_1$. While this cannot be avoided entirely as discussed before, its average error should at least be 0 µs in the average case according to Lemma II.1.

[13] The nodes with IDs 10, 11, and 72 were arbitrarily selected from our pool. The drift was measured via an oscilloscope tracking the delay between two periodically triggered I/O pins at both nodes. Note, that the drift of each pair corresponds perfectly to the other pairs' drifts: $918\frac{\mu s}{100s} + 1900\frac{\mu s}{100s} = 2818\frac{\mu s}{100s}$.

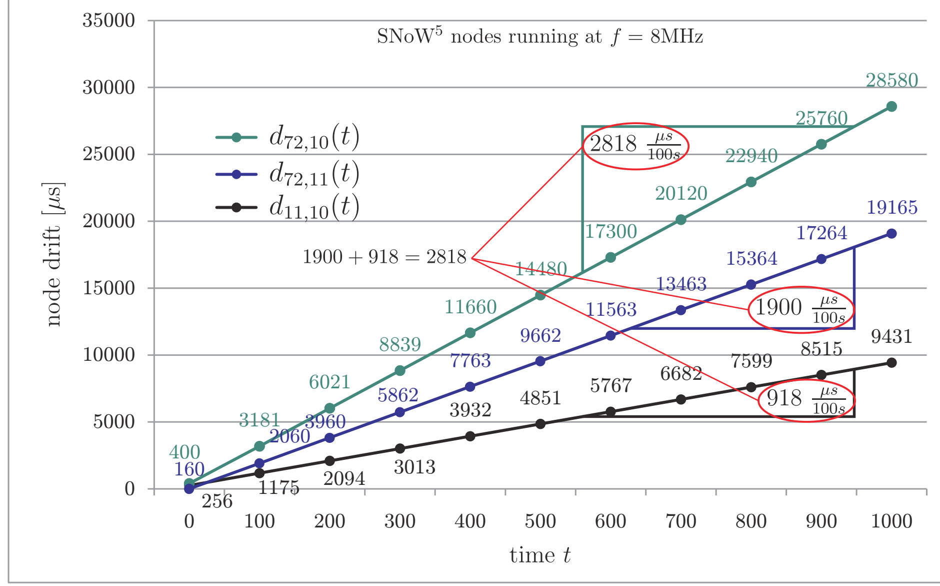



**Figure 5.6.:** Clock drift for three node pairs (environmental temperature: 24 °C)

## 5.4.1. Signal Emission and Self-Calibration

For the precisely timed signal emission, we propose a dynamic self-calibration scheme based on self-observation[14]. Therefore, the trigger signal will not only be captured by the other node where it is tagged with the timestamp $\tilde{t}_c$, but also by the emitting node itself. We denote the corresponding local timestamp as $\tilde{t}_r$. If the intended local response time for the current iteration has been computed as $t_r^*$, the lateness can be computed afterwards and used as compensation value $\Delta_{\text{comp}}$ to adjust the delay for the next iteration at emission time $t_r^*$:

$$\Delta_{\text{comp}} \quad := \quad \tilde{t}_r - t_r^* \tag{5.17}$$

$$t_r^* \quad := \quad \tilde{t}_c + \Delta_{\text{delay}} - \Delta_{\text{comp}} \tag{5.18}$$

Listing 5.3 shows the corresponding code sections[15]. In fact, the response time precision error ($E_{t_r^*} \in I_4$) depends not only on the two timestamps and their particular precision error ($E_{\tilde{t}_r}, E_{\tilde{t}_c} \in I_1$), but also on the error in the measured delay ($E_{\Delta_{\text{comp}}} \in I_2$) and the hard coded delay ($E_{\Delta_{\text{delay}}} \in I_3$) itself. Since we intentionally selected $\Delta_{\text{delay}} := m \cdot \lambda_C$ with $m \in \mathbb{N}^+$, at least this value is free from rounding errors and $I_3 := [0;0)$ for this special application.

[14] This technique will also be relevant for the precisely scheduled emission of slotted radio packets during the data aggregation stage of our indoor localization system SNoW Bat in Part III of this work. See Section 12.7 for details.

[15] Note that within the IRQ handler `ISRTrigger` (Line 10) calling `getIRQTime` will just take a copy of the *already captured* IRQ timestamp, and that the trigger task will not be resumed before `ISRTrigger` has returned. Also, calling `response` (Line 7) will indirectly invoke the self-trigger which will in turn update $\tilde{t}_r$ before it is used for the computation of $\Delta_{\text{comp}}$ (Line 8).

```
1 Time_t Δdelay=1000000, Δcomp=0;
2 while (1) {
3   waitEvent(&e); // wait for trigger
4   t*_r = t̃_c          // next emission
5      + Δdelay − Δcomp;
6   sleepUntil(t*_r); // absolute deadline
7   response();       // emit trigger
8   Δcomp = t̃_r − t*_r;  // self-calibration
9 }
```

**(a)** the inner loop of the trigger task

```
10 void ISRTrigger(int unused) {
11   getIRQTime(&t̃_c); // log timestamp
12   setEvent(&e); // resume trigger task
13 }
14 OS_DECLARE_IRQ_HANDLER( //ext. trigger
15      OS_IRQ_PIN10, ISRTrigger, 0);
16
17 OS_DECLARE_IRQ_HANDLER( //self-trigger
18      OS_IRQ_PIN11, getIRQTime, &t̃_r);
```

**(b)** IRQ Handlers (for demultiplexed I/O pins 10, 11)

**Listing 5.3:** The timing test bench: Trigger task and IRQ handlers

### 5.4.2. Pairwise Drift Calculation

For our tests we set up various node pairs $A, B$ as depicted in Figure 5.5, and we were interested in each nodes' $x \in \{A, B\}$ local timing error $e_x$ which was individually calculated by each node after $n$ iterations:

$$e_x \overset{(5.16)}{:=} \tilde{\Delta}_{\text{total},n} - \Delta'_{\text{total},n} = (\tilde{t}_n - \tilde{t}_0) - 2n \cdot \Delta_{\text{delay}} \tag{5.19}$$

Obviously, both timing errors $e_A, e_B$ have different sign unless the clocks are perfectly synchronous (then $e_A = e_B = 0\,\mu\text{s}$). Additionally, we define the symmetry error $e_{\text{symm}}$ as seen by an external observer as the average value over $e_A, e_B$. Since the average timestamp error $E_t \in I_1$ will accumulate over the two acquired trigger timestamps within each iteration,

$$e_{\text{symm}} := \frac{e_A + e_B}{2} = 2n \cdot E_t. \tag{5.20}$$

If we *indeed* achieved the timestamping error interval $I_1$ to be truly symmetric about 0, i.e. by selecting $\Delta_c = n \cdot \alpha + \frac{1}{2} \cdot \alpha$ properly in Eq. (5.11), we can consequently expect two observations for any pair of nodes $A, B$:

1. If both values $e_A$ and $e_B$ are made available to an external observer, their measured clock drift $d_{A,B}$, as depicted in Figure 5.6, can be verified through

$$d'_{A,B} := e_A - e_B \overset{!}{=} d_{A,B} \qquad \text{with} \qquad d_{A,B} = -d_{B,A}. \tag{5.21}$$

2. According to Eq. (5.20), $e_{\text{symm}} \overset{!}{=} 2n \cdot 0\mu s = 0\mu s$, and thus both values $e_A$ and $e_B$ will show the same absolute values. In direct consequence, each node can autonomously estimate its own drift towards the other node by simply calculating

$$\tilde{d}_{A,B} = 2 \cdot e_A \quad (\text{for node } A) \qquad \text{and} \qquad \tilde{d}_{B,A} = 2 \cdot e_B \quad (\text{for node } B). \tag{5.22}$$

   In particular, the exchange of any additional data, such as timestamps, between the nodes is not necessary to obtain this information (since $\Delta_{\text{delay}}$ is constant).

The reason becomes clear when considering the involved error intervals over $n$ iterations:

$$\begin{array}{cccccccc}
 & t_r^* & \overset{(5.18)}{:=} & \tilde{t}_c & + & \Delta_{\text{delay}} & - & \Delta_{\text{comp}} \\
\text{iter} & I_4 & & I_1 & & I_3 & & I_2 \\
1: & [-\lambda_C;\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [0;0) & & (-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) \\
 & \vdots & & \vdots & & \vdots & & \vdots \\
n: & [-n\lambda_C;n\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [0;0) & & (-(n-\frac{1}{2})\lambda_C;(n-\frac{1}{2})\lambda_C)
\end{array}$$

$$\begin{array}{cccccc}
 & \Delta_{\text{comp}} & \overset{(5.17)}{:=} & \tilde{t}_r & - & t_r^* \\
\text{iter} & I_2 & & I_1 & & I_1 \\
1: & (-\frac{3}{2}\lambda_C;\frac{3}{2}\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [-\lambda_C;\lambda_C) \\
 & \vdots & & \vdots & & \vdots \\
n: & (-(n+\frac{1}{2})\lambda_C;(n+\frac{1}{2})\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [-n\lambda_C;n\lambda_C)
\end{array}$$

Obviously, all error intervals remain symmetric about 0 µs throughout the entire test. In particular, the average error for each variable is 0 µs, and consequently $e_{\text{symm}} = 0$ µs, too.

In contrast, if we intentionally violate Eq. (5.11) and use e.g. $\Delta_c := n \cdot \alpha$ instead, the average timestamp error interval will be symmetric around $E_t = \frac{1}{2}\lambda_C$:

$$\begin{array}{cccccccc}
 & t_r^* & \overset{(5.18)}{:=} & \tilde{t}_c & + & \Delta_{\text{delay}} & - & \Delta_{\text{comp}} \\
\text{iter} & I_4 & & I_1 & & I_3 & & I_2 \\
1: & [-\frac{1}{2}\lambda_C;\frac{3}{2}\lambda_C) & & [0;\lambda_C) & & [0;0) & & (-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) \\
 & \vdots & & \vdots & & \vdots & & \vdots \\
n: & [-(n-\frac{1}{2})\lambda_C;(n+\frac{1}{2})\lambda_C) & & [0;\lambda_C) & & [0;0) & & (-(n-\frac{1}{2})\lambda_C;(n-\frac{1}{2})\lambda_C)
\end{array}$$

$$\begin{array}{cccccc}
 & \Delta_{\text{comp}} & \overset{(5.17)}{:=} & \tilde{t}_r & - & t_r^* \\
\text{iter} & I_2 & & I_1 & & I_1 \\
1: & (-\frac{3}{2}\lambda_C;\frac{3}{2}\lambda_C) & & [0;\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{3}{2}\lambda_C) \\
 & \vdots & & \vdots & & \vdots \\
n: & (-(n+\frac{1}{2})\lambda_C;(n+\frac{1}{2})\lambda_C) & & [0;\lambda_C) & & [-(n-\frac{1}{2})\lambda_C;(n+\frac{1}{2})\lambda_C)
\end{array}$$

Consequently, $e_{\text{symm}} \overset{(5.20)}{=} n \cdot \lambda_C$, and neither the autonomous drift computation through Eq. (5.22) nor the external drift verification through Eq. (5.21) will work any more.

### 5.4.3. Real-World Test Bed Analysis

Figure 5.7 shows the test bed results for the three already mentioned node pairs from Figure 5.6, and for various values of $\Delta_c$ after $n = 50$ iterations with $\Delta_{\text{delay}} = 1$ s ($\Delta'_{\text{total},n} = 100$ s). Note that the results repeat in a cyclic manner with period $\alpha = 8$, and thus the values for $\Delta_c = 10$ are similar to those for $\Delta_c = 18$.

When using $\Delta_c = 1 \cdot 8 + \frac{8}{2} = 12$, we did indeed achieve the expected symmetry error $e_{\text{symm}} \approx 0$ µs for all pairs. At least we received $|e_{\text{symm}}| < \lambda_C = 1$ µs, which is the timeline resolution and

| $\tilde{d}_{A,B}$ | local information[1,2] node 10 | node 11 | node 72 | observer[3] $d'_{A,B}$ |
|---|---|---|---|---|
| $\tilde{d}_{72,11}$ | 1900.0 | 1902.0 | 1900.0 | 1900 |
| $\tilde{d}_{72,10}$ | 2818.0 | 2818.0 | 2816.0 | 2818 |
| $\tilde{d}_{11,10}$ | 918.0 | 916.0 | 916.0 | 918 |

[1] black: measured according to Eq. (5.22)
[2] red: calculated according to Eq. (5.23)
[3] true drift as expected from Figure 5.6

**Table 5.2.:** Drift calculation for $\Delta_c = 12$

| $\tilde{d}_{A,B}$ | local information[1,2] node 10 | node 11 | node 72 | observer[3] $d'_{A,B}$ |
|---|---|---|---|---|
| $\tilde{d}_{72,11}$ | 1884.0 | 1836.0 | 2032.0 | 1900 |
| $\tilde{d}_{72,10}$ | 2724.0 | 2870.0 | 2922.0 | 2818 |
| $\tilde{d}_{11,10}$ | 840.0 | 1034.0 | 890.0 | 918 |

[1] black: measured according to Eq. (5.22)
[2] red: calculated according to Eq. (5.23)
[3] true drift as expected from Figure 5.6

**Table 5.3.:** Drift calculation for $\Delta_c = 16$

thus the best precision a node can reach. Furthermore, $d'_{A,B} \approx \tilde{d}_{A,B}$ verifies the measured values from Figure 5.6. Most important, as shown in Table 5.2, the autonomously measured drifts between two nodes are almost perfect. Indeed, the maximum error is ±2 µs. Another fact which we can verify from this table is, that since WLOG node $A$ knows its drifts $\tilde{d}_{A,B}$ and $\tilde{d}_{A,C}$ towards the two other nodes $B$ and $C$ respectively, it can also reliably derive the drift $\tilde{d}_{B,C}$ via

$$\tilde{d}_{B,C} := \tilde{d}_{A,C} - \tilde{d}_{A,B}. \tag{5.23}$$

For any other values of $\Delta_c$, the nodes can not gain reliable information about their relative drift on their own. When using $\Delta_c = 2 \cdot 8 = 16$ for example, Figure 5.7 shows values close to the expected symmetry error $e_{\text{symm}} = 2n \cdot \frac{1}{2}\lambda_c = 50$ µs. As a result, Table 5.3 summarizes the autonomously measured and computed drifts between the node pairs, and reveals quite large and asymmetric errors between to -132 µs and +94 µs.

Besides the precision of the autonomous drift estimation, another interesting metric is the resulting trigger frequency. The theoretical value

$$f_{\text{trig}} := \left(2 \cdot \Delta_{\text{delay}}\right)^{-1} \tag{5.24}$$

will not be visible in reality since neither node uses a perfect clock. However, we would at least like to achieve

$$f_{\text{trig, av.}} = \left(\Delta'_{\text{delay},A} + \Delta'_{\text{delay},B}\right)^{-1}, \tag{5.25}$$

which is definitely the best compromise two nodes $A, B$ can find if their true drift compared to the perfect global clock is unknown. Again, this is only possible if $e_{\text{symm}} = 0$. When looking at e.g. the graph for the nodes with IDs 11 and 72 in Figure 5.7, the extrapolation of $e_{\text{symm}}$ leads to a symmetry error of −345.6 µs for $\Delta_c = 12$ and 42336 µs for $\Delta_c = 16$ within one complete day. Thus, the larger $|e_{\text{symm}}|$ the larger the deviation from $f_{\text{trig, av.}}$. The effects are once more visible in Tables 5.2 and 5.3: For $\Delta_c = 12$ the values in each row are almost equal, while they exhibit significant variations for $\Delta_c = 16$.

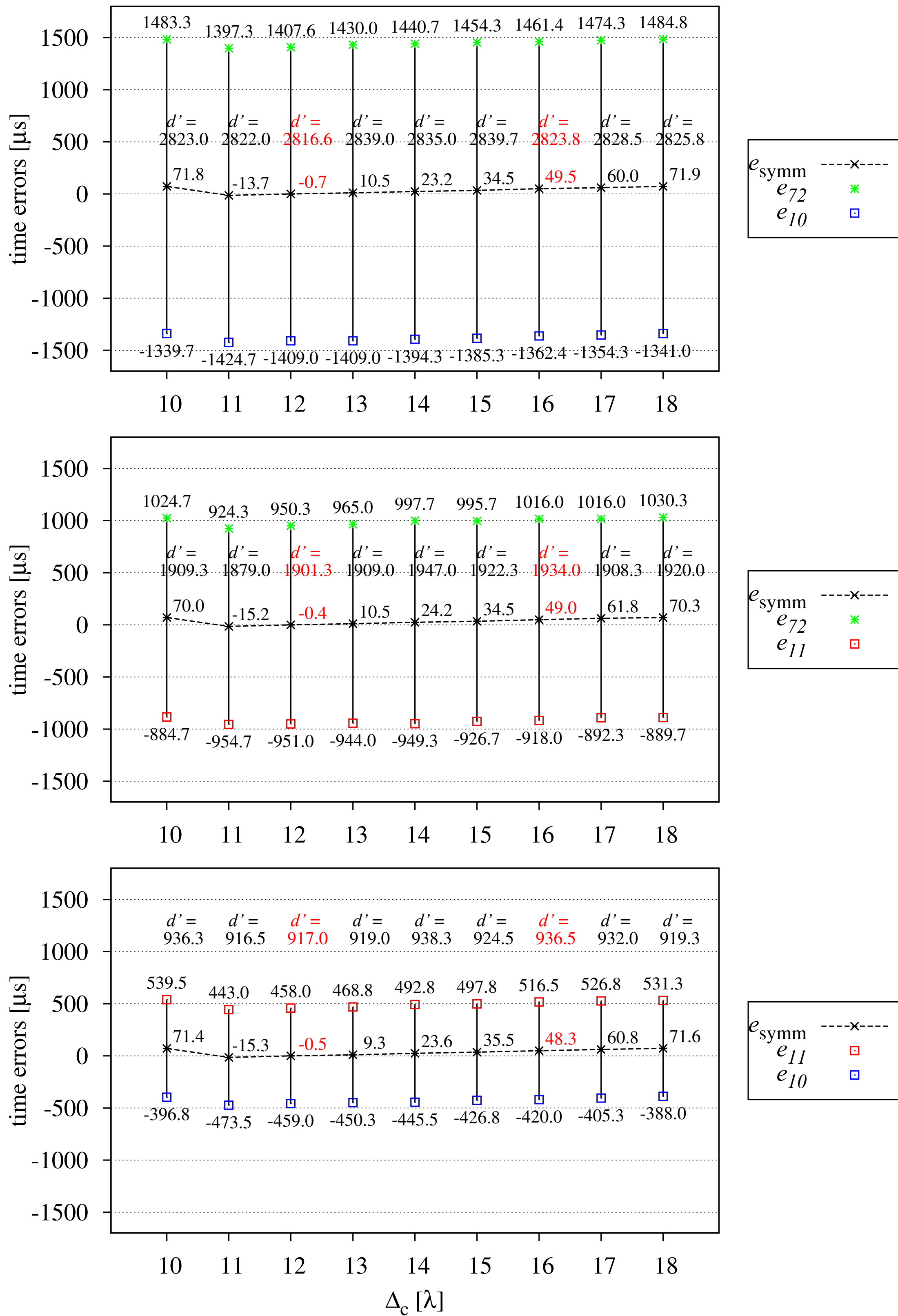


**Figure 5.7.:** The node timing error after 100 s as measured by each node (see Figure 5.6 for the expected $d'$ values)

### 5.4.4. Conclusion and Outlook

In this chapter we proposed a technique for obtaining precise timestamps $\tilde{t}_e$ for external events $e$, and for the precisely timed execution of reactions $r$ at scheduled times $\tilde{t}_r$. The error intervals for both $\tilde{t}_e$ and $\tilde{t}_r$ are symmetric about 0. While the first is achieved through the centralized and specially prepared preprocessing of interrupts by the kernel, the latter becomes possible through a simple self-calibration scheme at application layer. Throughout the remainder of this work, both approaches will show to be a great benefit for an inherent problem within distributed but interacting and time-critical (embedded) systems: In fact, as long as time is not properly manageable locally by the individual nodes, network-wide synchronization and event or state attribution will hardly achieve the potentially feasible precision.

A corresponding test bed verified, that it is possible to determine the drift between two nodes without the explicit exchange of any quantitative information (like e.g. timestamps or previously measured delays). Instead, it is sufficient to periodically pass events (i.e. interrupts) between the nodes. Since similar periodic behavior can also be found in several (wireless) communication protocols [142, 217, 269, 311], the proposed techniques can also be applied to support time synchronization among the involved systems. Regarding our demand for reliable environmental interaction from Section 1.1, the local time-awareness of any participating node improved significantly.

# 6. Real-Time Resource Management in Networked Sensor/Actuator Systems

**Abstract**

The increasing complexity of today's sensor/actuator network (SANet) applications does not only demand for a careful selection of the underlying embedded hardware; it also imposes considerable challenges on the subsequent software design. Due to various reasons (→ Table 2.1[p21]) the nodes are commonly very constrained in their computational power and available resources. Specific problems originate from these limitations and affect their modularity and real-time capabilities as demanded in Chapter 1. While preemptive operating systems like *Smart*OS are one approach to retain acceptable reactivity – within highly dynamic environments in particular –, their concurrency paradigm commonly leads to severe resource sharing problems. These are caused by the coexistence of tasks with interfering and even varying requirements.

To counteract these problems, we developed the DynamicHinting approach as entirely novel design and programming paradigm for managing the reactivity and real-time operation within compositional task systems. The key lies in the support for reflective and collaborative task behavior dynamically at runtime, and permits the efficient combination of preemptive task or CPU scheduling and the non-preemptive access to other temporarily shared resources. In summary, we facilitate compositional software design by providing independently developed but concurrently executing tasks with runtime information about their mutual influence on each other. As an extension to the purely cooperative resource access as described in Section 3.2, the resulting self-awareness allows tasks to relate their own requirements to superior objectives for yet collaborative and reflective resource sharing – e.g. supported by so called time-utility-functions. With respect to task priorities and the limited performance of sensor nodes (and comparable embedded devices), our technique significantly improves classical methods for handling task starvation, priority inversion, and deadlock conditions (where required), under both short- and long-term resource allocations. In many cases this even allows to reduce task blocking and resource allocation delays as otherwise imposed by bounded priority inversion.

## 6.1. Introduction

DynamicHinting

The ever increasing size, pervasiveness and service variety of today's sensor/actuator networks (SANet) significantly boosts the demands on and the complexity of the underlying nodes. To still allow their convenient and unobtrusive deployment in large numbers, while at the same time keeping costs, power consumption, and maintenance efforts low, these embedded systems are – and will probably remain – rather small in size, computationally weak, and severely resource constrained[1]. Thus, modular hardware and software concepts have become popular to manage their design, implementation and operation. For example service-oriented programming abstractions [152, 166] introduce several sub-layers within the application layer (→ Figure 3.1[p34]), or propose the almost independent development of subsystems by simply defining their interfaces. However, hiding too much complexity from the developer can rapidly become critical for autonomously operating systems: Adequate interaction between the various modules is essential to avoid typical compositional problems, but is hard to achieve and maintain automatically. Besides task scheduling [49], directly related issues comprise dynamic resource sharing or even real-time operation [172]. Concerning this, we find that current research in the field of severely resource constrained embedded systems is still too focused and limited to static design concepts. As already stated in [60] and discussed in Section 1.2, next-generation embedded systems will be more frequently used as reactive real-time platforms in highly dynamic environments. Here, the true system load varies considerably, and can hardly be predicted a priori during development. In fact, we definitely expect a clear focus shift from almost pure sensing in classic sensor networks (SNs) toward additional and intense proactivity in SANet applications (→ Figure 1.6[p13]). Integrated control systems for facility management, medical application, and military purposes are just few examples, but already comprise complex functionality like precise and DSP based on-demand measurements, time synchronization, real-time event recording and processing, dynamic routing, and reliable emergency handling throughout critical situations. Then, preemptive and prioritized tasks are required for fast response on various (sporadic or periodic) events, but further complicate resource assignment and reactivity. This is especially true for open systems where real-time and non real-time tasks must coexist in order to reduce hardware overhead, energy issues and deployment effort.

---

**Definition II.3: Real-Time-Critical/Capability/Scheduling**

We denote a task $T$ as *real-time-critical*, if it must complete at least one real-time-critical action or response within a well-defined temporal boundary $[\tau_{\min}, \tau_{\max}]$ with $\tau_{\min}, \tau_{\max} \in \mathbb{R}_0^+$ and $0 \le \tau_{\min} \le \tau_{\max}$, i.e. it must produce algorithmically correct results and system states within this interval. A task $T$ is not real-time-critical if $\tau_{\min} = 0$ and $\tau_{\max} = \infty$ for all actions. Throughout this work we limit ourselves to bounding the maximal execution or response time $\tau_{\max}$, and denote a task system as *real-time capable*, if it can provably complete all actions within this so called absolute *deadline* $\tau_{\max}$.

---

In this chapter we present the novel *DynamicHinting* approach for improving cooperative

[1] While related aspects were already discussed in Section 1.2.1, an exemplary sensor node prototype is presented in Section 2.2.

methods for the dynamic management of exclusive but temporarily shared resources among prioritized and preemptive tasks with real-time requirements. Thereby we support both periodic and aperiodic tasks (e.g. sporadically triggered in event-driven designs). In Section 1.2.1 we already demanded for improved quality awareness of networked embedded systems to conduct runtime self-evaluation, and to finally adopt to various self-x strategies. Projected to the application layer, our novel paradigm improves compositional software design by inducing resource-related self-awareness for independently developed, but concurrently running tasks concerning their mutual influence on each other. As often suggested (e.g. in [15]), we take advantage of the resource manager's enormous runtime knowledge about the system's current requirements. This information is carefully filtered and forwarded via so called *hints* to exactly those tasks, which currently block the execution of more relevant tasks due to a lasting resource allocation.

---

**Definition II.4: Blocking, Blocker, Critical Resource, and Blocking Delay**
We denote a task $h$ as *blocked*, if it waits for the allocation (i.e. it is suspended in a pending resource request) of a resource $s$ which can currently not be assigned due to the lasting allocation of a (not necessarily different) resource $r$ by another task $l \neq h$ with lower base priority, i.e. $P_l < P_h$. In this case, $l$ is called *blocker*, and $r$ is called *critical resource*. The time for which a task is blocked until its requested resource is assigned, is called *blocking delay*.

---

In turn, our hints allow blocking and even deadlocked tasks to adapt to the current resource demands and finally to contribute to the system's overall reactivity and stability in a collaborative manner. Furthermore, accounting for the task priorities as defined by the developer is simplified. In many cases, even delays which would otherwise occur due to bounded priority inversion can be reduced. The decision between following or ignoring a hint is made by each task autonomously and dynamically at runtime, e.g. by the use of appropriate time-utility-functions (TUF) [176]. In our opinion, the central weakness of all resource management approaches we found so far is, that tasks are not aware of their (varying) influence on the remaining system, and thus cannot collaborate adequately. In this respect, DynamicHinting follows classic reflexion concepts [15, 270], and introduces a new policy into operating system kernels, by which programs can become 'self-aware' and may change their behavior according to their own current requirements as well as to the system demands. Thus, DynamicHinting is not limited to embedded systems and the SANet domain, but can be applied to real-time operation in general.

In this work, we primarily present DynamicHinting as extension for the priority inheritance protocol (PIP) [262]. This way, we intentionally focus on long-term resource allocations (as frequently required for e.g. complex objects or hardware devices [51]), and avoid some related shortcomings of similar techniques. Yet, it may also be combined with most other policies and systems where task blocking can occur. Thus, we also compare results from using our concept with the priority ceiling protocol (PCP) [262]. Though both original protocols face several problems and failed almost completely within some of our test beds, DynamicHinting always achieved considerable improvements: It allowed a significantly higher resource load *and* increased task progress/utility. Just limited by the CPU performance, our approach improved

both resource allocation delays and task reactivity to be very close to the achievable best case values. In fact, by the nature of both original protocols, extending PIP commonly resulted in a much better performance gain than extending PCP. A comparison can be found in Section 6.7.

This chapter is organized as follows: Initially, we will further motivate the need for sophisticated resource management and real-time operation in reactive embedded systems, and sensor/actuator platforms in particular. Followed by an outline on some existing techniques, we'll motivate the selection of PIP as preferred basis for our novel approach. A detailed description of DynamicHinting and its integration into *Smart*OS will form the central part of this chapter. In this context, we'll also discuss the impact on the programming model, and evaluate the strengths and benefits through some synthetic stress tests and concrete application scenarios from real-world systems. The results will finally show that – despite of the problem's complexity – DynamicHinting is efficiently applicable even for low performance devices like sensor nodes.

## 6.2. Motivation and Requirements

An operating system has significant influence on the overall system performance since it coordinates task interactions (ITC) as well as the access to shared operational resources like hardware components or data structures. For some of them, exclusive access must be granted at least temporarily to avoid race conditions, resulting malfunctions or even system breakdown. Unfortunately, resource assignment in complex, modular systems with concurrently running tasks is hard to manage during development and runtime. This is particularly true, if tasks are allowed to use virtually any available resource in any order, and if they may even require exclusive access to several of them at the same time. As long as allocation times remain short-termed[2], or if the system's overall runtime requirements are roughly predictable, efficient methods already exist (→ Section 6.3.1). However, if long-term allocations collide with sporadic but time-critical on-demand requests in dynamic environments, smart adaptive techniques are needed to still provide good reactivity [60].

Whereas the main resource of each computing system, the CPU, is often managed by the task scheduler in a *preemptive* way, we believe that the operating system should also coordinate the access to other, *exclusive* resources (where automatic preemption is often not that easy) by contributing appropriate runtime mechanisms. Though some techniques were already implemented for resource constrained node platforms, most of them do not address the specific aspects of reactive real-time operation.

### 6.2.1. Terminology and the Problem Model

As already discussed in Section 3.2 we denote a system resource as *preemptive* if the resource manager is authorized at any time to temporarily withdraw the resource from a task, *and* if it is also capable of returning it in its previous state. In contrast, a *non-preemptive* resource must always be released voluntarily by its current owner task. Apart from the preemptive CPU, we

[2]According to Section 3.2[p35] task self-suspensions are forbidden during short-term allocations.

consider all other resources as non-preemptive. Additionally, we distinguish between *short-term* and *long-term* resource allocations: While the first term indicates that a process does not suspend itself while holding the resource (e.g. lock a data structure, process it, and release it eventually), the latter permits self-suspensions (e.g. lock a hardware component and release the CPU until an IRQ occurs).

Regarding these definitions, our optimization target for DynamicHinting is twofold: First, we aim on on-demand resource deallocations to reduce the duration of (bounded and unbounded) priority inversions, and to help software designers in avoiding prophylactic (e.g. regular/frequent) deallocations just to possibly serve other tasks. Second, we intend to improve the interleaved execution of preemptive tasks through the provision of a generous but priority aware resource assignment policy, which relieves the resource manager from the need to maintain a so called *safe state* on the resource pool. By combining both goals we achieve that persisting (long and short-term) resource allocations do not circumvent the assignment of further (free) resources, and finally observe a significantly improved task progression and utility compared to PIP and the conservative PCP in particular.

Compared to the classical problem model, which is commonly used in PIP/PCP related literature and where a task locks/allocates a resource and won't unlock/release/deallocate it until its work with this resource has been entirely finished, our modified problem model relaxes this constraint: We intentionally introduce the option for an early but strictly task-controlled release of non-preemptive resources if these cause the blocking of a task which is higher prioritized than the one which currently keeps the resource locked. In fact, the resulting the on-demand resource handover is the key to our concept's success and outperforms PIP, PCP, and similar resource sharing protocols significantly.

### 6.2.2. Requirements

During research and practical work, we found that reactivity and proactivity in modern embedded and SANet applications requires quite sophisticated real-time and resource management concepts. We'll give just a few examples from real-world application scenarios:

**Sporadic resource sharing under long-term allocations.** As exemplified in Figure 6.1, a radio protocol task commonly requires the long-term allocation of the used transceiver in combination with relatively short but sporadic accesses to the interconnection bus[3]. Obviously, both resources need specific configuration sets, and thus are non-preemptive. Although the radio task might suspend itself while holding the transceiver and waiting for certain interrupt-triggered events, using the bus on-demand becomes time-critical when radio transmission slots must be obeyed or when a receive buffer must be read and cleared quickly to allow the reception of successive radio packets without data loss. Concurrent to this communication task, other tasks might use exactly the same interconnection bus for data exchange or even continuous streaming (e.g. from an ADC to some external memory). Again, their resources are non-preemptive, but this time the bus is also locked in a long-term allocation. Even though this conflict involves just one single shared resource, the resulting compositional problem is

[3] In fact, the inter-arrival time of incoming packets and transmission requests is unknown in many cases.

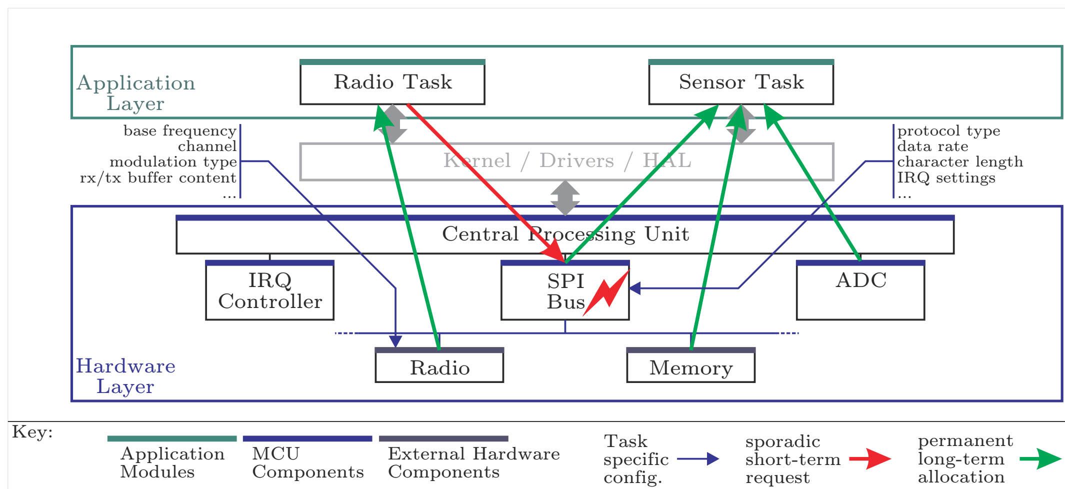



**Figure 6.1.:** Colliding resource requirement between two interfering tasks

already hard to solve. Even if task priorities can be selected carefully to indicate the desired relevance of each task, their compliance can not necessarily be guaranteed. Instead, knowledge about the overall system load and module interactions (including further tasks) must often be incorporated manually into the code – if this is possible at all. The regular and voluntary release of long-term resources could be one solution. However, while such prophylactic releases might simply be unnecessary in many cases, they might also impose considerable overhead when deallocation and reallocation are expensive in time (→ Table 4.2[p61] for *Smart*OS overhead) and energy, which we already indicated as a valuable resource in Section 2.2.2. Where data streams often require explicit termination (*trailers*) and initiation (*headers*), physical resources might require a time-consuming (de-)initialization procedure upon each (de-)allocation[4]. The additional use of so called *monitors* [276] for the managed operation of such resources is also no universal option, especially if either the performance for this abstraction is simply not available, or if their operation cannot be adjusted sufficiently well to the application demands[5].

**Dynamic base priorities.** Regarding the demand for quality awareness and the semantic use of data from Figure 1.7[p14], we found the support for adjustable task base priorities convenient and useful – though this complicates resource management even further: Besides a sensitive adaptation of tasks to changing environmental conditions, this would also allow server tasks to adapt to the priority of their (most relevant) clients at runtime. The *Smart*Net MAC protocol from Section 8.1 gives a concrete example for such an implementation under *Smart*OS. Another specific example is the sharing of hardware among virtual networks on routers or nodes: The idea is to largely isolate corresponding virtual subsystems on the same device for maintainability, security and safety reasons [223]. These, however have changing QoS demands, and in consequence need flexible access to still shared I/O ports. While variable base priorities can signal their relative importance, the prompt adaptation remains problematic: Techniques for coordinating the changing requirements of separate but interfering subsystems dynamically

[4]Under *Smart*OS we support this by providing resource-specific handler functions as described in Section 4.3.7
[5]In fact, the monitors themselves (and their internal data structures) are also comparable to shared resources then.

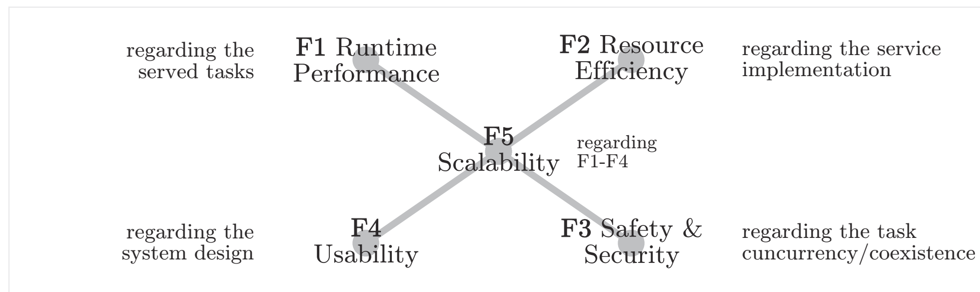



**Figure 6.2.:** Main challenges for many system service design problems

and on-demand have to deal with severe problems like priority inversions and deadlocks, and are still rarely found. With the increasing availability of multi-core architectures in embedded and SANet areas [51, 222] this must be considered even more carefully to efficiently utilize their advanced capabilities.

**Summary.** Fast reactions on internal and external events (e.g. inter-task triggers or interrupts) are frequently required within reactive sensor/actuator systems, but often suffer from severe delays and critical system states due to resources which are currently blocked in short-term and long-term allocations. Then, their fast handover to the reacting task is essential, and should at least be accelerated by the resource manager without damaging the atomicity of depending operations. According to Figure 6.2, which depicts the five main challenges for many system design problems[6], we request the following features to be supported by our novel approach:

F1 Runtime Performance

- Adjust allocation delays with respect to the task priorities.
- Support (hard) real-time demands at least for high priority tasks.
- Avoid the starvation of low priority tasks.

F2 Resource Efficiency

- Reduce the management overhead in terms of CPU load, memory requirements, and energy consumption.

F3 Safety and Security

- Handle deadlocks (e.g. through avoidance, prevention, detection & resolution).
- Coordinate and protect long-term and short-term resource allocations properly.

F4 Usability

- Do not impose unreasonable restrictions on the application design.
- Keep the impact on the programming model low.

F5 Scalability

- Maintain F1–F4 independent from the number of tasks and managed resources.

[6] In fact, we'll revert to these challenges in later chapters on e.g. memory management, localization, and wireless data aggregation.

Of course, it is almost impossible to support all features at the same time. While resources are not protected by *Smart*OS in general (→ Section 4.3.7[p57]), certain efforts must be undertaken at application layer to benefit from the already mentioned reflexion and collaboration concept. Having motivated our requirements for real-time resource management in embedded and SANet scenarios in particular, we'll now overview some related work before presenting our novel approach.

## 6.3. Related Work

In this section we focus on resource management, and tie in with both the kernel classification framework from Section 3.3 and the OS kernel comparison from Table 3.1[p46].

**Non-preemptive systems** with *run-to-completion* policy are very common in sensor/actuator systems[7]. These prevent some resource conflicts implicitly since task or process executions cannot be interleaved, and each allocated resource will be released implicitly upon task termination. If tasks need to hold exclusive access to certain resources over several runs, a frequent approach is to implement special *server tasks* (which implicitly own a particular resource for exclusive access) or stateful *function libraries* for managing these resources. Such an abstraction layer allows to share or even virtualize resources comparable to the TinyOS component concept [291]. In such systems, however, the so called *split-phase* software design and large resource hierarchies might result in severe inter-communication overhead and reduced overall performance, then. Additionally, run-to-completion tasks often provide bad reactivity on sporadic events since they can not be suspended arbitrarily for more important actions. Indeed, interrupt handlers might react quickly, but using resources therein is seldom wise since these might currently be unavailable, and the mere attempt could block the whole system forever. Many event-driven systems (like e.g. TinyOS) solve this problem by simply triggering (posting) an appropriate handler task during the interrupt service routine. However, its true execution delay is unknown or at least non-deterministic, and again depends on the currently running task and the scheduling policy. Note, that for reactive handler tasks the non-preemptive use of the CPU can already lead to some kind of priority inversion, then (→ Definition II.5). Therefore, TinyOS and Contiki [85], which natively also runs non-preemptive processes, both support so called *TOSThreads* [158], and *protothreads* [87] respectively. These are preemptive but lack priorities and native resource management entirely.

**Preemptive systems** potentially provide much better reactivity. Here, a task can be preempted at any time for a more important action implemented in another task. Therefore individual priorities commonly define each task's relevance. Yet, preemption yields no instant advantage if an important action requires a shared resource which is exclusively held by a less important task. Resulting problems like *priority inversion* [172, 321] might lead to the blocking of high priority tasks, and even deadlocks may occur. To cope with these issues, well studied approaches like the *priority ceiling protocol* (PCP), the *highest locker protocol* (HLP), the *priority*

[7] Due to their low CPU and memory overhead for task context switching and stack space.

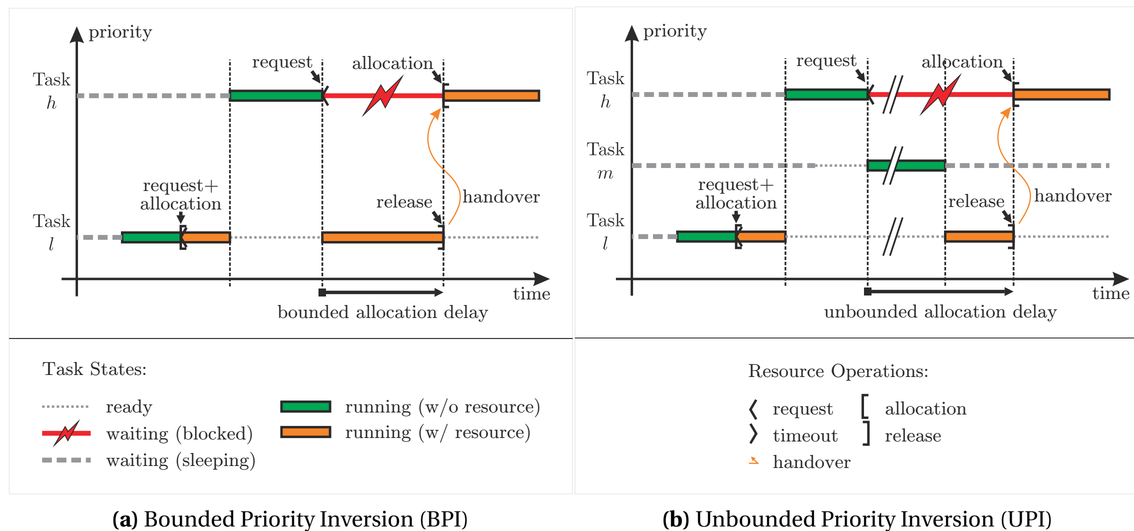
*inheritance protocol* (PIP) [63, 262], and the *stack resource policy* (SRP) [19] can be found in literature. Besides inflicting some runtime overhead, these techniques also suffer from certain weaknesses addressed in the next section. Additionally, these are hardly implemented in embedded operating systems with small memory footprint and low performance requirements. Apart from the *priority ceiling protocol emulation* (PCPE)[8] in Nano-RK [95], other preemptive sensor network operating systems like e.g. MantisOS [46], RETOS [62], or SenSmart [69] do not consider real-time and resource-related problems at all. At most they recommend to encapsulate the usage of resources in atomic sections, which executes tasks non-preemptively, and thus is similar to the also widely known *non preemption protocol* (NPP).

### 6.3.1. Resource Management in Preemptive Systems

Many scheduling algorithms, like e.g. *earliest deadline first* (EDF) and *rate monotonic scheduling* (RMS) [180], induce a strictly periodic execution model[9], and also assume that the processes they manage are entirely independent from each other: Only the CPU is implicitly considered as a shared resource which must be synchronized properly to meet all deadlines. Since further inter-task dependencies are neglected, the underlying scheduling problem can be solved and decided through schedulability analysis [180]. However it is often forgotten (or simply ignored), that various explicit and implicit dependencies can exist between the tasks. In fact, these can be always traced back to resource conflicts, and make scheduling hard: While *explicit dependencies* emerge from the purposive interaction between tasks through intentionally shared resources (e.g. the shared memory within the producer/consumer example from Listing 4.4[p57]), *implicit dependencies* emerge from the unconscious or even unnoticed interaction between tasks through unintentionally shared resources (e.g. peripheral components like the data bus and the radio unit from Listing 4.5[p58] and Section 6.2.2). A more complex example will follow in Section 6.7.

**Deadlocks.** In the context of resource dependencies deadlocks are a frequent problem, e.g. if two tasks hold at least one resource each while requesting another one which is currently held by the other task, respectively. As a simple solution, the frequently recommended *banker's algorithm* [79] grants a resource only if all resource demands of at least one task can always still be satisfied, then. More restrictive versions start a new task only if its worst case resource requirements can still be satisfied when all other running tasks also claim their demands entirely. Maintaining so called *safe states* avoids deadlocks, but implies two major problems: First, a (time-critical) task might not start promptly when required (e.g. an event handler task). Second, two tasks which *might* require the same unique resource may never run interleaved. In many statically linked task systems (like e.g. *Smart*OS) they could not even coexist. Another strategy is to require the allocation order for resources to be globally fixed *and* inverse to their release order (LIFO). In many scenarios this is neither possible nor desired, and would violate our demand for usability (F4[p93]). The constraint might even cause resources to remain allocated longer than

[8] PCPE is comparable to the highest locker protocol HLP.
[9] At least for real-time-critical tasks (→ Figure 3.3[p39]). EDF can also be used for non-periodic tasks.




**(a)** Bounded Priority Inversion (BPI) **(b)** Unbounded Priority Inversion (UPI)

**Figure 6.3.:** Types of priority inversions

really required by the task logic. So, either deadlock detection and recovery is required, or other techniques must be found.

**Priority inversion.** Mostly in the special context of implicit dependencies, *priority inversion* is another inherent problem of preemptive, prioritized task systems. Here, the competition for a single exclusive resource may already lead to the (temporary) blocking of a high priority task:

---

**Definition II.5: Bounded / Unbounded Priority Inversion (BPI / UPI)**
If a high priority task $h$ requests an exclusive resource which is currently allocated by a lower prioritized task $l$ (→ Figure 6.3a), $h$ is blocked, and this direct dependency is known as *bounded priority inversion*. If an additional task $m$ with medium priority prevents $l$ from running and thus from releasing the resource, this indirect dependency is known as *unbounded priority inversion* (→ Figure 6.3b), and might even result in the final suspension of $h$.

---

Since in both cases the task priorities, as defined by the developer, are not obeyed as desired, this might lead to unexpected behavior, reduced reactivity and real-time capability of the overall system. In particular this might not only affect directly involved tasks, but it can also induce far-reaching consequences beyond the time of occurrence. The described problem became famous in 1997 when the Mars Pathfinder mission had almost failed because of an inappropriately shared data bus [147]. The engineers' initial view on the synchronization problem was devastating:

> *"The data bus task executes very frequently and is time-critical – we shouldn't spend the extra time in it to perform priority inheritance!"* [10]

[10] Fortunately, the VxWorks [307] based software was configured in such a way that (via remote-management) the mutex for coordinating the access to the data bus could be reconfigured to use the priority inheritance protocol, which finally resolved the unbounded priority inversion.

In fact, especially for situations where time-critical behavior is expected or even mission critical, additional effort can commonly not be avoided without jeopardizing a proper operation. A truly critical property of priority inversion is that its occurrence depends on many variable runtime factors, and thus it is hard to reproduce. In addition, the effects depend on the situation, and vary widely from barely perceptible to catastrophic. Unfortunately, small or rarely observed abnormalities are frequently ignored in software development and classified as non-critical, if they no longer occur in subsequent (non-exhaustive) tests.

**Available brute force solutions.** So, what can be done in case of deadlocks or priority inversions? A common approach is to terminate a spurious low priority task or withdraw its resources for the benefit of a more important task. However, this might also result in highly critical system states: While undoing the work so far, it might even leave the resources in an undefined state, making a handover or re-initialization difficult. Just imagine the disastrous withdrawal of a currently active DMA or radio controller! Counteracting with checkpointing and roll-backs of whole tasks and involved resource states is hard or even impossible on tiny embedded systems. Even if rarely used, this would produce enormous system load and memory overhead issues, and seriously conflict with our feature requests F1 and F2[p93]. Not to mention the increased power consumption which is of special importance for energy constrained systems like sensor nodes.

### 6.3.2. Resource Synchronization Protocols with Priority Inheritance

Another option are resource sharing protocols which assign dynamic task priorities in order to handle deadlocks and deal with priority inversions. Since we present DynamicHinting as an extension for cooperatively managing this kind of exclusively accessed and temporally shared resources, we'll introduce some definitions and review some basic protocols first. In particular, we'll discuss their central policies with respect to their qualification for the demands from Section 6.2.2 as well as for being combined with and improved through DynamicHinting. A summary of these protocols – which basically rely on priority inheritance mechanisms – can be found in Table 6.1.

---

**Definition II.6: Priority Inheritance**

Depending on the concrete protocol, the general idea of *priority inheritance* is to temporarily, and either implicitly or explicitly, raise the active priority[11] $p(t)$ of a task $t \in T$ above its initial base priority $P_t$ to either

- the maximum active priority of all tasks it currently blocks (*true priority inheritance*),
- or a priority which would preventively avoid further blocking of a resource holder in the future.

---

Though (unbounded) priority inversions can be encountered through this idea, we also pay this improvement by some consequent problems:

[11] According to the *Smart*OS terminology, active priority means the priority according to which the scheduler and the resource manager execute a task and assign resources. Base priorities are initially left untouched.

**Definition II.7: Inheritance Related Inversion / Starvation (IRI / IRS)**
If any priority inheritance strategy raised a task $l$'s active priority $p(l) > P_l$, and consequently another task $m$ with $P_l < p(m) < p(l)$ cannot preempt any more, $m$ undergoes *inheritance related inversion*. If tasks with a priority below the priority of any resource owner will not be scheduled in general, these undergo *inheritance related starvation*.

**Definition II.8: Avoidance Related Rejection / Inversion (ARR / ARI)**
If a resource request for a currently unallocated resource is rejected in order to avoid potential problems (e.g. deadlocks) in the future, this is called *avoidance related rejection*. If the rejected task $h$ has higher base priority than the task $l$ which causes the rejection due to a resource allocation, i.e. $P_h > P_l$, this is called *avoidance related inversion* [73].

While IRI is obviously required to counteract the problem of unbounded priority inversion, IRS, ARR, and ARI impose severe problems for long-term resource allocations since these jeopardize our feature request F1[p93].

**Protocols.** A truly simple approach to avoid unbounded priority inversion is the *non preemption protocol* (NPP) which executes tasks non-preemptively as long as these keep resources allocated. While NPP can in general be emulated by encapsulating resource usage in atomic sections, it implicitly raises the owner's priority to the system-wide maximum, but also causes IRS. Deadlocks are reliably avoided since tasks may commonly not even suspend themselves while holding a resource. A relaxed version of the NPP allows the voluntary release of the CPU by means of yield-directives. However, this variation is only deadlock free if not more than one task at a time is allowed to allocate resources. Any other task's request will implicitly yield then, and resumes the current resource owner to at least ensure its own progression. This behavior turns IRS into ARR. The advantage of both versions is both the simple implementation and the convenient option to share one stack among the tasks[12]. The most significant disadvantage is their bad reactivity on sporadic events, and that they are obviously not suitable for long-term allocations because of inheritance related starvation and avoidance related rejection.

In 1990 Sha et al. [262] and Baker [19] presented various improved approaches for avoiding unbounded priority inversion in the context of exclusively managed resources: The already mentioned PIP, PCP and HLP techniques [262] face this problem by adjusting task priorities dynamically at runtime according to the current resource assignment situation. SRP [19] is an extension to PCP, but also supports multi-instance resources[13], and assigns so called *preemption levels* with regard to various selectable metrics (e.g. deadlines and periods under EDF and RM scheduling). Since these protocols are commonly still applied in their original form, we won't go into detail about their general policies. Instead we'll only mention some critical problems in the

[12]We have already discussed cooperative CPU scheduling in the context of the Contiki OS in Section 3.3.2 where yield-directives perform stack rewinding to share the stack, but invalidate all local variables.
[13]For this work we limit our consideration to single-instance resources.

| | NPP[1] | NPP[2] | PIP | PCP/OCPP | HLP/ICPP | SRP |
|---|---|---|---|---|---|---|
| reference(s) | | | [262] | [262] | [63] | [19, 296] |
| avoids bounded priority inversion | – | –[4] | –[4] | –[4] | – | – |
| avoids unbounded priority inversion | ✓ | – | ✓ | ✓ | ✓ | ✓ |
| avoids chain blocking | .[5] | ✓ | – | ✓ | ✓ | ✓ |
| deadlock avoidance | ✓ | ✓[6] | – | ✓ | ✓ | ✓ |
| avoids inheritance related inversion | .[5] | – | – | – | – | – |
| avoids inheritance related starvation | – | ✓ | ✓ | ✓ | – | –[10] |
| true priority inheritance[9] | – | – | ✓ | ✓ | – | ✓ |
| avoids avoidance related rejection | .[5] | – | ✓ | – | – | – |
| suitable for long-term allocations[11] | .[5] | – | ✓ | – | – | – |
| allows task self-suspension[8] | – | ✓ | ✓ | ✓ | – | –[10] |
| interleaved exec. of equal prio. tasks[8] | .[5] | .[5] | ✓ | ✓ | – | –[10] |
| support for dynamic base priorities | ✓ | ✓ | ✓ | –[7] | –[7] | ✓ |
| blocks on | preempt. | request | request | request | preempt. | preempt. |
| required a priori information[3] | none | none | none | alloc. graph | alloc. graph | alloc. graph |
| computation of allocation delay | simple | simple | hard[12] | simple | simple | medium[17] |
| expected implementation complexity | simple | simple | medium[12] | high[13] | medium | simple[18] |
| extensible via DynamicHinting | .[5] | ✓ | ✓ | ✓ | ✓[14] | ✓[14] |
| available in *Smart*OS | – | –[15] | ✓ | ✓ | – | – |
| available in e.g.[16] | | | POSIX<br>RTSJ | | POSIX(PPP)<br>RTSJ(PCE)<br>ADA95(CLP)<br>OSEK<br>Nano-RK(PCPE) | μC/OS-II<br><br>ADA05(PLP) |

[1] without voluntary yield (i.e. explicit self-preemption) during a lasting resource allocation
[2] with voluntary yield (i.e. explicit self-preemption) during a lasting resource allocation
[3] e.g. the computation of resource-specific ceiling priorities: $c(r) := \max\{P_t \mid t \in T \text{ might allocate } r\}$
[4] can be improved via DynamicHinting
[5] not relevant / cannot occur since only one task may hold resources at a time
[6] only if a denied request yields
[7] would require the dynamic adaptation of ceiling priorities (which is hard while resources are still allocated)
[8] during lasting resource allocations
[9] i.e. a task is not raised higher than to the priority of the highest prioritized task it blocks
[10] would corrupt the stack sharing feature of SRP
[11] according to Section 3.2 long-term allocations last over at least one task self-suspension
[12] because of potential chain blocking
[13] because of avoidance related inversion
[14] untested
[15] can be emulated
[16] system specific name in brackets
[17] depends on the scheduler: feasibility tests for RM and EDF are available
[18] simple if deallocations order is inverse to allocation order (LIFO)

**Table 6.1.:** A comparison of common synchronization protocols with priority inheritance (→ Section 6.3.2 for definitions and detailed explanations)

context of our demands from Section 6.2.2, and refer to [19, 73, 172, 262] for further information. Nevertheless, a concrete formalization of PIP and PCP in the context of their implementation for *Smart*OS will be presented in Sections 6.4.1, and 6.7 respectively.

Initially, the mentioned protocols have in common that they address the problem of bounded priority inversion only indirectly by raising the priority of blocking or simply resource holding tasks to potentially accelerate their operation and resource deallocation. For deadlock avoidance HLP, PCP, and SRP maintain a *safe state* (like NPP), and therefore use a rather conservative policy when adjusting task priorities or preemption levels, and deciding if a task is scheduled or if a resource request is granted or denied. While PIP and PCP will block a task not until it requests an already allocated resource (*block on request*), HLP and SRP will already block a task upon its attempt to preempt another task (*block on preemption*): Based on resource-specific *ceiling priorities* $c(r)$, which represent the maximum base priority over all tasks that might potentially request the resource $r$, a free resource is only assigned (PCP), and a task $t$ is only executed (HLP, SRP) if its active priority is higher than the current *system ceiling priority* $\bar{c}$, which in turn is the maximum ceiling priority over all resources which are currently allocated by other tasks $u \neq t$. Please note, that the computation of the ceiling priorities requires some static information about each task's potential resource requirements to which we refer as the *allocation graph* in Table 6.1.

Although the execution and assignment policies of PCP, HLP, and SRP perfectly meet our demand F3[p93] for safety, they severely conflict with F1[p93] which requests for reactivity and performance for both high and low priority tasks. Obviously, they can rapidly lead to the already introduced problems of avoidance related inversion/rejection from Definition II.8: If a task $t_1$ with active priority $p(t_1)$ holds a resource $s$ with ceiling priority $c(s) \geq P_{t_1}$, PCP at least refuses the assignment of any remaining but free resource $r$ to any other task $t_2$ with $p(t_2) \leq c(s)$. SRP and HLP (at least according to the implementation from [63]) implicitly act in a similar way by avoiding the execution of such a task $t_2$ entirely. Even though these implicit and anticipatory reservations would also ensure a fast assignment of further resources to $t_1$, i.e. each task is blocked at most once, they are critical in many respects: First, $t_2$ is rejected even if $r$ will not be allocated by $t_1$ for a long time. Second, for simplicity and performance in many implementations it is assumed that any task may request any resource, i.e. that the potential allocation graph is entirely meshed. In this case, $t_2$ would also be rejected if it does not even share a single resource with $t_1$. Finally, the penalty is even worse if the protocol already raised $p(t_1)$ above its original base priority $P_{t_1}$ while in fact, $t_2$ was specified to be truly more relevant than $t_1$ (i.e. $P_{t_2} > P_{t_1}$). In summary, a task implicitly prevents other tasks with equal or lower priority from being served while it simply holds a (maybe rarely shared) resource. When recalling our motivation for both real-time resource requests and long-term resource allocations from Section 6.2.2, it becomes obvious that such a behavior is not acceptable. Another problem with PCP and HLP is, that resource ceiling priorities complicate the support for dynamic task base priorities. For large task systems in particular, this feature will either cause significant algorithmic runtime effort to avoid the comeback of deadlocks, or it must be disabled while tasks hold resources. Finally, arbitrary resource allocation nesting is hard to manage in case of independent allocation and

deallocation orders. SRP in particular requires LIFO order for a simplified implementation and the option to share a common stack among the application tasks.

**Summary.** Though PCP, HLP, SRP, and NPP inherently prevent deadlock situations (→ F3), and even restrict allocation delays or blocking (of initially unknown duration) to every first requested resource per task, these techniques imply some serious problems. Apart from NPP they need initial information about each task's worst case resource requirements, which are often not even constant over several runs, but depend on various conditions and the code execution flow. Most notably, long-term resource allocations can immediately trigger inheritance related starvations and avoidance related rejections, and consequently result in disastrous performance effects on lower priority tasks (→ F1).

Thus, we selected the priority inheritance protocol (PIP) as basic technique for our DynamicHinting approach. In comparison to the presented alternatives, PIP is much more generous when granting resource requests, and often gains a better average case performance [184]: Since PIP treats safety against fairness, it achieves a good trade-off between serving high and low priority tasks. Here, requests for currently free resources are always granted immediately: The successful allocation of a resource $r$ by a task $l$ will initially leave $l$'s priority untouched. Then, as soon as a task $h$ with higher priority requests $r$, $l$ will be raised to the priority of $h$. This avoids unbounded priority inversion and allows $l$ a fast deallocation of $r$ (at least in theory) by granting more CPU time compared to scheduling with its initial priority. By doing so $l$'s priority is reduced again, $h$ obtains $r$ and is finally resumed. In turn $l$ is preempted and transits to ready state. However, since PIP will fortunately neither cause inheritance related starvation nor avoidance related rejection, it may consequently and unfortunately lead to chains of resource blocked tasks (*chain blocking*[14]) as depicted in Figure 6.5a, and even deadlocks may occur. Whether these problems can still be avoided or prevented entirely depends on the general resource management policy of the underlying operating system. Although this is not the case for *Smart*OS, we will show that both shortcomings can be handled and solved by our DynamicHinting approach in an efficient manner. In fact, though complete deadlock avoidance is a desirable property of autonomous systems (→ F3[p93]), it is not very practical in most reactive embedded applications.

So far, we presented our requirements for real-time-aware resource management along with some already available approaches and protocols for SANet and general embedded applications. Additionally, though the central idea behind DynamicHinting can also be combined with some other techniques, we motivated our decision for using the priority inheritance protocol as basis for our new approach, since it inherently allows long-term resource allocations without avoidance related inversions. Nevertheless, we will also confirm this decision by comparing performance results between a PIP and a PCP version within the test beds in Section 6.7.

[14] Regarding the allocation delay, a task can be blocked at most by $i$ critical sections of lower priority tasks or by $j$ critical sections corresponding to resources shared with lower priority tasks: If $U$ denotes the allocation time for each task and resource, then the blocking delay $\tau := \min\{i, j\} \cdot U$.

## 6.4. Resources and Priority Inheritance under *Smart*OS

This section presents the details about our novel resource management approach. Apart from embedded and real-time systems, the basic idea behind DynamicHinting may be applied as integral concept for many other operating systems in case these support truly preemptive and prioritized tasks, plus a timing concept that allows temporally limited resource requests. For our reference implementation we extended *Smart*OS (→ Chapter 4), since it provides appropriate task, timing and resource basics, and consequently allowed an easy integration. In addition, it is available for several microcontroller architectures, offers quite common OS characteristics, and thus is a good representative for the adaptation of similar systems: Since *Smart*OS was developed for reactive systems, it inherently applies a priority-based scheduler for fully preemptive tasks: Apart from syscalls which are executed in the kernel mode, atomic sections are entirely forbidden. Each task has its individual and dynamic *base priority*, which satisfies our request for a flexible adaptation to changing demands and environmental conditions at runtime, and an *active priority* which reflects dynamic task dependencies and dominates the scheduling process. Furthermore, the kernel maintains a local system time and provides a temporal semantic through the API which enables tasks to suspend themselves while waiting for events and resources with a certain (relative) timeout or (absolute) deadline. Thereby, tasks may react on resource allocation failures and event imponderabilities without jamming the whole system (e.g. by spinning request loops). Beyond, this already provides an early and simple method for delayed but guaranteed deadlock recovery.

With respect to our feature requests F1 – F5 from Section 6.2.2, we'll now formalize the extended resource management policy of *Smart*OS along with our novel strategy to extend the priority inheritance protocol with reflective task collaboration for on-demand resource sharing. See Section 6.7 for a comparison to the priority ceiling protocol version.

### 6.4.1. Extended *Smart*OS Specifications

This section formalizes the central properties of the *Smart*OS kernel as presented in Section 4.3, and extends the informal description by resource-related specifications.

S1 Each *Smart*OS system consists of a set of preemptive tasks $T$ ($|T| \geq 1$), events $E$ ($|E| \geq 0$), and non-preemptive resources $R$ ($|R| \geq 0$). Each task $t \in T$ is executed for the whole system runtime and can neither be started dynamically nor terminate entirely. As depicted in Figure 6.4, each task $t \in T \setminus \{t_0\}$ is always either in

- *waiting state*, if it can currently not be executed since it waits for the invocation of an event (i.e. a regular event, a resource allocation, or some time to elapse),
- *ready state*, if it is neither executing nor waiting for any event (instead it is in the ready queue, and can potentially be selected for execution on the CPU), or
- *running state*, if it is the first task in the ready queue, and thus itself or the kernel is currently executed on the CPU[15].

[15]In case of idle tasks (→ Section 4.3.2[p50]), this does not necessarily mean that the CPU is active.

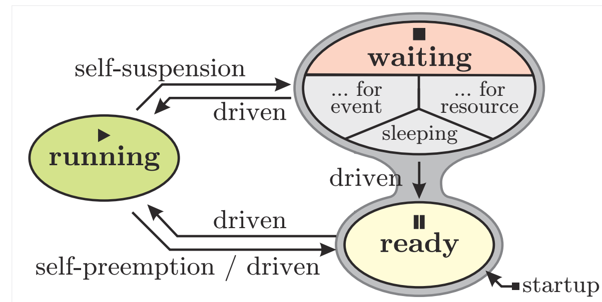



**Figure 6.4.:** Task state transitions under *Smart*OS

On startup, all tasks are in *ready* state. At runtime exactly one task is always in running state. The system idle task $t_0$ will never suspend itself, and thus can only transit between ready and running state. In particular, it will never wait for any event or request any resource.

S2 Each task $t \in T$ has a *base priority* $P_t$ which is defined at compile time, and can be changed at runtime by application code. An additional *active priority* $p(t)$ is used by the scheduler which always selects a task with highest active priority in ready state for execution. For tasks with equal active priorities either round robin (default) or strictly cooperative scheduling can be selected at compile time.

For our DynamicHinting approach we define $p(t) \geq P_t$, and assign active priorities according to a dynamic priority inheritance protocol like e.g. PIP. At system start $\forall_{t \in T} : p(t) = P_t$ holds. In case of the system idle task $t_0$, $p(t_0) = P_{t_0} = 0 = \text{const}$.

S3 State transitions can be triggered in three ways (→ Figure 6.4):

a) self-suspension: the *running* task transits to *waiting* state by either sleeping, requesting a resource which is currently allocated by another task, or by waiting for a still outstanding event.

b) self-preemption: the *running* task transits to *ready* state by releasing a resource, or by invoking an event for which a higher prioritized task is already waiting. Reducing its own base priority might also cause this transition if, by this action, another task in ready state receives a higher active priority.

c) driven: a task's state is changed due to any other task's or IRQ handler's operation (e.g. by invoking an event or releasing a resource), or if its deadline for a self-suspension in waiting state is reached.

S4 All resources in $R$ are treated as non-preemptive and will never be withdrawn. Once assigned, each owner task is responsible for releasing its resources. In particular, no other task or IRQ handler can force their release or handover[16].

[16]The question if and when a task will release its resources cannot be decided by *Smart*OS at runtime. If this is relevant for the correctness of the overall system other methods – like e.g. formal verification – must be applied at application level, and might even identify code positions which violate the specific requirements.

S5 If a task $t \in T\backslash\{t_0\}$ requests a currently free resource, it immediately succeeds and remains running or at least ready[17]. Otherwise it transits to waiting state. Waiting can be limited by the specification of a timeout. A task remains in waiting state until it receives the resource or until the timeout is reached. Then, depending on the other tasks, it transits to either ready or running state.

S6 Any application task $t \in T\backslash\{t_0\}$ may allocate any resource several times and must release it as often. Requests for already self-allocated resources are granted immediately without self-suspension.

S7 Each task $t \in T\backslash\{t_0\}$ may wait for at most one single resource $r \in R$ at the same time. The awaited resource is

$$\alpha_t := \begin{cases} \emptyset & \text{if } t \text{ awaits no resource} \\ r \in R & \text{if } t \text{ awaits } r \end{cases}.$$

S8 Several tasks $T_r \subsetneq T\backslash\{t_0\}$ may await the same resource $r \in R$ at the same time (due to S5, S6: $T_r \neq T$):

$$T_r := \left\{t \in T \mid \alpha_t = r\right\}.$$

S9 Each resource $r \in R$ may be assigned to at most one owner task $t \in T\backslash\{t_0\}$ at the same time. The owner task is

$$\sigma_r := \begin{cases} \emptyset & \text{if } r \text{ is not assigned to any task} \\ t \in T\backslash\{t_0\} & \text{if } r \text{ is assigned to } t \end{cases}.$$

S10 Each task $t \in T\backslash\{t_0\}$ may hold exclusive access to several resources $R_t \subseteq R$ at the same time:

$$R_t := \left\{r \in R \mid \sigma_r = t\right\}.$$

Allocation and deallocation orders of resources are arbitrary and independent from each other.

S11 As soon as a task $t \in T\backslash\{t_0\}$ releases a resource $r \in R_t$ entirely, $r$ will directly be handed over to the task $u \in T_r$ with highest active priority $p(u)$. For tasks with equal priorities the one which requested $r$ first will receive it and leaves waiting state (→ S3c).

In consequence to these specifications, adequate methods for dealing with deadlocks will be required at runtime since the four *Coffman conditions* Co1–Co4 [70] are fulfilled and deadlock prevention is not possible:

Co1 *mutual exclusion* (due to S9),
Co2 *hold and wait* (due to S7, S10),
Co3 *non-preemptive resources* (due to S4), and
Co4 *circular waits* (due to S5 and S10).

[17]For the special occasion that another task $u$ with $p(u) > p(t)$ reaches its deadline during the execution of the corresponding syscall, the scheduler will resume $u$ instead, and $t$ implicitly transits to ready state.

### 6.4.2. Priority Inheritance and Deadlock Occurrence

Having introduced the formal *Smart*OS specifications, we'll next define the resource-await-queue (RAQ) as central data structure for our resource management approach.

---

**Definition II.9: The Resource-Await-Queue $A(t)$**
The resource-await-queue $A(t)$ of a task $t \in T$ is an alternating list of tasks and resources for the representation of currently existing task-resource dependencies[18] (→ Figure 6.5a):

$$A(t) := \left( t, \underbrace{\alpha_t}_{\in R}, \underbrace{\sigma_{(\alpha_t)}}_{=u \in T}, \underbrace{\alpha_u}_{\in R}, \underbrace{\sigma_{(\alpha_u)}}_{=v \in T}, \alpha_v, \sigma_{(\alpha_v)}, \ldots \right) \quad \text{with length } |A(t)| \tag{6.1}$$

---

Due to the *Smart*OS specifications and the definition of RAQs, three important facts become obvious:

**Lemma II.2.** *On the structural properties of RAQs:*

*a) For each task $t \in T$, $A(t)$ is well-defined and cannot diverge, since*

$\forall_{x \in A(t)} : \text{outdeg}(x) \leq 1$ *due to S7 and S9.* □

*b) Two or more RAQs may converge (→ Figure 6.5b), since*

$\forall_{x \in A(t)} : \text{indeg}(x) \geq 0$ *due to S8 and S10.* □

*c) For each task $t \in T$, $A(t)$ ends either in a task or in a cycle (→ Figures 6.5a,d).*

*Proof.* Assume $A(t)$ ends in a resource $r \in R$. Then, $\sigma_r = \emptyset$ and $\exists_{u \in T} : \alpha_u = r$. This means that $u$ would await $r$ even though $r$ is free – a conflict to S5 and S11. □

Lemma II.2 directly leads to some more observations:

1. If $A(t)$ and $A(u)$ contain at least one common element $x \in R \cup T$, they also contain at least one common task $v \in T$. Finally, $A(v)$ controls the further execution of both $t$ and $u$. Within the example in Figure 6.5b the tasks $t_1, \ldots, t_6$ depend on $A(t_6)$ and finally on $t_7$.
2. If $A(t)$ does not end in a cycle, only its last task can be in *ready* or *running* state. All other tasks are currently *waiting*.

These observations are exactly the critical point when dealing with resource management under real-time conditions. The tail of a RAQ (cycles will be addressed later) always prevents all other tasks therein from running because of at least one certain resource. Until now, this was regarded entirely independent from any task priorities. However, we actually want all tasks to be

[18] For the internal representations of this dynamic data structures, each TCB contains a pointer to the resource the task currently waits for, and each RCB contains a pointer to the its current owner (→ Section A).

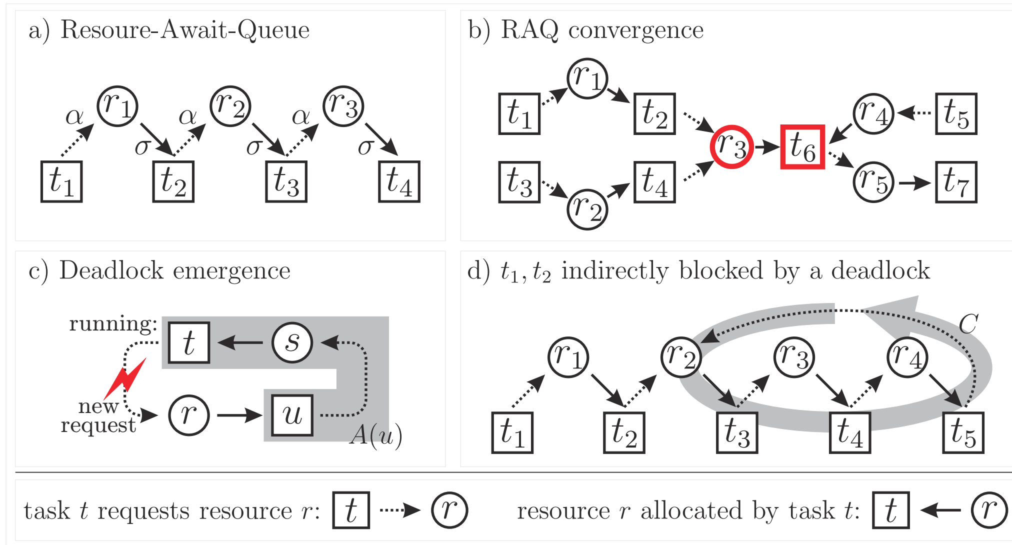



**Figure 6.5.:** Examples for Resource-Await-Queues

scheduled close to their intended *base priorities* (→ F1), but we also mentioned in Section 6.3.1 that this is not always possible due to priority inversions. Therefore, we adapt each task's active priority to the resource situation as follows:

Each task $t \in T$ always receives the maximum active priority $p(u)$ of all tasks $u \neq t$ it currently blocks. Obviously $t \in A(u)$ holds, and $t$ blocks *all* its preceding tasks in $A(u)$. In this case we want $t$ to release its resources quickly to grant a fast resumption of more important tasks.

With respect to F1 and F2[p93], we want to obtain the task active priorities efficiently, and without using additional memory or data structures except for those which are already provided by *Smart*OS, i.e. the task and resource control blocks from Appendix A[p321]. The problem with this is, that our demand for convenient usability (F4[p93]) complicates this procedure: While timeouts can split RAQs arbitrarily and modify system-wide task-resource dependencies asynchronously to any executing task, the support for dynamic base priorities and independent allocation/deallocation orders prohibits the use of stack based priority maintenance[19]. Apart from the memory overhead which would arise from maintaining task-specific priority stacks, each task's active priority may vary severely and arbitrarily, and will in particular not necessarily decrease in the opposite order in which it did increase previously. Consequently we need to re-compute any task's active priority each time a related RAQ changes. Therefore, as shown in Figure 6.6, we initially define

$$w(r) := \begin{cases} 0 & \text{if } T_r = \emptyset \quad (\text{indeg}(r) = 0) \\ \max\left\{p(t) \mid t \in T_r\right\} & \text{if } T_r \neq \emptyset \quad (\text{indeg}(r) \geq 1) \end{cases} \tag{6.2}$$

as the maximum active priority of all tasks $t$ currently waiting for the resource $r$. Hence, to

[19] E.g. similar to the proposed SRP implementation in [19]

support "true priority inheritance" (cf. Table 6.1), the optimum active priority $p(t)$ for each task $t \in T$ has its lower bound limited by its allocated resources at

$$W(t) := \begin{cases} 0 & \text{if } R_t = \emptyset \ \ (\text{indeg}(t) = 0) \\ \max\{w(r) \mid r \in R_t\} & \text{if } R_t \neq \emptyset \ \ (\text{indeg}(t) \geq 1) \end{cases}. \tag{6.3}$$

Furthermore, $p(t)$ is always limited to the bottom by its own base priority $P_t$. Finally, $t$'s active priority computes as

$$p(t) := \max\{P_t, W(t)\} \geq P_t. \tag{6.4}$$

This does not only solve the priority selection for priority inheritance, but it also leads to the fact that priority inheritance is transitive:

**Lemma II.3.** *Each RAQ $A(t)$ is always partially ordered by active priorities and thus its tail has highest active priority.*

*Proof.* If $|A(t)| = 1$ then $\alpha_t = \emptyset$, in particular no other task is in $A(t)$, and the Lemma obviously holds. Otherwise, according to Lemma II.2, for each resource $r \in A(t)$ exists exactly one owner task $v \in A(t)$ and exactly one task $u \in A(t)$ waiting for it:

$$\forall_{r \in A(t) \cap R} : \exists_{u,v \in A(t) \cap T} : \alpha_u = r, \sigma_r = v \quad \Rightarrow \quad u \in T_r \neq \emptyset, r \in R_v \neq \emptyset$$

According to Definition II.9 this covers all tasks in $A(t)$, and finally the Lemma holds since

$$p(v) \overset{(6.4)}{\geq} W(v) \overset{(6.3)}{\geq} w(r) \overset{(6.2)}{\geq} p(u).$$

□

Figure 6.6 gives an example, and also shows the mentioned chain blocking from Section 6.3.2. At the end of Section 6.4.1 we already mentioned that the extended *Smart*OS specification does not prevent deadlocks (→ Coffman conditions Co1–Co4[p104]). In fact, our resource inheritance strategy from Eq. (6.4) does not avoid them either, and thus they still require closer examination (→ Figure 6.5d):

**Lemma II.4.** *If any RAQ $A(t)$ contains a cyclic subsequence $C$, then:*

*a) $C$ contains at least two tasks* — *proof by S6.* □

*b) $A(t)$ contains no other cycle* — *proof by Lemma II.2a.* □

*c) All tasks in $A(t)$ are suspended* — *proof by Lemma II.2c.* □

*d) $\forall_{u,v \in C \cap T} : p(u) = p(v)$* — *proof by Lemma II.3.* □

*e) $C$ is a persistent deadlock if no task $t \in C \cap T$ specified a deadline. Otherwise it is a temporary deadlock.* — *proof by S5.* □

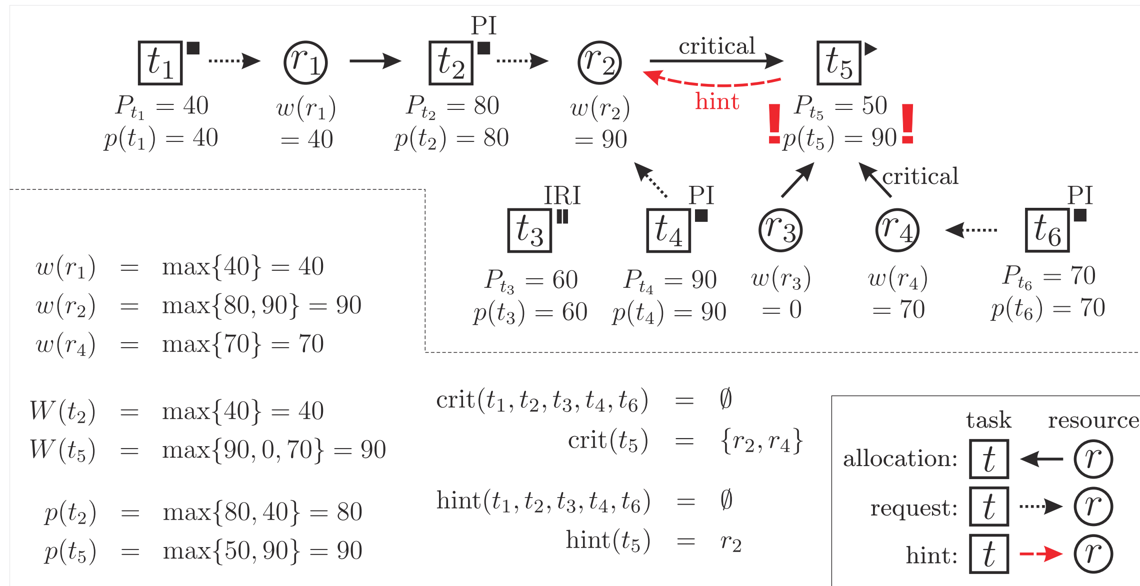



**Figure 6.6.:** Example for priority inheritance and DynamicHinting

Let's consider the consequences: By requesting a resource $r$ which is currently allocated by a task $u = \sigma_r$, the running task $t$ only produces a deadlock if it already holds another resource $s \in A(u)$ (→ Figure 6.5c). Then, $A(u)$ and $A(t)$ contain exactly the same elements. Thus, deadlocks are first considered when a task requests a resource.

As requested in Section 6.2, the formalization of our resource management policy along with the priority inheritance protocol is ready to support arbitrary and independent resource (de-)allocation orders plus dynamic base priorities. Next we'll present our DynamicHinting approach for improving bounded priority inversion, chain blocking, and deadlock situations, before implementation details follow in Section 6.6.

## 6.5. The DynamicHinting Approach

The central objective of DynamicHinting is to grant tasks the collaborative access to exclusively and temporally shared resources while, at the same time, it allows them to closely comply with their intended base priorities. We already introduced various related compositional problems within preemptive and concurrent task systems, but also motivated in Sections 6.2 and 6.3 why we accept and handle them dynamically at runtime.

### 6.5.1. Addressed Problems

Many conservative resource management approaches try to avoid deadlocks and chain blocking by simply refusing a resource request (or even a task execution) immediately if it would cause such problems or could do so in the future. Others take the risk and accept such situations, but simply suspend any requester until it can be served. In our opinion, both methods are not satisfying since exactly the just rejected or suspended task $h \in T$ alone has to cope with the situation. This is especially annoying if $h$ is truly more important than at least one other task $t$ in

the just averted cycle or extended chain. Independent from the applied resource management policy, this results in a violation of base priorities ($P_t < P_h$). Furthermore, resources are usually indispensable when requested, and these requests are commonly not meant to be replaced by any other proceeding. Then, developers, or tasks respectively, tend to retry infinitely until the request succeeds. This results in so called *active* or *spinning loops* or in long timeouts, and might not only starve other tasks, but even worse, this will simply shift the problem back from system level to task level without further information about how to proceed effectively. In fact, since the task-resource-dependencies and RAQ structures are highly dynamic and depend on the system-wide allocation order, another task therein might react much better than $h$ if it knew about the situation. Unfortunately, tasks are commonly not aware about their spurious influence on each other and so the RAQs are reduced successively, beginning at their very end (→ Lemma II.3)[20]. This is exactly where DynamicHinting applies.

### 6.5.2. Cooperation and Collaboration

Before we start, we'll clarify the basic terminology, and disambiguate the terms *cooperation* and *collaboration* from Figure 3.2[p37] in the context of concurrent task systems:

In today's literature, both terms are sometimes used differently and sometimes interchangeably. In [54], Brna presents a detailed overview on common meanings of the notion of collaboration. Among these we find two definitions which reflect our requirements adequately, since they allow a precise differentiation towards the notion of cooperation. Initially, according to Roschelle and Teasley [253], *"collaboration is a coordinated, synchronous activity that is the result of a continued attempt to construct and maintain a shared conception of a problem"*. In contrast, *"cooperative work is accomplished by the division of labor among participants, as an activity where each person is responsible for a portion of the problem solving"*. According to Grice [114], collaboration may be extended by a formal contract or a contract-like component, whereby the participants are obliged, at least to some extent, to participate actively in solving a problem. In the context of this work, both terms will play a substantial role, and thus we'll adapt them according to our requirements with tasks taking the role of the participants. While the differentiation can still be smooth we define them according to [114, 253] as follows:

---

**Definition II.10: Cooperation and Collaboration**

As *cooperation* we denote the intentionally planned and explicitly implemented interaction between tasks in order to conjointly fulfill a common objective, and consequently to comply with the system specification. In particular, these interactions are inherently part of the system design and known at development time. As *collaboration* we denote the dynamic arrangement and mutual engagement of tasks to solve one or more sporadically emerging problems upon their occurrence through coordinated efforts and on-demand interaction. In particular, the set of involved tasks and the degree of interaction is variable and situation specific, and thus unknown at development time.

---

[20] Please note: Though limited to at most one blocked resource per task ($\forall_{t \in T} : |A(t)| \leq 3$), this problem also exists for e.g. PCP and HLP.

Similar definitions within information sciences are known from information processing [115] and agent systems [59]. A common scenario for cooperation and explicit task dependencies is the exchange of information via a common and dedicated data structure, e.g. among the producer/consumer tasks from Listing 4.4[p57]. Here, the local objective to jointly fulfill a function was intentionally divided between a fixed number of tasks at development time, and is part of the system design. In contrast, in the context of DynamicHinting, collaboration means the distributed detection and handling of implicit task-resource dependencies. Since these are often unpredictable, tasks acquire the related information autonomously, but coordinate their operation on-demand for resolving the situation. Since this obviously requires some distributed knowledge, the information must be accessible for the involved tasks, and comprises

C1 the local objectives which can e.g. be derived from global objectives as depicted in Figure 1.1[p5], (defined within the system specification)

C2 a notion and an appropriate description of the current problem, (provided by dynamic hints)

C3 the knowledge about individually available solution strategies and techniques, (task and task state specific)

C4 an association for relating the objectives, problems, and potential solutions. This includes the time and quality awareness, and the self-evaluation through the semantic use of data (→ Figure 1.7[p14]), (e.g. specified through static rules or time-utility-functions)

C5 an optional contract for regulating the task-specific participation and engagement in the problem solving process [114]. (e.g. from real-time and WCRT specifications)

Since we address compositional software design in the context of dynamic resource management, we aim on solving sporadic concurrency problems emerging from implicit dependencies with on-demand collaboration strategies. The optional contract can be defined by the system designer, but commonly aims on reflecting the task priorities.

### 6.5.3. The Proposed Solution

Via so called *hints* our approach provides runtime information for each task about which resource it should release to improve the overall system reactivity and liveliness. Receiving and following these hints is always optional for each task. But if followed, it definitely breaks an RAQ and reduces direct, chain, or deadlock blocking of at least one task with higher priority (→ Figures 6.6, 6.7). In the case of direct blocking, even bounded priority inversions are reduced since the resource handover is accelerated. However, in order to allow blocking tasks to collaborate adequately, two preconditions must be fulfilled:

DH1 A lasting resource allocation must never prevent any task from requesting any resource. Otherwise, our approach lacks the knowledge about the overall system requirements[21]. Within our specifications, this is provided by S2, S5, and the priority inheritance protocol in general.

DH2 A spurious task must receive the time and opportunity to react on a hint. Here, the possibility to request resources with arbitrary timeout (→ S5) provides the time for the current owner $t$, and Eq. (6.4) provides the appropriate priority, i.e. exactly the active priority of the highest prioritized task blocked by $t$.

When the *Smart*OS resource manager determines the currently existing hints, the first step is to identify the *critical resources* $\mathrm{crit}(t)$ for each task $t \in T$ (→ Figure 6.6). These resources currently define $p(t)$ and thus, they directly or indirectly cause the blocking of any task $h \in T$ with higher base priority $P_h > P_t$ and $t \in A(h)$:

$$\mathrm{crit}(t) := \begin{cases} \emptyset & \text{if } p(t) = P_t \\ \{r \in R_t \mid w(r) > P_t\} & \text{if } p(t) > P_t \end{cases} \tag{6.5}$$

According to Eq. (6.4), the presence of a critical resource for a task $t \in T$ implies a raised active priority, and vice versa:

$$\mathrm{crit}(t) \neq \emptyset \Leftrightarrow p(t) > P_t. \tag{6.6}$$

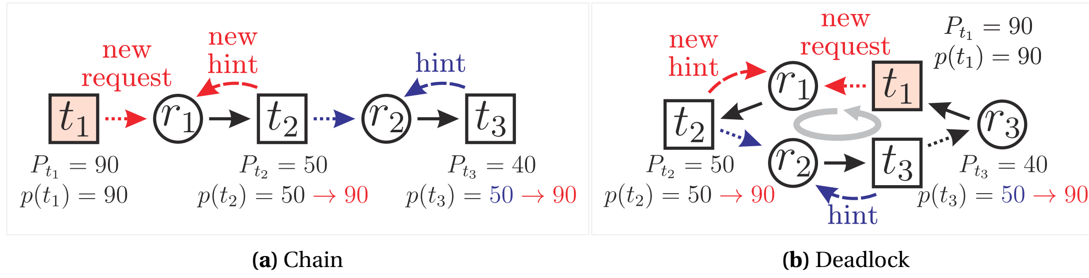
Then, $p(t)$ was raised by at least one pending resource request of a task $u$ with $p(u) = p(t) > P_t$. In turn, $t$ can reduce the blocking of at least one more important task by releasing any resource $r \in \mathrm{crit}(t)$. Finally, $t$'s hint can be selected in many ways, e.g. with regard to the blocked tasks' (remaining) timeouts and periods, or priority thresholds and RAQ characteristics. Even subsets of $\mathrm{crit}(t)$ could be selected and signaled to $t$. Yet, for this chapter our approach always selects the resource $r \in \mathrm{crit}(t)$ which blocks a most important task, and thus defines $p(t) := w(r)$, as follows (→ Figure 6.6):

$$\mathrm{hint}(t) := r \in \mathrm{crit}(t) \quad \text{with} \quad p(t) = w(r) \quad \text{and} \quad r \quad \text{was requested last.} \tag{6.7}$$

While this information is directly signaled to $t$, further critical resources can be queried by the task upon processing the hint by simply calling `getResourceHint(...)` again.

As soon as $t$ releases the indicated resource, it will directly be passed to the first task in the resource's priority queue of waiting tasks, WLOG $u$ (→ S11 and Section 4.3.7[p57]). Next, $p(t)$ is updated according to Eq. (6.2) – Eq. (6.4) and $u$ is scheduled promptly. This is true since then $u$ holds the highest priority of all tasks in ready state, and $t$ did let $u$ pass by (→ S3b,c). As soon as $t$ is scheduled again, it can immediately re-request the just released resource to continue its operation quickly. If, however, there is still another hint for $t$, our approach will signal this situation again. In any case, the untimely release of a hint definitely resolves a priority inversion, and accounts for the intended task base priorities.

[21] Since SRP, HLP, and NPP block on preemption (→ Table 6.1[p99]) these are ruled out entirely, and we have to rely on e.g. PIP and PCP which block on request.




**(a)** Chain **(b)** Deadlock

**Figure 6.7.:** DynamicHinting examples

The example in Figure 6.6 shows $\text{crit}(t_5) := \{r_2, r_4\}$ since both allocations block the execution of more important tasks: $t_4, t_2$, and $t_6$ suffer from bounded priority inversion. In particular, $p(t_5) > P_{t_5}$ was defined by $t_4$'s request for $r_2$. Releasing $r_2$ would instantly relax $p(t_5) := w(r_4)$. Then, $t_4$ is served and scheduled immediately since it indeed is the task with highest priority but currently still blocked by $t_5$. The allocation timeout which $t_4$ specified for $r_2$ grants $t_5$ the time to collaborate as described. If $t_5$ follows its hint $r_2$ prior to its regular release, it indeed reduces the bounded priority inversion toward $t_4$. At the same time, it also reduces the allocation delay of $t_2$ which is also more relevant ($P_{t_2} > P_{t_5}$) and will receive $r_2$ right after $t_4$. As soon as $t_5$ is resumed, it can either re-request $r_2$ or follow yet another hint $r_4$ to speed up the also still blocked task $t_6$ ($P_{t_6} > P_{t_5}$). Please note that according to Definition II.7, $t_3$ suffers from inheritance related inversion (not inheritance related starvation, though) as long as $t_5$ holds either $r_2$ or $r_4$ since then $P_{t_5} < p(t_3) < p(t_5)$. Task $t_1$ can simply be preempted because of its low base priority.

For a better understanding of the implementation details in Section 6.6 and the integration with PIP, we'll briefly address the situations in which a task's hint must be updated via Eq. (6.5) and Eq. (6.7). This is relevant since we want limit the computation of critical resources and hints to a minimum[22] to comply with the performance demand F1[p93].

1. A new $\text{hint}(t)$ evolves iff another task $u$ with $p(u) > p(t)$ requests any resource $r \in R_t$ while $p(t) = P_t$.
2. An already existing $\text{hint}(t)$ changes if another task $u$ with $p(u) \geq p(t)$ requests any resource $r \in R_t$ while $p(t) > P_t$. It also changes or even voids if another task's timeout for the hinted resource is reached or if $t$ releases it.

Two issues are obvious: First, $\text{hint}(t)$ may change each time when $p(t)$ is updated. Second, $\text{hint}(t)$ often changes while $t$ itself is not running. However, $t$ *might* become running then (→ S3c) and must get informed instantly about its spurious influence.

### 6.5.4. Receiving Hints

A special problem with dynamic resource sharing is, that task blocking can occur at virtually any time. From the blocker's view, this happens quasi-asynchronously and regardless of its current

[22]These computations include list operations, and depend on their current length and the position of the affected elements therein.

situation, task state or code position. Thus, we'll now describe three ways in which a task may receive and handle its hints quickly to collaborate, and to speed up more important tasks (code and usage examples will follow in Section 6.7). While a comparative property matrix can be found in Table 6.2[p116], the corresponding API functions are summarized in Listing 6.1[p115].

**Explicit Querying (EQ).** *First*, an *explicit query* can be invoked from within task mode, e.g. periodically or at distinct code positions, where its handling would be possible at all. While an explicit call to the corresponding `getResourceHint(...)` function allows the synchronous, and thus fine-grained and situation specific collaboration, a task can never react on a priority inversion as long as it is in waiting state. Yet, this is mandatory upon deadlocks (→ Figure 6.7b) and during many long-term allocations, where tasks sleep occasionally, or wait for some events or interrupts while holding a resource (→ Sections 6.2.2, 6.7). Then, their collaboration delay is unpredictable and deferred until the next query. Besides this severe weakness, the implementation effort and code pollution might be immense – a contradiction to F4[p93].

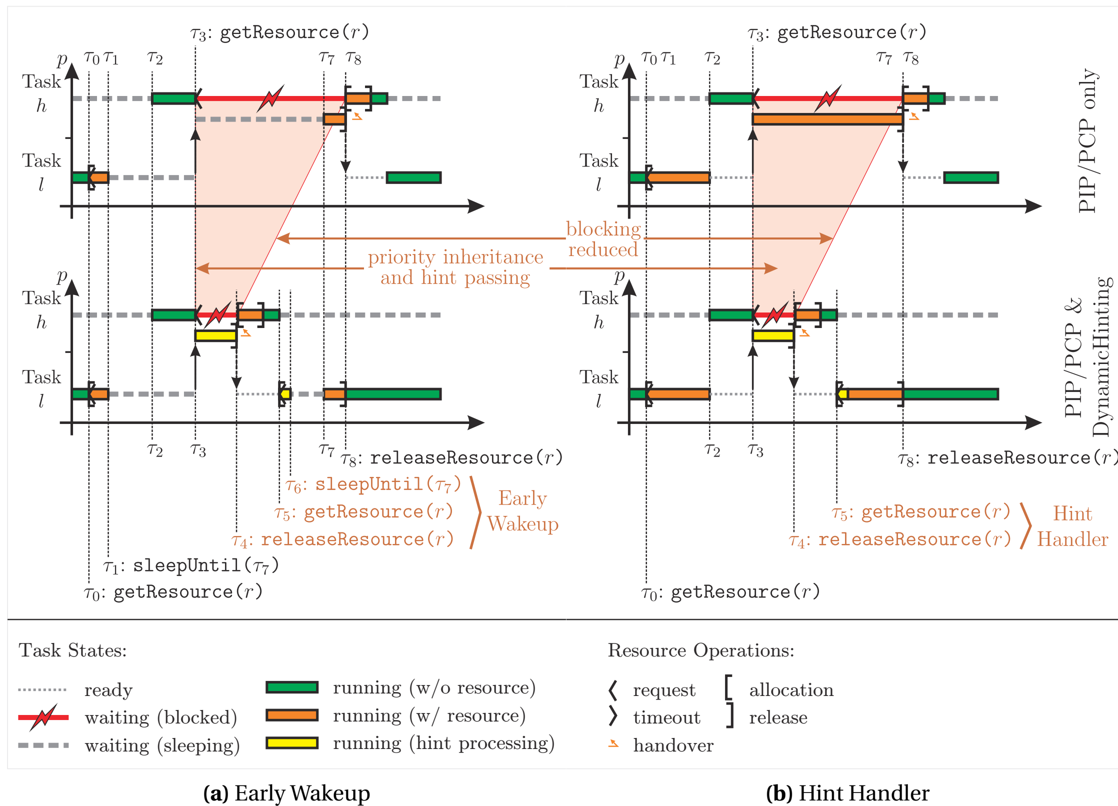
**Early Wakeup (EW).** *Second*, to reduce the collaboration delay from EQ, but to still preserve the advantage of synchronous reactions, we propose the *Early Wakeup* technique. When enabled, all functions by which a task suspends itself to waiting state (→ S3a) may return early upon a new or changed hint. A dedicated return value will indicate this special situation. The effort and impact on the programming model is similar to exception handling in various programming languages or under *Smart*OS in general: A task 'tries' to e.g. sleep but 'catches' an Early Wakeup to react on its blocking influence[23]. This way, coping with hints can be done instantly while it is entirely limited to those cases when they really occur. Figure 6.8a shows an example for a sleeping task $l$ (in waiting state) which nevertheless reduces the blocking delay of a higher prioritized task $h$ promptly: Though $l$ allocated a resource $r$ at time $\tau_0$ and subsequently requested at time $\tau_1$ to sleep until $\tau_7$, it wakes up early as $h$ requests the same resource $r$ at $\tau_3$. In consequence $l$ releases the hinted resource at time $\tau_4 < \tau_7$. The immediate handover suspends $l$ to ready state, but serves $h$ immediately. Finally, $l$ is resumed, reallocates the resource at time $\tau_5$, and continues to sleep from $\tau_6$ until $\tau_7$.

The use of Early Wakeup can be selected and tuned individually by each task $t$ and for each self-suspension. Therefore, we added a call specific threshold parameter $\varphi$ to the involved *Smart*OS syscalls and all subordinate API functions from Figure A.2[p327]. By passing $\varphi$, the threshold will be stored in the caller's task control block (`hintPrio` := $\varphi$), and the self-suspending function will only return early if

$$\varphi \neq 0 \ \wedge \ p(t) > P_t \ \wedge \ p(t) \geq \varphi(t), \tag{6.8}$$

i.e. if priority inheritance raised the caller's priority $p(t)$ to at least the specified threshold $\varphi$. In particular, these functions will also return right after calling if a hint is already available then. See Listing 6.1a for details.

[23] Compared to the ordinary *Smart*OS exceptions from Section 4.3.8, "trying" is implicit and covers just the self-suspending function. In fact, the internal implementation is entirely different, since it is not supported to throw exceptions from one task to another.




**(a)** Early Wakeup **(b)** Hint Handler

**Figure 6.8.:** Hint processing strategies (top: PIP/PCP only, bottom: PIP/PCP & DynamicHinting)

Regarding the RAQs from Figures 6.7, both new requests of $t_1$ will immediately resume $t_2$ if it has Early Wakeup enabled. Then, $t_2$'s own request for $r_2$ is withdrawn. If not enabled, $t_3$ may wake up early, instead. The same applies if $t_2$ refuses to release its new hint $r_1$, but simply requests $r_2$ again. So, beginning with the most promising task, i.e. the one which could reduce the blocking most, the hint is forwarded, and obviously a single collaborative task in a chain or cycle is already sufficient to improve or recover from the situation.

**HintHandler (HH).** *Third*, the last method for receiving hints are so called *HintHandlers*. So far, tasks in waiting state can use Early Wakeup, and running tasks can use explicit querying. The advantage of both methods is, that the hints can be received and handled directly within the task code, i.e. at positions where the developer knows the current task and resource situation, and thus can react in an appropriate way. However, a problem remains for tasks which are seldom in waiting state, or execute code which can not be modified for explicit querying. Indeed, code modifications are often not desired or simply neither allowed nor possible at all[24]. Then, a blocking task's priority is still raised according to PIP, but again the reactivity of other tasks remains reduced, since the resource deallocation is deferred at least until the next query or wakeup occurs. Our solution to this is to allow the resource manager to inject a special handler routine

[24]E.g. restrictive software licenses or the use of closed source might prohibit such modifications.

```
// For sleepX and waitEventX functions
void __syscall_waitEventUntil(
  Event_t *e, Time_t *deadline, φ)

// For getResourceX functions
void __syscall_getResourceUntil(
  Resource_t *r, Time_t *deadline, φ)
```

**(a)** Early Wakeup related syscalls

```
// Set/get dynamic hint handler
void setDHH(handler*)
handler* getDHH(void)

// Set invokation threshold (0 disables)
void enaDHH(φ);
```

**(b)** HintHandler related functions

```
Resource_t* getResourceHint(Priority_t* p, boolean* deadlock, Time_t* deadline);
```

**(c)** General API

**Listing 6.1:** *Smart*OS functions for the DynamicHinting API (apply for the running task)

into the task execution flow, which is capable of releasing the critical resource temporarily with respect to the current task situation. When enabled, these HintHandlers are similar to *Smart*OS IRQ handlers since they can also be invoked asynchronously at any code position. However, in contrast to ordinary IRQ handlers which may never access resources (→ Section 4.3.7 and Table 4.1[p56]), HintHandlers are always executed directly within their corresponding task's context. Thus, they are allowed to operate on the task's resources just like the task itself. In particular, they may release and (re-)allocate critical resources (→ S4), and also use DynamicHinting themselves for solving further potential conflicts. As soon as a handler returns, its corresponding task (or handler, respectively) is resumed where it was preempted before[25].

Figure 6.8b shows an example: Here, the high priority task $h$ preempts the low priority task $l$ at time $\tau_2$ and requests $l$'s allocated resource $r$ at time $\tau_3$. Instead of simply resuming $l$ after raising its priority $p(l) := p(h)$, the scheduler immediately invokes $l$'s task-specific HintHandler, and consequently $l$ prepares for releasing the hinted resource $r$ at $\tau_4$. Again, this suspends $l$ until $\tau_5$ where $l$ reallocates the just released resource, and continues its regular operation right after the HintHandler.

To make the HintHandler's execution entirely transparent and invisible to its corresponding task, it must fulfill two preconditions:

1. It must know how to safely release the hinted resource at the current time, and
2. it must be able to recover the resource's previous state.

Then, similar to the CPU scheduler in preemptive kernels, a HintHandler allows to operate literally non-preemptive resources in a quasi-preemptive way. Sometimes, similar techniques are also provided by drivers. However, these are commonly neither task-specific nor able to handle resource dependencies properly. During our work, we found that it is very common, that further resources depend on a critical resource and must also be handled (→ Listing 4.5[p58]). In fact, the current owner task (or its HintHandler) can do this best, since it has the required knowledge for providing an all-embracing solution.

[25] By preparing the task stacks adequately there is no further context switch when leaving the HintHandler, and its execution is entirely transparent and invisible to the corresponding task. The advanced option to combine HintHandlers with *Smart*OS exception handling will be addressed in Section 6.5.5.

| | Explicit Querying | Early Wakeup | Hint Handler |
|---|---|---|---|
| option while in *running* state | ✓ | – | ✓ |
| option while in *waiting* state | – | ✓ | – |
| option while in *ready* state | on resumption | – | ✓ |
| expected code modifications | many | few | handler only |
| execution of handling code | synchronous | synchronous | asynchronous[1] |
| delay until the hint is received | until query | prompt | prompt |
| task & resource-specific | ✓ | ✓ | ✓ |
| TUFs applicable | ✓ | ✓ | ✓ |
| support for throwing exceptions | ✓ | ✓ | ✓ |

[1] can be synchronized via *Smart*OS exceptions (→ Section 4.3.8)

**Table 6.2.:** Property matrix for the presented hint reception methods

Though it would be useful to associate one HintHandler per task-resource-combination, our current implementation WLOG supports only one handler per task to save memory (→ F2[p93]). While the handler must demultiplex the sources with respect to the hinted resource, it can also be exchanged dynamically at runtime. Similar to Early Wakeup, the current handler must be activated by the specification of a priority threshold $\varphi$. See Listing 6.1b for details. This way, each task can also adjust and fine-tune its individual acceptance for hints. Again, the HintHandler is only invoked if Eq. (6.8) holds.

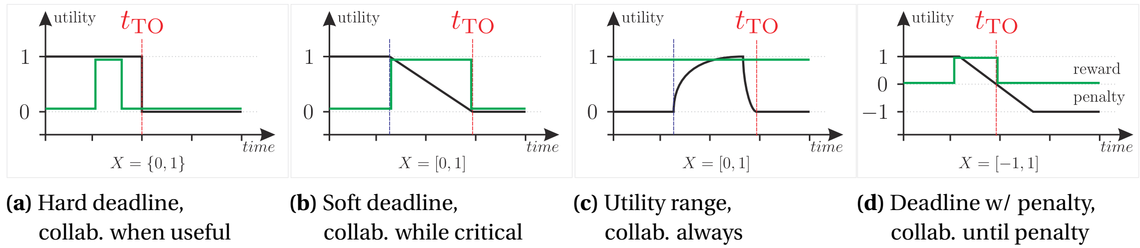
### 6.5.5. Processing Hints

While DynamicHinting can be limited to task-specific priority thresholds, another option is to introduce a common real-time priority threshold by initially defining $\varphi$ equal for all tasks. This inherently limits any potential collaboration to situations where tasks (directly or indirectly) block any real-time task $t_{\text{rt}}$ with $P_{t_{\text{rt}}} \geq \varphi$.

**Time-utility-functions and behavior functions.** Of course, priority thresholds are not the only useful metric for deciding between collaborative or egoistic behavior. Thus, besides the hinted resource itself, we grant each task $t$ access to some further information. According to Listing 6.1c, this comprises

- its current (raised) priority $p(t)$,
- a flag indicating that a deadlock might persist if the hint is not followed, and
- the absolute deadline for the hint to expire due to its latest request timeout.

This information is of special interest for applying time-utility-functions (TUF) as proposed in [113, 176]. First introduced by Gouda et al. [113], TUFs are a generalization of the common deadline constraint: They describe the utility of completing an action within a certain time limit, and therefore refer to the current system state, and static or dynamic demands. As exemplified in Figure 6.9, they are specified as a multivariable function $u : A^k \mapsto X$ which maps a set of $k$ current properties $a \in A$ (e.g. the current time) to a numeric value $x \in X$ indicating a *reward* ($x \geq 0$) or a *penalty* ($x < 0$). While TUFs are commonly used by kernel modules to coordinate




**(a)** Hard deadline, collab. when useful **(b)** Soft deadline, collab. while critical **(c)** Utility range, collab. always **(d)** Deadline w/ penalty, collab. until penalty

**Figure 6.9.:** Various shapes of time-utility-functions for the time domain (black, $u : T \mapsto X$), and potentially associated binary behavior functions (green, $c : T \mapsto \{0, 1\}$)

the scheduling processes, we forward certain resource-related properties to the tasks. This allows each task to reflectively relate its own demands and utility to other tasks and local system objectives, and to decide between collaborative or egoistic behavior. For example a hint could be ignored, if the remaining time until the deadline of the blocked task is still sufficient to complete the own operation regularly, or if it is too short for a timely deallocation. Otherwise it is either always followed, or the decision depends on further factors like the priority difference, task states, energy effort, etc. In fact, we propose the implementation and application of a binary *behavior function* $c : A^n \mapsto \{0, 1\}$ which depends, among other factors, on the current time and on the properties provided by the `getResourceHint(...)` function from Listing 6.1c:

$$c(t_{\text{now}}, \text{hint}, \text{priority}, \text{deadlock}, \text{deadline}, \ldots) = \begin{cases} 1 & \text{collaborate (follow hint)} \\ 0 & \text{otherwise (ignore hint)} \end{cases} \tag{6.9}$$

The test bed in Section 6.7.3 gives a concrete example for such decision criterions.

**Direct hint processing.** By providing the three presented techniques for hint reception, our approach covers all potential states in which a task can be when it starts blocking another more important task (→ Figure 6.8). Table 6.2 gives a summary. Thereby, it avoids the brute force withdrawal of resources or the termination of tasks while it potentially reduces blocking delays caused by resource conflicts. Up to now, handling a hint always follows the same procedure, and targets on a possibly continuous operation of the resource and the task execution flow:

- Query the critical resource, and decide between following or ignoring the hint by using Eq. (6.9). When following the hint:
  1. Save the resource state (if necessary) and stop its operation.
  2. Release the resource. *This will cause an implicit task self-preemption (→ S3b) due to the resource handover (→ S11).*
  3. Reallocate the just released resource upon resumption.
  4. Restore the resource state and restart its operation.

```
1  OS_DECLARE_TASK(t, 200, 100);
2
3  OS_TASKENTRY(t) {
4    Exception_t e;
5    while (1) {
6      getResource(&rResource);
7      TRY {
8        spinFunc(); // throws EX_HH
9                    // via hint handler
10       waitFunc(); // throws EX_EW directly
11     } CATCH (e) {
12       Resource_t *h = getResourceHint(...);
13       switch (e) { // situation sensitive
14         case EX_HH:  /* specific code */
15         case EX_EW:  /* specific code */
16       }
17     }
18     releaseResource(&rResource);
19   }
20 }
```

(a) A task $t$ with centralized hint handling ($P_t = 100$)

```
21 OS_DHHANDLER(hintHandler) {
22   if (behaviorFunction(...) == 1)
23     THROW EX_HH; // cond. collab.
24 }
25
26 /* Spins but can be interrupted
27    by a hint */
28 void spinFunc() {
29   setDHH(&hintHandler);
30   enaDHH(1);
31   while (condition) doSomething();
32   enaDHH(0);
33 }
34
35
36 /* Sleeps but can wake up early */
37 void waitFunc() {
38   if (sleep(3000000, 1) == -1)
39     THROW EX_EW; // uncond. collab.
40 }
```

(b) Hint forwarding via exceptions ($\varphi(t) = 1$)

**Listing 6.2:** Combining DynamicHinting with *Smart*OS exceptions

**Combining DynamicHinting with *Smart*OS exceptions.** Another convenient option apart from the direct processing of hints at the point of their detection (i.e. synchronous or asynchronous) is the combination of DynamicHinting with the *Smart*OS exception concept from Section 4.3.8. Instead of implementing situation specific handlers at each potentially affected code position, exceptions can be thrown (from within a task or HintHandler) in case of collaborative behavior. These can be caught and processed at centralized – and thus easier to maintain – sections of the application task. While throwing an exception will obviously work for Explicit Querying and Early Wakeup, HintHandlers can also do so without any further concerns (→ Section 6.6.4 for implementation details). Most notably, the use within HintHandlers can even allow to pass the initially asynchronous processing of hints back to the task, which synchronizes on the catch block, and handles the request just like any other exception. Listing 6.2 gives an example. The difference between processing a hint directly and raising an exception instead is, that for the first option, the intermediate release of the critical resource is transparent to the task logic, while for the latter option, the resource will most probably not be reallocated, but the task will continue to some kind of failure recovery mode. Then, we have a slightly modified hint handling procedure:

- Query the critical resource, and decide between following or ignoring the hint by using Eq. (6.9). When following the hint, throw an exception. In the exception handler:
  1. Stop and Release the resource. *This will cause an implicit task self-preemption (→ S3b) due to the resource handover (→ S11).*
  2. Do some failure recovery (e.g. restart the try block).

### 6.5.6. Summary

In this section we described, how DynamicHinting can help to reduce resource allocation delays and even to recover from deadlocks. By accepting hints from the resource manager tasks are still free to decide between collaborative or egoistic behavior at runtime. As demanded in Definition II.10[p109] this happens without explicit knowledge about each other. Since our approach gives no guarantee about that, but initially depends on the behavior of the involved tasks, deadlocks might consolidate and blocking persists if all involved tasks behave egoistic and ignore their hints. However, a single collaborative task is already sufficient for the effective improvement of these problems. To reduce this deficit, contracts between tasks may be used to regulate the hint handling, and to account for a stable system behavior even under hard real-time conditions (→ Chapter 7). When considering our demands F1–F4[p93] and the *Smart*OS resource policy, a soft advise is a good chance to facilitate a reflective operation with regard to long-term allocations. This also avoids complex brute force recovery methods which often cause high system load and energy consumption along with still reduced reactivity and hard-to-control system states. Examples and performance tests follow in Section 6.7.

## 6.6. Implementation Details

This section shows the central algorithmic aspects and complexity details about our resource management approach within our reference implementation under *Smart*OS. Regarding the tight performance and memory constraints of many embedded systems, the DynamicHinting API is limited to three central functions[26], the timeout handling, and the support for the presented hint processing strategies. As proposed in Section 4.3.3, the related functions are atomic syscalls and, since kernel data structures like RAQs and task priorities might get changed, always terminate by calling the scheduler.

The algorithmic problem is how to *efficiently* select each task's active priority and dynamic hint simultaneously. Thus, we first consider the situations in which $p(t)$ might change at all:

1. $p(t)$ might increase, if a task $u \neq t$ requests a resource $r \notin R_u$, and thus $t \notin A(u) \rightsquigarrow t \in A(u)$. Since the priority order within RAQs is transitive (→ Lemma II.3[p107]), $r \in R_t$ (direct blocking) needs not necessarily to hold.

2. $p(t)$ might decrease, if $t$ releases a resource $r \in R_t$, or if a task $u \neq t$ with $t \in A(u)$ waiting for a resource $s$ times out. Then, $u \in T_s \rightsquigarrow u \notin T_s$. Again, $s \in R_t$ needs not necessarily hold.

3. $p(t)$ might change in either direction if any base priority $P_u$ is changed at runtime while $t \in A(u)$. We'll omit further details here, since this just requires the application of Eq. (6.4) and Eq. (6.7) to tasks in $A(u)$ until the RAQ's transitive priority order is consistent again.

Before heading to the functional details, we'll consider the computational complexities for $w(r)$ from Eq. (6.2) and $W(t)$ from Eq. (6.3): Since the internal representation for each $T_r$ is a priority queue (of active priorities), retrieving $w(r)$ is in $O(1)$. In contrast, each $R_t$ is a list and thus $W(t)$ is in $O(\text{indeg}(t)) = O(|R_t|)$.

[26] `getResource`, `releaseResource`, and `changeBasePrioritiy`

```
1  int doBreak = 0;
2  Task_t *i = t;         // iterate over A(t) where t is the running_task
3  Task_t *u = NULL;
4  while (i != NULL) {
5    Resource_t r = α_i; if (!r) break; // end of RAQ reached (Lemma II.2c)
6    u = σ_r;
7    if (p(u) < p(t)) {          // Priority order in RAQs is transitive => update
8
9      hint(u) = r;              // r is critical for u and will raise p(u). If enabled
10                                  // by u, hint(u) ≠ ∅ will also cause the hint handler
11                                  // injection on resumption of u from ready state.
12
13     if (p(t) ≥ φ(u))          // Eq.(6.8) will be fulfilled in Line 17, and thus we
14       doBreak =                  // have to wake up u early if it is currently
15         earlyWakeup(u);          // waiting (u will become head of the ready_queue).
16
17     p(u) = p(t);              // priority inheritance: u blocks t
18     update_priority(u);       // Update u's position within its priority queue
19   } else doBreak = 1;         // RAQs are again sorted by increasing priority since
20                                  // tasks with higher active priority than t
21                                  // won't be affected anyway.
22
23   if (doBreak == 1) {
24     hint(u)->deadlock =
25       isInRAQ(u,u);           // u ∈ A(u) <=> deadlock condition detected
26     break;                    // done.
27   }
28   i = u;               // prepare for next iteration
29 }
```

**Listing 6.3:** getResource(...): Main loop for processing the affected RAQ

### 6.6.1. Resource Allocation

`int getResourceUntil(Resource_t *r, Time_t *d, Priority_t φ)`

This function either returns 1 (success), 0 (timeout), or -1 (Early Wakeup) to its caller $t$. Two fundamental cases must be considered:

1. If $r$ is free ($\sigma_r = \emptyset$) or already allocated by $t$ ($\sigma_r = t$), the request succeeds immediately without suspending $t$. Updating any priorities or dynamic hints is not required.

   This is obvious due to Lemma II.2 and S6. In particular, $t$ is at most tail of other RAQs, and thus the partial order of all RAQs remains implicitly valid. Complexity: $O(1)$. ☐

2. If $r$ is occupied by $\sigma_r \neq t$, $t$ is suspended, and must wait for the current owner to release $r$ (→ S9). Thus, priority management and hint selection must be executed for $A(t)$.

   Here $p(t)$ remains unaffected, but $t$'s state is changed from running to waiting. Thus, $\alpha_t := r$ and $t$ is inserted into $T_r := T_r \cup \{t\}$[27]. Due to Eq. (6.2), $w(r)_{\text{new}} \geq w(r)_{\text{old}}$.

   If $w(r)_{\text{new}} > p(\sigma_r)$, the partial order for $A(t)$ is violated and must be fixed by raising $p(\sigma_r) := p(t)$. In this case, $\text{hint}(\sigma_r) := r$ is also updated since $p(\sigma_r)$ is now limited by $r$ due to Eq. (6.3). Both changes might propagate over further task-resource-dependencies,

[27] If the specified deadline $d < \infty$, $t$ is also inserted into the global timeout queue (→ Figure 4.2[p52]).

```
1  Task_t *v = σ_r;
2  α_t = NULL;                  // stop waiting.
3  if (p(t)==p(v)) {
4    while (v) {                // iterate over A(v)
5      Priority_t p=W(v);       // implicitly updates hint(v)
6      if (p == p(v)) break;    // no further changes
7      p(v) = max(P_v,p);       // update p(v)
8      if (α_v != NULL) v=σ_{α_v};
9      else break;
10   }
11 }
```

**Listing 6.4:** Timeout handling: Processing of $A(t)$ if $t$'s request for resource $r$ times out

and so we iterate over $A(t)$ until a task $u \in A(t)$ with $p(u) \geq p(t)$ is found. Other RAQs than $A(t)$ need not be considered due to Lemma II.2a,b. Complexity: $O(|A(t)|)$. □

Listing 6.3 shows the corresponding code, and also indicates the application of Early Wakeup and the HintHandler. According to Section 6.5.4, we already resume the execution of the first *waiting* task $u \in A(t)$ with Early Wakeup enabled and for which Eq. (6.8) is true. If $u$ follows the hint, DynamicHinting was already successful. Otherwise, $u$ will simply continue its execution or restart its own just aborted request. By doing so, a hint can be passed to the next task in $A(u)$.

### 6.6.2. Resource Deallocation

`void releaseResource(Resource_t *r)`

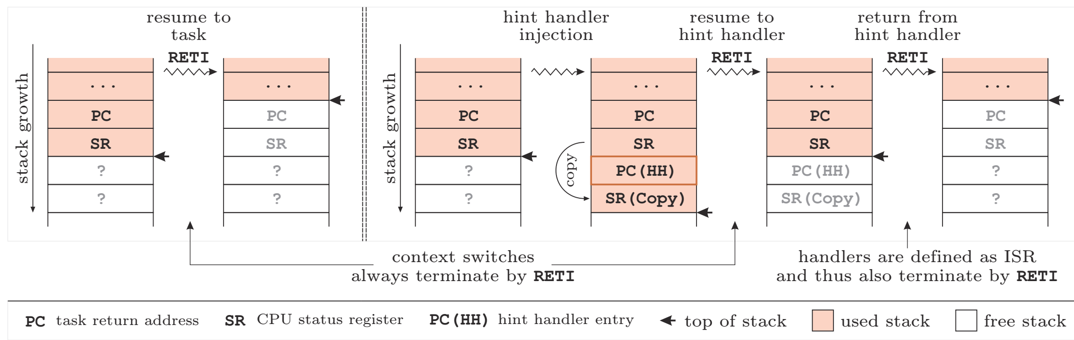
Releasing a resource $r$ is always initiated by its owner task $t = \sigma_r$ (→ S4). If $t$ holds $r$ several times, one is freed and no priority adjustments are required. The same is true if $t$ frees $r$ entirely while $T_r = \emptyset$, since then $w(r) = 0$, and $p(t)$ was obviously not defined through $r$. Complexity: $O(1)$. □

If $T_r \neq \emptyset$ and $t$ releases $r$ entirely, the resource is directly handed over to a task $v \in T_r$ with highest active priority (→ S11). Thus, $\sigma_r := v, \alpha_v := \emptyset, T_r := T_r \setminus \{v\}$ and finally $w(r)_{\text{new}} \leq w(r)_{\text{old}}$. Yet, $p(v) \leq p(t)$ still holds due to Lemma II.3 and priority management remains to be done. First, $p(v)$ of the new owner remains unaffected. But only if $p(v) = p(t)$ holds, $p(v)$ might have defined $p(t)$ in the past and thus $p(t)$ and hint$(t)$ might need an update according to Eq. (6.4) – Eq. (6.7). RAQs need not be iterated since $t$ is the running task, i.e. $|A(t)| = 1$. As desired, $P_t \leq p(t)_{\text{new}} \leq p(t)_{\text{old}}$ finally holds due to Lemma II.3. Complexity: $O(\text{indeg}(t)) = O(|R_t|)$. □

### 6.6.3. Timeout Handling

If a task $t$'s request for a resource $r = \alpha_t$ times out, we have to check and possibly update the priorities and hints for several tasks in $A(t)$. Indeed, $\exists!_{v \in A(t)} : v = \sigma_r$ (→ Lemma II.2) and $p(t) \leq p(v)$ (→ Lemma II.3).

If $p(t) \lneq p(v)$, then $t$ and $v$ are neither on a common cycle nor was $p(v)$ defined by $p(t)$. Hence, neither $p(v)$ nor hint$(v)$ need to be updated and we are done in $O(1)$. □




**(a)** ordinary resumption

**(b)** HintHandler injection

**Figure 6.10.:** Stack layout and evolution when resuming a task (cf. Figure 4.5[p66])

Only if $p(t) = p(v)$ holds, $p(v)$ is currently limited at least by $p(t)$. Listing 6.4 shows the corresponding code: In this case, all tasks in $A(v)$ need to be checked and updated iteratively by Eq. (6.4) – Eq. (6.7) until the partial order is satisfied again. Complexity: $O(|A(v)|) = O(|A(\sigma_r)|)$.

□

## 6.6.4. HintHandlers

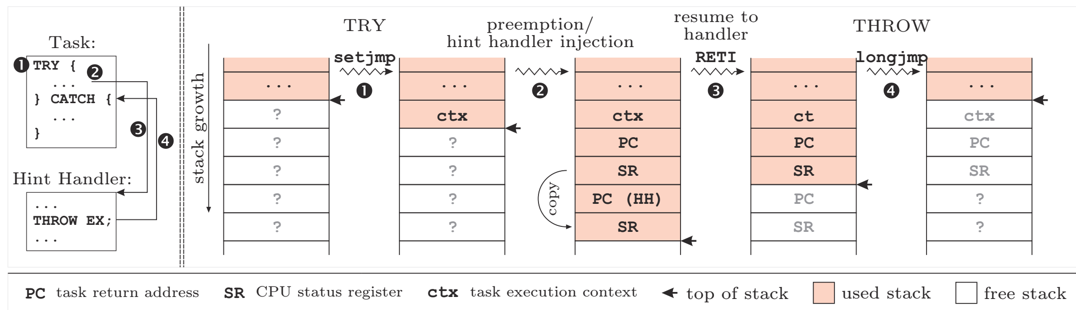
If a task $t$ transits from ready to running state while hint$(t) \neq \emptyset$, $t$'s HintHandler is injected into its ordinary execution flow. To be entirely transparent and to avoid lasting modifications of the task's stack or the register set it uses, handlers always operate according to the callee-saves strategy[28].

Although handlers are implemented like ordinary functions without any parameters or a return value, they are compiled as ISRs and thus return via the `RETI` instruction instead of the expected `RET` (→ Listing 6.6, Line 14). Comparable to many CPU architectures which automatically save machine status flags on IRQ occurrence, and finally restore them via `RETI`, the MSP430 also uses this strategy. Since syscalls and IRQ Handlers are executed in kernel mode just like interrupts (→ Section 4.3.3), we consequently have to preserve the status register across the handler execution, and need to prepare the stack as depicted in Figure 6.10b. Otherwise, the handler would consume these flags (e.g. the overflow flag) and corrupt the subsequent task execution. Resuming into the handler is done by the dispatcher as usual (→ Figure 4.5[p66]): Its final `RETI` instruction follows the context switch to the corresponding task, and, in particular, it consumes only a copy of the status register[29]. In Section 6.5.4 we already requested this behavior in order to execute the HintHandlers in task mode, and to grant them full access to their task's resources. Yet, returning from the handler abuses the `RETI` instruction to finally restore the original status register and resume the original task – although neither a context switch nor a 'true' interrupt return occurs at that time.

Another advantage of this implementation strategy is the inherent compatibility to the

[28]I.e. the called (handler) function has to save and restore the stack frame and the registers it uses.

[29]Since the HintHandler is declared as ISR, it won't depend on the status flags at all.




**(a)** execution flow **(b)** stack evolution

**Figure 6.11.:** Stack layout evolution when throwing an exception from within a HintHandler

*Smart*OS exception concept from Section 4.3.8. If a hint handler is executed within (i.e. injected into) a try block of its corresponding task, and an exception is thrown (but not processed) therein, the `THROW` macro restores the task's context at the beginning of the try block, and continues to the catch block, where the hint can be processed synchronously within the ordinary task execution flow. Leaving the HintHandler this way requires no additional action. In particular, it is not required to return from the handler as depicted in Figure 6.10b. Instead, Figure 6.11 shows the stack evolution in case of an exception throwing HintHandler.

### Memory and Response-Time Overhead

Since the main data structure (i.e. the RAQ) must be maintained anyway in order to support the classic PIP/PCP protocols, there is almost no additional RAM overhead for the DynamicHinting approach itself. In fact each task's control block contains 3 additional machine words: a flag field, a pointer for the associated HintHandler, and another pointer for the current critical resource (i.e. the hint). Of course the additional ROM requirements depend on the code for processing the hints.

The absolute response time overhead for passing a hint depends on the underlying hardware, but equals the time for an ordinary context switch since invoking a HintHandler or resuming a task early is nothing else than switching to the corresponding task context (≈ 35 µs on the test bed hardware from Section 6.7).

## 6.7. Real-World Applications and Test Beds

The novel DynamicHinting programming paradigm as introduced in Sections 6.5 and 6.6 combines temporally limited resource requests, priority inheritance, and the idea of task collaboration to support long-term allocations, arbitrary allocation orders, and dynamic base priorities. At the same time it reduces allocation and blocking delays, avoids inheritance related starvation (IRS), avoidance related rejection (ARR), avoidance related inversion (ARI), and allows tasks to handle deadlocks as these occur. Therefore it was designed to always support and comply with the task base priorities as defined at development time, or as adjusted at runtime.

In order to analyze the benefit of our approach we extended the *Smart*OS kernel as described before. In addition to the proposed use of PIP as underlying protocol, we also implemented a PCP version according to [262]. This allowed us to perform a comprehensive and meaningful comparison between two very common strategies with different claims (e.g. generous resource assignment for PIP versus deadlock prevention for PCP). In fact, according to Table 6.1[p99], only PIP and PCP allow task self-suspensions *and* avoid unbounded priority inversion at the same time. However, since PCP showed significantly less performance in most test beds, especially when dealing with long-term or rarely shared resources, we omit algorithmic and implementation related details here but refer to e.g. [262] instead.

Again, both versions were implemented for the TI MSP430 family of microprocessors, and tested on SNoW5 (→ Section 2.2[p22]) sensor nodes (10 kB RAM, 48 kB ROM) running at 8 MHz. Requiring 4 kB of ROM and 40 B of RAM for the whole kernel, the typically low computational performance and small memory footprint of sensor nodes was considered carefully to leave sufficient space for the actual application. For detailed performance analysis at runtime, we used the integrated *Smart*OS timeline with a resolution of 1 µs (→ Section 4.4.2[p62]). Please note, that besides the definition of the ceiling priorities under PCP, the used test bed sources were exactly identical, and just the kernel was configured to apply either PIP or PCP.

### 6.7.1. DynamicHinting and the Priority Ceiling Protocol

For the integration of PCP (instead of PIP), some specifications from Section 6.4.1 were slightly modified as follows:

- Adds to S2: At system start, each task $t \in T$ *registers* itself for each resource it might require during runtime:

  $$\rho_r = \{t \in T \mid t \text{ is registered for } r\}. \tag{6.10}$$

  $\rho_r$ is not stored explicitly, but allows to incrementally compute the fixed ceiling priority for each resource $r \in R$:

  $$c(r) = \begin{cases} 0 & \text{if } \rho_r = \emptyset \text{ (i.e. } r \text{ is unused)} \\ \max\{P_t \mid t \in \rho_r)\} & \text{otherwise} \end{cases} \tag{6.11}$$

  In consequence, task base priorities are not dynamic any more. Otherwise, this would not only require the re-computation of ceiling priorities at runtime but also jeopardize the deadlock freedom of PCP when changing $P_t$ of any $t \in T$ during lasting resource allocations.

- Replaces S5: If a task $t \in T$ requests a resource $r \in R$, it immediately succeeds and remains running if it already owns $r$ ($\sigma_r = t$) or if, according to the PCP policy, both of the following conditions hold:

  $$\begin{aligned} &1. \quad \sigma_r = \emptyset \\ &2. \quad p(t) > \max\left\{c(s) \mid s \in R \wedge \sigma_s \notin \{\emptyset, t\}\right\} \end{aligned} \tag{6.12}$$

I.e. $r$ is free, and no other task holds any resource with ceiling priority equal or above the requester's active priority. Otherwise, $t$ transits to waiting state until it receives the resource or the timeout is reached. Yet, a blocking task $v$ inherits at least $t$'s active priority: $p(v) \geq p(t)$.

- Replaces S11: If a task $t \in T$ releases a resource $r \in R_t$, the system checks if it can serve another task $u \in T$ waiting for a resource $s = \alpha_u$ (this is not necessarily $r$) which was previously not granted due to the PCP policy. If so, $\sigma_s := u$ and $u$ leaves waiting state. Anyway, $p(t)$ is updated according to the PCP policy.

### 6.7.2. Test Bed I – Continuous Data Streaming

Our first test bed addresses a quite frequently encountered real-world situation: A task $S$ is used to continuously transfer some data over a shared bus $b$ to an external device. The stream is rather long, or even infinite, but it can be interrupted and resumed at any time for more important communication over the same bus. Therefore, however, it always needs some bus setup plus a (complex) header/trailer for proper initiation/termination. During the transfer, $S$ obviously requires exclusive access to $b$. We first encountered this problem within our SNoW Bat localization system (→ Chapter 10): As depicted in Figure 2.8[p31], the top SNoW[5] sensor node had to stream position information to the node below while at the same time, it had to operate a CC1100 radio controller attached to the same bus.

A common solution is to split the stream payload into atomic packets. Then, $S$ would terminate the stream and release the bus temporarily after each packet. This way, other tasks may receive the bus regularly. However, since $S$ does not really know if it currently blocks a more relevant task, the temporary stream interruption and release of $b$ might be completely unnecessary. It is also obvious that the selected packet length has significant influence on the extent of potentially resulting bounded priority inversions as defined in Section 6.3.1. By using short (long) packets, the overhead increases (decreases) while it improves (degrades) the reactivity of higher prioritized tasks when these request $b$. In fact, a fixed length is often selected during development with regard to the individual application requirements. These must be known exactly, then.

Using a server task for coordinating the bus access might even result in slightly worse performance due to the client-server communication overhead. The mentioned problems remain the same, but they are concentrated at the server which commonly also creates atomic packets or grants exclusive bus reservations.

Finally, DynamicHinting provides two improvements. Since our approach knows about pending bus requests, the task $S$ could query its current blocking state periodically and react only if necessary. This time the query interval must still be selected carefully, but the overhead for useless stream interruptions is already avoided. The additional use of HintHandlers or Early Wakeup even gains the desired reactivity, as both inform $S$ *instantly* if it blocks a task with truly higher priority. Therefore, these options must be enabled during the entire transmission, or at least for the delay or suspension between two subsequent data words.

```
1  /* Function for streaming data (executed in the context of task S) */
2  void streamData() {
3    int stop = 0;
4    /* allocate the bus and initiate the stream */
5    getResource(&SPI, 0);                  // no timeout, no early wakeup
6    cfgBus(); header(); startDMA();
7    while (stop != 1) {                    // 1 indicates the stop event
8      /* "Try" to wait infinitely for the stopStream event.
9         Enable early wakeup for p(S) ≥ φ = 100. */
10     stop = waitEvent(&stopStream, φ);
11     if (stop == -1) {                    // "Catch" a hint!

11       Resource_t *hint = getResourceHint(NULL, NULL, NULL);
12       if (hint == &SPI) {                          // conditional hint handling
13         /* stop the stream and release the blocked resource */
14         stopDMA(); trailer(); releaseResource(hint);
15
16         /* ---------- THE TASK WILL BE PREEMPTED HERE SINCE AT ------------
17            ---------- LEAST ONE OTHER TASK WAITS FOR THE RELEASE ---------- */
18
19         /* continue streaming as soon as possible */
20         getResource(hint, 0);
21         cfgBus(); header(); startDMA();
22       }

20     }
21   }
22   /* done. stop the stream */
23   stopDMA(); trailer(); releaseResource(&SPI);
24 }
```

**Listing 6.5:** Streaming test in operation mode 1 (DMA): Use Early Wakeup to react on hints

**Application Setup.** For the concrete application we had to stream 8 bit wide ADC data (sampled at 10 kHz) continuously over an SPI bus. The overhead for each header and trailer was 1 B. Besides, a radio transceiver R and a motor controller M shared the same SPI bus (at different settings) for short communication. Yet, both associated tasks $R$, $M$ had to process sporadic events (average inter-arrival time ≈5 ms) which were much more time and safety critical, since radio packet loss had to be avoided, and especially failures in the motor control were disastrous. So, we initially defined $P_S < \varphi = 100 < P_R < P_M$. For our application and test bed, we implemented the streaming task $S$ to operate the bus (at best effort) in two completely different ways:

**Operation mode 1 (DMA).** To reduce the CPU load, we first used a DMA channel for sending the ADC values continuously to the bus controller. Thus, the streaming task $S$ simply had to allocate and configure the DMA, ADC, and SPI bus resources for a new stream. After starting the DMA transfer, $S$ did sleep until an event signaled to finalize the stream or until a hint occurred. The code in Listing 6.5 shows the relevant implementation details for $S$ when using Early Wakeup. The hint handling itself is highlighted. Most notably, it includes the proper handling of the DMA resource (Line 14) which depends on the current state of the shared bus.

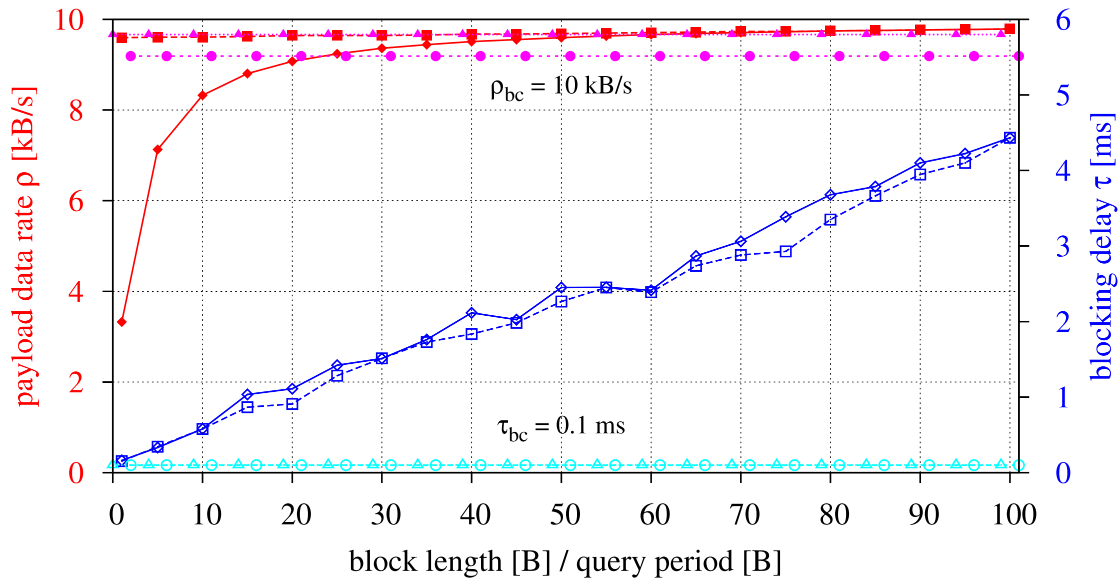
```
1  /* Function for streaming data (executed in the context of task S) */
2  void streamData() {
3    /* register and enable dynamic hint handler */
4    setDHH(&DHH_Stream); enaDHH(φ);
5    /* allocate the bus and initiate the stream */
6    getResource(&SPI, 0); cfgBus(); header();
7    /* get, process & send data until stopStream occurs */
8    while (!checkEvent(&stopStream)) SPI_TX(nextData());
9    /* done. disable hint handler and stop the stream */
10   enaDHH(0); trailer(); releaseResource(&SPI);
11 }

13 /* task-specific hint handler (executed in the context of task S) */
14 OS_DHHANDLER(DHH_Stream) {                    // define the handler function
15                                               // as ISR (return by RETI)
16   Resource_t *hint = getResourceHint(NULL, NULL, NULL);
17   if (hint == &SPI) {                         // conditional hint handling
18     /* stop the stream and release the blocked resource */
19     trailer(); releaseResource(hint);
20
21     /* ------------ THE TASK WILL BE PREEMPTED HERE SINCE AT --------------
22        ------------ LEAST ONE OTHER TASK WAITS FOR THE RELEASE ------------ */
23
24     /* continue streaming as soon as possible */
25     getResource(hint, 0); cfgBus(); header();
26   }
27 }
```

**Listing 6.6:** Streaming test in operation mode 2 (DSP): Use a HintHandler to react on hints

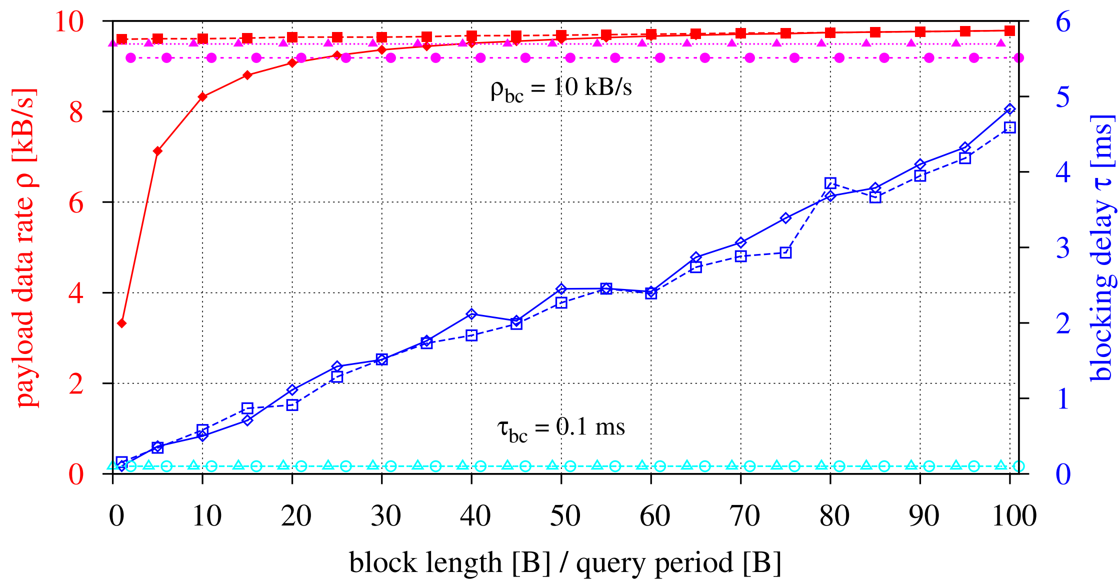
**Operation mode 2 (DSP).** In our second implementation we required the streaming task to also process the data before sending it over the bus. Therefore, it fetched each sample from the ADC and applied some functions first. The additional CPU load did not only reduce the maximal achievable payload rate compared to the DMA version. It also reduced the sleeping time (i.e. the task self-suspension) between two data words, and, in consequence, it also limited the use of Early Wakeup. Thus, we implemented an additional HintHandler and associated it with the streaming task $S$ for a prompt reaction in case of blocking situations. The handler operated fully transparent to $S$, and temporarily released the critical bus resource $b$ – even during the execution of the original foreign code. Again, we selected $\varphi = 100$. For improved readability, the example code in Listing 6.6 shows the streaming function and the HintHandler only.

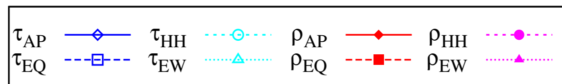
**Test bed analysis.** Processing data while simultaneously executing some sporadic but highly reactive tasks might already cause extreme system load for low performance embedded systems like sensor nodes. In particular if, besides the CPU, other exclusive resources must also be shared. Yet, the test bed results will show that our approach can still gain good reactivity and high throughput without manual task tuning during development. First, we implemented the just described application with *atomic fixed-length packets* (AP) but without DynamicHinting. Then we used DynamicHinting with *Explicit Querying* (EQ), and finally we activated *Early Wakeup* (EW, Listing 6.5) and *HintHandlers* (HH, Listing 6.6).

For our analysis we consider the PIP based system first. Figure 6.12a shows the results in terms of the average blocking delay $\tau$ of the real-time tasks, and the achieved payload data rate




**(a)** Priority Inheritance Protocol (PIP)




**(b)** Priority Ceiling Protocol (PCP)

**Figure 6.12.:** Streaming test: Atomic fixed-length packets (AP) vs. DynamicHinting (EQ/EW/HH)

$\rho$ of the streaming task. Due to the fixed trailer length and sampling rate, the best case blocking delay is $\tau_{bc} = 100\,\mu s$, and the best case payload data rate is $\rho_{bc} = 10\,kB/s$. As expected for the packet oriented design without DynamicHinting (AP), its throughput $\rho_{AP}$ improves while the blocking delay $\tau_{AP}$ degrades rapidly with increasing packet length. When using DynamicHinting with periodic explicit querying (EQ) and on-demand release only, $\rho_{EQ}$ remains nearly constant and close to the achievable maximum $\rho_{bc}$. On the other hand, the selected query period causes the blocking delay to behave analogous to the AP version: $\tau_{EQ}$ almost matches $\tau_{AP}$, and is also not satisfying for long periods. When using Early Wakeup with the DMA version, the data rate is still held high, and additionally the blocking delay (which also includes stopping the DMA) is kept extremely low. Indeed, $\rho_{EW} \approx \rho_{bc}$ and $\tau_{EW} \approx \tau_{bc}$. Finally, when using the HintHandler with the DSP implementation, we still obtain roughly the same reactivity for the real-time tasks: $\tau_{HH} \approx \tau_{EW} \approx \tau_{bc}$. Yet, we can observe a slightly decreased (but still constant) data rate $\rho_{HH}$ close to $\rho_{bc}$. The reason is not the bus or resource management itself, but the fact that the real-time tasks produce variable CPU load which sometimes slows down the data processing within the streaming task.

For better comparability in Figure 6.12 the data rates $\rho_{EW}$, $\rho_{HH}$, and the blocking delays $\tau_{EW}$, $\tau_{HH}$ are visible as horizontal lines, though Early Wakeup and HintHandlers are independent from any block length or query period, but occur just on-demand.

It is worthwhile to note, that in this test bed (since it shares just a single resource among only three tasks), PIP behaves very similar to PCP since the long-term allocation is held by the lowest priority task. Consequently, both protocols produced roughly the same priority assignments, allocation delays, and overall program flow. While the evaluation of PIP implicitly applies to PCP, too, Figure 6.12b shows the corresponding results. In fact, the observed differences always ranged at $\pm$ 0.037 ms for $\tau$ and $\pm$ 0.044 kB/s for $\rho$.

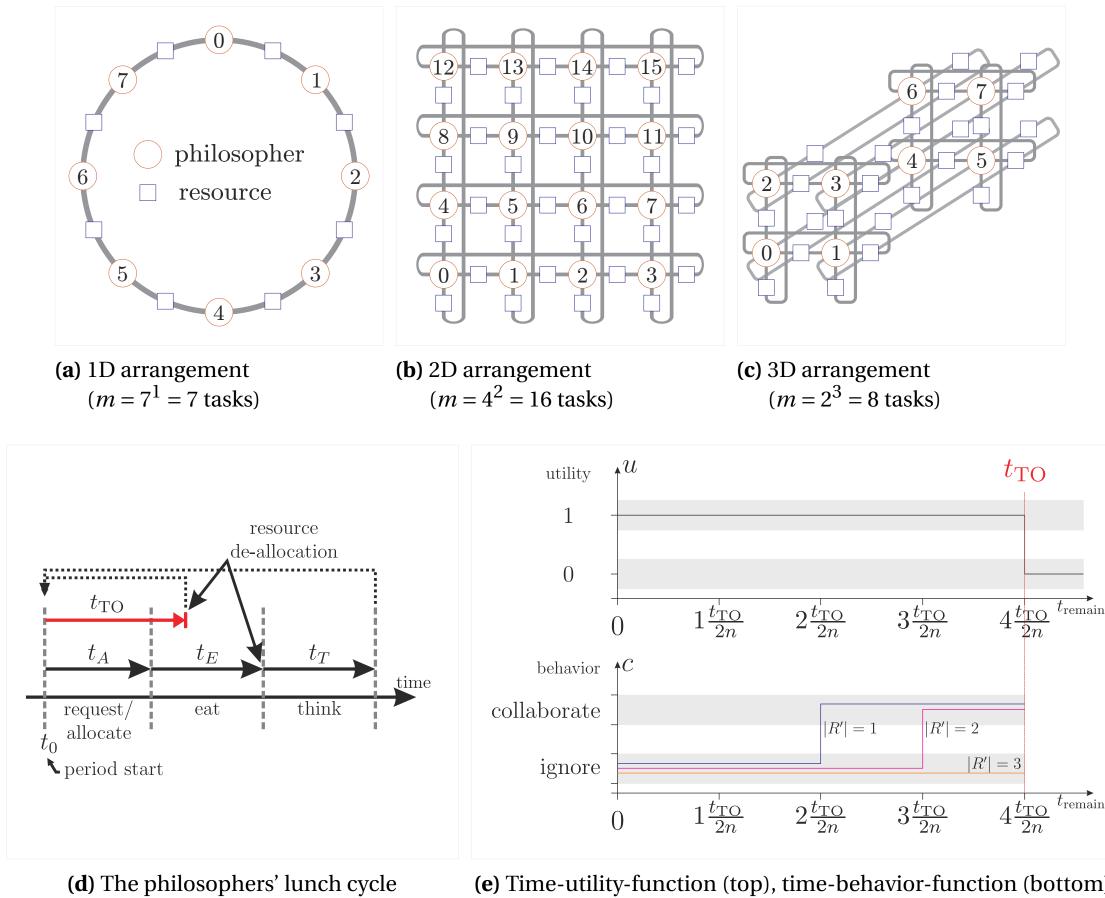
**Summary.** This stream test showed practical results from a real and quite common application scenario[30]. It comprises resource dependencies as well as resource blocking, which is mainly caused by one task in either running, ready, or waiting state. Our approach was able to significantly improve both PIP and PCP to achieve almost maximal data rates at minimal blocking delays. However, only few tasks and one shared resource were involved.

### 6.7.3. Test Bed II – The Dining Philosophers Stress Test

The next step is a tough stress test comprising many tasks, resources, and deadlock pitfalls. Though being more synthetic, the resulting test application still draws some parallels to real-world demands, and allows a deep analysis of our approach under extreme conditions. Furthermore, it demonstrates the supporting use of time-utility-functions (TUF) as described in Section 6.5.4.

Inspired by the well-known *Dining Philosophers Problem* [276], where 'thinking' philosophers (tasks) over and over compete for cutlery (resources) that would allow them to eat, we modified

[30] Note that this scenario also closely relates to the theoretical *sleeping barber problem* [276] with $S$ being the barber sleeping in his chair $b$ until the customers $R$ and $M$ demand a hair cut, and thus also require $b$. Yet, in addition to the original problem, priorities are involved here, and $S$ does not necessarily sleep while holding $b$.




**(a)** 1D arrangement ($m = 7^1 = 7$ tasks)

**(b)** 2D arrangement ($m = 4^2 = 16$ tasks)

**(c)** 3D arrangement ($m = 2^3 = 8$ tasks)

**(d)** The philosophers' lunch cycle

**(e)** Time-utility-function (top), time-behavior-function (bottom)

**Figure 6.13.:** The dining philosopher's problem

the scenario to be more complex. First, we extended the classic one dimensional problem to two and three dimensions. As Figures 6.13a – 6.13c show, this causes much more extensive task-resource-dependencies, and boosts the competition among the philosophers as well as the overall system load. For $c^n = m$ philosopher tasks $P_0, \ldots, P_{m-1}$ in an $n$-dimensional setup, the system will contain $|R| = n \cdot m$ resources, and each task requires $2 \cdot n$ resources for lunch. Then, each dining philosopher directly bars its $2 \cdot n$ neighbors from also doing so since it exclusively holds at least one of their shared resources. Obviously, the number of potential allocation cycles (deadlocks) also increases significantly along with the dimension and task count.

**Application Setup.** According to Figure 6.13d we implemented each philosopher task as follows: At the individual start time $t_0$ of each philosopher's lunch cycle, the corresponding task tries to quickly allocate its required resources. The *entire* allocation attempt for all its resources is temporally limited to $t_{TO}$. If the timeout $t_{TO}$ is reached, the philosopher gives up, releases all resources it allocated so far, and starts over. On success, the *allocation delay* $t_A$ is logged. Then the task takes some fixed time $t_E$ for *eating*, and finally releases its cutlery before *thinking* for a fixed time $t_T$. The relationship to real-world embedded applications are tasks executing

repeated actions for which they require the CPU and some exclusively shared resources with a certain period stability.

Again, the most interesting point is the applied resource allocation concept. Similar to common application logic, each philosopher requests its resources in a fixed order[31]. Therefore, it specifies the same absolute deadline $t_0 + t_{\mathrm{TO}}$ for each part $r$ of the cutlery while enabling Early Wakeup as follows:

```
allocationResult = getResourceUntil(r, t0+tTO, φ);
```

Within our implementation, we applied DynamicHinting with both PCP and PIP in the following ways:

1. PIP / PCP: We disabled the hints completely ($\varphi = 0$) to study the performance of the pure priority inheritance and priority ceiling protocol.
2. PIP+EW / PCP+EW: The philosophers used Early Wakeup ($\varphi = 1$) during their resource allocation stage. In fact, they were *always* collaborative and released each hinted resource on-demand. During each resource reallocation within the hint processing code, further hints were considered in the same way.
3. PIP+EW+TUF / PCP+EW+TUF: We applied a time-utility-function for dynamic runtime decision as follows:

   Initially, we considered the allocation timeout to be hard, and, according to Section 6.5.5, specified each task's utility at time $t_{\mathrm{now}}$ as binary function

   $$u(t_{\mathrm{now}}, t_{\mathrm{TO}}) := \begin{cases} 1 & \text{if } t_{\mathrm{now}} \leq t_0 + t_{\mathrm{TO}} \\ 0 & \text{otherwise} \end{cases}.$$

   Accordingly, for each required resource an average allocation timeout $\frac{t_{\mathrm{TO}}}{2n}$ can be accepted. Thus, whenever a task received a hint, it checked if its own remaining timeout $t_{\mathrm{remain}} := t_0 + t_{\mathrm{TO}} - t_{\mathrm{now}}$ was sufficient to allocate (in average case) its still required resources $R'$ plus the one which would be released when accepting the hint. To also resolve deadlocks we used the behavior function

   $$c(t_{\mathrm{TO}}, t_{\mathrm{remain}}, R', n, \mathrm{deadlock}) := \begin{cases} 1 & \text{if } \left(t_{\mathrm{remain}} > (|R'|+1) \cdot \frac{t_{\mathrm{TO}}}{2n}\right) \\ & \text{ or deadlock} == true \\ 0 & \text{otherwise} \end{cases} \tag{6.13}$$

   to decide between following (1) or ignoring (0) the hint. Figure 6.13e shows the corresponding graphs of $u$ and $c$ as functions of $t_{\mathrm{remain}}$. Still, $\varphi = 1$ was used, and further hints during the reallocation attempts were considered in the same way.

[31] We did also try randomized allocation/deallocation orders for each lunch cycle, but did not notice considerable changes in the test bed results.

In summary, we implemented each philosopher to potentially set back its entire meal for other more important tasks. But as soon as it has started lunch, it won't stop for anybody else. This behavior is similar to many real applications: A complex process might be deferred in time for the benefit of a more important task, but when in progress once, it is not aborted.

For each of the three presented alternatives, we inspected all 48 test bed setups from the following configuration space:

- Philosopher tasks: $m = c^n \in \{4^1, 9^1, 16^1, 2^2, 3^2, 4^2, 2^3, 3^3\}$
  $\Rightarrow$ Resources: $|R| \in \{4, 9, 16, 8, 18, 32, 24, 81\}$
- Resource allocation timeouts [ms]: $t_{\text{TO}} \in \{500, 1000, \infty\}$
- Eat and think duration [ms]: $t_E, t_T \in \{500, 1000\}$

**Test bed analysis.** Essentially, the observed basic characteristics were the same for any value of $m$. Hence, we'll just present a small but representative selection for $m = 4^2 = 16$ philosophers in a 2D setup (→ Figure 6.13b).

Obviously, a thinking philosopher allows each of its neighbors to eat. Thus, $\chi := \frac{t_E}{t_T}$ is an indicator for the average system load. For $\chi = 1$, the tasks might perfectly interleave their eat/think processes if this is allowed by the resource assignment policy (→ Table 6.1). However, due to some overhead (e.g. caused by context switches and the resource manager[32]), this will never be visible in a real run. Instead, the whole system already faces a slight overload condition, then. This overload even increases with $\chi > 1$ and turns into underload for $\chi < 1$.

As first metric for the achieved performance, we counted the number of each philosopher's successful lunch cycles $l$ in relation to the achievable maximum $l_{\max} = \frac{t_{\text{testbench}}}{t_A + t_E + t_T}$ with $t_A = 0$. Second, we considered the average allocation delay $t_{A,\text{av}}$ in relation to its maximum $t_{\text{TO}}$. As third metric we counted the number of persistent and temporary deadlock situations during each run. The presented results were averaged over 10 runs à 20 min. Figures 6.14 and 6.15 show the results for philosophers with increasing base priorities $P_{P_i} = 1 + i$ and identical values for $t_{\text{TO}} = t_E = t_T = 500$ ms. Since $\chi = \frac{t_E}{t_T} = 1$, a lunch count close to 100% might be possible[33].

*PIP.* Again, we start our analysis with the PIP based system: Using the pure priority inheritance protocol without DynamicHinting already supports the increasing task priorities partially as expected (→ Figure 6.14a). However, the values exhibit a clear jitter, and the average lunch count over all tasks settled at 47.4%. The jitter is even worse for the allocation delay $t_{A,\text{av}}$ in Figure 6.14b (the best case $t_{A,\text{bc}} = 4.0\%$ is only achieved if no blocking occurs while just the task load and allocation overhead is considered). In almost all of our setups (with $\chi \geq 0.7$) this phenomenon occurred when using PIP only. Obviously, the high variance arises from the tasks' missing knowledge about each other's requirements. The same is true for the system-wide temporary deadlock count[34] which reached an average of ≈ 161 per minute.

[32] See Table 4.2[p61] for some *Smart*OS timings.

[33] For a simplified comparison, both Figures 6.15 (PCP) and 6.14 (PIP) are based on the same test bed configurations.

[34] We received temporary deadlocks since these were resolved implicitly as soon as the first involved task reached its timeout $t_{\text{TO}}$.

Using collaborative resource sharing by means of DynamicHinting instantly improved all results while obeying the philosophers' base priorities much better. Of course, when following *each* hint, deadlocks are obviously avoided entirely (→ Section 6.5.4). Beyond, and in spite of the philosophers' generous behavior, the average lunch count increased to 79.5% at significantly less jitter. This is also true for the allocation delays $t_{A,\text{av}}$ which are more stable around 16.2% of $t_{\text{TO}}$ now.

Finally, by also applying the TUF described above, we observed additional improvements in most setups. Now, the philosophers did only collaborate if they could afford it. Let's consider the consequences: Along with falling priority, tasks tend to receive more hints and less CPU time. Thus, they also tend to get ever closer to their allocation timeout $t_{\text{TO}}$ and behave more egoistic when short in time. This results in a slight reduction of lunch counts for high priority philosophers but significantly increases the lunch count for the lower prioritized ones. Due to the selective collaboration, the number of temporary deadlock situations did also rise again: In our test bed we counted ≈ 2 temporary deadlocks per minute indeed, but nevertheless the overall lunch count further improved to 85.2% of the potential maximum. The allocation delays $t_{A,\text{av}}$ were also reduced further and stabilized even better around 12.4% of $t_{\text{TO}}$.

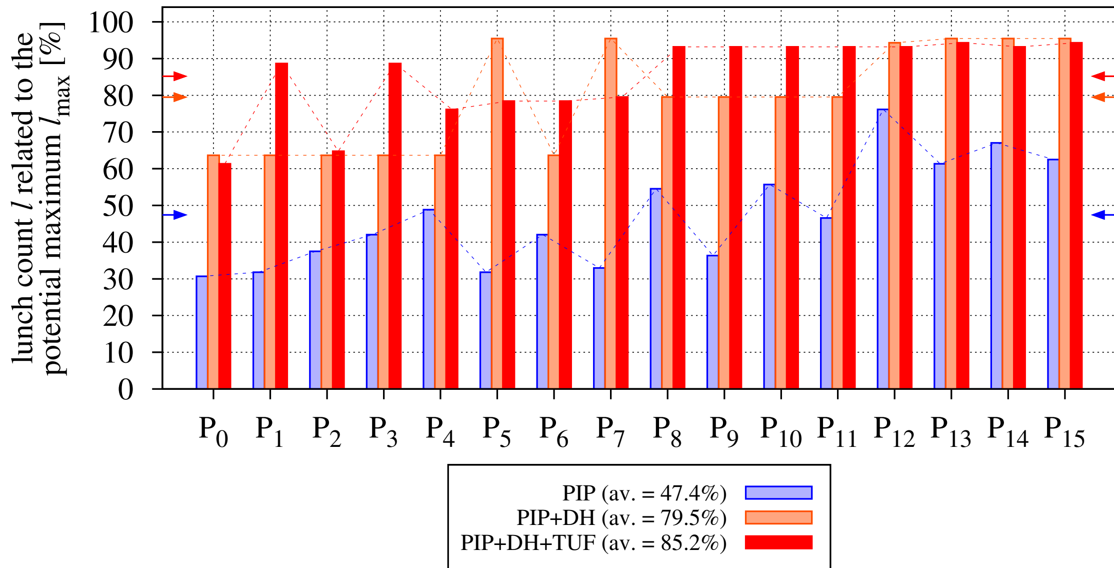
Unfortunately, for massive underload/overload setups (i.e. $\chi < 0.7$ or $\chi > 1.3$), the time-utility-function achieved only slight improvements for the lunch count compared to the PIP+EW method. Then, the load was either manageable anyway or it was simply too extreme. However, in any setup, DynamicHinting was always significantly better than the pure priority inheritance protocol. Especially when using $t_{\text{TO}} = \infty$, the pure PIP always got stuck in persistent deadlocks while our approach recovered reliably and still achieved good results[35].

*PCP.* Though extending PIP yielded much better results, we'll finally address and compare the results from the PCP based system: The already mentioned problems from Section 6.3.2 are particularly obvious within this test bed. Due to avoidance related rejection, PCP's conservative policy often lead to the unnecessary denial of free resources. In fact, this caused PCP to only serve the highest priority tasks which mutually pass their resource among each other, then.

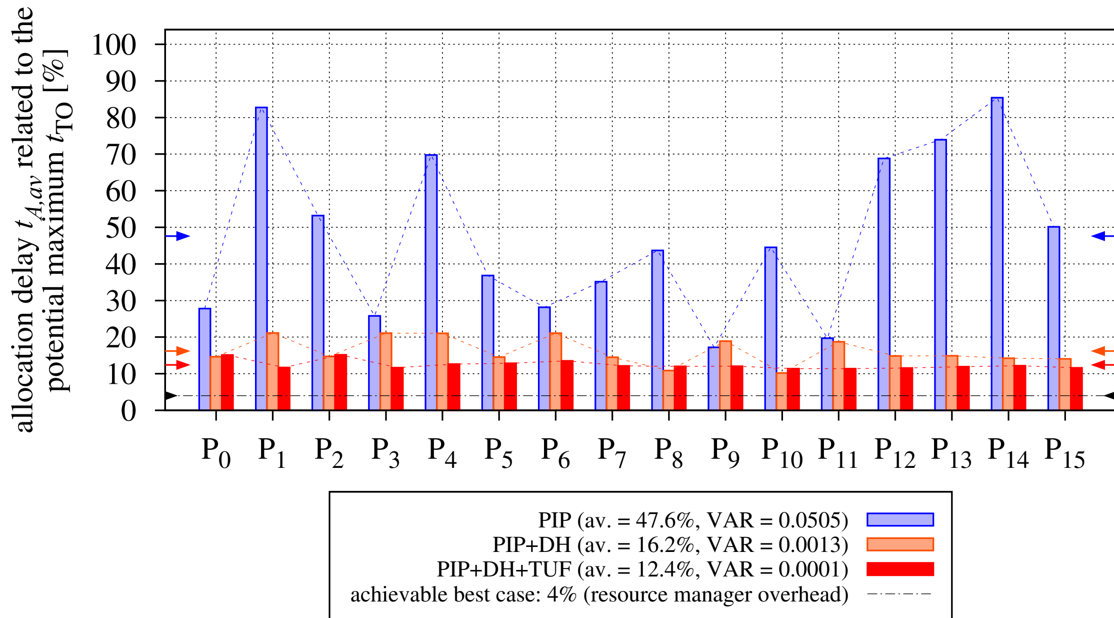
When choosing $t_T = \lambda \cdot t_E$ and thus $\chi = \frac{1}{\lambda}$, each thinking philosopher allows $\lceil \lambda \rceil$ others to start their lunch. As expected, exactly the $\lceil \lambda \rceil + 1$ tasks with highest priority will settle and eat alternately, leading to the final starvation of the others. Thus, for $\lambda = 1$ in Figure 6.15a, only the two most important philosophers were allowed to eat. Regarding the allocation delay, this value is minimal for perfect task interleaving, i.e. if $\lambda \in \mathbb{N}_0$. Indeed, Figure 6.15b shows small values close to the PIP version, but only for non-starved tasks! Yet, these are also not optimal due to the OS overhead and the high task load in general.

*Conditioning PIP and PCP.* To also allow lunch for *all* $m = 16$ philosophers under PCP, we had to choose $t_T = (m-1) \cdot t_E$, i.e. a load of just $\chi = \frac{1}{15} = 6.7\%$. As Figure 6.16a shows, this resulted in an average lunch count of $l_{\text{av}} \approx 96.3\%$, then. In comparison, to also achieve $l_{\text{av}} \gtrapprox 96.3\%$ under PIP, it was sufficient to reduce the initial load from 100% to $\chi = 66.7\%$. Then, the average allocation delay $t_{A,\text{av}}$ even dropped to stable $4.8\% \approx t_{A,\text{bc}}$ under PIP compared to unstable $85.3\% \gg t_{A,\text{bc}}$ under PCP (→ Figure 6.16b).

[35] For $t_{\text{TO}} = \infty$, the behavior function from Eq. (6.13) considered deadlock situations only.

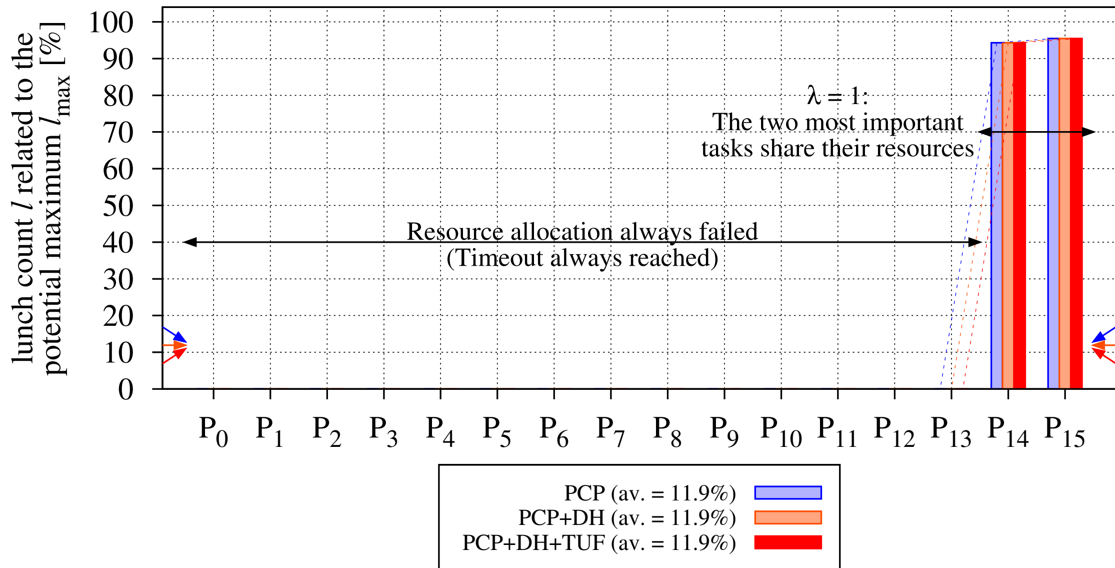



**(a)** Lunch Count $l$ related to the potential Maximum $l_{max}$

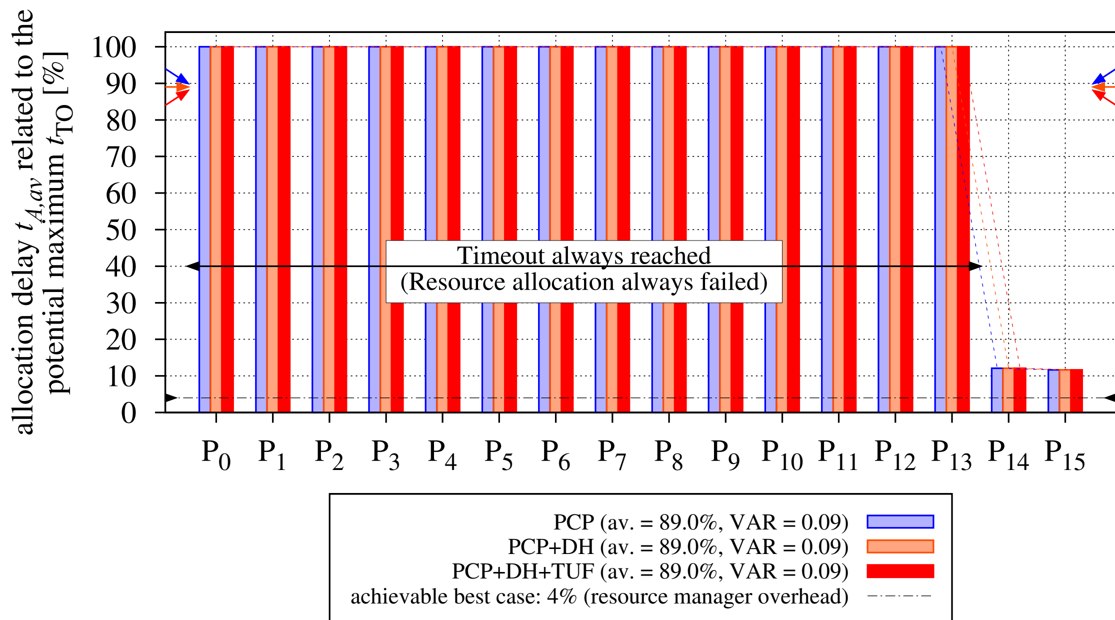



**(b)** Allocation delay $t_{A,av}$ related to the potential Maximum $t_{TO}$

**Figure 6.14.:** The dining philosophers test bed under PIP (2D, 4×4 tasks)
Configuration: $t_{TO} = t_E = t_T = 500$ ms, $\chi = 1$
(averages are indicated by the corresponding arrows)

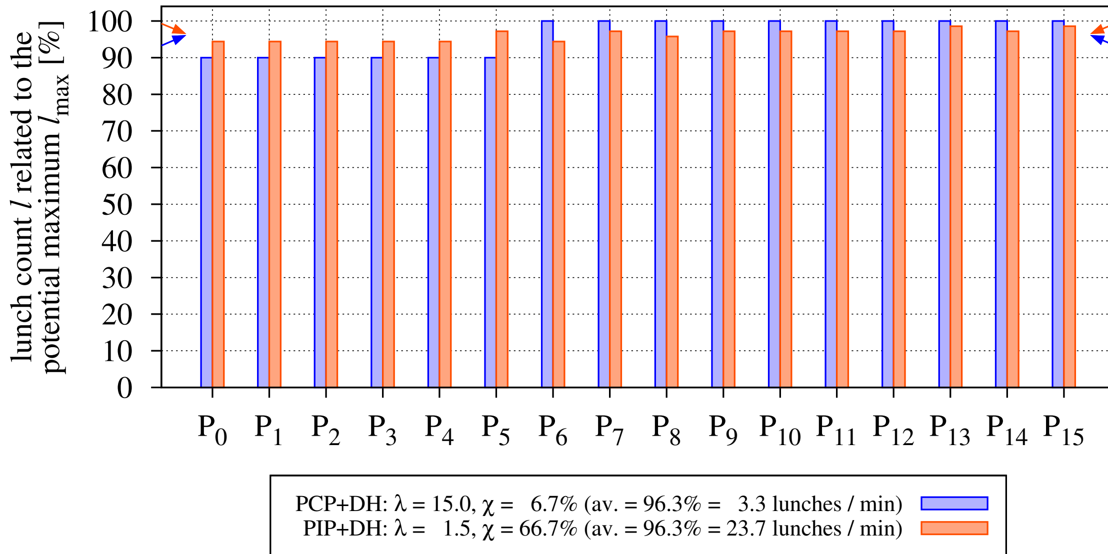



**(a)** Lunch Count $l$ related to the potential Maximum $l_{max}$

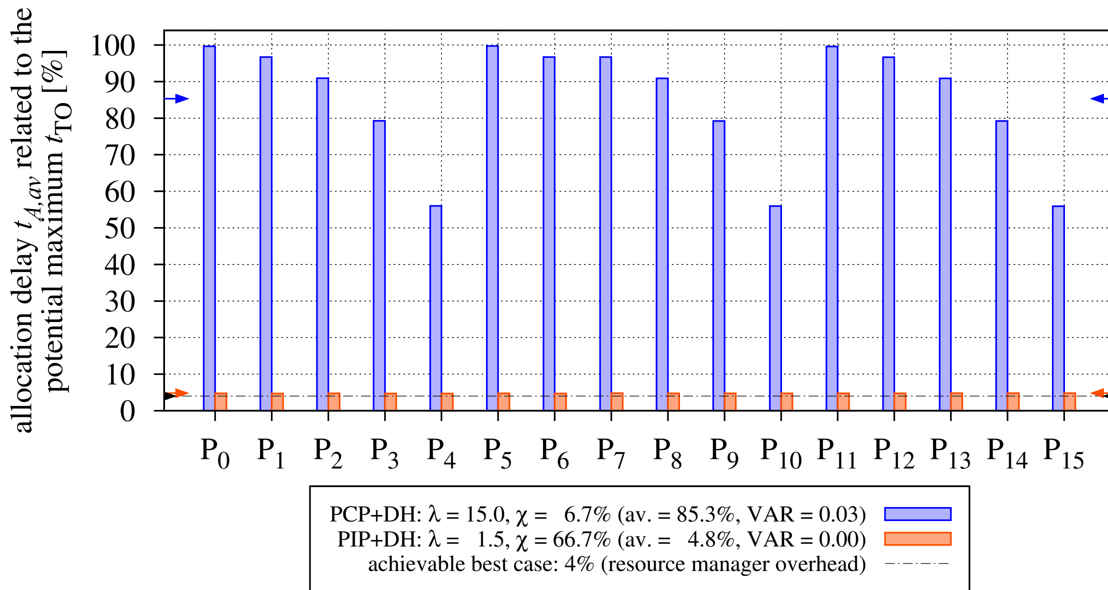



**(b)** Allocation delay $t_{A,\mathrm{av}}$ related to the potential Maximum $t_{\mathrm{TO}}$

**Figure 6.15.:** The dining philosophers test bed under PCP (2D, 4×4 tasks)
Configuration: $t_{\mathrm{TO}} = t_E = t_T = 500$ ms, $\chi = 1$
(averages are indicated by the corresponding arrows)




**(a)** Lunch Count $l$ related to the potential Maximum $l_{max}$




**(b)** Allocation delay $t_{A,\text{av}}$ related to the potential Maximum $t_{\text{TO}}$

**Figure 6.16.:** The dining philosophers test bed: Conditioning PIP and PCP (2D, 4×4 tasks)
Configuration: $t_{\text{TO}} = t_E = 500\,\text{ms}$, $t_T = \lambda \cdot t_E$
(averages are indicated by the corresponding arrows)

The direct comparison of PIP and PCP shows two issues: First, compared to the stream test from Section 6.7.2, the usage of DynamicHinting under PCP provides less advantage for the philosopher test. Since PCP is inherently very restrictive to prevent deadlocks, and prophylactically denies low priority tasks access to requested resources, these get starved more often, and consequently start less requests. Thus, hints are also rarely generated and the advantage of our concept vanishes. Second, PIP avoids performance reducing denials due to its generous policy, and thus serves low priority tasks more often. Though resulting problems like chained blocking or deadlocks are serious then, these produce hints when necessary; these are handled and resolved excellently by DynamicHinting at runtime.

**Summary.** As our test bed showed, choosing PIP as underlying protocol for our novel approach allows to 'combine' the advantages of PIP and PCP dynamically: Tasks achieve both a high progress rate and short resource allocation delays which are strongly related to their relative priorities. In summary, the introduced reflexion and collaboration concept always resulted in significantly better results compared to the pure classic techniques.

## 6.8. Conclusion and Outlook

In this chapter, we introduced the novel DynamicHinting approach for collaborative resource sharing among preemptive tasks in reactive systems. We showed that the underlying reflexion paradigm – preferably in combination with the priority inheritance protocol – can help to improve and stabilize the overall system performance. Therefore, our concept helps to reduce resource allocation delays and to recover from deadlocks at runtime. In particular, the individual task base priorities – as defined by the developer – are considered carefully to keep each task's performance and reactivity close to its intended relevance. Through DynamicHinting operating system kernels become able to put a reasonable part of the resource management efforts into the context of the affected tasks. Since these have the most complete knowledge for handling their assigned resources, this can be done more effectively, and the kernel is partially relieved from resolving emerging conflicts. E.g. the memory overhead is reduced since the kernel would have to store a standardized set of information on resource dependencies, while the tasks can store an optimal set of information (or do so anyway). In fact, in order to allow blocking tasks to be scheduled and collaborate adequately, the dynamic adaptation of task priorities remains to be done for the kernel or resource manager, respectively.

The basic idea behind DynamicHinting is to analyze emerging task-resource conflicts at runtime and to provide blocking tasks with information about how they can improve the reactivity and progress of more relevant tasks. Therefore, and depending on their current state (running, ready, waiting), we also introduced three central techniques for passing hints to them when required (→ Section 6.5.4). Thereby, our reflective approach allows each task to dynamically decide between collaborative or egoistic behavior with respect to its current conditions and the other tasks' requirements. By following the hints from the resource manager, tasks can implicitly collaborate without explicit knowledge of each other. However, our approach can initially not guarantee any time limits since these highly depend on the behavior of the involved tasks. Yet,

even if used sparsely, our approach does not modify the policy or reduce the performance of the applied resource management protocol. Thus it performs at least equal, and in most cases even significantly better, when compared to non-collaborative operation.

The test beds and the integration of all presented concepts into the preemptive embedded operating system *Smart*OS showed, that the effective use of prioritized tasks for creating reactive systems is quite possible on resource constrained and computationally weak devices like sensor nodes. Even time-critical resource sharing becomes feasible. Of course, a well-thought application design still remains elementary, but compositional software development is already facilitated through reflective on-demand resource handover. Using the *Smart*OS exception concept further simplifies the application of DynamicHinting. In general, our approach is not necessarily limited to networked sensor/actuator systems but may also extend other (embedded) systems.

Concerning real-world applications, we'll see DynamicHinting applied within the WSAN based indoor localization and vehicle steering system SNoW Bat (→ Chapter 10), where we obtained considerable benefits due to faster event handling and data transmissions. In this context, we applied our approach for collaborative memory management in open real-time systems: In Chapter 7 we'll present a framework for task-controlled heap reorganization in case of memory shortages. This avoids brute force memory withdrawals, data loss, and critical system states, while still considering task priorities and allocation deadlines.

Regarding further research, we are working on more sophisticated concepts for adjusting the acceptance of hints to the task and system situation. In particular, we see improvements concerning the hint selection itself and the application of TUFs by considering even more application-specific factors. For example, this might also allow us to relate the tight and varying energy reserves of autarkically operating wireless sensor/actuator systems to their reactivity requirements. Also, we plan to evaluate the use of DynamicHinting for remote resource management in distributed [42] and multi-core systems [90, 222], where blocking may induce hints between the subsystems. When considering database systems or software transactional memory (STM) techniques [103], an improvement might also emerge from controlled on-demand interruptions of interfering operations instead of the common but quite problematic aborts and restarts. Finally, another more general focus is the application of software model checking [10, 11, 264] for validating (distributed) system designs: For example, this technique can help to identify potential deadlocks or timing violations, and inherently also indicates code positions and system states where our novel DynamicHinting approach can be used to solve these problems dynamically at runtime.

# 7. Collaborative Memory Management for Reactive Sensor/Actuator Systems


**Abstract**

Besides the CPU, memory is a key resource for any computer system. However, its temporal sharing is hardly used under real-time conditions since unpredictable task execution flows may cause highly variable memory fragmentation at runtime, and consequently result in non-deterministic allocation delays and task reactivity. Moreover, finding a technique which reliably adapts to changing system demands is even harder. Regarding resource constrained embedded systems in particular, no concept offers a truly satisfying solution for handling allocation failures due to out of *continuous* memory conditions. Since such situations can often not be avoided or prevented entirely, more flexibility is required for highly integrated and autonomously operating devices within dynamic environments, and leaves runtime memory management as a central subject within current and next-generation sensor/actuator network research.

In this chapter we'll discuss various related problems, and recommend the *collaborative* sharing of dynamically allocated memory blocks at runtime. In the tradition of DynamicHinting from Chapter 6, our novel *CoMem* approach maintains high task reactivity in severely memory-constrained but time-critical systems, and even supports contracts for hard real-time specifications. With respect to task priorities and the typically limited performance of sensor nodes, it further facilitates compositional software design by providing independently developed tasks with runtime information for reflective memory sharing. Therefore CoMem creates a bidirectional communication link between tasks and the OS memory manager component, and triggers the on-demand but task-controlled heap reorganization in case of memory shortages. Yet, it always treats assigned blocks as strictly exclusive.

While CoMem requires no special hardware support like MMUs, it is not even limited to embedded systems and the SANet domain, but can be applied for concurrent task systems in general. The evaluation of CoMem under *Smart*OS will show, that our approach can achieve to keep memory allocation delays close to the allocator's achievable best case performance, and inversely proportional to the requester's task base priority.

## 7.1. Introduction

In the last chapter we introduced DynamicHinting as a novel paradigm for collaborative resource sharing among preemptive tasks. Throughout this chapter, and based on the DynamicHinting philosophy, we'll present and evaluate the also novel CoMem concept for runtime memory management under hard and soft real-time conditions. CoMem was designed to facilitate dynamic heap memory sharing among concurrently running and prioritized tasks, and therefore it also relies on the already introduced reflexion concepts from Chapter 6.5. This way, it further improves compositional software design by providing independently implemented tasks with valuable information about their mutual influences concerning one of the most important resources in each computer system: the *random access memory* (RAM).

When considering dynamic memory management, we will mainly focus on the on-demand treatment of sporadic heap memory bottlenecks during allocation attempts. Therefore, we'll use the terms *heap* and *memory* as synonyms within this chapter, and address out-of-memory situations where no continuous and sufficiently large memory block is available to serve a task's allocation request without jeopardizing the system stability. Even though the requested memory *was* available, its generous and careless assignment *could* possibly cause critical situations in the future, and the consequences are hard to predict. Though this particular problem is of vital importance for time and safety critical embedded systems, it has received little attention within the SANet community so far. Instead, it is mostly suppressed by using either static memory, or it is treated only with drastic measures like brute force memory revocation or even task termination. While the first measure interferes with the trend for executing complex software on cheap and resource constrained devices (→ Sections 1.2.1 and 6.2), the others are typically precluded for use within autonomous systems (→ Definition I.2[p4]), where no observing intelligence is available to recover from such situations[1]. In fact, the general research in the field of dynamic memory management has a long tradition, e.g. in large scale computing, multimedia systems, and multi-user environments. Hence, most programming languages and execution environments (e.g. operating systems and virtual machines) provide general purpose memory allocation algorithms – *allocators* for short – with sufficient performance for average applications[2] (e.g. Doug Lea's [170] under C). Anyhow, most traditional allocators silently assume powerful CPUs and truly large memory reserves (at least 32 Bit architectures), where shortages at runtime would be rare. For these, MMU supported virtual memory management eliminates external fragmentation, and the mere memory size makes internal fragmentation almost negligible[3]. Then, security and

[1]Though self-management and self-healing would be the relevant skills then, we better save them up for truly unforeseeable occasions, instead of relying on them for any memory request.

[2]Whatever "average" means! In fact the applied metrics are quite diverse. While average case performance is certainly a relevant factor, system designers often try to gain additional and highly task-specific advantages by using customized allocators to tune heap management beyond out-of-the-box techniques. Nevertheless, Berger et al. [44] found out that most applications should stick to just a few state of the art algorithms rather than using their own techniques. For us, an allocator is sufficiently good, if it reliably serves any application task with respect to its individual base priority, and if it provides a spatial and temporal allocation guarantee for hard real-time tasks and requests.

[3]For example, under Unix `sbrk` can be invoked at any time to increment the calling program's data space.

scalability aspects remain the main focus.

In contrast, embedded (SANet) applications involve entirely different demands: In this context, the event-driven character of many sensor/actuator systems (→ 3.3.2) can rapidly lead to significant memory dynamics. As summarized in Figure 7.1, the additional selection of energy efficient and resource constrained hardware collides with the desire for compositional software design. The consequences are hard to arrange with the high system reactivity which is typically required for handling sporadic events. Regarding the sparse memory of most sensor systems (up to 16 Bit architectures), most traditional allocators would reach their limits quickly and lead to weak memory versatility [69]. While fragmentation and memory shortages are just some obvious problems, emerging priority inversions among the tasks are rather hidden; though hardly considered, they may lead to severe system misbehavior and real-time violations. Under such adverse circumstances, and for compositional open systems in particular, it is quite difficult to find a strategy which boosts certain privileged tasks while not degrading others excessively. Analogous to the already mentioned design challenges for arbitrary system services from Figure 6.2[p93], it repeatedly showed to be a difficult venture to simultaneously support *conveniently usable, resource efficient, fast, safe, and secure* memory management within concurrent task systems. Without corresponding hardware support and energy reserves, which are most commonly not available on cheap and autarkic embedded devices (→ Chapter 2.1), it is a truly tough problem[4].

To face the mentioned problems in software, and to comply with a wide variety of further requirements, CoMem is initially not an allocator on its own, but provides a system extension to facilitate the on-demand memory reorganization in case of (time and space) critical memory situations. In fact, the internally applied allocation algorithm is not directly prescribed by our concept, but can be chosen almost arbitrarily as long as it respects the central CoMem philosophy and complies with the two basic preconditions DH1/DH2[p111] for using DynamicHinting at all (→ Section 6.5.3). This flexibility does not only simplify its adaptation for other storage-management systems apart from the one which we will use during our reference implementation under *Smart*OS, but the dynamic adjustment and replacement of the allocator can even be accomplished at runtime[5]. In any case we consider the memory management subsystem to be a separate component besides the operating system kernel, e.g. a library as depicted in Figure 3.1[p34]. To yet comply to the fundamental *Smart*OS philosophy, the necessary task synchronization and mutual memory access control will be accomplished via the kernel's internal resource manager as depicted in Figure 7.6. This keeps the allocated memory blocks strictly exclusive, and avoids an additional level of (address) indirection which would be required if the memory manager was allowed to shift assigned blocks without notifying the current owner explicitly (a comparable approach considering the stack space and its dynamic relocation can

[4]Of course there is the question of whether to treat the resulting problems in software, or if it would be wise to wait for suitable hardware support for embedded systems. Presumably, however, their integration will be a slow process: Firstly, any space and energy savings is initially associated with a loss of architectural properties (like MMUs) and vice versa, and secondly, even the actual memory size is hardly growing compared to non-embedded systems. While it is already in the range of several GB for desktop systems, the latest (ultra) low power MCUs are still restricted to a few KB. A significant improvement in this situation is therefore hardly to be expected in the foreseeable future.

[5]Although we won't implement this option within this work, it can certainly be utilized to e.g. tune for better performance after partial software updates and modified system specifications at runtime.

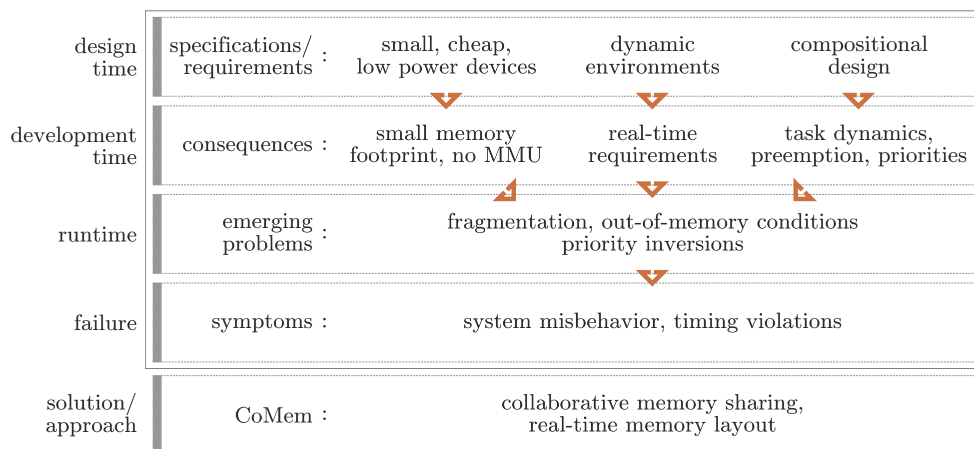



**Figure 7.1.:** Dynamic memory management: From system design to runtime failure

be found in [69]).

As we will discuss throughout the next sections, one central weakness of all existing memory management approaches is, that tasks are not aware of their (varying) impact on the remaining system, and thus cannot collaborate adequately when required in case of a critical situation. While tasks can directly interact with the memory manager via function calls, there is no communication channel in the reverse direction. Once a memory block is assigned, the memory manager has no chance to signal related problems to the owner task. In this respect, CoMem benefits from the DynamicHinting policy within software and the operating system kernel by which programs can become "self-aware", and may proactively change their impact and influence according to their own current requirements and the system's demands (→ Section 1.2.1 and Figure 1.7[p14]). Of course, there remains the question for a spatial and temporal guarantee regarding the allocation attempts: How can the memory manager ensure – under all circumstances – that a task receives its requested memory within a maybe hard deadline? As proposed in the collaboration prerequisite C5[p110], a contract based heap layout will help us to enforce the timely servicing of reactive tasks with sporadic and hard-to-predict memory requirements in open systems. Finally, a feasibility analysis allows the appropriate dimensioning of the required heap space.

During Section 7.2, we'll give a more detailed introduction on numerous problems and feature requests related to dynamic memory management within embedded real-time systems. In Section 7.3 we'll review some related techniques from existing work, before details about our novel approach will form the central Sections 7.4 and 7.5: A concrete implementation of CoMem under *Smart*OS will show that – despite of the problem's complexity – it is efficiently applicable even for low performance devices like sensor nodes. Therefore, Section 7.6 presents some application examples as well as the impact on the programming model, and some selected performance results from real-world test beds.

Regarding our final goal to implement a WSAN based localization system, CoMem will finally allow us to efficiently coordinate the heap memory usage among two very memory intensive

subsystems for ultrasound signal processing and remote updates. Although the real-time requirements of the DSP task would definitely indicate a static buffer allocation, this is simply not realizable due to insufficient hardware resources. In consequence, a dynamic memory allocation is essential, and becomes reliably feasible through CoMem.

## 7.2. Motivation and Requirements

Along with the CPU, time and memory are the central resources in any computer system[6]. While virtually any application requires some *statically* allocated memory of fixed size for code and data, most applications also make use of *automatic* and *dynamic* data structures which will commonly be placed on the so called stack or heap memory. While both memory types allow random data access, the stack is commonly filled incrementally at its current end (LIFO). Therefore, and to simplify its organization in hardware[7], it is designed as a strictly continuous memory area. Local variables, which are *automatically placed* on the stack, will implicitly be destroyed when leaving the code block in which they were created.

When considering the continuity aspect, heap memory is significantly less problematic. Though memory blocks must be allocated explicitly before use, these need not necessarily be placed adjacently[8] since the access is handled in software via address pointers to *each* assigned memory block. Furthermore, *dynamically placed* contents on the heap remain valid until released when not required any more (either intentionally by the application, or implicitly by a garbage collector as soon as the last reference has disappeared).

### 7.2.1. Existing Problems

**Fragmentation.** Of course, the mentioned flexibility on the heap is not for free. In fact, heap based memory management techniques suffer from several inherent flaws, stemming entirely from fragmentation. These problems become worse and more frequent with increasing fluctuation on the heap[9]. At runtime, they prevent the maintenance of a perfect heap memory partitioning, which satisfies all upcoming or persisting requests without impairing already established allocations. In this regard, it is worthwhile to note that fragmentation is not only a question of the current memory layout, but also of its future use. For multitasking systems in particular, it results in a severe lack of scalability due to high task dynamics and frequent (de-)allocations. This is why most concepts still limit their focus on developing an allocator, which assigns and releases the available heap space in a way to reject as few requests as possible despite of unpredictable competition for shared memory. To achieve this, the simple coalescing of neighboring free blocks, or the rather complex defragmentation of the entire heap space are just two approaches to gain larger continuous areas. While external fragmentation can also be

[6] When seen as a state machine, the CPU makes program progress possible by executing the algorithmically defined *transitions* between system *states* represented by memory contents. For synchronous architectures, the system clock is another key resource as it introduces *time* and triggers the transitions with a defined frequency.

[7] Via a dedicated stack pointer and associated atomic `push` and `pop` instructions within almost all ISAs.

[8] Although a foresighted placement can exploit the locality principle, and increases performance when using caches (which are hardly found within current sensor/actuator systems).

[9] For most allocators fragmentation cannot occur until the first memory deallocation took place.

reduced by increasing internal fragmentation[10], it is said to be in general a problem of poor allocator implementations: According to [146] most allocators would suffer from almost no fragmentation when using well-known policies like best-fit, first-fit, and next-fit for placing new blocks. However, up to now the worst case fragmentation is only known for best-fit and first-fit [251]. Though we'll also stick to a slightly modified first-fit strategy for our CoMem reference implementation, even little fragmentation is too much if it circumvents the compliance to task priorities and real-time operation at runtime.

**Resource constraints and real-time operation.** According to Wilson et al. [306], any allocator can face situations where continuous free memory is short while the total amount of free space would be sufficient to serve an allocation request. Especially for systems without MMU or virtual address space, a centralized heap reorganization by the memory manager is hard or even impossible then, since it lacks information about the actual memory usage by the current owner tasks[11].

Thus, the use of dynamic memory is largely avoided for time or safety critical systems [200]. For these, the allocator needs to be extended to support guaranteed allocations and the knowledge about its *worst case execution time* (WCET), also referred to as the *worst case allocation time* (WCAT) when seen from the application's view. Since both requirements are hard to provide in dynamic systems, memory is most commonly still allocated in a static way to avoid unforeseeable allocation failures or unbounded (de-)allocation delays at runtime. For modular systems this is done during program loading[12]; for monolithic systems it is done during compile time and linking. Since static memory as a unique resource is a well-known bin packing problem [65, 100, 245], the memory requirements are typically hard to predict by system designers, and often impossible to analyze automatically by profiling algorithms. Instead it is common practice to simply measure them based on worst case runtime profiling, and to add some extra space as tolerance. Of course, these preventative overestimations are one reason for unnecessarily large allocations of this highly valuable resource. Even worse, these allocations stay reserved throughout the whole program or system runtime. With increasing number of simultaneously running tasks, such a waste of the memory resource becomes unacceptable without losing flexibility and performance – the more so as it does not even guarantee their sufficient size.

If therefore dynamic memory sharing is not dispensable at all, real-time operating systems often provide so called *pools* of fixed-size memory blocks (e.g. for small but unified data structures like message buffers). Besides low external and high internal fragmentation, these often support a constant allocator execution time – at least in case of success. In contrast, blocks of arbitrary size commonly provide more flexibility at less internal but more external fragmentation (e.g. for dynamically sized sampling and DSP buffers). At higher management effort they might even partition the usually small heap space more efficiently – at least in theory[13]. Depending on the internal heap organization, three central techniques are commonly distinguished: Sequential fits, segregated free lists and buddy systems. However, since we primarily focus on

[10]For example, external fragmentation can even be avoided entirely by allocating blocks of fixed size only.

[11]A formalization of the resulting *fragmentation penalty* will be given Section 7.6.

[12]Which might also imply dynamic memory allocation when seen from the operating system's view.

[13]Although we aim on reactive systems, we yet support arbitrarily sized blocks for flexibility and convenience reasons.

collaborative memory reorganization in case of allocation failures, we won't go into detail about these techniques but refer to [200, 236, 306] instead. In [236] in particular, an analytical and quantitative evaluation of worst case allocation/deallocation times is presented. Based on worst case complexity analysis (pessimistic for hard real-time conditions) and real-world benchmarks (commonly less pessimistic for soft real-time conditions) they provide suggestions about which technique to use regarding the predictability aspect.

### 7.2.2. Feature Requests

According to e.g. Michael [211] and Lea [170], a good dynamic memory allocator should support and balance a number of performance relevant features. While we already mentioned some of them, there are still more aspects to remember. To further motivate the goals and design considerations for our CoMem approach, we'll give an extensive and distinguished summary of fundamental requirements for designing reactive sensor/actuator systems. With respect to the design challenges F1 – F5[p93] from Section 6.2.2, we will first introduce the general design space of dynamic memory management systems. Then, we'll divide the special feature requests into three groups: While group A contains general objectives, group B focuses on concurrent task systems as relevant for compositional software design, and group C addresses real-time requirements as relevant for reactive systems.

**The memory manager design space** comprises four main dimensions, and the need for scalability regarding any supported feature (→ Figure 6.2[p93]):

M1 RUNTIME PERFORMANCE. *Optimize the administrative overhead* to provide fast or even deterministic execution times.

  a) When calling related functions like `malloc` (allocation) and `free` (deallocation).
  b) When accessing already allocated memory blocks.

M2 RESOURCE EFFICIENCY. *Target on economic space and resource requirements*:

  a) Limit internal fragmentation, and maintain an efficient heap partitioning to also keep external fragmentation low.
  b) Optimize internal data structures and resource usage for managing memory areas. If possible, manage the entire unused system memory dynamically.

M3 SAFETY AND SECURITY. *Account for reliable application executions* through a well-controlled access to the system memory:

  a) Prevent tasks from inadvertently or intentionally corrupt or read foreign data.
  b) Account for properly (de-)initialized memory areas upon each (de-)allocation.

M4 USABILITY. *Provide useful functionality while avoiding trivializing assumptions.* I.e. make allocation success and task progress feasible without imposing unreasonable restrictions to the application design.

M5 SCALABILITY. *Maximize and maintain scalability* regarding the number of free or allocated blocks, and the number of concurrently running tasks.

Contributing to these general demands has always been the responsibility of the various allocators. Beyond, and depending the OS kernel's philosophy and the specifications of each individual application, various specific requirements must also be met. While some of the following feature requests F1 – F13 must be followed stringently to ensure the system integrity (marked with ■), others are less critical. Nevertheless, an effective treatment strategy is still desirable, and CoMem pays special attention to those marked with ▲:

### Group A – General objectives

F1 ■ ASYNC-SIGNAL SAFETY. *Ensure that memory management functions do not interfere with any ongoing operation when called asynchronously* (e.g. from within ISRs or injected HintHandlers).

F2 ▲ AVERAGE CASE OPTIMIZATION. *Minimize anomalies* to support good average case performance when using default settings.

F3 ▲ TUNEABILITY. *Support configuration options* to account for dynamic and task-specific requirements at runtime (e.g. on-demand and time-critical memory allocations).

F4 LOCALITY. *Maximize locality* by neighboring related blocks (e.g. from one task).

F5 PORTABILITY. *Maximize portability and compatibility* to other systems by using few but widely supported hardware and software features.

### Group B – Objectives for concurrent task systems

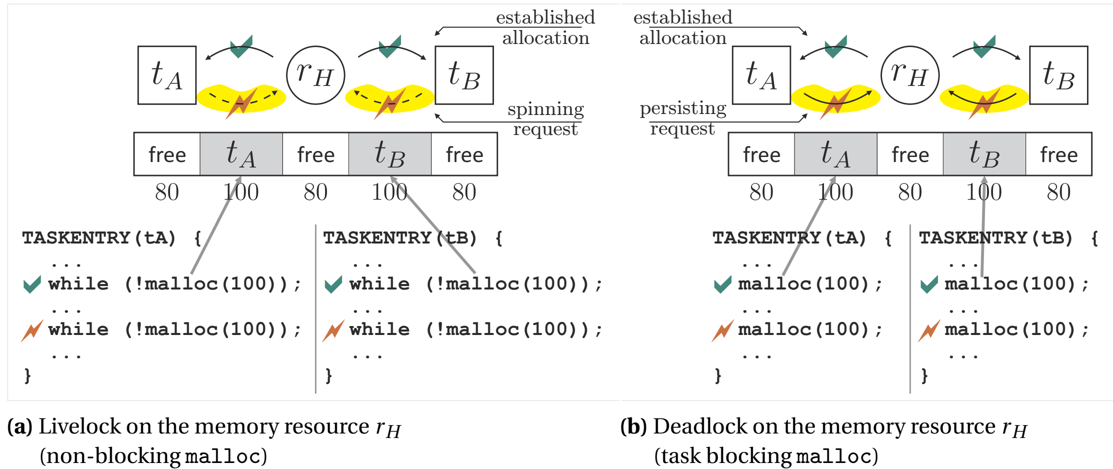
F6 ■ KILL TOLERANCE. *Avoid the final loss of memory* when terminating tasks with still allocated memory.

F7 ■ PREEMPTION TOLERANCE. *Avoid task starvation, livelocks, and high CPU load* caused by aggressively repeated calls to `malloc` in case of rejected memory requests (→ Figure 7.2a).

F8 ■ DEADLOCK TOLERANCE. *Avoid or handle task deadlock conditions:*

a) Handle *memory deadlocks* caused by tasks mutually requesting memory which is currently allocated by the other task respectively (→ Figure 7.2b).
b) Avoid *management deadlocks* caused by exclusive access to the memory manager's internal data structures.

F9 ▲ REORGANIZATION TOLERANCE. *Provide a means for controlled memory reorganization* in case of currently not grantable memory requests.




**(a)** Livelock on the memory resource $r_H$ (non-blocking `malloc`)

**(b)** Deadlock on the memory resource $r_H$ (task blocking `malloc`)

**Figure 7.2.:** Persisting memory allocation failures for various implementations of `malloc`

F10 ▲ PRIORITY REFLEXION. *Comply with task priorities* by serving high priority tasks first. Nevertheless, avoid prophylactic memory reservations or rejections leading to a potential starvation of low priority tasks[14].

### Group C – Objectives for reactive systems

F11 ▲ PRIORITY INVERSION HANDLING. *Handle situations where higher priority tasks cannot be served due to existing allocations by lower priority tasks.*

F12 ■ WORST CASE ALLOCATION TIME (WCAT). *Support real-time allocations* by providing *spatial and temporal guarantees* for time-critical tasks. Also provide an appropriate feasibility analysis at compile time or during the allocation attempts.

F13 ▲ OUT-OF-MEMORY HANDLING. *Facilitate the efficient and reliable handling of memory shortages.* With regard to the demanded system autonomy (→ Section 1.1), avoid brute force memory revocation, relocation, and task termination if this can result in critical system behavior which is hard to recover automatically.

Regarding the just requested feature requests we will show that a collaborative point of view on the varying system and memory situations can further improve existing techniques significantly.

However, before presenting our approaches for solving the related problems, we'll first discuss some available concepts for dynamic memory management in the domain of embedded systems and sensor networks in particular.

## 7.3. Related Work and State of the Art

Meanwhile, dynamic memory management is subject to intense research efforts for over thirty years [40]. Forced by the proceeding integration density of today's embedded systems, dynamic

[14] Regarding general resources, this would be comparable to the conservative PCP or HLP policy from Section 6.3.2.

| | simulation/ comparison | analysis/ technique | real-world test beds | focus on WSAN OS |
|---|---|---|---|---|
| stack specific | | | | [69] [46] [319] |
| application heap | [44] [245] | [211] [200] [111] [157] | [199] | [26] [3] [318] [120] [212] [165] [61] [317] |
| application heap / OS heap | [278] | | [43] | [120] |
| OS heap | | | | [85] |
| OS heap / general considerations | | | | [163] [72] |
| general considerations | | [306] | | |

**Figure 7.3.:** Related work for dynamic memory management systems
Red: SANet related, Blue: general memory management

memory management also plays an increasing role in the design and development of modular embedded software. Figure 7.3 presents an overview on some relevant work in this area. Indeed, it is a rather new research area for resource constrained and time-critical embedded systems, and was hardly considered for sensor/actuator networks before this work. In 2004 Masmano et al. [199] presented one of the first allocators for real-time systems at all, and in 2009 Ripoll et al. [245] developed a framework for providing both a temporal and a spatial guarantee for allocations; their approach will be discussed later within this section.

To get a better insight into existing work, we'll take a short look at the memory management concepts of some operating systems with real-time or embedded systems background. However, in this work we intentionally focus on conventional heap space management only: For energy, cost, and size reasons of modern sensor nodes both the data and program code is typically stored in the on-chip RAM and ROM, and won't be loaded or stored in an external memory. Thus, *overlaying* and *scratchpad* techniques [81] are largely irrelevant for the SANet domain.

### 7.3.1. Dynamic Memory Management in general Embedded Systems

- *VxWorks* [307] uses a best-fit memory allocator for reduced fragmentation and short but indeterministic allocation delays. Since there is no support for demand paging, at least the

memory access itself is always deterministic (→ M1b). To gain additional performance for real-time tasks, these share a separate memory space which is protected in case a MMU is available (→ M3a).

- *ChorusOS* [3] introduces a user level virtual memory approach. So called *pagers* allow the developer to provide code for an application-specific memory management policy. Although this also allows to reflect some individual needs at compile time (→ F3), the developer is still responsible for considering *all* potential system situations. Additionally, it might be hard to reflect inherent OS concepts, like task priorities and timing constraints, into such pagers.

- *FreeBSD* [235] and the GNU *libC* use ptmalloc [111] which is based on Doug Lea's malloc, and proved to be a good general purpose allocator for allocation-intensive programs (→ F2). Internally, it maintains several memory *arenas* (i.e. dynamically created heap areas) to reduce the chance for allocation conflicts (→ M1a). New arenas are created on-demand if a request finds all existing arenas locked.

- *Hoard* [43] is optimized for multi processor systems and manages one heap per core, as well as one global heap. These heaps are partitioned into *superblocks*, which, in turn, contain blocks of fixed size. A lock-free modification of Hoard is presented in [211]. By using atomic CPU instructions (e.g. `CAS`) to repeatedly check and change system variables in a non-interruptible way, functions like `malloc` and `free` were split into small atomic steps. Regarding the implementation this provides features like immunity to deadlocks (→ F8b), async-signal safety (→ F1), and preemption tolerance (→ F7). However, this strategy also requires adequate instruction sets which are often not available on typical sensor node MCUs, and causes considerable CPU load[15] while it treats the shared memory in a purely cooperative manner.

While from these representatives only VxWorks, and ChorusOS claim to support hard real-time operation – in case their allocators are used properly(!) – no operating system provides any mean for handling sporadic memory shortage without swapping (→ F13). In particular, since signaling mechanisms from the memory manager towards spurious tasks are missing entirely, not serviceable tasks must consequently be rejected independently from their relevance to the system. Even if task priorities are supported by the scheduler, they are not reflected for the memory allocations (→ F10, F11)[16]. Considering the deadlock tolerance for memory blocks (→ F8a) none of the mentioned operating systems treats allocated blocks as ordinary system resources. Thus, memory deadlocks are neither detected nor explicitly handled at runtime.

[15] Many do-while loops might need to execute quite often since the finalizing `CAS` operation can not succeed and commit the loop's modifications until the whole iteration has completed. Emulating `CAS`-like instructions, in case they are not supported natively, is yet another issue (→ Listing 4.2[p53]).

[16] At best, there is sometimes a regulating but non-binding communication in the other direction: Under Unix for example, an application can advise the kernel about how to handle paging within a specified address range (commonly via `madvise(...)`). Though the kernel is always free to ignore the advice, it *could* choose appropriate read-ahead and caching techniques to speed up the application.

### 7.3.2. Dynamic Memory Management in Sensor/Actuator Systems

When considering the WSAN operating systems from Section 3.3.2, their capabilities are even more reduced, and only few natively support dynamic memory for arbitrary use by application tasks. Most of them simply justify this with the lack of necessity in the field of sensor networks. Others, like SOS [120], make intense use of dynamic memory, and consider it as a key feature to flexibility and modularity in current and future sensor network applications. To give an adequate overview for a later comparison, we'll take a brief look on the most popular WSAN/SANet operating systems, but mainly focus on those with preemptive multitasking or multi-threading capabilities:

- *TinyOS* [291] offers no native support for dynamic memory within its kernel. In fact, it encourages purely static memory allocations within all components [174].

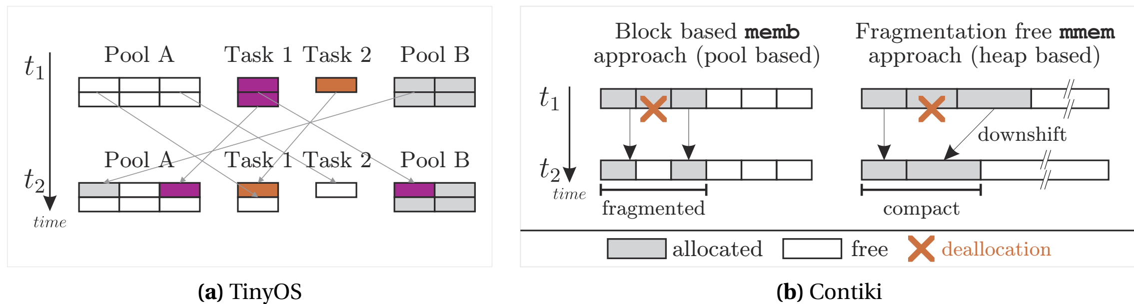
  However, since TinyOS 2.x, the PoolC data structure implements a semi-dynamic pool approach in which a fixed number of blocks can be statically assigned to each component [157]. At runtime, components can release their blocks to arbitrary pools, and reallocate blocks as long as their initial number is not exceeded (→ Figure 7.4a). Though this policy prevents out-of-memory and memory deadlock conditions implicitly (→ F8a, F13), and even data handover becomes possible, it also requires the worst case memory requirements to be known at development time and stay reserved during the whole system runtime. Furthermore, the block sizes are fixed, and, with decreasing size or increasing granularity, it becomes unlikely to receive a continuous number of blocks for memory intensive operations.

  In [72] Cooprider et al. propose a technique to provide memory safety for TinyOS (→ M3a). By using annotated nesC code and the Deputy [71] source-to-source compiler, they enrich application code at compile time to support appropriate runtime checks. However, these are not compatible with dynamic memory management approaches since block owners are not resolved at runtime, and any access to any block would always appear to be valid – even if it is not allocated at all.

- *Contiki* [85] uses dynamic memory for storing variables of dynamically loaded modules. While this option is not available for arbitrary use by processes or protothreads [87], two types of dynamic memory support are provided by libraries (→ Figure 7.4b):

  The block based `memb` approach requires static declarations at compile time: In fact, any number of uniquely named block sets with arbitrary block size and block count can be declared within the program code. At runtime, processes can request and release blocks by specifying the set name – block handovers are not intended. Since the block sizes are fixed, at least one block set is needed for data types of different size. This increases external fragmentation among the sets and requires the tasks to know the set names, which can possibly be understood as a contradiction Contiki's modularity concept.

  In contrast, the managed memory allocator `mmem` features a "fragmentation free" approach for allocations of arbitrary size. While only one system-wide memory area (default size:




**(a)** TinyOS

**(b)** Contiki

**Figure 7.4.:** Dynamic memory management examples

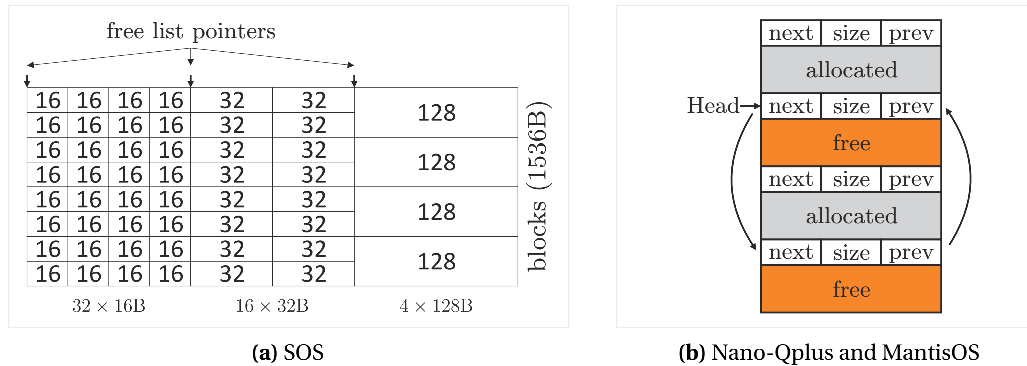
4096 B) is maintained, deallocations are always followed by a rearrangement of all still allocated blocks at higher addresses. Though this compaction eliminates memory holes, it produces high CPU load in case of frequent (de-)allocations and keeps internal data structures locked until finished. In addition, the load does not only depend on the amount of memory to be down-shifted, it is also exceptionally annoying if a rather "unimportant" task[17] releases a block at a low address. Finally, an additional level of base pointer indirection is required, since even already allocated blocks might get shifted asynchronously when seen from an owner's view. The impact on dependent peripherals (like e.g. DMA controllers) using such dynamic blocks is not considered at all.

- *SOS* [120] makes intense use of dynamic memory and uses a block based first-fit scheme to store module variables and messages (→ Figure 7.5a). The fixed and power-of-two sized chunks are used to minimize the allocation overhead to $O(1)$, but are likely to cause quite high internal fragmentation (especially in the context of severely memory-constrained sensor nodes). However, data transfer between tasks is supported by simply changing the block owner. Finally, allocation failures immediately trigger a so called kernel panic mode and stop the entire system – a behavior which is quite critical for inevitably self-managed systems like sensor nodes, as we already mentioned in Section 7.2. While data protection is initially not available, Kumar et al. [163] proposed a software based memory protection scheme: By taking a sandbox approach, they perform write address checks at runtime to protect the kernel space and the system stack area. Timing benchmarks can be found in [120].

- *Nano-Qplus* basically also relies on static memory allocation. However, in [317] and [212], a combination of sequential fits, segregated free lists, and the buddy system is proposed for dynamic memory usage within tasks (→ Figure 7.5b). In addition, dynamic task stack management was initially tested within Sesame [318] which (de-)allocates stack space on a per-function basis, and further extended by Sesame-P [319] which uses a pool based stack management with on-demand resizing.

- *MantisOS* [46] supports a best-fit approach to allocate blocks of arbitrary size for thread

[17] though task priorities are not explicitly supported




**(a)** SOS **(b)** Nano-Qplus and MantisOS

**Figure 7.5.:** Pool layouts for dynamic memory management

stacks and the networking subsystem only. Therefore, it uses a double-linked list to manage the free space (→ Figure 7.5b). For use within task code, it relies on pure static memory allocation.

- *SensorOS* [165] incorporates dynamic memory management for message envelopes, and for temporary buffers which are occasionally needed by tasks. A thread allocates and releases previously reserved blocks from a memory pool. While a binary buddy algorithm allows the allocation of different sized blocks, a more lightweight alternative uses fixed-size blocks, and is mainly targeted to message envelopes.

- *SenSmart* [69] supports a technique to adapt the stack sizes within multitasking environments at runtime and with regard to each task's current demands. During linking, some management code is added to memory access instructions and allows runtime memory protection, address translation, and stack relocation in case of shortages. Yet, there is no guarantee for a successful expansion, and tasks simply get killed when the physical memory is exhausted. Furthermore, this concept is neither reorganization tolerant (→ F9) nor does it consider execution speed (→ M1) or even real-time operation (→ F12).

- In [245] Ripoll et al. propose a contract based scheme to support dynamic memory management in real-time systems. For each block, the contract is proposed by the requesting task through the specification of the *allocation importance* $R_{\text{imp}}$ and a minimum *stability time* $R_{\text{stab}}$. If a sufficient amount of free space is found, the request is granted immediately, otherwise unused memory is reclaimed from allocated blocks which exceeded their stability time; lower priority blocks will be preferred. However, there are several problems with this approach: While the owners won't get informed about the revocation (→ F9), such a stability time is almost impossible to estimate in multitasking environments. In fact, a task might get slowed down by others and still uses its memory after it has been withdrawn. Furthermore, block priorities are independent from task priorities and might try to manage the memory conversely to the CPU scheduler. Worst case allocation times cannot be guaranteed at all (→ F12). Finally, `malloc` and `free` are implemented as so called *transactions*, and a scratchpad is used for each heap modification. While this would

allow on-demand heap compaction, it is quite expensive in time and resources when considering typical embedded sensor/actuator systems.

- *SDMA* [278] uses simulation for comparing several candidates by various metrics, and to find a suitable allocator for concrete WSN applications. Finally, the best performing allocator is selected and linked statically. However, static allocator selections will hardly be optimal in complex next-generation SANet applications since the choice cannot be easily adapted in case of dynamic changes to the application code [85] (e.g. after fine-grained software updates).

Besides SensorOS, no presented SANet OS provides the arbitrary use of dynamic memory for tasks. In particular, none provides any mean for dynamic memory management in case of time-critical sporadic requests, or priority inversions when low priority tasks block higher priority tasks by any memory allocation. To the best of our knowledge, no OS exists to support on-demand memory reorganization in small embedded systems without brute force methods like (energy intensive) swapping, memory revocation, or task termination with possibly critical side effects. This is exactly where CoMem applies.

## 7.4. The CoMem Approach

As we have already indicated in Chapter 6, reflexion based task collaboration is a mighty strategy to share general resources on-demand and "upwards" along with the task priorities. In this chapter, we'll adapt the strengths and benefits for the special case of dynamic memory management where tasks may allocate any number of blocks with arbitrary size. By addressing various specific problems and the feature requests from Section 7.2, we'll motivate and discuss the design and implementation decisions behind our novel concept. Beyond, CoMem is in general an example for the integration of the DynamicHinting programming paradigm into system services which require priority-based task synchronization due to their dynamic dependencies on exclusively shared resources.

### 7.4.1. Basic Design Criterions and Application Example

We want to emphasize once more that CoMem was initially intended for systems with severely limited memory and no hardware support for its protection or virtualization. Keeping the typical specifications of today's sensor/actuator systems in mind (→ Chapter 2), we yet seek to respect the dynamics of concurrently running tasks without generating significant overhead in terms of energy consumption (for e.g. data swapping), CPU load (for e.g. runtime address translations), and memory (for e.g. scratchpad operations). Nevertheless, for the reasons given in Section 1.2 and 4.2, we also wanted to provide a means for bounding the WCAT under real-time conditions.

**Internal component structure.** As central kernel components, resource managers are deeply involved into the scheduling policy (→ Figure 3.1[p34]): They are responsible for monitoring and enforcing the compliance of all application modules to the system specifications – either strictly

or at best effort. We take advantage of this tight interaction, and treat each individual memory block as an ordinary (*Smart*OS) system resource, just as described in Section 4.3.7. As central idea, we'll enrich the resource manager's runtime knowledge about currently persisting memory allocations, but this time we let the CoMem subsystem analyze emerging task/memory conflicts based on certain contracts between the application tasks and the memory manager.

Therefore we split the CoMem memory management component into two parts named *allocator* and *collaborator*: The heap memory itself is managed by the *allocator* which is also the layer in charge of negotiating new requests regarding their size. The contracts however will be proposed by the application tasks, and checked for conformity by the *collaborator* according to the current memory state and layout. While the allocator is reliable for the actual heap partitioning, and can be selected almost arbitrarily with some interface adaptations, the collaborator acts as an intermediate layer towards the application, offers a unified API, and applies DynamicHinting to selectively provide spurious tasks with information about how they can help to improve the allocation progress of more relevant tasks. In this context, the combination with blocking based priority inheritance techniques already proved to reliably enhance and stabilize the overall and average system performance. Since memory allocations are commonly long-termed, i.e. tasks often suspend themselves while holding blocks, we'll stick to the priority inheritance protocol (PIP), since it avoids inheritance related starvation and avoidance related rejection towards low priority tasks when these also request resources concurrently (→ Table 6.1[p99]). In general, the combination of DynamicHinting and PIP reduces resource allocation delays by resolving (bounded and unbounded) priority inversions, and even provides an option to recover from deadlock situations as required by F8[p146]. Further reasons for this decision will be discussed in Section 7.5.2.

**Heap allocation and reorganization.** Finding a suitable block allocation and heap maintenance strategy (like e.g. free lists and defragmentation) is indeed a critical core element within all memory managers, and was already considered in many ways, e.g. in [200, 306]. Regarding the special case of on-demand heap reorganization in case of memory shortages (→ F2, F9, F13) various options exist:

1. Do nothing, but simply indicate an allocation failure to the requester.
2. Kill one or more tasks to free all their memory.
3. Withdraw or relocate certain spurious memory blocks.
4. Swap out some data, reassign the gained free space, and keep the owner task suspended until the data was made available again.
5. Ask an owner task to voluntarily release or relocate some memory.

Apart from the first option, which is the only one where the request is simply rejected, most of them are hardly considered for embedded applications [69, 245]. In fact 2. – 4. are most commonly not useful for autonomous, time-critical, and energy constrained systems: For option 4 in particular the suspended task cannot easily be resumed if it e.g. causes a new blocking and should be executed due to a priority inheritance policy. For option 3 it is important to notice, that revoking or moving memory without signaling this action to the owner task is

critical or even impossible in most cases since related addresses will become invalid. Not even data structures which are just accessed relative to block base addresses (e.g. C structures) can simply be relocated by moving the data and updating the start address: Expired addresses might still reside in registers or CPU stages, then. Much worse, affected peripherals like e.g. DMA controllers can often not be updated automatically, and would still transfer data from or to old addresses. In such situations not even option 2, i.e. terminating and restarting a spurious task, would be a valid solution. Instead, modifications to a memory block can best be handled by the owner task which has complete knowledge about its current usage and *all* dependencies. Thus, and for the reasons given in Sections 7.1 and 6.2, only the last option is supported by our CoMem approach[18].

Before we go into the details, we'll briefly recall the general idea behind our collaboration concept: According to the basic priority inheritance protocol policy, a task $u$'s *active priority* $p(u)$ is raised to $p(t)$ if $u$ blocks at least one other task $t$ with truly higher *active priority* $p(t) > p(u)$ by means of at least one critical resource. Only then, DynamicHinting passes a hint to $u$, indicating this priority inversion, and "demands" for releasing at least one critical resource quickly. While this was initially intended to facilitate the on-demand release and handover of blocked resources, we'll also use this indicator to trigger some situation specific memory management and reorganization functions. According to the *Smart*OS specification S4[p103], which forbids the revocation of any resource as well as its use by non-owners in general, modifications to any allocated memory block must also be authorized by the corresponding owner and always be conducted in its task context. In particular, this allows for adequate preparations prior to any heap reorganization and avoids critical system states[19]. Their completion will finally be signaled back to a blocked task, and implicitly triggers its re-request for memory.

For both voluntary actions (release and relocate), we need to discuss which blocks to select, and how to treat them adequately with respect to their current owner task. Since our concept targets on respecting and enforcing the compliance to task priorities only those blocks will be considered for reorganization which belong to lower prioritized tasks, and which would lead to sufficient continuous space for serving higher prioritized requesters. Among these we first signal the one with lowest base address, and the capability to produce sufficient free space. If a hint is rejected, it will cyclically be passed to the next appropriate task[20]. In contrast to the frequently used *free lists* for currently unallocated areas (→ Figure 7.5b), our strategy therefore applies a linear *used list* of currently allocated blocks since these must be known anyway in order to dynamically select spurious owner tasks for DynamicHinting. Details will be discussed in Section 7.5.

For the sake of tuneability (F3), the requester's remaining timeout and active priority will also be passed along with the hint to the blocking tasks. Furthermore, we advise the blockers whether releasing or relocating their memory blocks would solve the actual problem most

[18]Of course the other options can (easily) be added by replacing the hint passing with another operation. However, the collaborative character is entirely lost then, and other recovery strategies might be required.

[19]Adequate authorization checks can be incorporated into the corresponding code (e.g. via `testResource(...)`).

[20]Signaling tasks with their lowest priority first (compared to lowest base address first) did not make much difference, but increased the average execution time of the selection algorithm.

suitably, and thus would account for the reactivity and progress of more relevant tasks. If both reactions are acceptable, relocating blocks will take precedence over releasing them: While a relocation is less damaging and imposes less side effects on the blocker, releasing memory is at least as effective when seen from the blocked task's view. Eventually, the generated hints trigger a self-controlled but on-demand heap reorganization by means of some further CoMem API functions (e.g. `relocate()` and `free()` from Listing 7.4) which are legally called by and in the context of the current owner task. Of course, specific performance and energy issues must still be taken into account, but depend on the task's actual hint processing strategy.

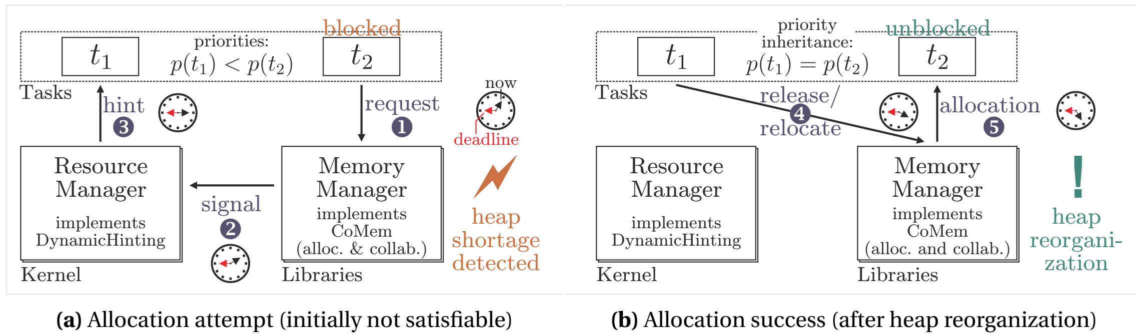
**Usage and operation example.** Exemplified by an initially not satisfiable memory request, Figure 7.6 shows some typical interactions within our modular concept to still grant the allocation in time: The task $t_2$ requests some heap memory directly from the memory manager (❶). Though it specified an absolute deadline, $t_2$ can not be served immediately due to a lasting memory shortage. However, with our approach the memory manager identifies a lower priority task $t_1$ with $p(t_1) < p(t_2)$, which might help to relax the situation, and signals this option to the resource manager (❷) which is reliable for the task synchronization. While $t_2$ remains blocked in suspended state, a hint is passed to $t_1$ (❸) and possibly triggers a self-controlled heap reorganization (❹). This can especially include the adaptation of referencing data structures, or the deactivation and reconfiguration of an autonomously operating on-chip resource (e.g. a DMA controller) which would otherwise continue to access expired block addresses. Under guidance of the memory manager, this finally leads to sufficient space for serving and unblocking $t_2$ (❺). Regarding potential real-time constraints and the concomitant spatial and temporal guarantee to serve $t_2$ in time, $t_1$'s block might have been selected intentionally since a valid contract engaged $t_1$ to react on the hint before $t_2$'s deadline has been reached. Details on hard real-time operation can be found in Section 7.5.4.

Yet, passing hints is not trivial in preemptive systems. From the blocker's view, this happens quasi-asynchronously and regardless of its current situation, task state, or code position. Given that a blocking task itself can be in ready or even in waiting state while a new blocking comes up, we already presented three techniques for receiving hints in Section 6.5.4. Since Explicit Querying is cumbersome and hardly applicable for real software designs, and we intend to comply to the usability feature M4, we won't consider it for CoMem at all. When using Early Wakeup and HintHandlers[21], hints are passed automatically when blocking occurs. While this is much more reliable and yields improved reactivity, we need to provide async-signal safety then to comply with feature request F1. Nevertheless, our reflective approach still allows each task to dynamically decide between collaborative or egoistic behavior with respect to its current conditions and other tasks' requirements – e.g. by using appropriate behavior functions.

### 7.4.2. Collaborative Memory Sharing

Since memory is commonly a scarce resource in small embedded systems, it needs to be shared among the tasks in order to achieve a higher integration density for future versatile systems and

[21] Both methods can be combined with the *Smart*OS exception handling concept from Section 4.3.8.



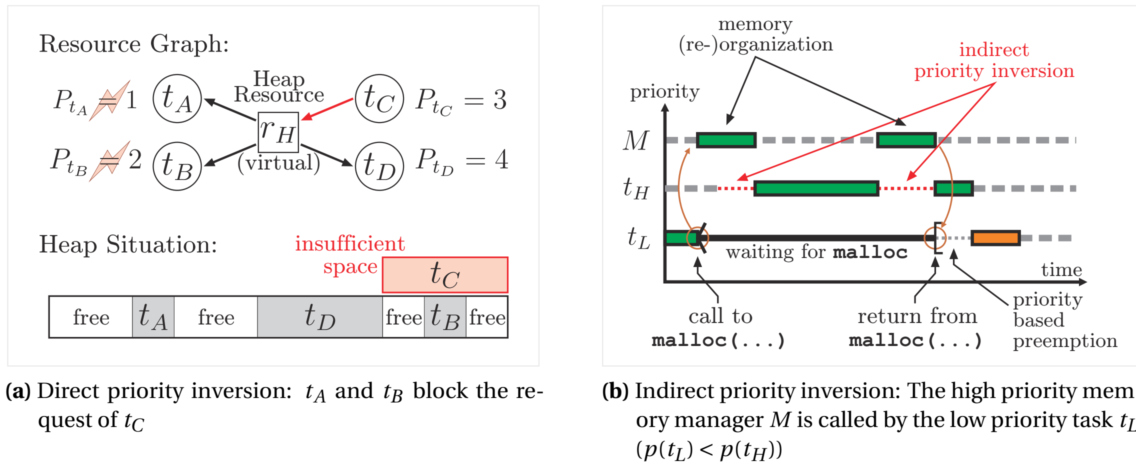

**(a)** Allocation attempt (initially not satisfiable) **(b)** Allocation success (after heap reorganization)

**Figure 7.6.:** Delayed memory allocation, and related interactions between system components

pervasive applications (→ Section 1.2). This is already true if some tasks run rather seldom and a static memory allocation would leave valuable space unused for long periods. Nevertheless, tasks in general might also be subject to tight timing constraints when requesting memory upon certain events. Triggered by environmental interactions, even situation specific tuneability might be required then (→ F3):

**Priority compliance and task starvation.** In this context, the first problem we want to address is priority inversion concerning such an request (F11). According to Definition II.5[p96] this term was initially used upon blocking on exclusive but temporarily shared resources as described in Sections 3.2 and 4.3.7. Considering the heap memory resource the available space will not only become partitioned and fragmented at system runtime, but the number of allocated blocks and owner tasks will vary also. In such cases a single-instance resource is insufficient for synchronizing the interleaved but mutually exclusive access to dynamically assigned blocks. Instead, virtualization or multi-instance resources are used by many approaches. These support simultaneous allocations by different owners, and allow the spatial sharing of the internally split (and otherwise monolithic) memory among concurrently running tasks. However, it is often forgotten that priority inversion can also occur in conjunction with e.g. virtualization, and that its handling is even more complex then, since it can originate from multiple tasks at the same time.

Figure 7.7a shows an example scenario: Tasks $t_A$, $t_B$, and $t_D$ hold memory blocks protected by the multi-instance resource $r_H$. Due to the actual heap situation, $t_C$'s memory request can currently not be granted, and we see a *direct priority inversion* originating from $t_A$ and $t_B$ towards $t_C$. The question whether to reject $t_C$ immediately or whether to grant the blocking tasks some time to resolve the memory shortage, was already discussed in the context of general resources in Chapter 6. In the case of multi-instance resources, simply using e.g. PIP or PCP for raising the active priorities $p(t_A)$ and $p(t_B)$, and to potentially accelerate their deallocation while $t_C$ remains blocked in suspended state (→ Figure 6.4[p103]), imposes some additional questions: Which task's active priority should be adapted? Raising just one blocking task requires a reasonable selection strategy, and might still select the "wrong" one. Raising all blocking tasks $t_A$ and $t_B$ means setting them to equal priorities $p(t_A) = p(t_B) = P_{t_C}$, and leads to e.g. round robin or strictly cooperative




**(a)** Direct priority inversion: $t_A$ and $t_B$ block the request of $t_C$

**(b)** Indirect priority inversion: The high priority memory manager $M$ is called by the low priority task $t_L$ $(p(t_L) < p(t_H))$

**Figure 7.7.:** Critical scenarios during dynamic memory management

scheduling (→ Figure 3.3[p39]) despite of intentionally different *base priorities* $P_{t_A} \leq P_{t_B} \leq P_{t_C}$. Furthermore, the chance for inheritance related inversions, which are fortunately minimized under PIP[22], increases along with the number of unnecessarily raised priorities. Finally, the option for simultaneous allocations of a single resource by several tasks will result in increased complexity and management overhead, and thus is not supported under *Smart*OS. Especially in combination with priority inheritance protocols, severe adaptations to the specifications and internal data structures would significantly increase the execution time of the related algorithms[23].

Going back to the initial problem from Figure 7.7a, $t_C$ could be served if either lower prioritized task $t_A$ or $t_B$ would release or just relocate its memory block. Yet, in all existing approaches we found within the embedded systems domain, especially in those from Section 7.3, tasks do not know about their spurious influences, and thus cannot react adequately. In consequence, most allocators do not even try to find a solution for improving the situation, but return immediately and indicate this failure via an error code[24]. In turn, developers tend to retry aggressively until the allocation succeeds:

a) Using e.g. plain C functionality within preemptive systems would result in *spinning loops* calling the allocator, and cause the unintentional (and maybe infinite) starvation of lower prioritized tasks. As depicted in Figure 7.2a, such behavior can even lead to livelocks of spinning tasks.
b) If the underlying operating system supports timing control for tasks, spinning might be

[22] In fact this is one reason for which we prefer PIP over PCP as the proposed resource synchronization protocol (→ Table 6.1[p99]) under *Smart*OS and DynamicHinting.
[23] In particular the at most linear execution times for processing the resource-await-queues (RAQ) in Section 6.6 arose from the fact that this data structure is well-defined for each Task $t \in T$ (→ Lemma II.2). Potential divergences in the RAQs would prohibit a linear traversal, and result in higher computational complexity.
[24] For example, the allocator from the GNU standard C library glibc (`void* malloc(size_t size)`) returns a `NULL` pointer in case of an allocation failure [104].

relaxed by periodic *polling* for free memory. While this would still cause significant CPU load upon short periods, it can potentially miss sufficiently large areas of free memory upon long periods: If the memory manager does not know that a task $t$ actually still waits for a memory block between its polls, it can neither serve $t$ nor would it be a good advice to reserve memory in advance.

c) If task blocking has to be avoided entirely, *lock-free* methods, as applied by the modified Hoard allocator [211] for multi-core architectures, can be an option. A shared resource is defined as *lock-free* (*non-blocking*) if the resource manager guarantees that whenever a task executes a finite number of steps, at least one operation on the resource by some task must have made progress during the execution of these steps. Basically, the implementation of lock-free methods applies do-while loops which are preemptible themselves (e.g. by higher priority tasks) and (re)try to commit some changes to a resource until the code within their loop body was executed without any interfering interruption, i.e. comparable to a critical section where no other task modified the resource during the last iteration. In Listing 4.2a[p53] we have already seen an example for the lock-free modification of a list head: While the old head is saved right before the modification attempt, a `CAS` instruction tries to commit the update and simultaneously checks for success or failure to directly control the break condition at the loop's end.

Even though these three techniques are quite common and reflect the state of the art, they produce a high system load in case of not satisfiable requests, result in reduced CPU time for blocking tasks, and consequently make it difficult to comply with real-time requirements.

**Reducing CPU load through task self-suspension.** To ease the problem of starved blocking tasks, CoMem works similar to the lock-free technique, but instead of letting blocked tasks spin aggressively, it puts them into waiting (i.e. suspended) state and defers each iteration until certain indicators announce a potential allocation success during the next retry. The provision of task suspending `malloc` API functions minimizes the CPU load caused by currently not serviceable tasks, while it still preserves the possibility to signal them a delayed success as soon as possible:

Corresponding to the *Smart*OS kernel API, we allow temporarily limited memory requests by extending the allocator's interface with an optional deadline or timeout parameter (→ Figure 7.8), and, in the tradition of the self-suspending kernel functions from Figure A.2[p327], each task consequently transfers the memory request to the memory and resource manager subsystem[25]. Through this temporal limitation a task not only provides the CoMem collaborator with information about how long it is willing to wait in worst case, but it also supplies the allocator with a defined amount of time for the reorganization of the heap space[26]. While the concept of using self-suspending functions for operations which might not complete immediately, but depend

[25] As Figure 3.1[p34] depicts, we consider the memory manager as an OS component which will not necessarily be executed in kernel mode.

[26] In order to support DynamicHinting, the requesting task's deadline will also be passed to the blocking tasks which in turn can use this information within their TUFs or behavior functions (→ Section 7.6.2, F3).

on other tasks than the caller, already showed in Section 4.3 to provide preemption tolerance (F7) through *Smart*OS's priority-based scheduler, kill tolerance (F6) is implicitly supported since tasks cannot terminate entirely under *Smart*OS (→ S1[p102])[27].

**Avoiding indirect priority inversion.** Regarding the requested priority reflexion (F10), another point is to decide whether the CoMem subsystem itself should be implemented as task (server), as kernel function (syscall), or if it operates entirely within the context of each task (library).

If the memory manager was implemented as server task or atomic syscall, the internal data structures would be implicitly protected against concurrent access[28]. On the other hand, servers and syscalls commonly run at higher priority than ordinary application tasks, and *indirect priority inversion* would still emerge from handling each request immediately and independently from the requester's priority: As Figure 7.7b shows, this would allow a low priority task $t_L$ to implicitly slow down a higher priority task $t_H$ by simply calling `malloc()`. To avoid this problem, it is at least wise to design the memory manager as server task $t_M$, and to adapt its base priority $P_{t_M}$ dynamically to the maximum active priority of all tasks with pending memory requests[29]. To further reduce the runtime overhead in terms of task count, context switches, and stack space, we decided to implement CoMem as a library, and to execute all contained functions entirely within the context of the calling tasks. While this improves portability (F5), it will implicitly treat the corresponding operations with adequate priority in relation to other tasks (F3). Yet, async-signal safety issues must be considered for this approach since concurrently executing tasks and HintHandlers might interfere[30].

### 7.4.3. Summary

Throughout this section we already pointed out how we intend to fulfill most feature requests from Section 7.2.2. From the remaining points we still have to discuss async-signal safety (F1), to deal with the handling of potential deadlock situations (F8), and to provide a means for the specification and enforcement of a WCAT for time-critical requests (F12). Since these depend on the concrete CoMem implementation, we'll present the technical details and central algorithms first, and address them separately afterwards starting with Section 7.5.3. Previously, however, we will summarize our CoMem design decisions and make sure that these correspond to the preconditions DH1 and DH2[p111] from Section 6.5.3 for using DynamicHinting at all:

[27]Even if task termination was implemented under *Smart*OS, it would still be possible for the task or memory manager to traverse the list of allocated blocks first in order to to query each well-defined owner task and to possibly release the corresponding blocks (→ Section 7.5.1). Apart, we could also introduce some kind of "terminate-signal" as e.g. known from the Unix world, and which could be sent as a hint to instruct a task to cleanly terminate itself.

[28]Comparable to the server task design pattern (→ *Smart*Net from Section 8.1), client tasks would just indicate their request, and wait for an event to signal its completion.

[29]If dynamic base priorities are supported at all – like e.g. under *Smart*OS when using PIP. In addition to the discussion about an appropriate resource synchronization protocol from Section 6.3.2, this is just another example for the benefit of dynamic base priorities.

[30]Note that IRQ handlers are not affected since these may never access any resource through a potentially self-suspending kernel function (→ Section 4.3.7 and Figure A.2[p327]).

1. Persisting memory allocations must not prevent further requests (→ DH1). While this avoids task starvation, the collaborator can always keep track about all system-wide requirements, and produce adequate hints and advices.

2. Extending `malloc()` by a timeout or deadline for limited waiting adds a temporal semantic to the API (M4), and gives blocking tasks the time to react on a hint (→ DH2).

3. Executing the memory management functions directly within the callers' task contexts reduces overhead (M1), and implicitly reflects the task priorities (F10).

Although requested in M3a, our concept does not even attempt to *protect* dynamic memory blocks against unauthorized access as long as no MMU or MPU is available. While some approaches (e.g. [69]) replace memory access instructions by jumps into special management functions and allow to detect false sharing, they also cause significantly increased CPU load. Beyond, perfect security cannot be provided anyway since indirect access (e.g. via a DMA controller) remains hard to control. Thus, we prefer the direct and constant time access to allocated blocks (→ M1b), and only *coordinate* the exclusive sharing of the available heap space.

## 7.5. CoMem Implementation and Usage

This section presents the central implementation details about our novel memory management approach. Just like in the case of DynamicHinting the basic idea behind CoMem can be applied as integral concept for many (embedded) real-time operating systems if these support truly preemptive and prioritized tasks plus a timing concept which allows temporarily limited requests for exclusively shared resources.

For our reference implementation we further extended *Smart*OS from Chapter 4 since it fulfills all these requirements. As also demanded in feature request F5, its kernel architecture offers quite common characteristics, is available for several MCU architectures, and thus is a good representative for the adaptation of similar systems. CoMem supports the *Smart*OS resource sharing philosophy by entirely omitting centralized modifications to allocated memory blocks by the memory manager: Once assigned, each owner task is responsible for releasing, resizing, and relocating its memory block(s) explicitly and voluntarily. To still achieve priority aware and on-demand memory reorganization both the priority inheritance protocols from Section 6.3.2 and the DynamicHinting paradigm from Section 6 were applied as described previously. In the following, we'll discuss how to integrate our collaborative resource sharing approach and the *Smart*OS specification to meet the CoMem design considerations from Section 7.4.2.

In general, the implementation of CoMem serves as an example on how the execution of an (intrinsically atomic) operation within an application task can be interrupted by another task or by the operating system by utilizing an ordinary synchronization resource as a mediator to coordinate this (extraordinary) influence.

### 7.5.1. Internal Data Structures

Regarding the tight performance and memory constraints of many embedded systems, CoMem is limited to a central list data structure, a tiny API, and one control block for each dynamic memory allocation. The heap memory itself will be placed by the linker, and – as demanded in M2b – occupies the entire unused system RAM, i.e. the free space between the global variables and the kernel stack (→ Figure A.1[p324] for the MSP430 memory layout).

**The Memory Control Blocks (MCB): Brokers as collaborative synchronization primitive.** As shown in Listing 7.1, each *memory control block* (MCB, data type: `MCB_t`) takes either 7 or 13 machine words in RAM[31], and can be located anywhere in the system memory as long as it remains accessible for its allocating task. Compared to other approaches, which often require just about 2 – 3 machine words for managing each block (→ Knuth's boundary tag method [159] and Figure 7.5b) an MCB might appear to be quite large. However, storing some additional information gives us the option to support (timing) contracts, and to establish the already mentioned *bidirectional* communication link between the memory manager and the block owners. Through this strategy, which is quite novel in the area of dynamic memory management, we are able to inform tasks immediately if they block a task of higher priority due to a dynamic memory allocation. In fact we can simply use a special dynamic hint generation scheme for this, and associate one ordinary *Smart*OS resource (→ Section 4.3.7) – a so called *broker resource*, or simply *broker* – with each memory block. In case of successful memory allocation the broker will be allocated automatically by the memory manager, but implicitly belongs to the requesting task since it is exactly this task which executed the corresponding `malloc` library function within its own context. The broker stays long-term allocated throughout the entire block allocation time, and will finally also be released automatically by the memory manager during the block deallocation process which, once more, is conducted by the `free` library function in the owner task's context. In direct consequence to this, we directly profit from five important advantages:

1. Each allocated memory block – though dynamic in size and location – is known to the resource manager as an individual system resource: The underlying resource management policy (e.g. PIP from Section 6.4.1) will implicitly be applied for the memory management, and *all* operational resources will be treated in exactly the same way. In particular, the synchronization scheme and task priorities will also be respected equally.
2. Memory blocks remain strictly exclusive: According to the *Smart*OS philosophy neither the kernel nor the memory manager will ever touch an allocated block, but always notify the owner in case of critical situations (e.g. priority inversions). This can be done by simply requesting a task's broker resource for a currently disturbing memory block, which, according to Definition II.4[p89], is one of the owner's critical resources then.
3. A notified owner task is free to decide dynamically between collaborative or egoistic behavior: In particular it can adjust its reaction carefully by relating its current situation and its very own requirements to the existing contracts and to the demands of the task(s) it is blocking.

[31] On the currently supported Atmel AVR (8Bit), TI MSP430 (16Bit), and Renesas SH2A (32Bit) architectures.

```
1  /* CoMem MCB advice types and block types */
2  typedef enum { adv_nothing, adv_doRelocate, adv_doRelease,  // for non-RT blocks
3                 adv_RTBlock,                                  // for RT blocks
4                 adv_doClear,                                  // for any block
5               } advice_t;
6
7  /* The CoMem MCB data structure */
8  typedef struct {
9    size_t          size;      // 1W: block size in machine words
10                              //     (const if advice == adv_RTBlock)
11   volatile int    *base;     // 1W: block start address (fixed for RT blocks)
12                              //     (const if advice == adv_RTBlock)
13   Resource_t      *broker;   // 1W: associated broker resource
14                              //     (NULL if advice == adv_RTBlock,
15                              //     otherwise +6W for Resource_t)
16   union {                    // 2W: timing contract for real-time operation
17     Delay_t       WCRT;      //     WCRT R in [μs] for hinting non-RT blocks
18                              //     WCRT = -1 for unknown or R ≥ 2^(8·sizeof(Delay_t)) - 1
19                              //     (valid if advice != adv_RTBlock)
20     Delay_t       WCAT;      //     WCAT A in [μs] for allocating RT blocks
21                              //     (valid if advice == adv_RTBlock)
22   }
23   advice_t        advice;    // 1W: advice for what to do upon a hint
24                              //     (const adv_RTBlock indicates a RT block)
25   MCB_t           *next;     // 1W: linked list pointer (next block in MCL)
26 } MCB_t;                     // Total RAM requirement:
27                              //         7W + 0W =  7W for RT blocks
28                              //         7W + 6W = 13W for non-RT blocks
```

**Listing 7.1:** The CoMem `MCB_t` data structure for Memory Control Blocks (MCB)

4. Though involved in the general resource management process, CoMem can entirely be implemented as library: In compliance with F5 it does not produce additional code or runtime overhead within the OS kernel (→ Figure 3.1[p34]).
5. By providing common or individual resource handler functions (i.e. `fGet` and `fRelease`) for each broker resource (→ Section 4.3.7) we can ensure the automated and reliable initialization and destruction of data stored in memory blocks for safety and security reasons (→ M3).

The `MCB_t` data structures will either be created by the allocator (for non real-time requests) or at compile time (for real-time requests). While the necessity for declaring real-time-critical blocks at compile time will be discussed in Section 7.5.4, the macros in Listing 7.2 already exemplify the creation and initialization of the `MCB_t` data structures and the corresponding brokers for non-RT blocks: According to the handle's individual broker declaration, each deallocation of such a memory resource will clear the corresponding block through `mem_clearMemory` and system-wide standards. Since *security* is not only an issue of proper memory protection, but also affected by the accessibility of information left over in released blocks, this automation is important to provide compliance with M3. In contrast, *safety* is not only a problem of e.g. correct address translation, but also of proper memory initialization during its allocation. However, to reduce runtime overhead we avoid a common `fGet` function for this, and clear the entire heap once throughout the initialization of the central memory management resource `rCOMEM` (→ Listing 7.3). An individual `fGet` function could still be supplied manually for each block's broker.

```
1  /* CoMem memory block deinitialization. Fills blocks with 0. */
2  int mem_clearMemory(Resource_t *broker) {
3    MCB_t *dm = mem_getMCBForBroker(broker);  // complexity: O(#allocatedBlocks)
4    if (dm->advice == adv_doClear)
5      mem_fill(dm, 0);                         // complexity: O(|dm|)
6    return 1;
7  }
8
9  /* CoMem block declaration macro for non-RT block m̌ (dynamic addr. and WCRT) */
10 #define COMEM_DECLARE_MCB(_name, _size)
11   OS_DECLARE_RESOURCE_EXT(__rCOMEM_##_name, NULL, NULL, mem_clearMemory);
12   MCB_t _name = { _size, NULL, &__rCOMEM_##_name, -1, adv_nothing, NULL }
13                  dynamic base                    broker  R(m̌) = ∞
14 /* CoMem block declaration macro for RT block m̂ (fixed address and WCAT) */
15 #define COMEM_DECLARE_RT_MCB(_name, _size, _base, WCAT)
16   MCB_t _name = { _size, _base + __heap, NULL, WCAT, adv_rt_block, NULL }
17                                  fixed base  no broker  A(m̂)
```

**Listing 7.2:** The CoMem memory control block (MCB) declaration and initialization

```
1  /* The fInit function for the CoMem management resource rCOMEM. */
2  int rCOMEM_fInit(Resource_t *unused) {
3    disableHint(&rCOMEM);    // disable DynamicHinting for this resource
4    clearHeapSpace()         // initialize the entire heap space
5    return 1;                // indicate initialization success
6  }
7
8  /* The CoMem management resource (protects critical sections)       RAM: 6W */
9  OS_DECLARE_RESOURCE_EXT(rCOMEM, rCOMEM_fInit, NULL, NULL);
10
11 /* The CoMem management event (signals modifications to the MCL)    RAM: 1W */
12 OS_DECLARE_EVENT(evMCLChanged);
13
14 /* The head of the memory control list (MCB with lowest address)    RAM: 1W */
15 MCB_t *COMEM_MCL = NULL;
```

**Listing 7.3:** The CoMem management resource (protection) and event (signaling)

**The Memory Control List (MCL): A "used list" as the allocator's main data structure.** Though the CoMem concept is initially independent from the applied allocator, we use a simple first-fit scheme for studying the effectiveness of our collaborative approach. In contrast to many other approaches, like e.g. [46], which maintain a so called *free list* of unallocated memory areas, our reference allocator utilizes a *used list* for currently allocated blocks. Internally, this *memory control list* (MCL) is organized as a linked list of preallocated MCBs sorted by block base addresses, and thus allows the linear scanning for continuous free areas of sufficient size for new requests. Beyond, removing an MCB from the linked list implicitly results in the *automatic coalescing* of adjacent free areas without any additional efforts. Though other data structures (like e.g. totally or partially ordered trees [306]) might scale better for a large number of simultaneous allocations, a simple list's low maintenance complexity is in line with the typically weak sensor node hardware, and still provided good performance within our test beds and real-

world applications[32]. In fact, allocated blocks must be scanned anyway to select one particular for reorganization in case of insufficient free space (→ Section 7.5.4) for blocked but pending requests. Complexity: $O(n)$, where $n$ is the number of currently allocated blocks.

The concurrent access to the MCL is synchronized and protected against race conditions through the unique *management resource* `rCOMEM`, which is acquired by each CoMem API function prior to any MCL read or write operation to create a surrounding critical section[33]. Its declaration is shown in Listing 7.3 along with the MCL head pointer, and the central management event `evMCLChanged` for signaling modifications to the MCL; the total RAM requirement for these static variables and management data structures is 8 machine words[31,34]. The obvious deactivation of DynamicHinting for `rCOMEM` in Line 3 is acceptable since – according to the classification scheme from Section 3.2 – it will *only* be used as a short-term resource. In fact, the avoidance of hints for `rCOMEM` is even required due to the request for async-signal safety (→ F1) which will be discussed in Section 7.5.3.

### 7.5.2. API Functions

In the following we refer to Listing 7.4 which shows the central algorithms of the CoMem collaborator, and to Figure 7.9 for some selected execution flows. Unlike most memory managers whose API functions require just the size of the requested memory block or the block address pointers themselves, the CoMem API relies on MCB pointers[35] which will be passed to most of its public collaborator functions. Let's start with `free` and `relocate` since these are rather simple compared to the memory allocation.

**Memory deallocation and relocation.** For releasing memory the `free(m)` function simply removes the specified MCB $m$ from the MCL and releases the broker resource $b_m$ (`free`:6). Finally, it triggers the `MCLChanged` event to indicate the MCL update. In contrast, `relocate(m)` seeks a new location for the specified block $m$ (cyclic next-fit) by which more continuous free space would become available (`relocate`:3), and moves the contained data. Finally, it temporarily releases its own broker resource $b_m$ (`relocate`:6-7) *and* triggers an event (`relocate`:9) to indicate the modification and to resume waiting tasks. For subsequent address updates by the caller, the block and data shift is returned in bytes.

Note that within both functions `free` and `relocate` DynamicHinting is disabled for the broker before releasing it since any potential and subsequent handover to a blocked task means just a short-term allocation (`mallocUntilDH`:21-22)[36]. Also note that both the broker release

[32] While the evaluation of more advanced data structures is left for future research, most SANet applications seem to make use of only a few dynamically allocated memory blocks at the same time.

[33] Do not confuse this with atomic sections, which are intentionally not supported for application tasks under *Smart*OS. Critical sections prevent simultaneous execution but are still preemptive under *Smart*OS.

[34] The ROM requirement depends on the architecture-specific code size (e.g. ≈2 kB for the TI MSP430).

[35] MCB pointers are comparable to file handles under e.g. Microsoft Windows [110] and the C standard library libC [104].

[36] Any interference by yet another task with higher priority would cause entirely avoidable overhead since the "borrowed" broker will be released immediately anyway. A hint would only delay this action through the additional hint processing action (i.e. deallocation and reallocation) between the lines `mallocUntilDH`:21-22.

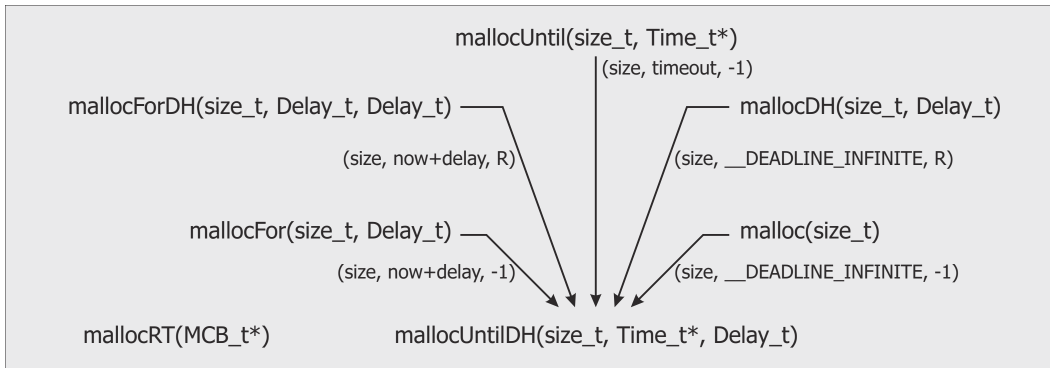



**Figure 7.8.:** API and call hierarchy for the CoMem `malloc` functions

and the event trigger are used to resume tasks with pending requests (depending on their current situation).

**Memory allocation.** The CoMem API functions for memory allocation are summarized in Figure 7.8. As usual under *Smart*OS, most available variations map to one basic function `mallocUntilDH(...)` which requests memory with an absolute deadline but without hard real-time guarantees for the caller. For those non-RT blocks it accepts three parameters:

1. The requested size $s$ of the continuous non-RT memory block.
2. The absolute deadline $\tau$ for this request.
3. The requester's WCRT $R$ for processing a hint once the block has been allocated.

Internally `mallocUntilDH` checks if the caller is authorized to request dynamic memory and throws an exception otherwise (`mallocUntilDH`:2). Then, comparable to lock-free methods, the collaborator code loops until the request for dynamic memory succeeded or the deadline $\tau$ has been reached (`mallocUntilDH`:6): Each retry attempts to insert the new block $m$ into the MCL (`mallocUntilDH`:7). Therefore it reverts to the allocator (which uses a first-fit approach in our case) and ends up in one out of three situations:

**Case 1**: On immediate success (`mallocUntilDH`:8), i.e. if the allocator returned an MCB $m$, the corresponding broker resource $b_m$ is locked by the calling block owner $\sigma(m)$ and we are done. Figure 7.9 shows an immediately successful request by task a $t_2$. Considering our resource management policy, it will be sufficient for another task with higher priority to request this particular resource if it finds itself blocked by $\sigma(m)$ and thus requires $m$ to be removed.

**Case 2**: Indeed, this is exactly what will happen if sufficient space is not available but a spurious memory block $m'$ was found (`mallocUntilDH`:15). In case the broker $b_{m'}$ is currently allocated (Case 2b) the short-term resource request (`mallocUntilDH`:21-22) adapts the active priority $p(\sigma(b_{m'}))$ of the blocking owner $\sigma(b_{m'})$ through the priority inheritance protocol. In Figure 7.9 PIP raises $p(t_1) := p(t_2) = 200$. If DynamicHinting is enabled for $b_{m'}$ (`mallocUntilDH`:10), the resource manager immediately passes a hint to $\sigma(b_{m'})$ (i.e. to $t_1$ in Figure 7.9) and indicates its disturbing influence. If $\sigma(b_{m'})$ reacts by releasing or relocating its block $m'$ before the blocked

task's deadline $\tau$ has expired, it also releases $b_{m'}$ either ultimately (`free`:6) or temporarily (`relocate`:6-7) to indicate the adjusted memory situation. This deallocation once more adapts its priority, resumes the blocked task, and triggers a new retry for allocating $m$. If the deadline $\tau$ has expired the pending resource request is canceled (`mallocUntilDH`:21), retrying stops, and `NULL` is returned (`mallocUntilDH`:31)[37]. In Figure 7.9 $t_1$ relocates its block, and through the subsequent broker handover PIP reduces $p(t_1) := 100$ again. Flow 1 in the example shows that $t_2$ immediately releases the "borrowed" broker, and finds sufficient space during the next allocation attempt; eventually $t_1$ reallocates the broker for its relocated block and we are done.

During both Flows 2 and 3 yet another task $t_5$ with even higher priority $p(t_5) = 250$ interferes with $t_2$'s pending request. Most important for those examples, CoMem allows $t_5$ to still overtake $t_2$ as demanded by our priority reflexion feature request F10: While $t_2$ still holds $t_1$'s broker "borrowed" in Flow 2, it blocks the request of task $t_5$ and receives a hint itself which it follows for the benefit of $t_5$. Finally $t_2$ completes its own allocation by requesting $t_1$ to remove its block. At the beginning of Flow 3 task $t_5$ also requests the broker of the spurious block which, however, is already locked in a short-term allocation by the non-owner $t_2$ (borrowed from $t_1$), and thus won't generate a hint towards $t_2$. After the broker has been handed over to $t_5$ and released immediately thereafter, $t_5$ encounters case 2a and suspends itself until the next MCL update. Instead the priority-based scheduler executes $t_2$ which also runs into case 2a and likewise has to wait for the `MCLChanged` event. As soon as $t_1$ reallocated its broker $t_5$ is resumed again and passes a hint to $t_1$. In turn $t_1$ releases its block and sets both $t_5$ and $t_2$ ready through the broker deallocation and the event trigger, respectively. After $t_5$ succeeded first, $t_2$ retries but this time it requests $t_6$ to free some space.

The remaining problem regarding the allocator is how to *reasonably* select and return a blocking MCB $m'$ to the collaborator for generating a hint on. While scanning the MCL for free space (`mallocUntilDH`:7), the allocator searches for two types of MCBs: The first type would already produce the requested space if it was relocated, and the other one would only produce the requested space if it was released entirely. Among these the first type takes precedence, and $m'_{\text{advice}}$ will be set to either `adv_doRelocate` or `adv_doRelease`. Following the first-fit approach for the allocation itself, the first block with a lower priority owner than the currently blocked task is selected for hinting. Thus, along with the hint, its owner also receives the advice for a suitable reaction. Of course, and from the view of the blocked task in particular, a release is always at least as effective as a relocation.

**Case 3**: If no spurious task, or block $m'$ respectively, was found (`mallocUntilDH`:27) at all, the collaborator simply waits for the next modification to the heap space. Triggered by another task through the `MCLChanged` event from within `free` or `relocate`, one more retry is started if this happens within the deadline $\tau$. Otherwise waiting is canceled and `NULL` is returned to indicate the timeout. Once more Figure 7.9 gives an example where the low priority task $t_3$ cannot hint the higher prioritized block owners, but has to wait for a voluntary release; which is finally done by $t_1$ and lets $t_3$ succeed.

[37] Note that forwarding the deadline $\tau$ to all self-suspending (kernel) functions allows us to immediately exit from the inner loop.

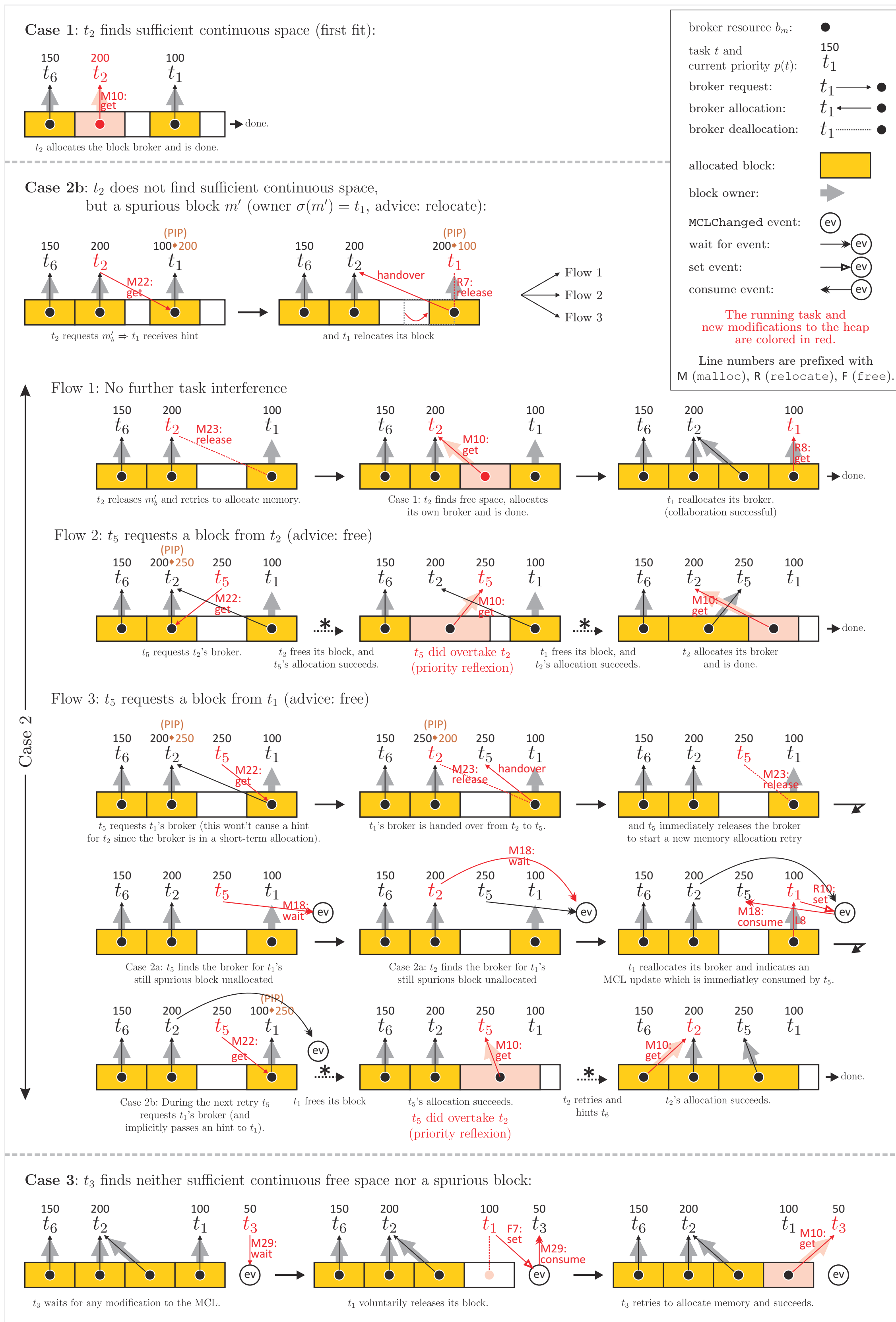



**Figure 7.9.:** Various memory allocation flows related to the source code from Listing 7.4 with DynamicHinting enabled, i.e. `USE_HINTS = REQUEST_BROKER = 1`

```
MCB_t* mallocUntilDH(size_t s, Time_t *τ, Delay_t R) {
  if (os_inKernelMode()) THROW EXC_COMEM;
  MCB_t *m' = NULL;

  retry = 1;
  while (retry == 1) {
    MCB_t *m = insertIntoMCL(s, R, &m');
    if (m != NULL) {
      getResource(m->broker);
      if (USE_HINTS == 1) enableHint(m->broker);
      return m;
    }
    if (&τ == 0) return NULL;
    clearEvent(&MCLChanged);
    if (m' != NULL) {
      if (testResource(m'->broker) != 0) {
        retry = waitEventUntil(&evMCLChanged, τ);
        continue;
      }
      if (REQUEST_BROKER == 1) {
        retry = getResourceUntil(m'->broker, τ);
        if (retry) releaseResource(m'->broker);
      } else {
        retry = waitEventUntil(&evMCLChanged, τ);

      }
    } else {
      retry = waitEventUntil(&MCLChanged, τ);
    }
  }
  return NULL;
}
```

will always succeed immediately

Prohibit requests by IRQ handlers, and create pointer to a spurious block.

Retry until success or timeout...

Try to find continuous free space:

**Case 1**: Continuous free space found:
Lock the broker $b_m$, enable DynamicHinting, and return the handle.

**Case 2**: Found a spurious block $m'$:

**a)** Broker free: Wait for an MCL update (owner is already working on $m'$ and $b_{m'}$ is currently not allocated).

**b)** Adapt the blocker's priority by requesting its broker resource $b_{m'}$ temporarily. This will also pass a hint.

pass hint (→ Listing 7.7)

**c)** Omit a hint and just wait for MCL update. (For test beds without DynamicHinting).

**Case 3**: Neither free space nor a spurious block found:
Wait for the next MCL update.

Timeout.

```
void free(MCB_t *m) {
  if (os_inKernelMode()) THROW EXC_COMEM;
  removeFromMCL(m);
  disableHint(m->broker);
  m->advice = adv_doClear;
  releaseResource(m->broker);
  setEvent(&MCLChanged);
}
```

Release the block $m$.

Release the broker $b_m$.

Indicate the MCL update.

```
int relocate(MCB_t *m) {
  if (os_inKernelMode()) THROW EXC_COMEM;
  delta = relocateInMCL(m);
  moveDataBy(m, delta);
  disableHint(m->broker);
  releaseResource(m->broker);
  getResource(m->broker);
  if (USE_HINTS == 1) enableHint(m->broker);
  setEvent(&MCLChanged);
  return delta;
}
```

Find new space and copy the block content.

Temporary broker release.

Indicate the MCL update.

**Listing 7.4:** The CoMem API/collaborator functions and their interactions:
Calls to allocator functions are protected through the short-term management resource `rCOMEM`, and highlighted in gray (critical sections).
Green arrows: Interaction via the global `MCLChanged` event.
Red arrows: Interaction via a block specific broker resource.

**Summary.** This section introduced the CoMem API functions. While relevant implementation consequences regarding our feature request from Section 7.2.2 will be discussed next, the support for time-critical memory allocations will be presented in Section 7.5.4, and application examples can be found in the test bed Section 7.6.

### 7.5.3. Implementation Consequences

Having introduced the general algorithms behind the CoMem collaborator, we'll now discuss the implementation consequences for the remaining feature requests from Section 7.2.2. From these we still have to discuss the usability aspect, in particular the selection of PIP, the deadlock freedom for the management resource, and the async-signal safety of the library functions.

**Selecting PIP to avoid unreasonable restrictions.** Within our CoMem reference implementation we applied the priority inheritance protocol (→ Section 6.3.2) for synchronizing the brokers, and hence we intentionally focused on long-term memory allocations as frequently required for complex objects like memory blocks (the related advantages of PIP were already discussed in Chapter 6). Though CoMem can in principle also be realized with other policies where task blocking can occur and DynamicHinting is supported, we want to illustrate once more how the selection of PIP performs in the special case of CoMem and why it avoids some significant shortcomings of similar techniques (→ Table 6.1[p99]):

At system runtime, each broker $b_m$ is likely to remain locked in a long-term allocation by the owner task of the corresponding MCB (→ `mallocUntilDH:9` in Listing 7.4). Meanwhile, higher prioritized tasks might signal their demand for currently unavailable dynamic memory by requesting such a foreign broker $b_{m'}$, and keep it locked briefly in a short-term allocation (→ `mallocUntilDH:21-22` in Listing 7.4). When using e.g. PCP (or any other protocol which requires a priori knowledge like allocation graphs as introduced in Section 6.3.2), the *true* ceiling priority for each broker resource $b_i$ is hard to compute, since the actual heap fragmentation and the resulting broker requests are hardly predictable. To be prepared for the worst case, we can only compute pessimistic ceiling priorities $c(b_i)$ by considering all potential requests among the tasks $T_{DM} \subseteq T \setminus \{t_0\}$ with dynamic memory utilization:

**Lemma II.5.** *For resource synchronization protocols which require the knowledge of ceiling priorities $c(b_i)$ for all brokers $b_i$, these are identical and compute according to Eq. (6.11) as*

$$c_b := \max\{P_t \mid t \in T_{DM}\}. \tag{7.1}$$

*Proof.* Eq. (7.1) is true since the highest prioritized task in $T_{DM}$ might potentially request *each* broker, and thus $\forall_{b_i} : c(b_i) := \max\{P_t \mid t \in T_{DM}\} = c_b$. □

Even worse, only one task could allocate one or more blocks at a time, since

$$\forall_{t \in T_{DM}} : p(t) \leq c_b \tag{7.2}$$

would hold then, and PCP's resource assignment condition from Eq. (6.12)[p124] would fail immediately for each additional request from another task. Even serviceable requests (from the

allocator's view) would be denied preventively since they could lead to the later rejection of higher priority tasks. Similar to the DynamicHinting test bed results from Section 6.7, PCP would perform unbalanced and worse than PIP due to its conservative policy. In fact, it would cause unacceptable restrictions for any memory manager (→ M4[p145]) and starve tasks. This is why we continue to use PIP, and benefit once more from its generous resource assignment policy.

**Deadlock freedom and async-signal safety.** Within the introduction of this chapter we requested for deadlock tolerance and async-signal safety as indispensable features of a dynamic memory manager. Referring to our implementation from Listing 7.4 and the resource policy from Section 6.4.1, we can show both properties for CoMem:

**Lemma II.6.** *CoMem is free from management deadlocks (→ F8b).*

*Proof.* According to Lemma II.4, a task $t$ can only produce a deadlock cycle, if it already owns any resource $r_a$ while requesting another resource which is currently allocated by a task $u \neq t$. Furthermore, such a cycle contains at least $n \geq 2$ tasks, and $n$ resources as well. Thus, let $t, u \in T \setminus \{t_0\}$ be two arbitrary *Smart*OS application tasks and let $r_a \in R$ be any *Smart*OS resource. Most important, let $r_M \in R$ be the *Smart*OS resource `rCOMEM` for protecting the internal CoMem data structures as depicted in Listing 7.4. In order to produce a deadlock by simply requesting some memory and putting $r_M$ into an RAQ cycle, a resource situation as depicted in Figure 7.10a must be reached. However, this scenario can never occur under *Smart*OS, since $u$ would obviously have allocated $r_M$ prior to requesting $r_a$, then. As clearly visible within the CoMem code from Listing 7.4, the algorithms will never request any resource while still holding $r_M$; in particular, the broker resources are always allocated outside of the critical sections (gray areas). In consequence, $|A(r_M)| \in \{1, 2\}$, and according to both Lemmata II.2c and II.4a CoMem is free from management deadlocks regarding its internal protection resource `rCOMEM`. □

**Lemma II.7.** *CoMem provides async-signal safety for its API functions (→ F1).*

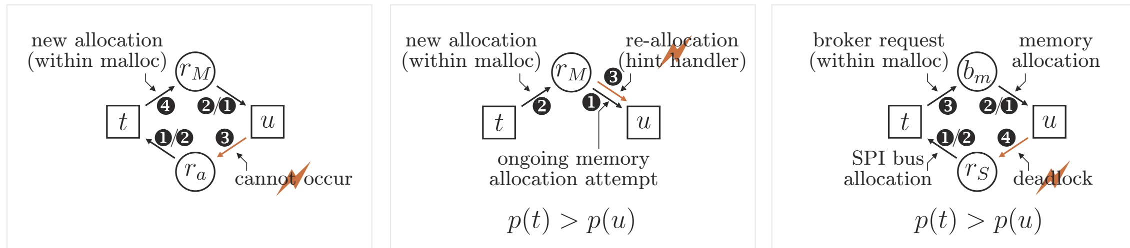
*Proof.* Regarding this property, we have to distinguish between the asynchronous occurrence of IRQ handlers (→ Section 4.3.5) and HintHandlers (→ Section 6.5.4) under *Smart*OS:

a) For IRQ handlers, resource (de-)allocations are prohibited entirely according to the *Smart*OS specification. Since this restriction also applies for dynamic memory allocations, and for broker resources in particular, CoMem is async-signal safe for IRQ handlers[38].

b) Regarding HintHandlers we have to distinguish between the management resource $r_M$ and the broker resources[39]. Within the example from Figure 7.10b two critical situations might emerge from two tasks' interleaving request for memory. Here $p(t) > p(u)$, and $t$ preempted $u$ while the latter already held $r_M$ during a memory allocation attempt (❶). If $t$ also requests

[38] As depicted in Listing 7.4, each CoMem API function will check for the correct execution context (i.e. task mode), and throw an exception if the test fails. The exception might finally result in a kernel panic (if it is not caught within the IRQ handler) to indicate a bad software design.

[39] Remind that HintHandlers also occur asynchronously just like interrupts, but have more resource-related privileges. In fact they are meant for resolving resource conflicts, and thus they will intentionally be executed within the context of their associated task.




**(a)** Deadlock freedom for the management resource $r_M$ **(b)** Async-signal safety regarding the CoMem resource $r_M$ **(c)** Async-signal safety regarding any broker resource $r_b$

**Figure 7.10.:** CoMem deadlock freedom and async-signal safety
(The numbers ❶ – ❹ denote the order of resource requests and allocations.)

$r_M$ for a memory allocation (❷), a hint will be passed to $u$ indicating this priority inversion. If $u$'s HintHandler decides to react on the hint, and requests $r_M$ once more in $u$'s context (❸), it will succeed immediately since, according to the *Smart*OS specification S6[p104], $u$ already owns $r_M$ and will simply receive it twice. In consequence, $u$ and its own handler would concurrently execute critical sections protected by $r_M$. This problem cannot be solved by using synchronization primitives: Any attempt to synchronize between a signal handler and a regular task execution would immediately result in a deadlock[40]. Thus, CoMem avoids the multiple allocation of the management resource, and in general disables DynamicHinting on $r_M$ for performance reasons and to provide async-signal safety (→ Listing 7.3, Line 3).

□

Though DynamicHinting is disabled for $r_M$ the priority inheritance protocol is still applied, and adjusts any blocking task's priority regarding this and other resources: Since we strictly avoided task self-suspensions by the allocator within CoMem's critical sections, the execution delay of any interleaving but blocked task will increase, but the extra time in waiting state (for $r_M$) is mainly bounded by the worst case execution time for insertion, removal or relocation of a single block. Since these operations will be executed at the priority of the blocked task, comparable problems are inherent to any allocator design when used in combination with preemptive tasks.

Compared to the async-signal safety and deadlock freedom for its API functions, CoMem does not inherently provide these properties for brokers in general. Let's take a look at Figure 7.10c where the task $u$ allocated some memory along with the corresponding broker resource $b_m$ (❶), and the task $t$ allocated a communication bus resource $r_S$ (❷). Next (❸), $t$ requests some memory which is currently not available, and implicitly triggers a hint for $u$. If $u$ accepts the hint, but decides to save the contents of its memory block to an external flash memory first, this operation might also require the communication bus. In this case, $u$'s request for $r_S$ (❹) leads to a deadlock situation between $u$ and $t$, and must be handled as described in Section 6.5.5.

[40]While the handler holds exclusive access to the CPU, the task holds a resource which is requested by the handler but cannot be released since the task won't receive the CPU. A deep discussion of this problem can also be found in the multithreaded programming guide for POSIX compatible operating systems [271]).

### 7.5.4. Hard Real-Time Heap Organization

So far we have introduced the basic collaboration concept and implementation details behind our novel memory management approach. However, until now the acceptance of hints and the subsequent heap reorganization are neither guaranteed nor bounded in their execution time. While we'll already obtain remarkably low and task priority dependent memory allocation times without further efforts (→ Section 7.6), the worst case allocation times (WCAT) remain uncertain – a fact which is at most acceptable for non real-time or soft real-time operation.

In this section we present an extension for CoMem to support the reliable and efficient sharing of the valuable heap space in open systems, i.e. among coexisting tasks with time-critical and non time-critical requests for blocks of arbitrary size. Therefore our goal is threefold:

1. We *need* to save the commonly sparse memory reserves by truly sharing a single memory space between real-time and non real-time tasks[41].
2. We *must* provide a spatial and temporal guarantee for time-critical requests – even in out-of-memory or high load situations.
3. We largely *want* to avoid the prophylactic rejection of requests from non time-critical tasks – at least as long as free space is available.

While the 1st goal is just a predefinition we make, the others are objectives which can be enforced through CoMem if some specific information about the memory related timing constraints is available, *and* if we are ready to accept additional runtime effort resulting from a special adaptation of the allocator. According to Figure 7.11 the idea is based on the one hand on static *compile time contracts* between the allocator and time-critical tasks, and on the other hand on dynamic *runtime contracts* between the collaborator and non time-critical tasks: Without reserving the required space permanently, a static heap layout – the so called *RT heap layout* – is created for time-critical allocations to definitely avoid their mutual interference and deadline violation at runtime. Also at runtime the allocator relies on the RT heap layout to save heap memory by dynamically co-locating non time-critical blocks in a way to ensure that an on-demand heap reorganization for freeing space at shared but potentially colliding addresses is temporarily bounded *and* below a co-located real-time task's block allocation timeout. This is achieved by maintaining some kind of *safe state* regarding the heap partitioning and the available timing information. In the following we define this special RT heap layout to assure timely memory allocations for real-time tasks with hard deadlines.

**The static RT heap layout** is based on some information which is commonly available for hard real-time tasks anyway (e.g. through static code analysis): Initially we bipartition the set $M$ of memory blocks into time-critical blocks $\hat{M}$ (*RT blocks* for short) and non time-critical blocks $\check{M}$ (*non-RT blocks* for short) with $\hat{M} \cap \check{M} = \emptyset$ and $\hat{M} \cup \check{M} = M$. While $\hat{M}$ must be known

[41] In fact providing various heap areas or pools for critical and non-critical blocks would only attenuate the problem since the time-critical tasks would still compete for memory and might still run into severe allocation failures. Besides it would lead to increased external fragmentation as already discussed in Section 7.2.1.

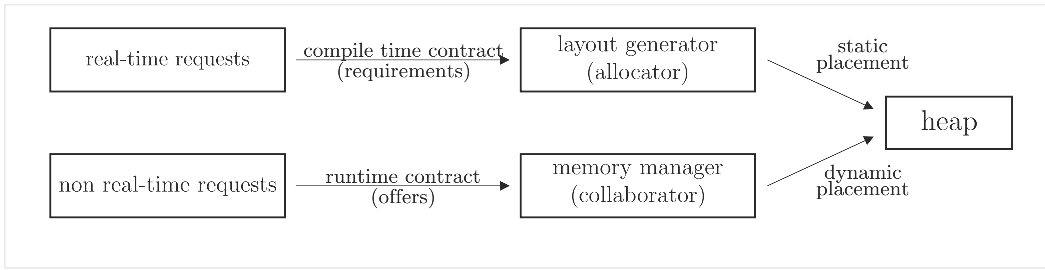



**Figure 7.11.:** Memory allocation contracts under CoMem

during the heap layout generation[42], $\check{M}$ is arbitrary at any time. We also assume that RT blocks won't be superseded on-demand, and thus do neither trigger hints nor require a broker resource (→ Listing 7.1). For each RT block $\hat{m} \in \hat{M}$ we define the acceptable *worst case allocation time* WCAT $A(\hat{m}) \in (0, \infty)$, i.e. the value which will be used as hard timeout for the memory request. Reflecting these timing constraints, the directed *RT memory graph* $G_{\hat{M}}$ specifies the potentially simultaneous allocations of RT blocks as follows:

$$\begin{aligned} G_{\hat{M}} &:= (\hat{M}, \alpha) \quad \text{with} \\ \hat{M} &:= \text{the set of memory blocks with hard allocation timeouts} \\ \alpha &:= \{(\hat{a}, \hat{b}) \mid \hat{a}, \hat{b} \in \hat{M} \ \wedge \ A(\hat{a}) \leq A(\hat{b}) \\ &\qquad \wedge \ \hat{a}, \hat{b} \text{ might be allocated simultaneously}\} \end{aligned} \tag{7.3}$$

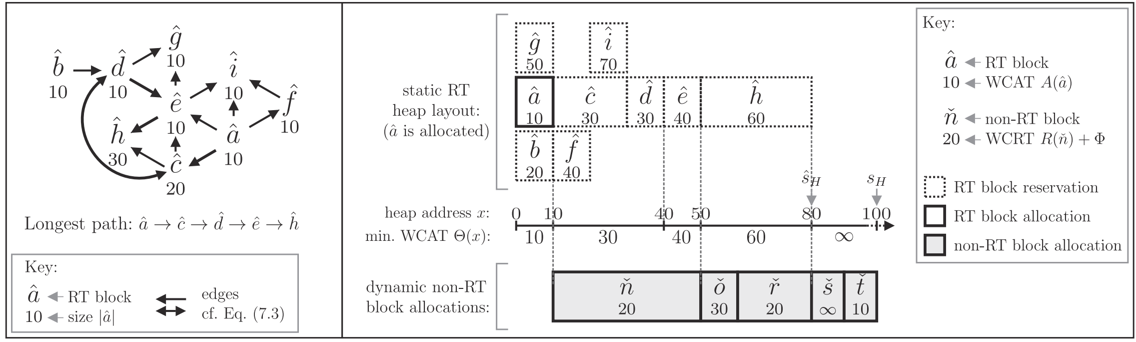
As an example Figure 7.12a shows a set $\hat{M} = \{\hat{a}, \ldots, \hat{i}\}$ of RT blocks, for which each path in $G_{\hat{M}}$ is partially ordered by its nodes' WCAT[43].

Of course it is not always easy to determine which blocks will never be allocated simultaneously. While we won't present an algorithm or tool to analyze an arbitrary set of *Smart*OS tasks for potential interference but leave this for future work, corresponding information can easily and automatically be deduced for tasks which allocate and deallocate blocks successively (since these won't interfere by nature), and – as a more complex challenge – for tasks with a well-known execution interleaving (e.g. through mutual triggering as e.g. implemented for the SNoW Bat task system in Section 10.2.4).

The next step is to generate the RT heap layout. To avoid spatially colliding requests for RT blocks (i.e. to support any valid allocation scenario for blocks in $\hat{M}$), but to also obtain as large continuous areas of free memory as possible for non real-time tasks, we need yet another metric for placing the RT blocks efficiently. For each heap address $x$ we define $\Theta(x)$ as the minimal

[42]Within our current implementation the required static information must be provided by the system designer at compile time. Modifications at runtime might require the computation of an entirely new heap layout from scratch, and are currently not yet supported by our implementation. However, this option could be added e.g. through an additional server task $t_S$ with high priority $P_{t_S}$ which requests the entire heap, and passes hints to all current owners. If these tasks are designed to always release or relocate their blocks in case of a sufficiently high threshold priority $\varphi \leq P_{t_S}$, the new layout can also be computed on-demand by $t_S$.

[43]If there is no appropriate information, we assume the worst case where all allocations in $\hat{M}$ might exist simultaneously; then $G_{\hat{M}}$ is still directed but fully meshed.




**(a)** The RT memory graph $G_{\hat{M}}$ (considering block sizes)

**(b)** The RT heap layout and some exemplary block allocations (considering timing constraints)

**Figure 7.12.:** The CoMem heap organization for RT blocks and co-located non-RT blocks

WCAT over all RT blocks spanning over $x$:

$$\Theta(x) := \begin{cases} \min\{A(\hat{m})\} & x \text{ is reserved for } \hat{m} \in \hat{M} \\ \infty & x \text{ is not reserved for any } \hat{m} \in \hat{M} \end{cases} \tag{7.4}$$

**Placing RT blocks at static addresses.** After these preliminary considerations, the next step is to generate the RT heap layout. As requested in M2[p145], our goal is to keep the scarce memory in mind, and to efficiently share the heap space between the RT blocks $\hat{M}$ and non-RT blocks $\check{M}$ while not jeopardizing other feature requests[44]. To still avoid colliding requests among RT blocks while obtaining large continuous areas of assignable memory for non real-time tasks, we introduce two rules C1 and C2 based on $G_{\hat{M}}$ and $\Theta(x)$:

C1 No two different RT blocks $\hat{m}_1, \hat{m}_2 \in \hat{M}$ with $(\hat{m}_1, \hat{m}_2) \in \alpha$ may span over a common heap address $x$. I.e. RT blocks must never interfere with each other.

C2 Reservations for the RT blocks $\hat{M}$ must be partially ordered by $A(\hat{m})$ as follows: $\forall_{x \le y}$ : $\Theta(x) \le \Theta(y)$. I.e. the shorter $A(\hat{m})$ the lower $\hat{m}$'s base address.

Both rules are easily obeyed and the RT layout is quickly created in $O(|\hat{M}|\log|\hat{M}|)$ by sorting $\hat{M}$ by ascending $A(\hat{m})$, and placing the RT blocks successively at the lowest address permitted by C1 and C2. While C1 already solves the memory allocation for the RT blocks $\hat{M}$, we still have to show how C2 simplifies the allocator's dynamic placement algorithm for non-RT blocks $\check{M}$.

As an example Figure 7.12b shows the statically generated RT heap layout for the RT memory graph from Figure 7.12a. In contrast to e.g. $\hat{f}$ and $\hat{i}$ with $(\hat{f}, \hat{i}) \in \alpha$, the RT blocks $\hat{a}, \hat{b}$, and $\hat{g}$ can be placed at the same address since they won't be allocated simultaneously (but maybe alternately through synchronized tasks, or successively within a single loop). Though the RT

[44] Therefore some allocators (e.g. under VxWorks [307]) maintain several heaps to place memory blocks depending on their owner's priority. While this would once more raise problems regarding their individual size and the resulting fragmentation, a temporal allocation guarantee can still not be given. Due to the limited memory we won't accept the concomitant overhead, and seek for another solution.

block base addresses are fixed, the memory is not assigned statically but only reserved while being shared with non-RT blocks at runtime[45].

**Placing non-RT blocks at dynamic addresses.** While $\hat{M}$ must obviously be known during RT layout generation, non-RT blocks $\check{M}$ need not to be known then. However, to allow the emergence of hints toward blocking non-RT tasks at runtime, we assume a strict separation between RT and non-RT tasks regarding their initial *base priority P* as defined by the developer[46]:

$$\forall_{\check{m}\in\check{M},\hat{m}\in\hat{M}} : P_{\sigma(\check{m})} < P_{\sigma(\hat{m})}$$

where $\sigma(m)$ denotes the requester or owner task of block $m$. Then PIP raises the active priority $p(\sigma(\check{m})) \geq p(t)$ if a RT task $t$ with $P_t > P_{\sigma(\check{m})}$ indirectly requests $\sigma(\check{m}) \neq t$ to clear its exclusively allocated space through CoMem[47].

Upon each allocation attempt, the requester $\sigma(\check{m})$ of a non-RT block must provide an individual contract offer by specifying the WCRT $R(\check{m}) \in (0,\infty]$ of its hint handling routine for clearing $\check{m}$ (by either `free` or `relocate` from Listing 7.4). Since CoMem relies on this information, the specification of $R(\check{m})$ offers and accepts a contract at the same time. If $R(\check{m})$ is unknown, $\infty$ must be specified to be prepared for the ultimate worst case. The offer is negotiated by the memory manager, and allows an appropriate placement of the non-RT blocks as follows:

When considering some temporal overhead $\Phi$ for the memory management itself (e.g. for the broker (de-)allocation), the lowest possible base address $x_{\min}(\check{m})$ for any $\check{m} \in \check{M}$ is

$$x_{\min}(\check{m}) := \min\{x \mid \Theta(x) \geq R(\check{m}) + \Phi\}. \tag{7.5}$$

E.g. $\check{n} \in \check{M}$ in Figure 7.12b is placed at its lowest possible address $x_{\min}(\check{n}) = 10$. Since however several non-RT blocks may share a RT block's reserved range (e.g. $\check{o}, \check{r}, \hat{h}$ in Figure 7.12b), we have to be careful with such multiple co-locations, and select each base address according to the additional rule

C3 Maintain a temporal and spatial safe state for RT blocks by placing any $\check{m} \in \check{M}$ at a base address $x \geq x_{\min}(\check{m})$ so that the reserved space for any RT block $\hat{m} \in \hat{M}$ can reliably be freed within $A(\hat{m})$:

$$\forall_{\hat{m}\in\hat{M}} : \sum_{\substack{\check{m}\in\check{M} \\ \check{m} \text{ is co-located with } \hat{m}}} R(\check{m}) + \Phi \leq A(\hat{m}) \tag{7.6}$$

This way, all potentially disturbing non-RT blocks can be removed on-demand for the guaranteed and timely success of any RT task's request. If all WCRTs of the affected non-RT tasks are held, the hard WCATs of the RT blocks are also safe.

[45]Note that if $G_{\hat{M}}$ was fully meshed, the RT blocks $\hat{a}, \dots, \hat{i}$ would be placed successively in this order with $\hat{a}$ at the lowest address.

[46]A property, which is commonly the case for hard real-time operation, anyway.

[47]Even if $\sigma(\check{m})$ would not be scheduled and thus break its contract, this would only occur due to another even higher prioritized task. Schedulability analysis might be applicable to detect and avoid such system behavior at development time.

Since we placed the RT blocks according to C2, Eq. (7.5) holds and the allocator can start to seek the required space at the highest heap address. Like for $\check{s}, \check{t}$ in Figure 7.12b it might not even share memory with RT tasks then, but as soon as it needs to co-locate a non-RT block $\check{m} \in \check{M}$ with any $\hat{m} \in \hat{M}$ it can stop at address $x_{\min}(\check{m})$ at the latest. This is why the MCL contains its blocks sorted by base addresses.

If there is currently no space available for $\check{m}$, the memory manager tries to hint and eliminate any other allocated $\check{m}' \in \check{M}$ with $R(\check{m}') + \Phi \leq A(\check{m})$ and lower owner priority. In case of success $\check{m}$ is allocated while C3 must still be followed to maintain the safe state. If an allocation is not possible within $A(\check{m})$, the request is rejected (timeout).

**Real-time feasibility and heap size dimensioning.** To ensure the feasibility of our approach, the total heap size $s_H$ must at least be sufficient for the RT blocks under C1 and C2. While the exact size requirement $\hat{s}_H$ for $\hat{M}$ is known through the generated RT heap layout ($\hat{s}_H = 80$ in Figure 7.12b), there is also a theoretical upper bound $\lceil \hat{s}_H \rceil$ which can be computed by using the individual block sizes $|\hat{m}|$ as node weights in $G_{\hat{M}}$, and finding the longest acyclic path $P_{\hat{M}}$ therein. Then,

$$\lceil \hat{s}_H \rceil := |P_{\hat{M}}| = \sum_{\hat{m} \in P_{\hat{M}}} |\hat{m}| \geq \hat{s}_H. \tag{7.7}$$

In addition, some extra space $\check{s}_H$ should be reserved for those non-RT blocks which cannot be entirely co-located with RT blocks due to Eq. (7.5) and C3, respectively. At least for those blocks $\check{m} \in \check{M}$ with known size $|\check{m}|$, the still non-allocatable fraction computes as

$$u(\check{m}) := \max\{0, |\check{m}| - (\hat{s}_H - x_{\min}(\check{m}))\}.$$

With it,

$$\check{s}_H := \max\{u(\check{m}) \mid \check{m} \in \check{M}\} \tag{7.8}$$

is the lower bound for the extra space, and finally

$$s_H \geq \hat{s}_H + \check{s}_H \tag{7.9}$$

must be chosen as minimal heap size to be sufficient in terms of (timely) allocations for all RT blocks, but also to provide at least one suitable location for each non-RT block of known size. Still, allocations can never be granted for non-RT blocks.

For Figure 7.12a, $P_{\hat{M}} = \hat{a}, \hat{c}, \hat{d}, \hat{e}, \hat{h}$ with $\lceil \hat{s}_H \rceil = |P_{\hat{M}}| = 80$. The RT heap layout in Figure 7.12b also requires $\hat{s}_H = 80$. Thus, $s_H \geq \hat{s}_H + \check{s}_H = 80 + u(\check{s}) = 80 + 10 = 90$ must be selected. Since we provided $s_H = 100$, the allocator was able to place (the chronologically last request) $\check{t}$ at base address 90; with $s_H = 90$, $\check{t}$ could not have been allocated since $\hat{a}$ is already located at address 0. Besides, if e.g. $\hat{h}$ is requested, both $\check{o}$ and $\check{r}$ will be removed in time since according to Eq. (7.6) $R(\check{o}) + R(\check{r}) + 2\Phi = 50 \leq 60 = A(\hat{h})$. Likewise, $\check{n}$ will be removed in time for the request of any RT block $\hat{c}, \hat{d}, \hat{e}, \hat{f}$, or $\hat{i}$.

## 7.6. CoMem Evaluation and Benchmarks

Based on the just described CoMem concept and the operation details, we'll next discuss some performance results from concrete application scenarios. Just like in our previous benchmarks, we'll once more use one stress test to demonstrate the overall stability and behavior under extreme conditions, and one real-world example which will also become relevant for the indoor localization system SNoW Bat in Part III of this work.

**Metrics.** For analyzing the benefits of our collaborative memory management concept compared to the classic approach, i.e. the combination of temporally limited memory requests and on-demand heap reorganization vs. immediately returning allocator calls, various metrics were of special interest:

1. The minimum, maximum, and average allocation delay $\delta(i)$ for a task $t_i$ when requesting memory,
2. the iteration count $s(i)$ (during a defined time) for a task $t_i$ which requests memory in a periodic manner,
3. the hint count $h(i)$ for a task $t_i$,
4. the total fragmentation penalty $X(P)$ when either heap reorganization was required or when tasks could not be served due to the current memory partitioning $P$, and finally
5. the memory overhead for the internal CoMem management data structures.

While the metrics 1 – 3 are particularly relevant for evaluating the collaborator's runtime performance, metric 4 is mainly an indicator for the applied allocator's quality. Since however both components exert a strong influence on the task and memory states – which are highly dynamic anyway – they inevitably affect each other, too, and must thus be evaluated regarding their specific combination within the test beds. In contrast, metric 5 is constant for a given implementation.

As already mentioned several times, the typically small memory of sensor nodes had to be considered carefully during the reference implementation under *Smart*OS to leave sufficient space for the actual application. Once more our test scenarios were executed on the SNoW[5] platforms from Section 2.2 where we required 4 + 1 kB of ROM[48] and 40 + 16 B of RAM[49] for the whole *Smart*OS kernel plus the CoMem library.

For a detailed performance analysis of the metrics 1 – 3 we used the integrated *Smart*OS timeline with a resolution of 1 µs, and let each task log its related values at runtime to compute its individual statistics after the test run.

Metric 4 is more complex and requires some preliminary considerations: Especially for storage devices, such as hard drives, various metrics to assess a specific fragmentation have already been examined. However, these always rely on the number of non-contiguous files, or on their

[48]Compiled without benchmarking functionality using mspgcc [292].
[49]MCL head pointer (1×1W), MCL changed event (1×1W), protection resource (1×6W)

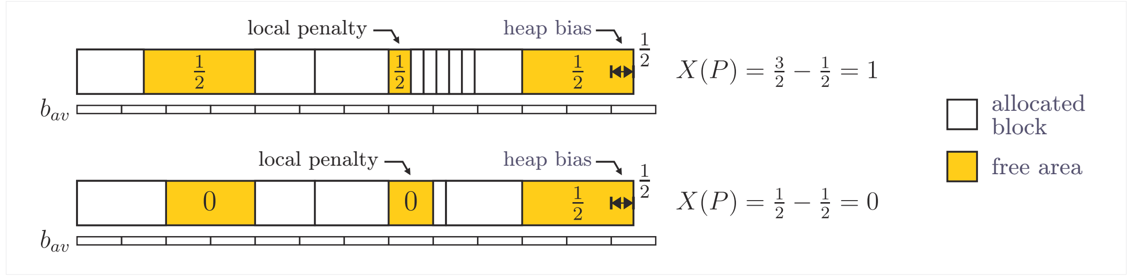



**Figure 7.13.:** Examples for heap fragmentation penalties

individual number of occupied clusters. As memory blocks on the heap (in contrast to files on the medium) must always be contiguous, these metrics are rather useless for us. Nevertheless, we want to make a statement about how good a specific heap fragmentation is in comparison with an optimal arrangement of the occupied areas, e.g. if all allocated blocks are adjacent and there is only one contiguous free area:

A partitioning $P$ of a heap $H$ with total size $|H| > 0$ consists of a finite but dynamic set of $n$ disjunctive memory areas $b_1, \ldots, b_n$ of maximum size $|b_i| > 0$. According to our block management data structure MCL from Section 7.5, we assume the automatic coalescing of free areas, and thus the sets of allocated and unallocated blocks are well-defined:

$$A(P) := \{b \in P \mid b \text{ is allocated}\} \quad \text{and} \quad U(P) := \{b \in P \mid b \text{ is unallocated}\}. \tag{7.10}$$

When considering an average block size $|b|_{av}$ for the allocations within a certain application, the heap's *total fragmentation penalty* $X(P)$ is the number of average sized blocks which could *only* be allocated in addition to the already allocated blocks if the current placement of the latter was perfect, or, in other words, the number of average sized blocks which can *not* be allocated since the used blocks are not perfectly placed[50]. Therefore, we initially define the *local fragmentation penalty* of a free memory area as the fraction of an average size block which would not completely fit inside if the area was filled up with such blocks. Consequently, $X(P)$ is defined as the sum over all local penalties for each unallocated area $b \in U(P)$ reduced by the constant heap bias, which is the penalty for an entirely free heap:

$$X(P) := \max\{0, \underbrace{\left\lfloor \sum_{b \in U(P)} \overbrace{\left( \frac{|b|}{|b|_{av}} - \left\lfloor \frac{|b|}{|b|_{av}} \right\rfloor \right)}^{<1} \right\rfloor}_{\text{local penalties}} - \underbrace{\overbrace{\left( \frac{|H|}{|b|_{av}} - \left\lfloor \frac{|H|}{|b|_{av}} \right\rfloor \right)}^{<1}}_{\text{constant heap bias}} \} \tag{7.11}$$

According to this definition, any $X(P) < 1$ can be considered as optimal, since not even one more block could be allocated in case of a perfect block placement.

[50] In this context "perfect" does not necessarily require the adjacent placement of these blocks. We also ignore the program logic and the concomitant locality principle (→ F4), but assume all blocks to be independent.

Figure 7.13 gives an example: The depicted heap can hold 12.5 blocks of average size $b_{av}$ and thus exhibits a constant bias of 0.5. The free areas within the upper partition can hold 2.5, 0.5, and 2.5 blocks of average size, and thus put a penalty of 0.5 each. Finally, the heap could hold $X(P) = 1$ more such average sized blocks if the fragmentation was optimal. In contrast, the lower partition is optimal since no other placement of allocated areas would allow to allocate more blocks of average size. For pool based approaches in particular their unique block sizes $|b| = |b|_{av}$ avoid external fragmentation, and consequently always yield a constant fragmentation penalty $X(P) = 0$.

### 7.6.1. Dynamic Memory Stress Test

**Setup.** The first test bed scenario analyzes our CoMem approach under extreme conditions with $n$ cyclic tasks $t_0, \ldots, t_{n-1}$ and many concurrent but irregular memory requests. We intentionally omitted dedicated RT blocks as well as explicit timing specifications and deadlines, to study the average performance when using arbitrary allocation sizes (→ F2). Instead, we simply assigned ascending base priorities $P_{t_i} = 100 + i$ for the tasks to analyze their relative success when requesting heap memory of arbitrary size. According to the state diagram from Figure 7.14a, the tasks were rather simple, and executed the same code repeatedly:

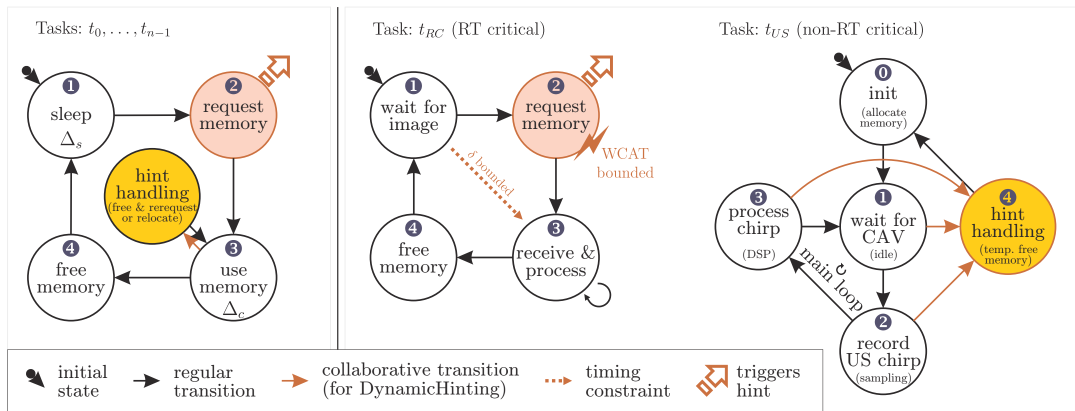
❶ Sleep for a randomized time $\Delta_s$.
❷ Request a memory block of randomized size $b_m$ without any finite timeout (i.e. $\tau = \infty$).
❸ Operate on the memory for a randomized time $\Delta_c$.
❹ Release the memory.

Both durations $\Delta_s$ and $\Delta_c$, as well as the requested block sizes $b_m$ were randomized for each iteration. By using various randomizer seeds, we obtained significant heap space dynamics, fragmentation, and task blocking which needed on-demand handling at runtime. Of particular interest in this respect is state ❸, which – in case of a memory related priority inversion as described in Section 7.4.2 – should be left prematurely to speed up the allocation delay of a higher priority task blocked in state ❷. Since we specified infinite timeouts $\tau$ for the requests, each task measured its specific execution times of `malloc`, and logged its minimum, maximum, and average allocation delays $\delta_{\min}$, $\delta_{\max}$, and $\delta_{av}$ separately. Furthermore, the number of received hints $h(i)$ and successful iterations $s(i)$ were counted. If CoMem performs according to our expectations, the results should exhibit the following relations:

$$\delta_{av}(i) \propto \frac{1}{P_{t_i}} \quad \text{and} \quad s(i) \propto P_{t_i} \tag{7.12}$$

The allocation delay would therefore decrease with increasing task priority; at the same time, the number of iterations should be directly proportional to the individual task relevance. In contrast, the number of triggered hints will show to be more difficult to predict.

Similar to the DynamicHinting test bed from Section 6.7 we once more applied two non-collaborative policies (P1, P2) and two collaborative policies (P3, P4) for comparing the allocation delays in relation to the task priorities when requesting memory according to Listing 7.4:




**(a)** Stress test (→ Listing 7.5) **(b)** Real-world application (→ Listings 7.6, 7.7)

**Figure 7.14.:** CoMem test bed task state diagrams (reduced to the relevant states)

P1 **Classic approach**: For this policy, we disabled the CoMem approach entirely (`USE_HINTS == 0`) and simply omitted the request for any blocking task's broker (`REQUEST_BROKER == 0`). To still avoid spinning loops of currently not serviceable tasks, we just waited in suspended state for heap modifications (Case 2c), and started another retry then. Consequently, blocks were no longer treated like regular resources, i.e. the priority inheritance protocol was not available for them. Furthermore, related hints as well as the chance for collaborative memory sharing vanished completely. This policy is comparable to many common approaches and its performance solely depends on the used allocator.

P2 **Priority inheritance**: For this policy, we reactivated the broker requests to signal upcoming memory shortages immediately (Case 2b, `REQUEST_BROKER == 1`). However, we let the blocking tasks simply ignore the emerging hints (`USE_HINTS == 0`). Though a blocking task $t$ would not collaborate on-demand then, its active priority $p(t)$ was at least raised to the priority of the task it blocked, and it received CPU time for step ❸ more quickly. Please note that the sole use of PIP is of almost no use, if only little CPU load is generated by the blocking task and it mainly stays suspended while holding the memory block[51].

To exploit the potential of CoMem we let each task activate DynamicHinting on its broker (`USE_HINTS == 1`) and triggered hints in case of out-of-memory situations (`REQUEST_BROKER == 1`):

P3 **CoMem & HintHandlers**: Here, each task generated CPU load but supplied a HintHandler for immediate injection into its own execution flow when blocking a higher prioritized task. This resulted in blocking while in ready/preempted state.

P4 **CoMem & Early Wakeup**: Finally, the tasks did sleep while holding a memory block. Yet, they were resumed immediately when blocking a higher prioritized task. This resulted in blocking while in waiting/suspended state.

[51]The same would be true for similar policies, like e.g. PCP, if task self-suspensions are allowed while holding resources exclusively (long-term allocations).

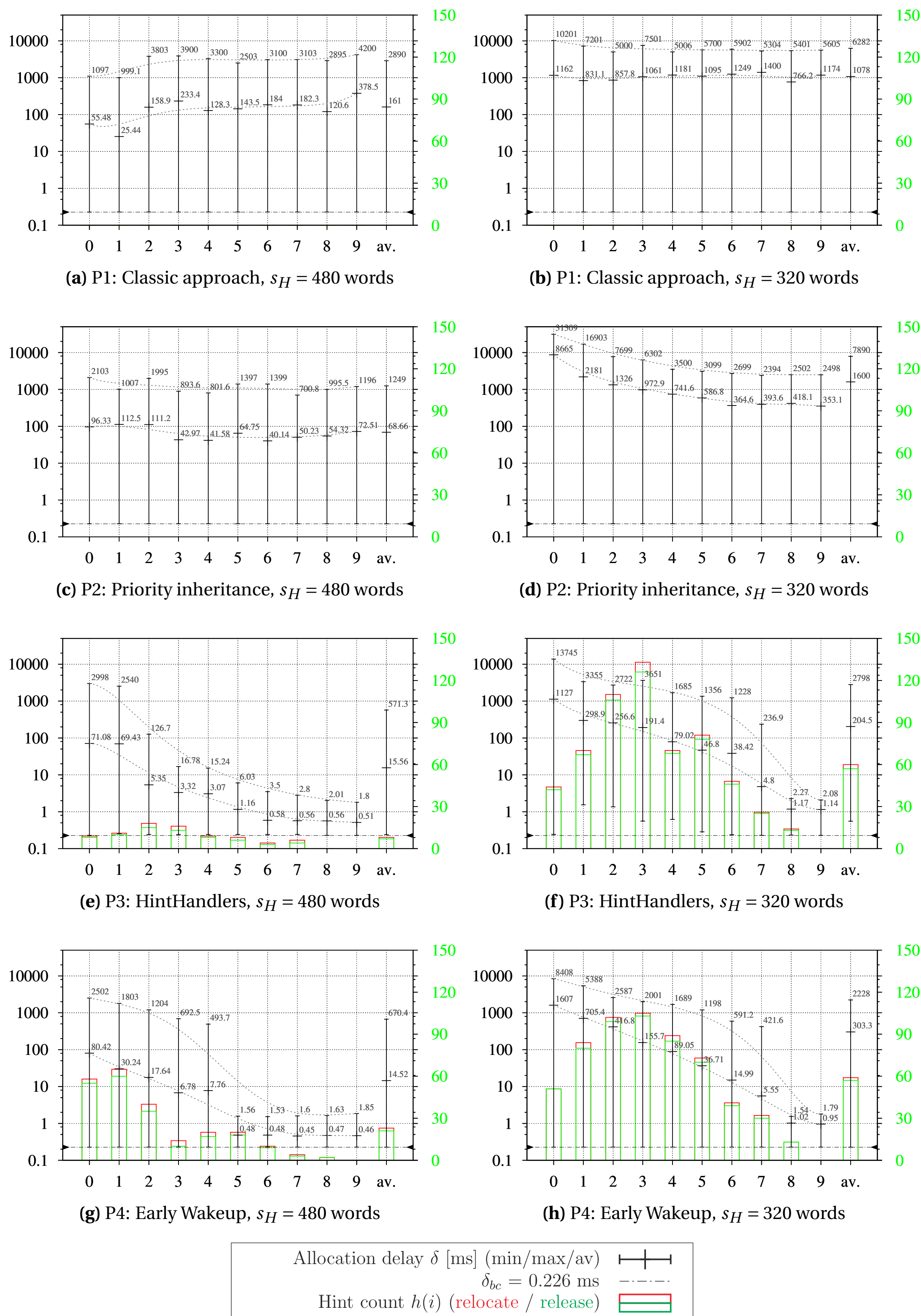



**Figure 7.15.:** CoMem stress test results for various policies and heap sizes: Allocation delay and Hint count (ascending base priorities for $n = 10$ tasks, test bed runtime: 10 min)

```
1  #if (TB_USE_HINTHANDLERS == 1)     // use as hint handler (→Section 6.6.4)
2    OS_DHHANDLER(DHH_Memory) {
3  #else
4    void DHH_Memory() {              // use as regular function
5  #endif
6    MCB_t* dm = getMCBForBroker(getResourceHint(NULL));
7    if (dm != NULL) {
8      // stop operation on the memory
9      if (dm->advice == adv_doRelocate) {
10       int delta = relocate(dm);
11     } else if (dm->advice == adv_doRelease) {
12       size_t size = dm->size;
13       free(dm);
14       /* ---------- THE TASK WILL BE PREEMPTED HERE SINCE AT ----------
15          -- LEAST ONE OTHER TASK WAITS FOR THE MEMORY/BROKER RELEASE -- */
16       malloc(size);
17     } // ignore other advices
18   }
19 }
```

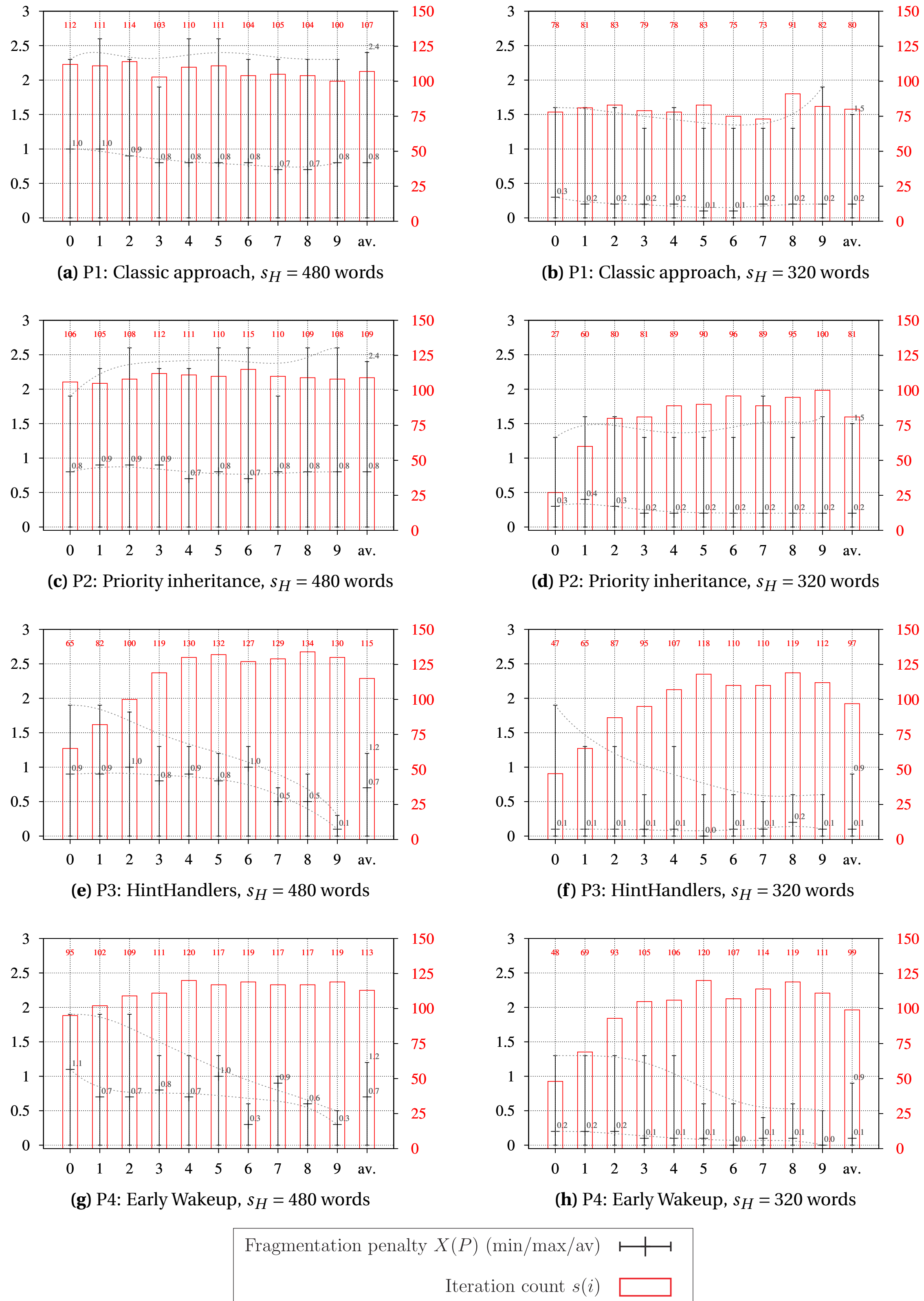
**Listing 7.5:** Hint handling for the CoMem stress test: The function can either be declared as asynchronous HintHandler, or as regular function for explicit calling during Early Wakeup.

Under P3 and P4, a task $t_L$ treated its hints according to Listing 7.5: After querying the affected MCB $t_L$ stopped the operation on the memory block, and – depending on the advice from the CoMem subsystem – it either called `free` or `relocate`. As intended, this caused the immediate allocation success as well as the scheduling of a directly blocked task $t_H$ with higher priority. This is always true since $t_H$ then held the highest priority of all tasks in ready state and $t_L$ did let $t_H$ "pass by". When scheduled again, $t_L$ tried to continue or restart its operation quickly. In case of `relocate` it reused the old but shifted block. In case of `free` it re-requested a block of the old size. Please note that the data integrity within the memory blocks was not considered by this test (→ Section 7.6.2 instead). Just the allocation delay was analyzed for reactivity and response time evaluation.

**Evaluation.** We configured the test bed using several task counts $n$, heap sizes $s_H$ and randomized block sizes $s_B$ under the policies described above. Since the results always showed similar main characteristics, we just present the analysis for $n = 10$ tasks, block sizes $s_B \in \{32, 64\}$ words, and heap sizes $s_H \in \{320, 480, 640\}$ words. Each setup was executed for 10 min.

As expected, all allocation attempts succeeded immediately when sufficient heap space $s_H = 640$ words was available to serve all requests even in the worst case. Though static memory assignments would suit much better then, we did this cross-check to see if the influence on the CPU load is already observable: Indeed, while the hint count $h(i)$ remained 0 for each $i \in \{0, \ldots, n-1\}$, the average allocation delay already settled around $\delta_{av}(i)$=280 µs for each task $t_i$ and policy. In comparison, the best case allocation time was $\delta_{bc}$=226 µs when executing only one task (with immediate success and without any preemption or self-suspension).

Selecting $s_H := 10 \cdot \frac{32+64}{2} = 480$ words (the required heap size for the average case) already shows the benefits of our collaborative approach (→ Figure 7.15, left column). Please note the




**(a)** P1: Classic approach, $s_H = 480$ words

**(b)** P1: Classic approach, $s_H = 320$ words

**(c)** P2: Priority inheritance, $s_H = 480$ words

**(d)** P2: Priority inheritance, $s_H = 320$ words

**(e)** P3: HintHandlers, $s_H = 480$ words

**(f)** P3: HintHandlers, $s_H = 320$ words

**(g)** P4: Early Wakeup, $s_H = 480$ words

**(h)** P4: Early Wakeup, $s_H = 320$ words

**Figure 7.16.:** CoMem stress test results for various policies and heap sizes: Fragmentation penalty and Iteration count (ascending base priorities for $n = 10$ tasks, test bed runtime: 10 min)

logarithmically scaled time axis. While the non-collaborative policies resulted in almost uniform average allocation delays around 161 ms (P1) and 69 ms (P2) in Figures 7.15a and 7.15c, both do not reflect the tasks' intended base priorities at all. In contrast, using hints manages to reliably signal tasks about their spurious influence and allows them to react adequately. Considering both the average *and* maximal allocation delays, the task priorities are visibly reflected by both collaborative policies P3 and P4 in Figures 7.15e and 7.15g. By following their hints, low priority tasks obviously allowed higher priority tasks to achieve significantly reduced allocation delays. Nevertheless, compared to P1 and P2, not even the lowest prioritized task $t_0$ suffers from significantly increased allocation delays (≈100 ms). Instead, several high priority tasks are very close to the achievable best case of $\delta_{BC}$=226 µs, now. In average, $\delta_{av}$ roughly improved by factors 10.7 (compared to P1) and 4.6 (compared to P2), respectively.

Reducing $s_H := 10 \cdot 32 = 320$ words further increased the task competition and allocation delays to be even more demanding (→ Figure 7.15, right column). While the different task priorities were still not visible for P1 in Figure 7.15b, the sole application of the priority inheritance protocol showed slight improvements for P2 under this heavy load in Figure 7.15d. However, compared to the larger heap we've been using previously, both policies' average allocation delays increased by factor 7 (for P1) and even 23 (for P2), respectively. Since blocking obviously occurs more often now, the hint count increased significantly for the collaborative policies P3 and P4 in Figures 7.15f,h. Yet, these still managed to serve all tasks according to their intended relevance: The two most important tasks still achieved an average allocation delay of $\delta_{av} \approx 1$ ms while even the lowest prioritized ones were still at least as reactive as with the non-collaborative approaches. Again, similar results were also observed for $\delta_{\max}$.

Considering each task's iteration count, Figure 7.16 clearly continues the already discussed effects from Figure 7.15. While the pure PIP once more reflects the task priorities only for the small heap (→ Figure 7.16d), the remaining runs with the non-collaborative policies P1 and P2 still do not seem to consider this metric at all. In contrast, the DynamicHinting concept in both P3 and P4 boosts the overall performance by ≈ 6% (108 → 114) for the large heap, and by ≈ 22% (80.5 → 94) for the small heap where the competition among the tasks is significantly higher. In particular, as the iteration count $s(i)$ increases along with the base priorities, the graphs 7.16e – 7.16h show the expected shape from Eq. (7.12), and indicate the preferred servicing of the more relevant tasks.

Regarding the fragmentation penalty it should first be mentioned that the selected first-fit strategy seems to be a good choice for our CoMem allocator under all policies P1 – P4: The average value $X(P)_{av}$ remained below 1.0 in most cases, and consequently not even one more block of average size would have fit into the remaining free space if the heap partitioning had been optimal. Conversely, the first-fit scheme also profits from our collaboration concept, since higher priority tasks face less fragmentation in general. While the overall average fragmentation penalty decreased by just ≈ 0.1 when comparing P3 and P4 to P1 and P2, the maximum values $X(P)_{\max}$ were reduced notably: For both $s_H = 480$ words and $s_H = 320$ words the task collaboration resulted in an overall improvement of ≈ 50% (2.4 → 1.2) and ≈ 40% (1.5 → 0.9), respectively.

This test bed addressed allocation delays for dynamic memory in case of periodic requests among tasks with varying base priorities. We pointed out that intentionally created dependencies (via broker resources) between blocking and blocked tasks can already reduce these delays in general and account for the specific task priorities in particular. While PIP already showed rudimentary success for heavy load situations, the additional introduction of dynamic hints facilitated explicit task collaboration and boosted this effect significantly. Our novel concept even allowed almost best case delays for high priority tasks while hardly degrading lower priority tasks (→ F10). Up to now explicit RT specifications have intentionally been omitted.

### 7.6.2. Real-World Application under Real-Time Conditions

**Setup.** The second test bed considers a problem from one of our real-world projects: The infrastructure of our ultrasound based indoor vehicle tracking system SNoW Bat (→ Part III) comprises several static anchors as references for the applied localization algorithms. These anchors execute six preemptive tasks for several software components (radio communication, sensor reading, etc). Among these, the two tasks $t_{\mathrm{US}}$ and $t_{\mathrm{RC}}$ are exceptionally memory intensive.

With a maximum frequency of 3 Hz, $t_{\mathrm{US}}$ performs the ultrasound signal detection, recording, and processing for e.g. calculating the time of flight (ToF) between a synchronizing radio packet and the corresponding ultrasound chirp. Each time it uses an on-chip capture compare unit to trigger an ADC/DMA combination (→ Figure 2.4[p25]) which in turn samples the wave signal into a buffer of 4 kB; then, a DSP algorithm operates on the sampled data. In parallel, each node runs our SNoW Ghost (→ Section 8.2) remote node management system which comprises a task $t_{\mathrm{RC}}$ for over-the-air software updates. Compared to other parts of the system this service is rarely used, but as soon as a new firmware image (max. 48 kB for the SNoW[5] nodes) is announced via radio, $t_{\mathrm{RC}}$ requests $n \cdot 256$ B of RAM and successively fills this buffer with image fragments received by radio. As soon as the buffer is full it is transferred to an external flash memory (block size: 256 B); this is repeated until the entire image was received. For optimizing the data rate and energy consumption, $n$ should be chosen as large as possible: This reduces the frequent switching of the SPI-communication between the MCU as bus master and the radio transceiver or flash memory as slaves[52], as well as the spacing delay between successive radio packets. Furthermore, the external flash consumes less time and energy when accessed less frequently but for longer burst writes. In fact we use $n = 20$ and thus require 5 kB for the software image buffer.

From the controller's 10 kB RAM, the OS components and the application modules already require about 4 kB of static memory. The remaining 6 kB will be used as heap space. Thus, the chirp sampling buffer $\check{m}_{\mathrm{US}}$ (4 kB) and the image data buffer $\hat{m}_{\mathrm{RC}}$ (5 kB) must be co-located and dynamically share the available heap memory. In this regard the development of an appropriate software design requires some prior knowledge:

[52]On the SNoW[5] both devices are connected via the same SPI bus (→ Figure B.3[p331]), but need different protocol configurations. Though their synchronization and setup is accomplished automatically through *Smart*OS resources and adequate handler functions (→ Section 4.3.7 and Listing 4.5[p58]), this inevitably consumes some time which can be critical and should be optimized.

1. SNoW Bat. The sporadic missing of an incoming chirp is acceptable in most situations: Since other nodes within the localization system are still available, the failure of some anchor nodes can probably be tolerated by the SNoW Bat localization system. In fact, the node will become available again for later measurements.

2. SNoW Ghost. In contrast, missing even a single code image fragment is highly critical indeed: Though SNoW Ghost supports some recovery strategies, an incomplete image reception would cause avoidable time and energy overhead through expensive retransmissions and additional write accesses to the external data flash. For software updates via (single-hop) radio broadcasts in particular, we cannot wait for (an actually unknown number of) ACK packets after each image fragment transmission, but have to depend on a reliable reception on each first try[53].

To meet these considerations, $t_{\mathrm{RC}}$ imposes a hard upper bound $A(\hat{m}_{\mathrm{RC}})$ for its memory allocation delay, and $t_{\mathrm{US}}$ guarantees to release its possibly allocated memory on-demand within $R(\check{m}_{\mathrm{US}}) \leq A(\hat{m}_{\mathrm{RC}})$. Both real-time requirements can easily be guaranteed with our CoMem approach. While Figure 7.14b shows the tasks' state machines, the Listings 7.6 and 7.7 present the corresponding code for `tUS` and `tRC`. Both tasks make use of DynamicHinting and *Smart*OS exceptions as described before.

Since $t_{\mathrm{US}}$ requires its buffer quite frequently (up to 3 Hz), but tries to limit the overhead for frequent re-initializations, it allocates the memory at task start in Line `tUS:10`, and at the same time it offers and accepts a contract which guarantees the on-demand deallocation within 1.3 ms. Then it configures the DSP process and the DMA controller according to the assigned block base address. Since $t_{\mathrm{RC}}$ is more time-critical, it received a higher base priority $P_{t_{\mathrm{RC}}} > P_{t_{\mathrm{US}}}$. As soon as $t_{\mathrm{RC}}$ requests dynamic memory with a deadline of $\tau = 2$ ms in Line `tRC:19` – which can obviously not be granted immediately but causes an out-of-memory situation since $t_{\mathrm{US}}$ holds its sampling buffer – CoMem immediately passes a hint to $t_{\mathrm{US}}$:

- State 1. If the sampling buffer $\check{m}_{\mathrm{US}}$ is currently not in use, i.e. if $t_{\mathrm{US}}$ waits for an incoming radio packet[54] in Line `tUS:17`, Early Wakeup will resume this self-suspension and an exception is thrown by the task code. This will leave the main loop from Figure 7.14b, transit into the hint processing state, and release $\check{m}_{\mathrm{US}}$ in Line `tUS:28` to serve $t_{\mathrm{RC}}$ quickly.

- States 2 and 3. If $\check{m}_{\mathrm{US}}$ is in use while $t_{\mathrm{US}}$ blocks, the SNoW Bat helper functions from Listing 7.6b initiate an untimely but controlled abortion of the current measurement process: For the sampling function `doSampling()` in particular, this includes the adequate handling of running DMA and ADC operations before its internal exception handler forwards the situation to the task entry function. The DSP function `doDSP()`, which operates at 100% CPU load, receives the hint on the broker through the injected HintHandler `DHH_Memory()` which also throws an exception and directly ends up in the task's catch block.

[53] According to Section 8.2, over-the-air broadcast updates via SNoW Ghost allowed to reprogram the entire system of $n = 45$ nodes in about 22 s compared to $n \cdot 22$ s = 16.5 min for the serial update within the SNoW Bat installation. Energy considerations can also be found there.

[54] These so called Chirp Allocation Vectors (CAV) are e.g. required for node synchronization.

```
1  OS_DECLARE_TASK(tUS, 200, 100);
2
3  OS_TASKENTRY(tUS) {
4
5    /* Prepare for DynamicHinting */
6    Exception_t e;
7    os_DHSetHandler(&DHH_Memory);
8
9    /* initial memory allocation */
10   MCB_t *m̆US = mallocDH(4096, 1300);
11   ... // init buffer, DMA, DSP
12                          |m̆US|   R(m̆US)
13   while (1) {
14     TRY {           // chirp processing
15 receive
16 hint  /* 1. wait for radio or hint */
17       if (waitEvent(&evCAV) == -1)
18         THROW EX_BAT_ABORT_WAIT;
19       /* 2. sample incoming chirp */
20       doSampling();
21       /* 3. process recorded chirp */
22       doDSP();
23
24     } CATCH (e) {  // hint handling
25
26       /* on-demand memory release
27          (e.g. due to tRC:19) */
28       free(&m̆US);
29 /* == tUS WILL BE PREEMPTED HERE ==
30     since tRC waits for the memory  */
31       /* re-request the memory */
32       mallocDM(4096, 1300);
33       ... // reinit buffer, DMA, DSP
34     }                   |m̆US|   R(m̆US)
35   }
36 }
```

**(a)** The simplified SNoW Bat task $t_{US}$ task with $P_{t_{US}} = 100$

```
int doSampling() {
    Exception_t e;
    TRY {
      /* config. & start DMA/ADC/CC */
receive ...
hint  /* wait for CC trigger */
      if (waitEvent(&evCC) == -1)
        THROW EX_BAT_ABORT_SAMPLING;
      /* stop sampling */
      ...
    } CATCH (e) {
      /* stop the DMA and ADC */
      ...
      /* forward exception to tUS:24 */
      THROW EX_BAT_ABORT_SAMPLING;
    }
    return 1;
}


OS_DHHANDLER(DHH_Memory) {
  Resource_t *hint =
    getResourceHint(NULL);
  if (behaviorFunction(...) == 1)
    /* issue exception to tUS:24 */
    THROW EX_BAT_ABORT_DSP;
  else { /* ignore hint */ }
}

int doDSP() {
  /* Do some DSP:
     Runs at 100% CPU load.
     Hint handling is entirely done
receive within the handler. */
hint
  ...
}
```

**(b)** Some SNoW Bat helper functions

**Listing 7.6:** The SNoW Bat ultrasound chirp detection subsystem (→ Figure 7.14b[p181])

Since we apply the priority inheritance protocol for `rCOMEM` and the broker resources, its policy raises $p(t_{US}) := P_{t_{RC}}$ while $t_{RC}$ blocks on its allocation request. During the memory deallocation in Line `tUS:28` the priority inheritance protocol will reduce $p(t_{US}) := P_{t_{US}} < P_{t_{RC}}$ again, and $t_{RC}$ is served and scheduled promptly. As soon as $t_{US}$ is scheduled again it will re-request its sampling buffer for further measurements in Line `tUS:32`, and will receive it as soon as $t_{RC}$ has completed the image reception after Line `tRC:31`. On allocation success, $t_{US}$ leaves the hint handling state from Figure 7.14b, i.e. the catch block in Lines `tUS:24`–`34`, and continues to execute its main loop. While the use of behavior functions to decide between accepting or ignoring a hint is only briefly indicated in the HintHandler code, the underlying self-reflexion concept could also be used anywhere else in the code[55]. Further details can be found in Section 6.5.5.

[55] In our implementation, $t_{US}$ requested the blocked task's allocation timeout $\tau$, and simply ignored the hint if it could not release the memory block in time. While this would lead to an unavoidable deadline violation for the blocked task $t_{RC}$ this situation never occurred in our test beds.

```
// Declare the RT block: size = 5kB, base address = 0, τ = 2ms
COMEM_DECLARE_RT_MCB(m̂RC, 5120, 0, 2000);

OS_DECLARE_TASK(tRC, 100, 250);
OS_TASKENTRY(tRC) {
  while (1) {

    /* Prepare rx buffer and wait for incoming command or data.
       See Sections 8.1 and 8.2 for details. */
    GhostRxHandle->event = &evGhostRX;
    GhostRxHandle->appID = APPID_GHOST;
    if (!SmartNet_ReceivePacket(&GhostRxHandle)) /* error handling */;
    waitEvent(&evGhostRX);

    if (newImageStarts) {
      /* Try to allocate dynamic memory
         (the request will issue a hint towards tUS although
          both tasks do initially not know about each other) */
      if (mallocRT(&m̂RC) != 1) {
        ... // alloc. failure handling
      }
    }

    /* Process received command or image fragment */
    ...

    if (imageReceptionDone) {
      /* Release the previously allocated dynamic memory
         (this will implicitly return the memory to tUS from
         Listing 7.6 which already issued a re-request in tUS:32) */
      free(&m̂RC);
    }
  }
}
```

$|\hat{m}_{RC}|$ $A(\hat{m}_{RC})$

stack size base priority

might pass a hint (→ Listings 7.4 and 7.6)

**(a)** The simplified SNoW Ghost task $t_{RC}$ task with $P_{t_{RC}} = 250$

**Listing 7.7:** The SNoW Bat remote-management subsystem SNoW Ghost (→ Figure 7.14b[p181])

**Timing selection and test bed evaluation.** Table 7.1 shows the results for $t_{RC}$'s worst case allocation delays: If $t_{US}$ would only release its memory after each completed measurement $t_{RC}$ would be blocked for $\delta_{max} \approx 141.3$ ms in the worst case. Using hints from our CoMem approach allows an almost immediate memory handover which is just limited by the already discussed memory manager overhead $\Phi = 0.226$ ms, and the required time for aborting any currently running operation. Static analysis of the handler code revealed $R(\check{m}_{US}) \approx 1.3$ ms and corresponds well to the measured values from Table 7.1. Thus, we declared $\hat{m}_{RC}$ as RT block as described in Section 7.5.4. According to C3[p176] and Eq. (7.6), $A(\hat{m}_{RC}) \geq R(\check{m}_{US}) + \Phi = 1.3$ ms + 0.226 ms also bounds the minimal tolerable delay $\Delta$ between any software image announcement and the first image fragment. Thus, we selected $\Delta = 3.0$ ms for the SNoW Ghost transmission protocol, and specified the hard timeout $\tau = 2.0$ ms $= A(\hat{m}_{RC}) \geq 1.526$ ms for $t_{RC}$ when requesting $\hat{m}_{RC}$. Indeed, we always observed $\delta_{max} \leq 1.351$ ms during our tests, and no timeout violation was detected. Regarding the heap dimensioning, $s_H = |\hat{m}_{RC}| + 0 = 5$ kB was sufficient according to Eq. (7.9).

This test bed showed that CoMem allows tasks to coordinate sporadic and time-critical

memory requirements without explicit and time-consuming inter-task-communication. As in the previous test bed, a blocking task not even needs to know which task it blocks. Our approach provides sufficient information (via hints) and adequate task priorities (via e.g. PIP) to allow tasks a reflective resolution of their blocking influence at runtime. Besides the advantage of time-aware on-demand memory handover in sporadic real-time systems, the sometimes complex termination and reconfiguration of dependent resources (e.g. ADC and DMA peripherals) or subsystems (e.g. DSP algorithms) is limited to a minimum.

## 7.7. Summary and Conclusion

In this chapter, we introduced the novel CoMem approach for dynamic memory management in reactive systems. We showed, that our technique can help to improve and stabilize the overall system performance by optimizing memory allocation delays with respect to the dynamic priorities of preemptive and concurrently executing tasks. As "each allocation policy is motivated by an allocation strategy and implemented by an allocation mechanism" [187], CoMem establishes a new philosophy in the area of embedded real-time systems: It applies the DynamicHinting collaboration paradigm (→ Chapter 6) for on-demand but task-controlled heap reorganization in case of blocking out-of-memory situations. Due to their hard to predict nature dynamic heap operations, like e.g. compaction, are often associated with high effort. Thus we request this process and the corresponding actions only in those cases where moving or releasing a task's memory block(s) would account for the progress of a higher prioritized task.

While various techniques for dynamic heap management have already been developed throughout the past decades, either priority reflexion (→ F10) or reorganization tolerance (→ F9) are often neglected though both properties are of vital importance for concurrent task systems and highly integrated embedded devices with true hardware parallelism regarding their peripherals. If at all, then the treatment of emerging hardware and software dependencies is outsourced to device drivers or to the resource manager as a central intelligence. As an improvement for flexibility and reduced resource manager complexity, CoMem introduces the chance for application self-awareness as already demanded in Section 1.2.1:

| heap memory | $t_{US}$ state | $t_{US}$ hint handling | $\delta_{max}(t_{RC})$ (measured)[1] | $\delta_{max}(t_{RC})$ (computed)[2] |
|---|---|---|---|---|
| free | - | - | 226 µs | –[3] |
| allocated by $t_{US}$ | idle | just free the memory | 1301 µs | 1526 µs |
| allocated by $t_{US}$ | sampling | stop DMA/ADC & free | 1351 µs | 1526 µs |
| allocated by $t_{US}$ | DSP | abort DSP & free | 1342 µs | 1526 µs |
| allocated by $t_{US}$ | sampling/DSP | free after measurement | 141284 µs | –[3] |

[1] The measured best case $\delta_{min}$ = 226 µs, i.e. with only one application task running and immediate allocation success.
[2] Computed through static analysis of the compiled code produced by the mspgcc toolchain version 3.2.3 [292]. Instruction cycles were taken from [279].
[3] The value was not computed since the code analysis would have been to complex.

**Table 7.1.:** Memory allocation delays for $t_{RC}$ within SNoW Bat

Through the establisment of a bidirectional communication link between the memory manager and the application code, CoMem facilitates the on-demand but task-controlled and resource-dependency-aware memory reorganization. By monitoring and analyzing emerging task/memory conflicts at runtime, our concept provides spurious tasks with information about how to specifically reduce the blocking of more relevant tasks. Following these hints allows them to collaborate implicitly without explicit knowledge of each other. The advantage is paid with some additional overhead in terms of broker reservations for each allocated memory block.

As a reflective concept, CoMem also allows each task to decide autonomously between collaborative or egoistic behavior with respect to its current conditions and other tasks' requirements. Thus, we can initially not guarantee any worst case allocation times, since these highly depend on the behavior of the blocking tasks. However, as requested by F2, the possible adaptation to varying system situations commonly results in a good average case performance without prior compile time information about e.g. task-memory relations and priorities. The concept even reduces (bounded) priority inversions reliably, and achieves memory allocation delays which are mainly limited by the pure resource handover overhead. To still support hard allocation timeouts for RT tasks – even if these share the heap with non-RT tasks in open systems(!) – we introduced a special RT heap layout and a contract negotiation mechanism based on a mixture of static and dynamic timing specifications. In this context the combination of a collaborator and an allocator introduces dynamic and static contracts to provide both strictly exclusive allocations and bounded allocation delays for hard real-time requirements.

Apart from the demanded memory protection (M3a) and the locality principle (F4), which mainly depend on hardware support and specific allocator improvements regarding the task and application logic, our concept and reference implementation considers the entire design space and all feature requests from Section 7.2. In particular, individual task base priorities as defined by the system designer are considered carefully to keep each task's progress and reactivity close to its intended relevance.

Based on the reference implementation under *Smart*OS, the presented test beds showed that the effective use of prioritized tasks for creating reactive open systems is quite feasible on small embedded devices like sensor nodes: High priority tasks almost achieved the theoretical best case allocation delays and reactivity while low priority tasks did hardly lose performance. Even if used sparsely, CoMem always proved to be better compared to non-collaborative task operation. Though a well-thought application design still remains elementary, compositional software development is already facilitated. In general, our approach is not necessarily limited to sensor/actuator networking, but may also extend other embedded systems as well.

**Outlook and future work.** Up to now, CoMem was only analyzed for a simple first-fit allocator[56] in combination with a two stage hint generation scheme in case of memory shortages. While stage one considers task priorities only, stage two also adheres to WCAT/WCRT contracts. One option to further improve the overall memory management performance might be the

[56]According to [146] this choice would keep the chance for allocation failures low, and result in good average case performance.

implementation and analysis of different allocators. Depending on the application design and system state, lock-free methods as presented in [43] and pool based approaches might yield various advantages but will certainly lead to entirely different hint generation effects. The off-line evaluation of various policies as proposed in e.g. [278] and [236] might also help to better reflect specific application requirements.

Regarding the collaboration concept itself, which is the main contribution of CoMem, the adaptation of time-utility-functions and behavior functions to more application and system specific factors, like remaining timeouts and allocation frequencies, might also tune the hint acceptance on the blocker side. In this regard, a good balance of cooperation and egoism will definitely contribute to the overall application performance, and programming-by-contract, as already applied for the RT heap layout, should be extended to other relevant requirements.

From the perspective of static code analysis for e.g. software validation purposes it should also be mentioned that the on-demand memory reorganization is likely to cause increased task interaction and the enormous expansion of the application's state space. The consequences for verification techniques like software model checking still remain to be investigated.

Finally, a more distant research area is the application and evaluation of CoMem for shared memory in multi-core systems [90, 222], where blocking may induce hints between the subsystems across the cores. Just like in the software domain, runtime collaboration is hardly known there. Nevertheless, first test benches using a hardware implementation of the DynamicHinting concept on a Xilinx Spartan-3 FPGA [312] revealed promising results when sharing a common data bus among three dynamically prioritized cores of a custom educational CPU [12, 80].

# 8. Selected *Smart*OS Software Design Examples

**Abstract**

For the sake of completeness, this chapter will briefly introduce two software components which are relevant for wirelessly communicating sensor/actuator nodes in general, and for the design of our distributed SNoW Bat localization system throughout the next part of this work in particular: While *Smart*Net is the radio MAC protocol which we do commonly apply for *Smart*OS applications, SNoW Ghost is the common remote maintenance subsystem which can be linked into any *Smart*OS application.

The main advantage of *Smart*Net is its support for cross-node inter-task communication through a scheme which is comparable to TCP ports in TCP/IP based networks, and which reflects the special demands of preemptive multitasking systems.

The main advantage of SNoW Ghost is its support for arbitrary communication interfaces (as long as these provide a *Smart*Net compatible API), and the ability to support both directed software updates through a dedicated sender (push mode) as well as the autonomous cloning of foreign software by each node (pull mode).

While the theoretical background for both concepts is rather lightweight, they demonstrate once more how to implement system services for *Smart*OS.

## 8.1. The *Smart*Net Radio MAC Protocol

*Smart*Net is a lightweight radio media access control (MAC) protocol for the wireless transfer of arbitrary data in the context of tiny embedded systems, and with special focus on the requirements of multitasking environments[1]. On the hardware side it provides groupcasts and broadcasts with optional flooding as well as routed point-to-point connections between two devices. On the software side it supports directed point-to-point transfers between application tasks on both ends. While any transmission is organized in packets of dynamic payload length as illustrated in Figure 8.2[2], the achievable data rate is actually independent from *Smart*Net, but mainly depends on the transceiver device, and the CPU performance or application load respectively. Since most details about *Smart*Net would go beyond the scope of this work, we limit ourselves to just a few selected aspects with particular relevance for the event timestamping and the higher level HashSlot protocol which relies on the *Smart*Net MAC layer as depicted in Figure 8.1.

**Functional protocol overview and *Smart*OS integration.** As we have thoroughly discussed in Part II of this work, a common issue in embedded system design is the dynamic sharing of exclusive resources among concurrently running processes in multitasking environments. While special care must be taken to avoid closely related problems like race conditions or deadlock situations, the manual tuning of software components and system configurations, which might create implicit runtime dependencies between intrinsically independent code, is frequently found, but hard to control and almost impossible to verify. For example, the communication channel between two or more devices is such an exclusively shared resource: It is not only accessed by coexisting devices but also by various software components on both the sender's and the receiver's end. Interleaving the data or superseding the signals on the channel would corrupt the information and must therefore be avoided. While sending data might in many cases be accomplished directly in the context of each task (which would consequently have to keep the communication resource allocated during the entire transfer), waiting for a packet reception would also require the resource to stay allocated and thus starves other tasks when these try to communicate simultaneously. Furthermore the order of incoming packets is commonly unknown in advance, and assigning the bus to an arbitrarily selected task is thus likely to deliver the contained information to the wrong receiver. While discarding the information would be no adequate option in general, forwarding the packet to the "true" receiver would once more involve expensive task coordination (especially in the context of real-time systems).

*Smart*Net addresses these problems by providing a server task $T_{\text{NET}}$ for handling any send and receive request as a system service for other application tasks (→ Figure 8.1). According to the client/server design concept, any application task can simply transfer one or more data buffers (either empty for receiving or filled for sending) along with some associated configuration parameters to the *Smart*Net server task. The API functions for the service request are non-blocking, i.e. they will transfer the buffer, trigger its processing by notifying $T_{\text{NET}}$ about the

[1] In fact, this section also serves as an example on how to implement system services under *Smart*OS.
[2] Up to 234 B payload for the SNoW Bat adaptation using the TI CC1100 radio transceiver.

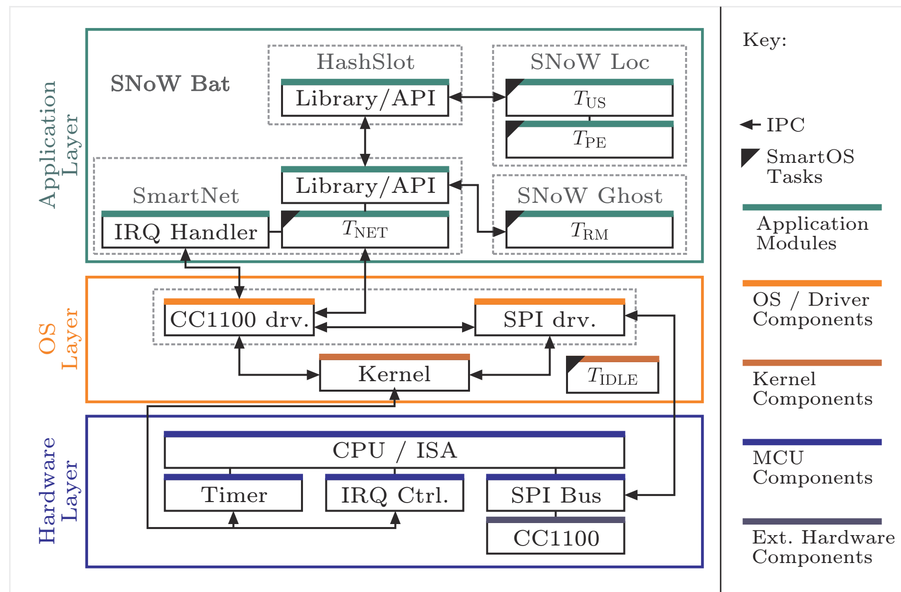



**Figure 8.1.:** SNoW Bat communication subsystems and task interactions for a mobile client (extract)

new request, and return immediately[3]. Conversely, a client will be notified through a request specific *Smart*OS system event as soon as the corresponding rx/tx action has been completed[4]. In the following we refer to the combination of a data buffer and its configuration (including the not necessarily unique notification event) as a *handle*, and distinguish between *rx-handles* for packet reception and *tx-handles* for packet transmission.

Referring to the OS kernel classification from Section 3.3.1 the integration of the radio communication unit shows once more the advantage of the exokernel principle within *Smart*OS: As illustrated in Figure 8.1 the *Smart*Net task and IRQ handler acquire direct access to the exclusively shared radio module and interconnection bus by allocating *Smart*OS resources which coordinate the exclusive access to the hardware and the corresponding device drivers. As soon as the access has been granted – i.e. the resources have been assigned by the kernel – *Smart*Net can access both the CC1100 and the SPI directly at full rights and without any additional delay through further intermediate layers[5]. The general resource allocation process has already been sketched in Listing 4.5[p58]. Of course it must be noted that *Smart*Net itself serves as an intermediate layer for higher level protocols, and, from their perspective, also produces some overhead *and* restricts the use of the wireless communication channel to its specific abilities. Since however *Smart*Net is no inherent part of the operating system it can easily be replaced by compatible alternatives, or even operate concurrently to other MAC protocols within the same

[3]While the API functions will be executed in the context of the client task, the actual packet processing takes place in the context of the *Smart*Net server task. As an important side effect for system design considerations – and comparable to the IRQ handler execution which is always accomplished in the *Smart*OS kernel context – this keeps the stack requirement for the calling client tasks fixed.

[4]If a tx request demanded for an acknowledgment, the event will not be triggered unless the ack has been received.

[5]For portability reasons *Smart*Net nevertheless relies on the API of the corresponding device drivers. In fact, this design concept proved to operate reliably when porting the MCU independent protocol code to the SuperH based multi-radio switch SuperG [210] with four TI CC1100 [281] and two TI CC2520 devices [282].

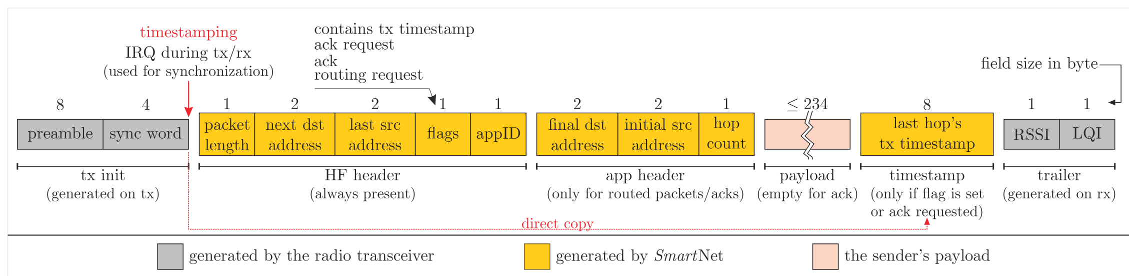



**Figure 8.2.:** The *Smart*Net packet layout (as implemented for the TI CC1100)

application as long as their resource demands are coordinated properly[6].

**Packet structure, transmission, and reception.** *Smart*Net uses a standardized packet structure for each transceiver type and thereby adapts to various hardware specific features. Figure 8.2 shows the layout for the CC1100 radio where we find the packet length and the upper byte of the next hop or destination address at a predefined position to exploit e.g. the hardware support for broadcast detection. Apart from the 16 bit destination address (which is comparable to an IP or MAC address) each *Smart*Net packet header contains an 8 bit appID (which is comparable to a TCP port in TCP/IP based networks) and the payload length to retrieve a suitable data buffer from the available rx-handles at the destination side, and to demultiplex the incoming packets among the potential receiver tasks.

The *Smart*Net server task manages the committed tx-handles in the so called *tx queue*, a priority queue which reflects each client task's active priority (→ Specification S2[p103]) at the instant when the transmission was requested. In case of several pending transmission requests – and in the tradition of the priority inheritance philosophy from Chapter 6 – these will take place in the order of their tasks' relative relevance. Apart from the destination address, appID and routing request, *Smart*Net respects each handle's individual tx strength and retry count limitation, and optionally performs CCA on the specified tx channel. Other techniques for sharing the radio channel with further nodes must be provided through protocols at higher software levels like e.g. Extended Desync [77, 207] or HashSlot from Chapter 12. An example for the integration of the HashSlot library into the task system of SNoW Bat anchor nodes is visualized in Figure 10.4[p229].

In contrast, the committed rx-handles are kept in the so called *rx pool* which is implemented as a linked FIFO list. While receiving is always done on the so called base channel (which can however be changed dynamically) the CC1100 will be put to idle mode to save energy and CPU time in case the routing functionality is disabled *and* the rx pool is empty (i.e. no task would be available anyway to process incoming data). As soon as a packet with the node's address is received the *Smart*Net server task scans the rx pool for an rx-handle with matching appID

[6]While potential conflicts regarding the communication medium must obviously also be considered then, this is not the reliability of the local task or software design.

and sufficient buffer size (i.e. buffer size $\overset{!}{\geq}$ payload length to avoid buffer overflows)[7]. If no such buffer can be found the incoming data is silently discarded and no acknowledgment will be returned. Otherwise the incoming data is transferred to the data buffer which is associated with the rx-handle, and an acknowledgment will be scheduled if requested by the sender through the packet flags[8].

**Task notification.** Since the preemptive *Smart*OS scheduler interleaves the *Smart*Net server task execution with its clients, the support of *non-blocking* send and receive requests results in various advantages which can be exploited in many ways:

1. A client task can achieve progress while it simultaneously “accepts” incoming data or allows transmissions to complete. E.g. a strictly periodic task can convert the initially event-driven *Smart*OS or *Smart*Net philosophy into a conditional packet processing by regularly querying its rx-handle’s event state.

2. A client task needs not check for a handle’s event as long as it does not require access to the handle. This is e.g. useful for messages which can be sent according to a best effort strategy, but which won’t cause critical system states in case of a transmission failure.

3. If a task supplied several rx-handles to the *Smart*Net task (independent from whether the notification events or appIDs are equal or not), any packet reception order will be accepted and notified accordingly.

4. Regarding the performance impact on the remaining system, the *Smart*Net server task largely avoids indirect priority inversions (→ Section 7.4.2) by dynamically adapting its own base priority to the priority of the most important pending handle.

While the hardware resources remain exclusively allocated by the *Smart*Net server task, the access rights regarding the rx/tx-handles must be transferred from the client to the server and vice versa. Thus, it is absolutely essential that a client task leaves the contents of a handle untouched unless it received the notification indicating the completion of a service request. Listing 12.1[p275] gives an example for the use of *Smart*Net in the context of the HashSlot protocol.

**Service request cancellation.** In case a task requires to cancel the reception or transmission of data after the corresponding handle has already been passed to the *Smart*Net server task, *Smart*Net offers a stop function which synchronizes on the rx pool or the tx queue respectively, and removes the handle from the particular data structure. If the indicated handle is currently being processed while the stop function is called, the stop request will be rejected.

[7] If the routing request flag is set and the node is not indicated as the final destination while the hop count is greater than zero, an arbitrary but initially registered routing protocol can be queried for the next hop address. For flexibility reasons *Smart*Net does not offer such a protocol, but an interface for registering a compatible callback function at runtime.

[8] Matching a packet and its ack is done via the included tx timestamp which is unique for each individual sender.

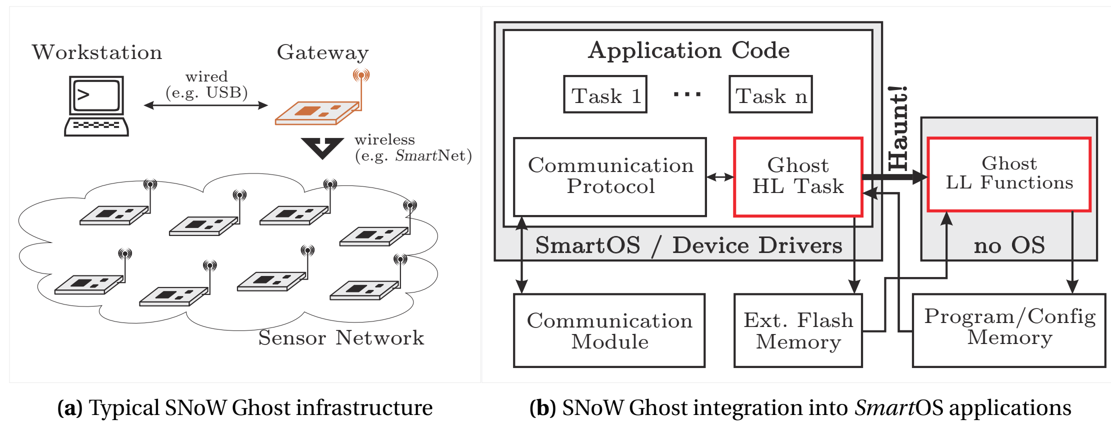
**Timestamping and synchronization** As indicated in Figure 8.2 a local timestamp is taken each time a sync word has been sent or received. According to the *Smart*OS timestamping mechanism from Chapter 5 this is done by the *Smart*Net IRQ handler with a precision of ±0.5 µs and logged into the corresponding handle for optional post-processing through the client task[9]. During a transmission, the just captured timestamp can also be included into the packet and allows e.g. the computation of the nodes' relative time drift at the receiver as well as the scheduling of dependent actions like the emission of an ultrasound chirp in SNoW Bat.

## 8.2. The SNoW Ghost Remote Maintenance System

Wireless sensor/actuator networks commonly consist of a rather large number of more or less widely deployed nodes, and, as already mentioned in Sections 1.2.1 and 3.3.2, the resulting spatial complexity arises extensive setup and maintenance demands for each individual embedded device and for the network as a whole. Thus, in order to achieve a long lifetime for these distributed systems, they are subject to regular maintenance cycles: Besides hardware related issues like the renewal of power supplies or the replacement and attachment of modules, software related actions comprise application code and configuration updates or just remote control capabilities. These allow to integrate new functionality or to fix bugs. Sometimes, there is just the need to reset a node or to modify its configuration for changing its behavior or individual role in the overall system.

The problem's scope and intensity differs according to the evolution stage of the system. During software development, frequent updates for few nodes within a test bed can be expected. The frequency diminishes rapidly with the final release, but then affects significantly more nodes within the original environment. The conventional method to gain direct physical access to the nodes and to update them by means of mobile computers and debug-interfaces is safe and secure indeed, but also annoying and time-consuming; sometimes it is even extremely complex and expensive, or just impossible.

Therefore it is no surprise that code dissemination protocols have received wide attention within the WSN community – where, at times, whole swarms of mobile sensors must be re-programmed [76]. An early survey can be found in [121]. Even some operating systems like Contiki [85], MantisOS [46], and SOS [120] as well as various middleware concepts like Impala [182, 183] and FiGaRo [214] facilitate or provide native support for complete or partial reprogramming. Solutions for the special case of TinyOS were proposed in [215] (Dynamic TinyOS) and [197] (FlexCup). More general approaches like Molecule [320], Typhoon [178], or Deluge [131] provide dynamic reconfiguration schemes, or use well-known tools like rSync [145] or diff [160] to achieve the central goal of energy efficiency [241] by reducing the amount of memory which must be rewritten. According to [84] the virtual machine concept (as e.g. implemented by Maté [173]) is just perfect in this regard: While VM code is often smaller than native code, the




**(a)** Typical SNoW Ghost infrastructure **(b)** SNoW Ghost integration into *Smart*OS applications

**Figure 8.3.:** SNoW Ghost in action

communication cost is reduced (at increased interpretation cost).

From our viewpoint of reactive sensor/actuator systems however, VM concepts steal too much real-time performance, and partial software updates in general are hard to control regarding the compositionality aspects from both Chapters 6 and 7[10]. Apart, as discussed in Section 12.1, a common disadvantage of most presented approaches is that they require their own customized communication or routing protocol. In our opinion these problems are unacceptable, since remote maintenance is important indeed, but it should neither define the actual application's communication nor affect the overall system performance.

Thus SNoW Ghost was designed to interface any already available communication protocol with a compatible API (like e.g. *Smart*Net) and needs no modifications of the actual application. Instead, it aims on resource-aware integration into any software, and accomplishes software and configuration modifications by transferring complete firmware or configuration images.

## 8.2.1. Concept and Operation

Figure 8.3a shows a typical infrastructure for the SNoW Ghost remote maintenance system: In this case, data is transferred from a workstation computer to a dedicated gateway node which in turn creates protocol compatible data packets and transmits them to the destination nodes. Depending on this protocol, both unicasts and broadcasts including routing are possible. If supported, groupcasts offer the special possibility to modify software in a role specific manner simultaneously for certain subsets of nodes.

Implemented as add-on for the seamless integration into any *Smart*OS based application, the SNoW Ghost subsystem performs three central operation steps:

[9]According to [218] and our own observations [207] there is a hardware dependent but virtually constant delay of about 20 µs ± $\frac{1}{2}$ µs between the two sync word IRQs at the sender and the receiver. This delay is directly compensated by our CC1100 low-level driver and invisible for the *Smart*Net protocol implementation.

[10]While e.g. dynamic address resolution through jump tables are just an obvious performance problem, we must also not forget the inevitable ROM fragmentation which results from successive code modifications and eventually requires severe management and energy effort (→ Table 2.2[p27] for example values).

1. receive and check command and data packets for integrity
2. execute commands, and buffer image or configuration data into an external flash memory[11]
3. update affected memory blocks during the so called *haunting* process

According to Figure 8.3b it is therefore organized in two modules: While steps 1 and 2 are executed during regular node operation (high-level task), the operating system along with the entire application is stopped for step 3 (low-level functions). Since *Smart*OS is fully preemptive, it is sufficient to link the SNoW Ghost modules into the actual application.

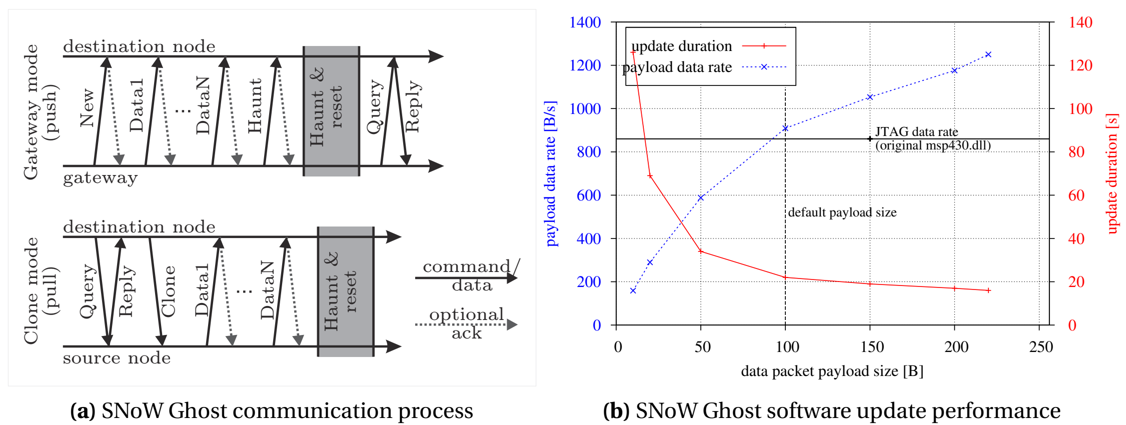
Though the most central SNoW Ghost operation details were already discussed in the context of dynamic memory management and resource sharing in Chapter 7, we'll revert once more to the implementation of the high-level task $t_{\mathrm{RC}}$ from Listing 7.7[p189] (Lines 10 – 13): To be always available for remote commands, $t_{\mathrm{RC}}$ transfers an rx-handle to *Smart*Net from Section 8.1, and waits for the associated event to indicate a packet reception on the configured application ID. This way it runs "in parallel" to each node's individual software, and interleaves its execution just when required. Depending on its current state, it accepts five basic commands:

1. **New** initiates a new image transmission and assigns an ID[12].
2. **Data** packets contain successively numbered image fragments for a given ID.
3. **Haunt** initiates the update process for any previously received image ID.
4. **Reset** simply restarts a node.
5. **Query** requests hardware, software, and runtime related information about the node[13].
6. **Clone** requests the currently running application code from a node.

Figure 8.4a shows examples for remote software updating via a gateway node (push mode) or via cloning (pull mode): By first sending a query, a node can check its neighborhood for available software types and versions. Equipped with an initial firmware containing the SNoW Ghost subsystem and the knowledge about its future role within the WSN (e.g. anchor or client within the SNoW Bat localization system) newly deployed nodes can autonomously integrate themselves into a running installation by requesting the appropriate software from surrounding nodes. Likewise, nodes can stay up to date by observing their environment for newer versions of their own software. This desired virus like spreading is particularly useful for very large networks where the individual handling of each single node would become too complex. Of course, the network protocol must remain compatible between the firmware versions.

Though safety, security, and reliability are truly relevant factors regarding the communication vulnerability and the update process we won't go into detail here but refer to [29] instead.

[11] External buffering is necessary at least for those MCUs with less RAM than ROM, and for storing several images.
[12] Various preconditions like e.g. the CPU type will be checked first to avoid the accidental installation of incompatible software or configuration which would render the node inoperable.
[13] This includes, inter alia, the application type, version, build number, CPU type, and uptime. The information can e.g. be used to check for a successful update or to identify a node as potential source for cloning.




**(a)** SNoW Ghost communication process

**(b)** SNoW Ghost software update performance

**Figure 8.4.:** SNoW Ghost update flow and performance evaluation

### 8.2.2. Resource Demands and Performance Evaluation

Keeping in mind that software remote maintenance subsystems must stay active for the entire system runtime (although the fraction of their active operation time is relatively small), the complexity and permanent resource requirements (CPU, RAM, ROM) must be adjusted carefully to the underlying embedded system, and a well-balanced cost-benefit-ratio must be found.

Apart from the demand for a 5 kB buffer when receiving a software image – which is allocated as dynamic memory to keep this valuable resource available for other tasks whenever possible – our implementation takes about 1 kB of ROM, and 200 B (100 words) of static RAM for the task stack including a buffer for the data packet payload (→ Listing 7.7[p189]). As Figure 8.4b shows, the latter has significant impact on the data dissemination time. Thus, we commonly chose image fragment sizes of 100 B as a trade-off between speed and memory consumption.

For performance evaluation under real-world conditions, we used the SNoW Bat indoor localization system from Figure 10.1[p225] where reprogramming *all* 45 static anchor nodes (i.e. the nodes at the ceiling) simultaneously through an image groupcast in push mode commonly took the same time as reprogramming a single one (≈22 s). In comparison, manual flashing of the same WSN installation successively node by node via e.g. JTAG (and a student who didn't manage to run away quickly) took about 1 hour and more. Apart, it demanded for a mobile laptop, meant annoying cabling, and offered no option to efficiently query the software state and version of each node just for the sake of information retrieval and consistency checking. In fact, the achieved speed up for the SNoW Bat development process was the initial main motivation for implementing the SNoW Ghost system at all – and a truly appreciated help in the end.

## 8.3. Summary

This chapter introduced the *Smart*Net wireless MAC protocol as well as the SNoW Ghost remote maintenance system as representative examples for designing system services under *Smart*OS. The opportunities and pitfalls of the multitasking aspect were addressed in particular, and a

client/server model was proposed to help avoiding various related runtime conflicts among the client tasks when accessing shared services. Throughout the next part of this work, both software subsystems will be applied intensively: While *Smart*Net will form the basis for the wireless data aggregation protocol HashSlot in Chapter 12, SNoW Ghost will show to be a great advantage for e.g. deploying node software within a real-world SNoW Bat installation as presented in Chapter 14.

# Part III.

# The Design of Time-Critical Sensor/Actuator Networks: Indoor Localization Concepts

"The biggest difference between time and space is that you can't reuse time."

---

*(Merrick Furst, computer scientist)*

# 9. Localization Systems

## 9.1. Introduction

Living creatures are instinctively able to determine the location of other creatures and objects in their environment, and consciously or unconsciously use the gathered information to adapt their behavior in regard to numerous aspects. Nature brought forth a large variety of senses which would allow them to detect the close presence by means of characteristic indicators (e.g. through touch, smell, and taste), and to perceive the direction and extent of spatially distant events (e.g. through sight and audition). Though all kinds of organisms developed their senses to different degrees, they are always distinctively applied depending on the current situation. This natural *situation awareness* allows for instance the tracking and tracing of predators or potential prey to launch a specific attack or an escape. Thus, the conscious or unconscious three dimensional localization of objects within the various habitats is already fundamental for the existence of life on earth. While the required and achieved accuracy and precision[1] of these innate localization abilities mainly depends on the size of the perceiving creature and the objects it has to deal with, it is also limited to relative directions and distances. In consequence, the gained information is at first only useful for each individual being. Therefore, some species try to preserve, enrich, and accumulate this information, and make it available to conspecifics by choosing appropriate descriptions and attributes for locations or places, and by relating these to other information. The involved kind of "*early semantics*" does not only allow to retrieve or avoid special places at a later time; advanced intelligence even facilitates the prediction of future events and environmental conditions through experience, and allows to make specific appointments. Beyond, location information from different sources can be combined and compared to obtain a higher quality and reliability, and to improve the utility of short-term and long-term actions based on it.

Mankind thus developed and introduced well-defined length units and coordinate systems to divide the natural environment into artificially created but more convenient and comparable subspaces. Thereby it became possible to achieve almost any resolution for the location specification and position estimation process – at least in theory: Under realistic conditions the achieved precision always depends on accurate measurement techniques (compared to natural senses) and sophisticated mathematical methods (rather than the mere instinct). Both prerequisites must be satisfied to reliably convert the original and commonly erroneous sensor information – as previously obtained from the environment – into meaningful position information. In the age of automatic signal and data processing the temporal performance (i.e. the

[1] See Figure 9.6 for a distinction of these terms.

computation time) as well as the practical complexity of the involved processes (as e.g. expressed through memory demands) is yet another problem.

Years of research in the field of automated and computer aided *location estimation*, *position estimation* and *tracking* has resulted in a large variety of systems with very different features, advantages, and problems – and there are still more to come. Depending on the applied measurement system some of the provided services are available virtually always, everywhere, and for everyone (e.g. outdoor navigation systems with an accuracy of a few meters); others are highly specialized for certain applications (e.g. surface analysis systems with sub-millimeter precision), or explicitly designed for use inside of buildings (e.g. for emergency systems and person tracking). Thereby, it is always essential to distinguish between the two fundamental terms *location* and *position* which we will use as follows:

---

**Definition III.1: Position and Location**

*Position* or *position estimation* refers to the *numerical* specification or determination of a physical position $P$ within a specific coordinate system[2]. Examples are the geographical coordinates $P$ =(29°58'45.05"N, 31°08'03.10"O) within the World Geodetic System WGS84 [135], or $P = \{1, 1, 1\}$ in any three dimensional reference system with currently not specified origin. In general, numerical position information is mainly used for the well-defined and automatically processable attribution of both stationary and mobile objects. The demands on the precision, accuracy, and update frequency are highly application-specific. Common values and requirements range from a few meters down to several micrometers and from seconds down to milliseconds, respectively.

*Location* or *location estimation* refers to the *descriptive* specification or determination of a symbolic position in relation to another (well-known) place. Examples are "next to the Great Pyramid" where the physical location is irrelevant at first, but the place can nevertheless be found. In the area of automatic information processing, location information is mostly used for context-aware services, and therefore the demands in terms of precision, accuracy, and update frequency are rather low. For instance Ward et al. [304] proposed a system which enables employees in an office to find some just required equipment which is currently available and free in their vicinity.

---

Many applications benefit greatly from the ability to determine the position of mobile objects in the environment; some do even depend on it and rely on specially developed sensors and supporting infrastructure, respectively. As we have already discussed in Section 5.1, sensor readings and event notifications are commonly worthless unless tagged or annotated with their date and place of origin. Therefore, the attribution of events and states with meaningful spatial and temporal information is one of the central purposes of any sensor system[3].

[2]The applied coordinate system can itself depend on another one, e.g. to express the relative position of objects.

[3]The development and provisioning of such services might also be a good chance to further establish the WSAN/SANet technology in other areas, as already envisioned in Section 1.2.

While the capturing of precise timestamps was already discussed in Chapter 5, this part of the work addresses the spatially precise and temporally frequent capturing of positions within WSAN environments. Before we go into the details, we will briefly highlight some concrete application scenarios first to illustrate the enormous variety in the underlying objectives and implementations, which has led to the broad acceptance and utilization of such localization or position estimation systems – for example in the private, industrial, medical, and military sector:

**Home and office automation** uses – depending on the required resolution – either locations (e.g. rooms) or positions (e.g. within a room's specific coordinate system) to capture and trace people and objects on their ways through a building, and to adjust for example the temperature, light and sound of the environment according to various factors such as the current use and the time of day. If we abandon the anonymity of the users, even individual preferences can be taken into account, and unauthorized or unidentified persons can be reported. For such systems, mainly optical and radio-based hardware like cameras and RFID tags are used.

**Social monitoring** is a closely related area which often relies on RFID tags and near-field communication in general to capture and trace relative neighborhood information. Observing the proximity between humans or animals allows to deduce potential short-term interactions, and even long-term relationships (like group forming processes and habitat monitoring) can be evaluated through data fusion and data mining processes [14].

**Consumer electronics** has recently started to also record the physical actions of users. A real hype can be observed particularly in games consoles [230, 298]. By locating the user itself or special controllers, which are moved through the room by the players, actions can be captured in detail and transferred to virtual characters. For such motion tracking and gesture recognition systems, mainly optical and acoustic hardware like (stereoscopic) cameras and 3D microphones is used. Additional information is collected from e.g. acceleration sensors.

**Navigation systems** in the military sector already rely on Global Navigation Satellite Systems (GNSS) since 1964[4] [231]. While simple GPS receivers for civil devices already achieve accuracies of about 15 m, certain extensions improve the precision to $1-3$ m (for GPS[5]), and even 10 cm (for Galileo) are planned. Today the purpose of satellite navigation is to guide and support humans and vehicles in passenger traffic, transportation, and military as well as rescue operations. Taking current information into account, e.g. from the Traffic Message Channel (TMC) or by using eCall, can help to dynamically calculate detours, avoid traffic jams, and report accidents.

**Air traffic control and air surveillance** already rely on RADAR systems (RAdio Detection And Ranging) since 1934. Even at high distances of more than 100 km these can locate smaller

[4] Transit 1967-1996, GPS (USA) since 1990, GLONASS (Russia) and Compass (China) since 2011, Galileo (Europe) expected in 2013/14.

[5] in combination with WAAS (Wide Area Augmentation System), EGNOS (European Geostationary Navigation Overlay Service), or MSAS (Multi-Functional Satellite Augmentation System)

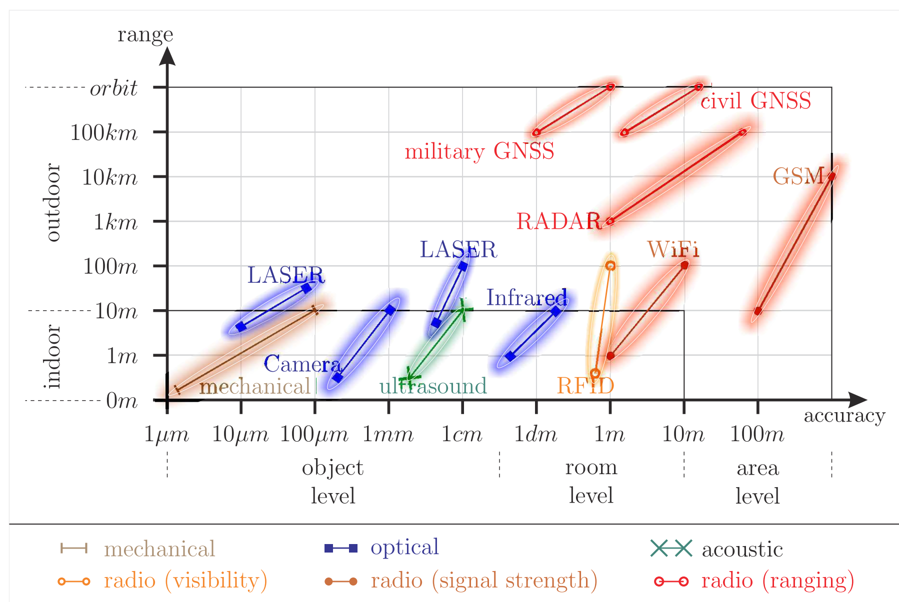



**Figure 9.1.:** Localization systems: Range vs. accuracy (based on [202])

objects with a precision of about 50 m. Here, the detection reliability, velocity estimation, and direction determination is much more relevant than the absolute precision.

**Emergency and rescue systems** can also benefit from location information about humans in critical situations, and about various sources of danger. In combination with detailed maps of buildings and the environment, information can be combined to predict mass panics and the propagation of threats, and to adapt escape routes, guide rescue teams, and to coordinate counter-measures in general. While special infrastructures are commonly required to obtain a sufficient spatial resolution inside of buildings, outdoor teams can once more revert to GNSS, or even cell phone tracking: Location based services (LBS) of mobile operators offer the automatic association of emergency calls with the location of the caller (e.g. according to the E-911 standard [213]).

**Safety and surveillance systems** for goods and equipment often use the radio-based tracking of markers (e.g. RFID tags). These are attached to the products and inventory to be monitored in order to detect their conditions and presence in certain areas (the so called fencing [232]), and to possibly trigger an alarm. While the absolute accuracy is of less importance here, these systems focus on high reliability and robustness against random disturbances and targeted attacks like e.g. spoofing.

**Medical engineering** applies various localization techniques to trace the position of probes inside the human body and to associate the obtained information (e.g. from imaging techniques) with time and location information. While avoiding surgical intervention this non-invasive process allows unclouded investigation and detailed diagnostics. Another application scenario

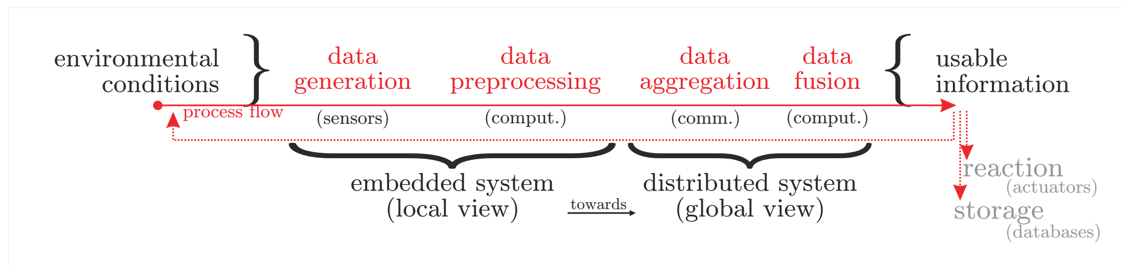



**Figure 9.2.:** The WSN/WSAN process flow: From environmental conditions to usable information

is the monitoring of medical equipment in e.g. operating rooms [66].

**Industrial plants** use position estimation systems to track and control material flows as well as the movement of vehicles, robots, and tools. High update rates allow many of the served systems to even gain a certain autonomy regarding their mobility (→ Def. I.2[p4]). Since accuracies in the centimeter range are commonly sufficient for dynamic path control, tightly placed induction loops or ultrasound based systems are widely used. The latter depend on statically deployed anchors which must be calibrated as precisely as possible during their installation to ensure a high quality and reliability for the actual localization process[6]. In this context, the automatic calibration of machines and tools in general must be mentioned: As a result of extensive research, accuracies up to a few micrometers can be achieved nowadays. If this procedure must be non-contact, optical or laser methods are commonly used, otherwise mechanical measuring systems are a potential fall back.

As we have seen from our rather small selection of real-world scenarios, many services and applications depend on the precise knowledge about the spatial location or position of (mobile) creatures and objects in a more or less wide environment. Based on [202] Figure 9.1 gives an extensive overview regarding the range and accuracy of typical techniques: Where large scale radio ranging systems are designed for coarse but globally available outdoor navigation, optical and acoustic systems are more suitable for indoor applications and commonly achieve a significantly higher precision.

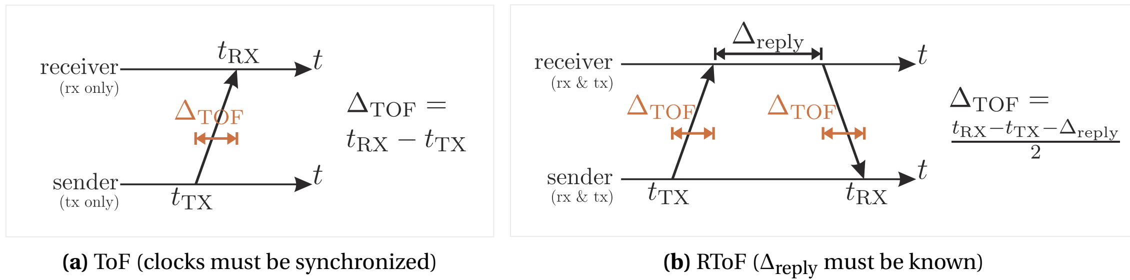
## 9.2. Motivation, Requirements, and Evaluation Metrics

Despite of the significant differences in their concrete realization as well as in the underlying hardware and software architectures, all localization systems share a common goal:

*Extract spatial information from observed environmental conditions.*

Conceived and created by humans, this ambition is quite obvious and only natural since it helps to support and extend the innate senses for an improved perception, situation awareness, and reaction capability. In fact, even the technical accomplishment is comparable and seems to

[6]Related research which was conducted as part of this work – but is not included in detail – can be found in [257].




**(a)** ToF (clocks must be synchronized) **(b)** RToF ($\Delta_{reply}$ must be known)

**Figure 9.3.:** Measuring distances based on signal propagation delays

be "nature inspired" as it follows an almost "unified" process flow as depicted in Figure 9.2: Generate and (pre-)process data first, collect it from several sources, and fuse it thereafter. What will finally be done with the obtained information – e.g. coordinate reactions immediately or store it for later use – is out of scope for now, and depends on the application scenario.

Remembering the basic paradigms, objectives, and the design space of networked sensors – and localization systems are nothing else indeed(!) – as discussed in the introductory Chapter 1 of this work, the relevant aspects can be separated into three main points of view. These will be discussed next.

### 9.2.1. Points of View

When seen from a **physical/technical point of view**, each localization process starts with the collection of sensor readings from the environment (*data generation* / *data aggregation*). As Figure 9.5 depicts, the initial raw values describe e.g. signal strengths and propagation delays, transmission and reception angles, as well as cell or proximity relations between the participating sensor nodes. In turn, the obtained sensor values can be converted into distance, direction or neighborhood information, and will finally be (pre-)processed by locally executed algorithms (*data preprocessing*). A comprehensive overview on wireless measurement techniques is given in e.g. [300].

Since radio RSSI[7] (Received Signal Strength Indicator) and sound signal ToF / RToF (Time of Flight / Roundtrip Time of Flight) measurements are very common in WSN based position estimation, and since SNoW Bat in particular relies on ultrasound metering after we found RSSI to be too "unsteady", we'll give just two examples for the conversion between raw sensor values and distance information:

**ToF / RToF:** If both the time of flight $\Delta_{TOF}$ and the velocity $v$ of any signal is known, then its traveled distance $d$ computes according to Figure 9.3a as

$$d = \frac{v}{\Delta_{TOF}}. \tag{9.1}$$

The precision and accuracy of $d$ depends on the measurement noise, i.e. on the knowledge

[7] [181] gives a detailed survey/overview on RSSI based systems like MoteTrack [186] and RADAR [18].

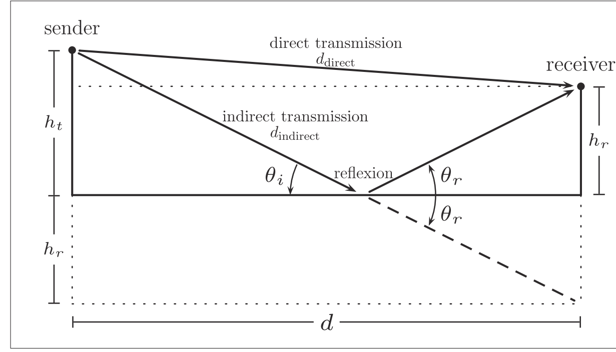



**Figure 9.4.:** The radio signal reflection model

about the signal propagation speed (which depends on various environmental factors) as well as on the timing and signal detection capability of the used hardware (e.g. temporal resolution, reactivity and DSP). Another critical factor is the synchronization between the signal's sender and its receiver(s). Measuring the RToF as shown in Figure 9.3b can be an option to avoid explicit synchronization. However, this technique is quite critical when measuring distances to several anchors simultaneously: Similar to the data aggregation problem from Chapter 12, returning the signal to the sender must be properly coordinated to avoid interference and collisions which would otherwise lead to the loss of information.

**RSSI** If the radio transmission power $P_t$, the antenna gains $G_t$ and $G_r$ of the sender and the receiver, and the radio wave length $\lambda$ is known, then the signal strength $P_r$ at the receiver can be captured and allows the estimation of the distance $d$ according to the path loss theory as described by the free space propagation model (FSPM) and the Friis transmission equation [239]:

$$P_r(d) = \frac{P_t}{d^2} \cdot \frac{G_t G_r \lambda^2}{(4\pi)^2} = \frac{P_t}{d^2} \cdot c \quad \Leftrightarrow \quad d = \sqrt{\frac{P_t}{P_r} \cdot c} \tag{9.2}$$

Unfortunately, this is only true under the precise knowledge of $c$, and for the optimal LoS (Line of Sight) case without *any* reflections. Under real-world conditions, the imprecision of the estimated distance $d$ is significantly influenced by multi-path effects and signal superposition. In fact, a single reflection as depicted in Figure 9.4 and described in [167, 239] already affects the received signal strength $P_{r,g}$ as follows:

$$P_{r,g}(d) = P_r(d_{\text{direct}}) + \cos(\Delta_\varphi) \cdot P_r(d_{\text{indirect}}) \cdot \Gamma_\nu \tag{9.3}$$

Here $\Delta_\varphi$ represents the phase shift between the reflected and the unreflected signal, and $\Gamma_\nu$ defines the material properties of the reflector. Since indoor environments suffer from many unpredictable reflections, and real antennas are never perfectly tuned, the achievable precision is highly variable and not sufficient for many applications. In [38] we give a detailed analysis on using RSSI distance measurements in indoor and outdoor environments, and reveal why we finally switched to ultrasound for SNoW Bat.

When seen from a **theoretical/mathematical point of view**, the localization process can be described as a *data fusion* function $\psi$ which computes a position estimation $\tilde{p}$ from measured or otherwise collected environmental information $\tilde{I}_{\text{env}}$ (like distances, anchor positions, etc.), and an optional prediction function $\theta$ based on prior estimations and the current time $\tilde{t}_{\text{now}}$:

$$\psi : \underbrace{\tilde{I}_{\text{env}}}_{\text{new measurements}} \times \underbrace{\left( \underbrace{H}_{\text{history}} \times \underbrace{\tilde{t}_{\text{now}}}_{\text{current time}} \right)}_{\text{prediction } \theta \text{ (optional)}} \rightarrow \underbrace{\tilde{p}}_{\text{position estimation}} \tag{9.4}$$

While the prediction is commonly used for tracking purposes, it may describe a single potential location as well as an expectation or exclusion area for filtering. It may even provide a so called *quality indicator* for its expected reliability. Thus, $\theta$ requires some historic position values $p^*$ in association with their corresponding timestamps $t^*$ and further annotations $a^*$ (if available). In fact, a trail of position estimations can be constructed over time and be used as history $H$:

$$H := \left\{ \underbrace{(p_0^*, t_0^*, a_0^*)}_{h_0}, \ldots, \underbrace{(p_{|H|-1}^*, t_{|H|-1}^*, a_{|H|-1}^*)}_{h_{|H|-1}} \right\} \text{ with } t_0^* \leq \ldots \leq t_{|H|-1}^* \tag{9.5}$$

Obviously, each position estimation relies on more or less prefect input values, and in turn also produces imprecise and inaccurate results. Thus, one goal of any localization algorithm is to compensate for measurement noise, errors, and failure. The generation of largely error-free position estimations is mandatory for almost any application to be served. Available methods use proximity analysis (like e.g. neighborhood relations), scene analysis (like e.g. fingerprinting or maps) and geometric analysis (like e.g. lateration and angulation), and often rely on various mathematical models (like e.g. filters or maximum likelihood estimators).

Finally, the **algorithmic/computational point of view** considers how to integrate the physical measurements and mathematical methods into an application. In this context data aggregation, processing, and fusion will once more turn out as central aspects. While the first one is mainly considered by an appropriate communication or network protocol, the latter necessitate both a reasonable filtering of available information as well as optimizations regarding the CPU and memory requirements. Considering each single node and the entire distributed system as a whole, these aspects will seriously affect the scalability, reliability, quality, energy requirements, and update frequency of the localization service. In most cases, a trade-off must be found (either statically or dynamically) with respect to the system situation and the actual application requirements. In fact, two main approaches are generally distinguished:

- In *centralized approaches* the clients are relieved from estimating their own position or location. Instead, dedicated systems are available as part of the infrastructure. These execute the localization algorithms and forward the results to the particular clients. While these central systems are commonly more powerful than ordinary sensor nodes, they

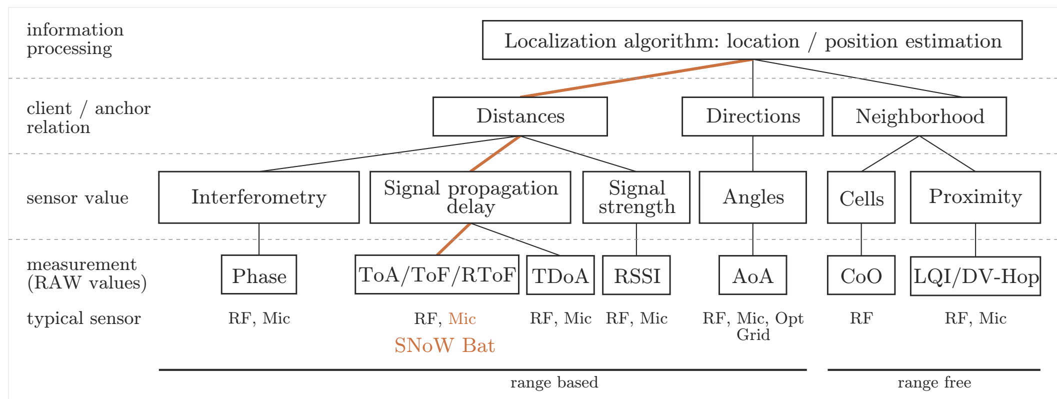



**Figure 9.5.:** Wireless measurement techniques for localization systems

might even have access to global information. Thus, they can generate an extensive view of the world, and process much more information by executing more complex computations (e.g. map stitching [168], multidimensional scaling (MDS) [263], and semidefinite programming (SDP) [48, 161, 265]).

- In *decentralized approaches* the clients localize themselves, and do not rely on supplementary hardware for computation. While this causes higher CPU and memory load on the sensor nodes, they stay autonomous in their operation, and avoid (resource) bottlenecks as well as a single point of failure. Also, depending on the remaining system specification, there is commonly more anonymity. Common algorithms for this use case – like trilateration, multilateration, (weighted) centroid methods, MinMax, and some hybrids – will briefly be compared to our pVoted approach in Chapter 13.

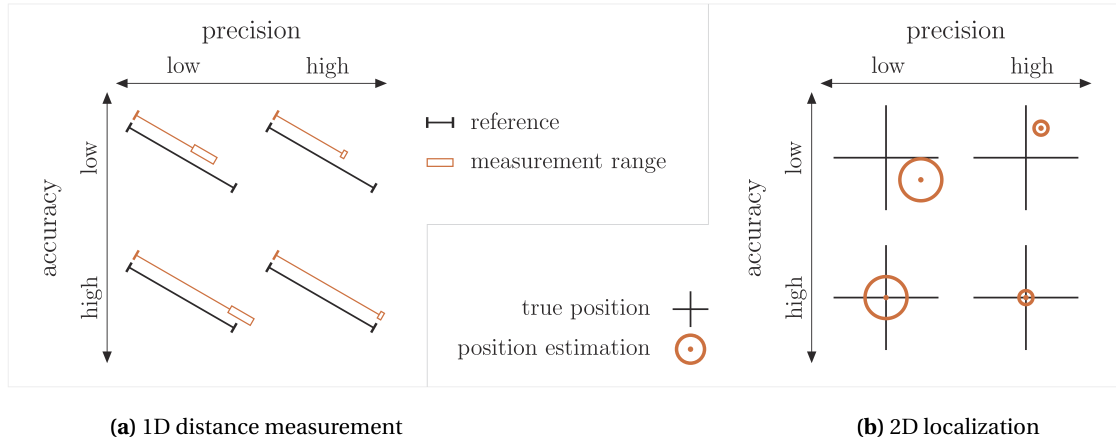
Having described the various points of view on localization and tracking systems, we can summarize the demands regarding the WSAN optimization problem from Definition I.5 as:

*Optimize the data generation, communication, and data fusion cost to jointly save resources while providing a well-balanced temporal performance along with spatial accuracy and precision.*

To achieve this long-term goal, we'll next recommend concrete evaluation metrics and feature requests regarding the design and implementation of such systems.

### 9.2.2. Evaluation Metrics

Besides the request for more or less relevant and "nice to have" features, the design of (indoor) localization systems also involves the definition of feasible target specifications to finally comply with. These depend on the application context and the customer's individual requirements on the one hand, but they are also limited by environmental constraints and physical laws on the other hand. Thus, the implementation and configuration parameters for the underlying hardware, software, and networking subsystems must be selected for both efficiency and interoperability, and tested carefully according to certain metrics.




**(a)** 1D distance measurement

**(b)** 2D localization

**Figure 9.6.:** Accuracy vs. precision

Although most of these metrics can directly be derived from the just presented design space, we nevertheless want to emphasize some selected issues, which – according to our experience – are indispensable for rendering a localization and tracking system suitable for practice. More specific metrics will be considered where appropriate throughout the next chapters.

**Spatial performance: Precision and accuracy.** In literature [196, 216, 224] as well as for real installations (→ Table 9.1) the absolute *accuracy* is probably the most central concern during both the initial specification and the concluding evaluation process. It is commonly defined as the *mean error* (ME) or *mean square error* (MSE) in the Euclidean distance between the true positions $p'_{m_i}$ of a node $m$ to be localized and its estimated positions $\tilde{p}_{m_i}$ over $n$ iterations:

$$e_L = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \left\| p'_{m_i} - \tilde{p}_{m_i} \right\|^1 \qquad \text{or} \qquad e_L = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \left\| p'_{m_i} - \tilde{p}_{m_i} \right\|^2 \tag{9.6}$$

In contrast, the *precision* is an indicator for quantifying the numerical stability of the estimated position coordinates when neither anchors nor the clients have moved during several localizations. Figure 9.6b illustrates the differences between those two terms for the 2D case; it behaves analogously for the 3D case.

Since the just mentioned accuracy and precision of each estimated position is always influenced by the quality of the previously acquired measurement data, the metric from Eq. (9.6) can easily be adapted for e.g. measured and true distances $d_{m_i,a_j}$ between two nodes $m_i$ and $a_j$ over $n$ measurements:

$$e_d = \frac{1}{n} \cdot \sum_{k=0}^{n-1} (d'_{m_i,a_j} - \tilde{d}_{m_i,a_j})^1 \qquad \text{or} \qquad e_d = \frac{1}{n} \cdot \sum_{k=0}^{n-1} (d'_{m_i,a_j} - \tilde{d}_{m_i,a_j})^2 \tag{9.7}$$

Also comparable, the measurement precision describes the scattering at constant distances as depicted in Figure 9.6a.

**Energy consumption.** In this regard we once more refer to Figure 9.2, and consider the cost for data generation, preprocessing, aggregation, and fusion as relevant factors. As mentioned in Section 1.1 energetic autarky plays an important role in the design of autonomous embedded systems. With regard to WSAN based localization systems the energy consumption is mainly defined by three major subsystems: While today's MCUs are designed for energy efficiency anyway, the most promising potential for further optimization can be found in the measurement hardware and the (wireless) communication. Since the energy consumption increases with the localization rate $f_L$, the communication stages should be kept short in general, and radio retransmissions for e.g. error recovery should be avoided whenever possible. Since indoor systems often rely on a fixed infrastructure, our energy consideration will mainly relate to the mobile nodes.

**Localization frequency and multi client support.** According to e.g. [5, 261] the achievable update rate $f_L$ and various related communication issues must also be considered. This is particularly relevant for the realization of tracking and control systems with a temporally and spatially high resolution. Our experiments in [191][8] have shown that for tracking a mobile target moving at rather low speed of about 10 cm/s (i.e. 0.36 km/h) a localization frequency $f_L \approx 2$ Hz is already required to precisely steer a tiny LEGO™ vehicle with a maximal deviation of about 10 cm along a predefined path – exclusively based on position estimations. In most cases it is argued that the applied data fusion algorithms or CPUs are the limiting factors, and must therefore simply be replaced by sufficiently fast alternatives. We will show that in both centralized and decentralized systems, the data aggregation stage can quickly become a comparably relevant issue which offers an enormous potential for optimization.

A closely related metric is the number of clients which can be supported simultaneously, i.e. served with a "sufficiently" high localization frequency which in turn can depend on the application, on each single node, or even on changing demands. A discussion as well as a system optimized for numerous devices can be found in [261].

**Deployment, service coverage and environmental integration.** Another issue is to efficiently and reasonably dimension and install the localization system in order to guarantee a reliable service coverage at minimal cost and impact on the surrounding or on other systems. In this regard we have to minimize the number of anchor nodes while keeping the service quality constant at any position. While this would reduce environmental pollution caused by the emission of radio and ultrasound signals and simplify the anchor calibration process [256][9] (e.g. during the deployment or at runtime), it is even likely to attenuate the competition on the radio channel and the system's overall energy consumption.

[8] Diploma thesis conducted in conjunction with this work.
[9] Diploma thesis conducted in conjunction with this work.

### 9.2.3. The Design Space

A localization system's design space comprises four main dimensions (LS1 – LS4), and the need for scalability (LS5) regarding any supported feature[10]. Note, that several requirements impose different challenges when seen from an anchor's (i.e. a node or device in the installation's infrastructure) or a client's (i.e. a node or device using the localization service) view:

LS1 SPATIAL AND TEMPORAL PERFORMANCE. *Maximize precision and accuracy while minimizing runtime:*

a) Reliably obtain an adequate set of environmental information, and apply data fusion algorithms which are robust against false and noisy measurements.

b) Reduce the temporal overhead for measurements, inter-node communication, and CPU activity.

LS2 RESOURCE EFFICIENCY. *Optimize the overall system regarding both the local node and the global network point of view:*

a) At the distributed systems level: *Minimize the required infrastructure* to simplify its deployment and maintenance. Also account for energy efficiency which is often defined by the communication or data fusion cost.

b) At the embedded systems level: *Minimize the hardware and software overhead* to reduce production costs as well as operational resource requirements (e.g. for compositional reasons as discussed in Part II of this work).

LS3 SAFETY AND SECURITY. *Account for a reliable system operation:*

a) Ensure a sufficiently wide service coverage – even in critical areas (e.g. borders) – through a reasonably dimensioned infrastructure and efficient node interaction.

b) Provide failure-safety through a careful hardware and application design.

c) Preserve client anonymity where required, and protect inter-node communication against attacks like spoofing or overhearing[11].

LS4 USABILITY. *Provide useful functionality while avoiding trivializing assumptions:* Make position estimation feasible without imposing unreasonable restrictions or prerequisites on the hardware, software or natural laws. Aim for robustness regarding badly calibrated anchor positions, noisy sensor data, or unreliable communication.

LS5 SCALABILITY. *Maximize and preserve scalability* regarding both the number of anchor and concurrently operating client nodes as well as the demanded localization frequency.

[10] If you feel like having a déjà vu when reading these lines, then this is partially true! In fact, the design space we have been using for the resource and memory managers in Chapters 6 and 7 comprised the same "main dimensions". Though their peculiarity was different in the superordinate context, the most central objectives are closely related and remain almost the same.

[11] This aspect has been omitted entirely within this work. See [7] for a survey.

Since the presented aspects can obviously impose serious conflicts and mutual dependencies, the integration of suitable solutions will commonly demand for a reasonable trade-off. This will also affect the SNoW Bat design considerations, and thus we'll address various strategies for most of them throughout this part of the work.

## 9.3. Related Work

Since a lot of work on the various aspects of WSN based localization systems is available, we won't give a complete overview but refer to literature for general surveys [8, 126, 203, 226], algorithmic aspects [56, 224], tracking issues [289], evaluation metrics [300], security concerns [261], and specific challenges regarding indoor environments [202, 259] instead. For this work in particular, we strictly focus on WSN based indoor localization using ultrasound, and omit other systems (using e.g. optical techniques as summarized in [204]) entirely.

Especially in the area of contact-free indoor localization, ultrasound measurements apparently allow for sufficiently "high accuracy ranging, low cost, safety, and good user imperceptibility" [267], and consequently a large variety of such systems does already exist. While Table 9.1 gives a brief comparison of SNoW Bat and some selected approaches, these and other concepts are optimized for various application-specific demands, and can thus be classified with regard to the already depicted points of view from Section 9.2.1.

Apart from the demand to track either persons roughly (❶ – ❻) or objects precisely (❼ – ❾), some localization systems content themselves with determining their clients' positions or locations. Therefore, however, all systems make use of a static infrastructure of pre-installed reference anchors. While some systems allow their arbitrary deployment (❷ – ❹, ❻, ❽), others require a specific pattern to exploit geometrical relations during the data aggregation (❶), or during the actual position or location estimation (❺, ❼, ❾). Independent from the deployment, the spatial anchor calibration must be accomplished manually for most systems (❷ – ❹, ❻, ❽, ❾), while only a few provide an automatic self-calibration scheme with either explicitly triggered operation e.g. during the deployment (❺, ❼), or even a continuous self-observation and self-adaptation (❶) at runtime.

Measuring the spatial distances between the client and several anchors is done via the methods from Section 9.2. Using either an initial or a repeated time synchronization to determine the signal's time of flight, it can be triggered periodically by the environment (❷, ❻) or on-demand by the clients. Apart, either the clients or the anchors can be used as US transmitters or receivers, respectively. Letting the clients emit the signal (❶, ❹, ❺, ❼, ❾) allows for truly simultaneous measurements, but must be coordinated properly to avoid sound interference in case of closely operating clients. Letting the anchors emit the signal (❷, ❸, ❻, ❽) also demands for a proper coordination, but also raises the problem of potentially inconsistent distances in case of moving clients.

The two options for the trigger source and the sound emission direction also exert direct influence on both the privacy aspect and the data aggregation: Regarding the first, some systems do not even attempt to know about the presence of a client but simply provide the service (❷, ❻), others just detect their presence (❹, ❽), and yet others even collect position or identification

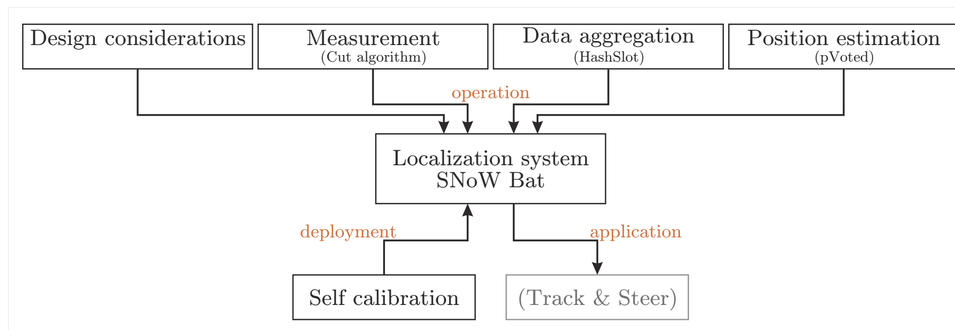



**Figure 9.7.:** The SNoW Bat localization system: An overview on Part III

information (❶, ❺, ❼, ❾). Regarding the latter, collecting the distance information from the anchors is done either wirelessly (❶, ❸, ❹, ❻, ❽) or via wired connections (❺, ❼, ❾); only in one case (❷) the data is collected from wired clients. The question remains who collects the information: As described in Section 9.2.1, the clients represent the data sinks in decentralized systems (❶, ❸, ❹, ❻, ❽). In contrast, centralized systems (❷, ❺, ❼, ❾) employ a central computational unit to fuse the data and forward the final position or location information to the client. Independent from this choice, all approaches apply lateration algorithms, which is quite obvious due to the fact that they use ultrasound to determine distances instead of angles or just the pure presence of clients within the serviced area. Finally, the position accuracy and update frequency is a highly interesting point. Few systems (❷, ❺) apply an ordinary PC (with enormous computational power compared to typical sensor nodes) to achieve a high frequency despite of executing complex data fusion algorithms for high position accuracies. Other systems accept either a lower frequency (if considered at all) or a lower precision (❸, ❹, ❻, ❽, ❾), but content themselves with sensor node hardware.

In order to improve both aspects by also just relying on exceptionally weak, but cheap and energy efficient sensor node hardware (→ Chapter 2), we implemented the entirely novel SNoW Bat system from scratch to thoroughly rethink the manifold design aspects, and to gain a deep understanding of related hardware, software, and network issues. Regarding e.g. energy, real-time, and scalability demands, we consider this step as indispensable for optimizing the sometimes subtle interactions and implications through an efficient co-design.

### 9.3.1. Scope of this Part of the Work

Based on the presented SNoW[5] sensor node and the operating system *Smart*OS from Part II of this work, a real-world installation comprising 45 static anchor nodes allowed us to perform an extensive analysis of some novel approaches (mainly the Cut, HashSlot, and pVoted algorithms for data generation, aggregation, and fusion) under realistic conditions – and to stress test the reliability and usability of the central building blocks.

Though we attach great importance to executing SNoW Bat on real WSN hardware, we also linked the real system to a simulator and visualizer. This allowed us to process real-world data (like e.g. distance measurements) within a partial simulation (e.g. of the localization algorithm),

and helped us to evaluate the capability of various techniques and algorithms at an early stage. At the same time it reduced the comparably complex and time-consuming implementation of embedded software just for testing purposes. We already discussed the pros and cons of simulation and real-world implementations in Section 2.

Figure 9.7 outlines the organization of this part of the work:

**Chapter 10** introduces the general SNoW Bat operation principle, followed by some considerations on the hardware platform, anchor deployment issues, service coverage, and the software architecture. Beyond, we'll investigate some critical algorithmic and physical factors which bound the maximum achievable localization frequency. In this respect, far-reaching decisions regarding compositional software design under real-time conditions and resource sharing demands become once more relevant to obtain sufficiently frequent *and* precise position information for demanding applications like autonomous robot or vehicle steering and path control.

**Chapter 11** covers the ultrasound signal detection and ranging for precise distance measurements. Physical aspects and the impact of using off-the-shelf transducer hardware will be discussed as well as DSP based event detection using real-time timestamping and curve matching.

**Chapter 12** considers the special requirements for fast, reliable, and energy efficient data aggregation at the client. The almost simultaneous arrival of the ultrasound signal at the anchors, and the subsequent emergence of distance information imposes considerable consequences on the wireless communication. The presented collision-free and almost optimal HashSlot protocol is a key feature for SNoW Bat's performance and allows to trade precision against speed.

**Chapter 13** presents the decentralized localization algorithm pVoted. Using prediction and a progressive voting scheme to progressively process the wirelessly received distance information allows to immediately detect and filter (probably) faulty sensor data. This saves valuable RAM and reduces distortions in the final position estimation. The numerical classification of each estimation allows to stop the data fusion process as soon as an adjustable threshold is reached, and provides a trust indicator for applications using the SNoW Bat subsystem as a service.

**Chapter 14** presents a real-world evaluation of a concrete SNoW Bat installation based on the SNoW[5] hardware and the *Smart*OS operating system from Part II of this work. Considering the temporal and spatial performance within a realistic environment, a conclusion on the previously introduced techniques and approaches from Part III will be given as well as an outlook on further research in the area of ultrasound based indoor localization in wireless sensor/actuator networks.

| | ❶ SNoW Bat | ❷ LOSNUS | ❸ DOLPHIN | ❹ Eckert's | ❺ Dragon | ❻ Cricket | ❼ Active Bat | ❽ SmartLOCUS | ❾ Beep |
|---|---|---|---|---|---|---|---|---|---|
| **General issues** | | | | | | | | | |
| references | [27, 35] | [261, 268] | [106, 107] | [92] | [267] | [233, 234] | [303, 304] | [53] | [185] |
| main purpose | obj. tracking | obj. tracking | obj. tracking | obj. tracking | obj. tracking | obj. tracking | person tracking | person tracking | person tracking |
| client privacy[1] | presence / z-coord. | –[5] | ? | presence | position | – | position | presence | pos. / IP address |
| triggering | by client | by environment | ? | by client | by client | by environment | by client | by client | by client |
| tracking support | 3D | 3D | 3D | 3D | 3D | Y (coarse) | 3D | 2D | 3D |
| **Infrastructure** | | | | | | | | | |
| anchor deployment | grid (continuous self-cal. [256]) | arbitrary | arbitrary | arbitrary/grid | grid (triggered self-cal. [324]) | arbitrary | grid (triggered self-cal. [75]) | arbitrary | lattice |
| **Measurement** | | | | | | | | | |
| method | US ToF | US TDoA | US ToF | US ToF | US ToF | US proximity | US ToF | US ToF | audible ToF |
| chirps per loc.[2] | 1 | $\|A\|$ | $\|A_m\|$ | 1 | 1 | $\|A\|$ | 1 | $\|A_m\|$ | 1 |
| range | 10m | ? | 10m | 9m | >2.6m | 10m | 9m | ? | 7.5m |
| time synchronization | each time (RF) | each time (US) | each time (RF) | at startup | each time (RF) | at startup | RF cue | each time (RF) | cont. (802.11) |
| **Communication** | | | | | | | | | |
| RF packets per meas.[2] | $\|A_m\|+1$ | 0 | ? | $\|A\|+1$ | 1 | $\|A\|$ | 1 | $\|A_m\|+1$ | 2 |
| data aggregation | wireless | wired[6] | wireless | wireless | wired | wireless | wired | wireless | wired |
| **Algorithm** | | | | | | | | | |
| localization type | position | position | position | position | position | location sector | position | position | position |
| decentralized | Y | N (Y optional) | Y | Y | N | Y | N | Y | N |
| pos. est. algorithm | lateration | trilat. | multilat. | lateration | fake spot | lateration | multilat. | lateration | multilat. |
| improvement | voting | various | – | Kalman filter | Kalman filter | – | nonlin. regr. | multi-measure | shrinking |
| **Sound signal** | | | | | | | | | |
| sender | client | anchor | anchor | client | client | anchor | client | anchor | client |
| form | beam | beam | cylinder | hemisphere | beam | beam | hemisphere | beam | arbitrary |
| receiver | anchor | client | client | anchor | anchor | client | anchor | client | anchor |
| **Benchmark** | | | | | | | | | |
| pos. accuracy (3D) | 1 cm (98%)[3] | 1 cm (90%) | 2-3 cm (95%) | 3 cm | 10 cm (90%) | 1.2m (100%) | 9 cm (95%) | 20 cm | 90 cm (95%) |
| loc. Frequency | 2.85 Hz | 10 Hz[4] | ? | ? | 10 Hz[4] | 4 Hz | → [303] | 1 Hz | ? |

[1] The information which must be revealed by the client
[2] $|A_m|$ is the number of anchors within the US range of a client $m$, $|A|$ is the total number of anchors within the system
[3] 3 mm within the simulation environment, 10 mm within the real-world installation right after the calibration, and 30 mm about one week after the calibration (→ Section 14.2).
[4] The system uses an ordinary PC for coordinating the wired data aggregation and the centralized position estimation.
[5] Only if the location server is considered as a trusted authority. Otherwise, the client's presence and entire location information is revealed.
[6] In contrast to the other systems where the data is taken from the anchors, LOSNUS wires the clients to report their data for centralized processing (using Zigbee is planned).

**Table 9.1.:** Comparison of various sound based indoor localization systems

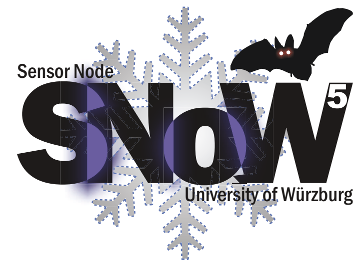
# 10. The SNoW Bat Indoor Localization and Tracking System


**Abstract**

The SNoW Bat system is intended for the localization and tracking of mobile objects within indoor environments, where e.g. GNSS is not available and an accuracy of a few millimeters is required.

Similar to many WSN based indoor localization systems, SNoW Bat requires a pre-installed infrastructure of *static anchor nodes* for estimating the position of *mobile client nodes* which are commonly mounted on various types of objects under observation. To be independent from further hardware, each SNoW Bat client operates autonomously, and localizes itself just when required, e.g. periodically or upon certain events. Therefore it triggers the simultaneous distance measurement between itself and several anchors, and applies a decentralized position estimation algorithm on the wirelessly collected distance information. Though using only typical WSAN components, the intended objectives were to achieve a high localization accuracy, precision, and frequency despite of an easy deployment process and low energy demands. At the same time we kept the focus on compositional and reliable software design as demanded in Part II of this work to simplify the integration with other complex software modules on the same resource constrained hardware.

## 10.1. Operation Principles




SNoW Bat will serve as a representative for decentralized indoor localization and tracking systems based on distributed networked sensors. As illustrated in Figure 10.2 it relies on an initially deployed infrastructure of spatially calibrated *anchor nodes* at static positions within a reference coordinate system, and allows concurrently operating mobile *client nodes* within the serviced area to localize themselves on-demand and based on ultrasound distance measurements between themselves and the anchors. While the central SNoW Bat service concept provides just the infrastructure to generate, preprocess, and aggregate spatial distance data at a client (→ Figure 9.2[p211]), the finally applied data fusion algorithm for position or location estimation is not predefined per se, but can – in the tradition of decentralization and autonomy – be chosen arbitrarily by each client. Nevertheless we recommend the pVoted approach from Chapter 13 since it is optimized for various system properties and completes the philosophy of *hardware/software/network co-design*.

Despite of our goal to support several clients in estimating their position simultaneously at minimal energy consumption, this process should nevertheless be feasible at high frequency, precision, and accuracy in all three dimensions. In this regard, runtime determinism and resource sharing issues will once more turn out as relevant aspects for seamlessly integrating SNoW Bat into existing applications as discussed in Section 1.2 – e.g. to allow vehicles on which the clients are mounted to steer autonomously along predefined paths within the serviced area.

Regarding the terminology from Definition II.10[p109] the *anchors cooperate* with the clients on-demand to offer them a service base for their self-localization. In contrast the *clients collaborate* with each other to retain the localization system's reliability and efficiency: Though literally independent they always seek to lock a minimum of the overall system resources[1] to also let coexisting nodes operate according to their individual demands. The resulting computational load is distributed among the anchors (data generation and preprocessing) and the clients (data fusion), and makes the entire system independent from additional hardware.

For performance reasons we thus attach particular importance to system parallelism; not only within the network, but also for each node: As a comprehensive example for the design and implementation of complex WSAN applications, SNoW Bat exploits the strengths of the previously introduced preemptive and event-driven operating system *Smart*OS to achieve reactive real-time operation through temporally deterministic task interaction and collaboration – these are essential properties regarding the quite dynamic environment in which SNoW Bat will finally operate. Nevertheless, a sophisticated software design is still required to optimize the wireless communication, and to dynamically adapt both the localization frequency and the spatial precision or accuracy to the varying requirements and potential of any individual client.

[1] Especially radio channels as well as anchors which may service only one client at any given time.

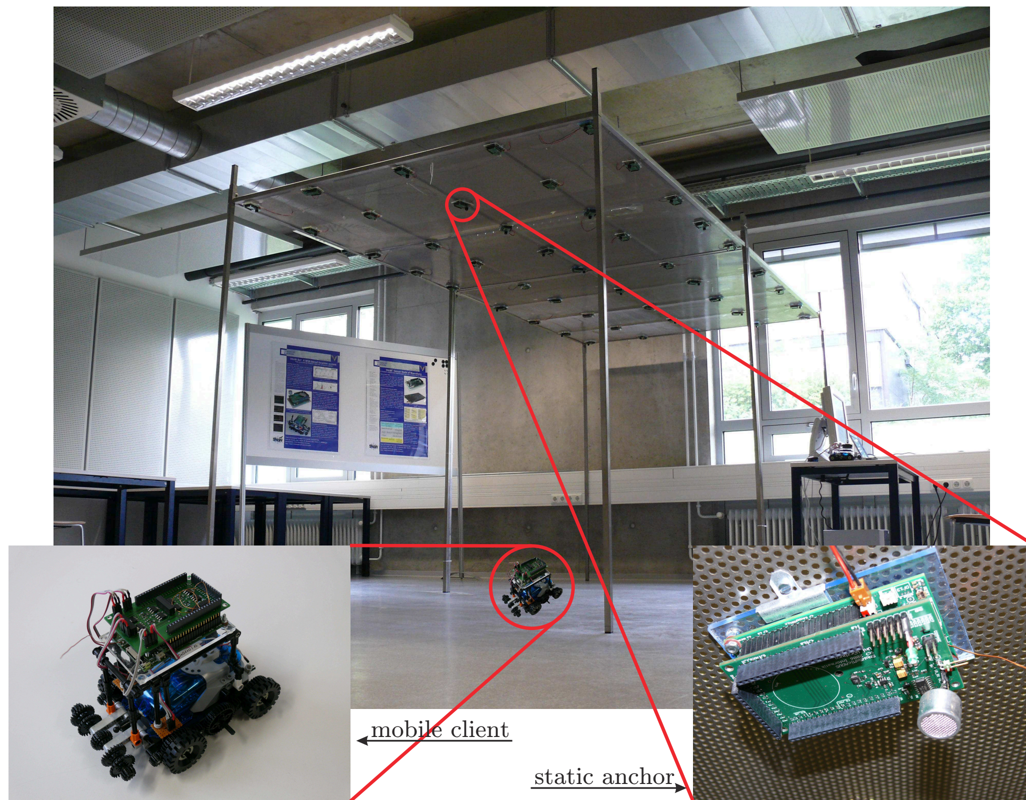



**Figure 10.1.:** The SNoW Bat test installation

### 10.1.1. SNoW Bat Operation Stages

At the clients, SNoW Bat provides the position estimation as a system service. It can be used by any application task and relies on four operational stages $P_1 - P_4$, which, depending on the application, can be triggered both periodically or sporadically. The anchors are involved in only the two stages $P_2$ and $P_3$ and react on-demand, i.e. event-triggered by the clients. Figure 10.3 shows both the parallelized node operation from the view of an external observer watching the mobile node $m \in M$ and a single anchor $a \in A_m$ serving $m$, as well as the interleaved task executions on both systems.

The client $m$ will finally annotate any position estimation $\tilde{p}(t)$ with the time $t = t^C_{\text{Chirp}}$ of the corresponding ultrasound signal emission[2]; preceding position predictions will thus also refer to the intended time $t^C_{\text{Chirp}}$ for the next iteration[3]. The corresponding sequential flow can be described as follows:

$P_1$ **Position prediction**: In the run-up to any position estimation $\tilde{p}_m(t)$ a prediction $p^*_m(t)$ is calculated from former annotated estimations $h_0, \ldots, h_{|H|-1}$ according to Eq. (9.5). While details will be presented in Section 13.2, this step serves to increase the spatial precision,

[2]The chirp emission time $t^C_{\text{Chirp}}$ is captured via the *Smart*OS IRQ timestamping functionality from Section 5.3.
[3]Scheduling the chirp emission precisely to the intended time can in turn be done as described in Section 5.4.1.

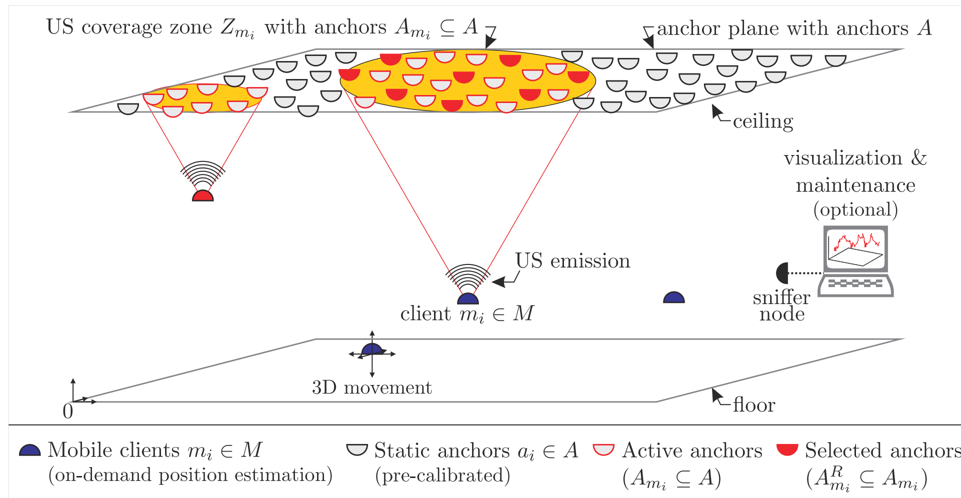



**Figure 10.2.:** Schematic design of the SNoW Bat indoor localization system

accuracy, and reliability, and to select an adequately sized subset of anchors $A^R_m \subseteq A_m$ for the immediately following distance measurement. Though less obvious, this situation aware on-line configuration of the radio communication will prove to be crucial for reducing both the data aggregation cost in stage $P_3$ (→ Chapter 12) as well as the data fusion cost in stage $P_4$ (→ Section 13).

$P_2$ **Distance measurement**: The distance measurement between $m$ and each anchor $A_m \subseteq A$ within both radio and ultrasound range is conducted simultaneously. This is particularly relevant since $m$ might otherwise move during several successive measurements; an avoidable modification to the overall system state which would complicate the subsequent calculations. Thus $m$ broadcasts a so called *chirp allocation vector* (CAV, → Listing 10.1[p228]) containing the prediction $p^*_m(t)$ precisely at the time $t^C_{\text{CAV}} = t^C_{\text{Chirp}} - \Delta_{\text{SYNC}}$ first to announce the measurement and to synchronize the involved anchors. Each (selected) anchor $a_i \in A^R_m \subseteq A_m$ within the ultrasound coverage zone $Z_m$ (→ Figure 10.2) consequently calculates the direct distance $\tilde{d}_{m,a_i}$ towards $m$ using the chirp's just measured time of flight $\Delta_{\text{TOF}}$. See Chapter 11 for details on the chirp detection.

$P_3$ **Data aggregation**: After the distance measurement each anchor $a_i \in A^R_m \subseteq A_m$ wirelessly returns a corresponding *distance vector* (DV, → Listing 10.2[p228]) containing inter alia $\tilde{d}_{m,a_i}$ and its own (calibrated) 3D position back to $m$. We apply the self-organizing HashSlot data aggregation protocol as introduced in Chapter 12 to assign definitely collision-free and tightly packed TDMA slots $s(a_i) \in [0; |A^R_m| - 1]$ for each involved anchor. As visualized in Figure 10.3 each slot is initially scheduled for time $t^A_{\text{DV},a_i} = t^A_{\text{DA}} + s(a_i) \cdot \Delta_{SLOT}$ under HashSlot, but may shift forward dynamically under HashSlot$^+$ to fill up unexpected channel idle times – for whatever reason these might appear[4]. The total number of requested DVs as well as the same number of required slots to be reserved has already been encoded as QoS value within the CAV. This reflects the (current) demands of the position estimation

[4]The begin of the data aggregation $t^A_{\text{DA}} := t^A_{\text{CAV}} + \Delta_{\text{DA}}$ is computed by each anchor according to the previously received CAV information from Listing 10.1, Line 18.

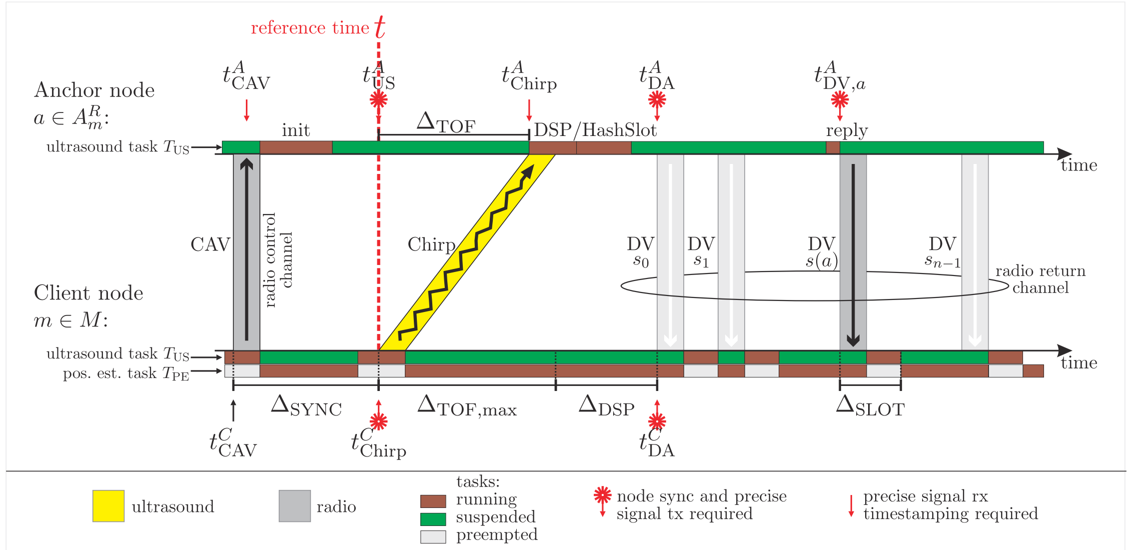



**Figure 10.3.:** The SNoW Bat localization process

algorithm, and allows to adjust the duration of both the data aggregation and the overall localization process for a deterministic limit.

$P_4$ **Position estimation**: During the last stage a localization algorithm computes a position estimation $\tilde{p}_m(t)$ from the prediction $p^*_m(t)$ (if available) and the received DVs. Using pVoted from Chapter 13 will – apart from the 3D coordinate $\vec{X}_m$ – also compute the current 3D velocity $\vec{V}_m$ and an indicator $\chi \in [0;1]$ which specifies the reliability of the estimation regarding its precision and accuracy – a valuable information for subsequent predictions and the actual application.

Regarding the call for real-time capability from earlier discussions (→ e.g. Chapter 5) the two stages $P_2$ and $P_3$ deserve particular attention: $P_2$ is critical in terms of time synchronization (→ “capturing of timestamps” in Section 5.3) through the CAV, the temporally precise emission of the chirp by the client $m$ (→ “specification of delays” in Section 5.2), and the precise capturing of the chirp arrival time by any anchor $a_i \in A^R_m$ (→ Chapter 11 or [41][5]). $P_3$ is critical in terms of the compliance with assigned TDMA slots (→ “scheduling of reaction times” in Section 5.4) since any violation would probably lead to radio collisions and result in the loss of valuable information.

## 10.2. System Design Considerations

### 10.2.1. The Sensor Node Platform Configuration

The SNoW Bat hardware is based on SNoW[5] wireless sensor nodes[6] from Chapter 2. Though – for economic, deployment, and interchangeability reasons (→ Section 2.1) – it is exactly the same

[5]Bachelor thesis conducted in conjunction with this work.
[6]TI MSP430F1611 MCU [280] (8 MHz CPU clock, 10 kB RAM, 48 kB ROM), TI CC1100 radio transceiver [281]

```
typedef struct {
  uint8_t qx;              // the QoS level in x direction, 0=unused
  uint8_t qy;              // the QoS level in y direction, 0=unused
  uint8_t gamma_ad;        // the adaptive grid module, 0=unused
  uint8_t dl;              // dist. from anchor plane, lower bound [cm], 0=unused
  uint8_t du;              // dist. from anchor plane, upper bound [cm], 0=unused
  uint8_t tTau;            // τ for HashSlot+ [ms]
} hashSlot_Request_t;

typedef struct {
  uint16_t cavID;          // measurement ID (autoincrement)
  uint16_t x;              // predicted x-coord of the client [mm], 0=unknown
  uint16_t y;              // predicted y-coord of the client [mm], 0=unknown
  uint16_t z;              // predicted z-coord of the client [mm], 0=unknown
  sint16_t temperature;    // ambient temperature [1/100°C]

  uint16_t Δ_SYNC;         // the delay between the CAV and the chirp [ms]
  uint16_t Δ_DA;           // the delay between the CAV and the first DV [ms]
                           // = Δ_SYNC + Δ_TOF,max + Δ_DSP

  /* data aggregation specific */
  batDAMAC_t DAProtocol;   // the data aggregation protocol (see Section 12.3)
  uint8_t retChannel;      // the radio return channel for the data aggregation
  uint8_t Δ_SLOT;          // slot length for TDMA methods [ms]
  hashSlot_Request_t HS;   // the HashSlot configuration

  /* various additional data */
  ...
} batCAV_t;
```

**Listing 10.1:** The SNoW Bat Chirp Allocation Vector (CAV) data structure of a client $m \in M$ (29 B)
See Figure 10.3 as reference and Listing 10.2 for the replying Distance Vector (DV)

```

typedef struct {
  uint16_t cavID;          // copy from CAV (for matching)
  uint16_t x;              // x-coord of the anchor [mm], 0=unknown
  uint16_t y;              // y-coord of the anchor [mm], 0=unknown
  uint16_t z;              // z-coord of the anchor [mm], 0=unknown
  sint16_t temperature;    // ambient temperature [1/100°C]

  uint8_t  DVSlot;         // the used TDMA slot s(a_i) (for slotted protocols)
  uint16_t Δ_TOF;          // the measured ultrasound TOF [us]
  uint16_t d̃_{m,a_i};      // the measured direct distance to the client m [mm]

  /* various additional data */
  ...
} batDV_t;
```

**Listing 10.2:** The SNoW Bat Distance Vector (DV) data structure of an anchor $a_i \in A_m^R$ (22 B)
See Figure 10.3 as reference and Listing 10.1 for the preceding Chirp Allocation Vector (CAV)

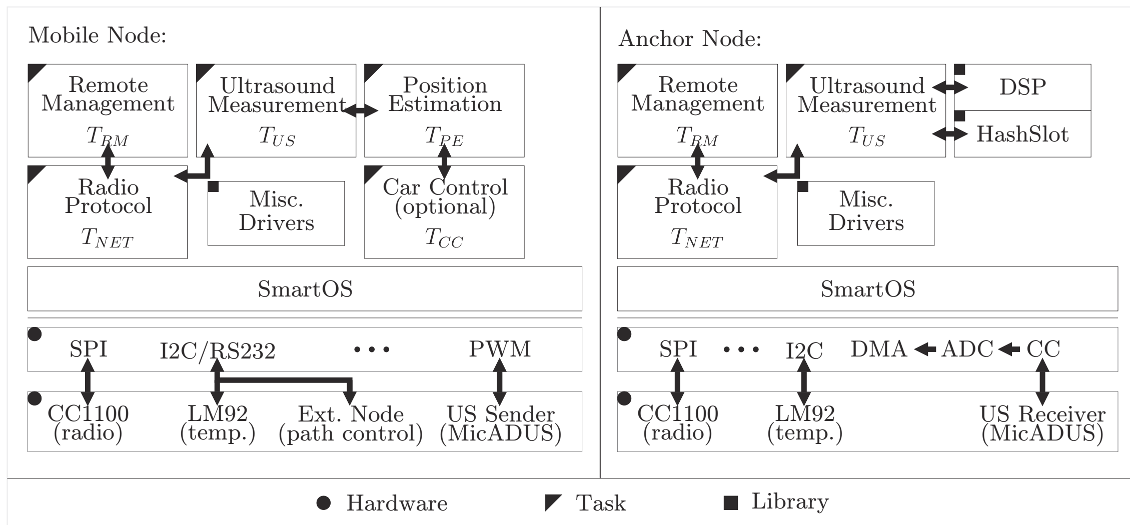



**Figure 10.4.:** The SNoW Bat task system for mobile and anchor nodes

for the anchors and the clients, an overview on the respectively relevant hardware/software components is summarized in Figure 10.4:

The sub-1 GHz radio transceiver is configured to operate at 915 MHz base frequency where it supports data rates up to 500 kbit/s at 255 freely selectable channels. To preserve a maximum of control over most configuration options regarding the channel access scheme we did intentionally not rely on a standardized MAC protocol like e.g. ZigBee, IEEE 802.15.4, or Bluetooth, but implemented a more appropriate solution based on the CC1100's proprietary PHY[7]: On each node the *Smart*Net low-level MAC protocol keeps exclusive control over any wireless communication as well as over the automatic rx/tx timestamping. Implemented as a system service *Smart*Net also serves as a flexible base for the implementation of higher level protocols like the already mentioned HashSlot and HashSlot$^+$ TDMA schemes. A short overview on *Smart*Net has already been given in Section 8.1.

The ultrasound transceiver unit is implemented as extension board as depicted in Figure 2.7a[p31]: Like the sensor node itself the so called MICADUS board is exactly the same for the anchors and the clients. Figure 2.8[p31] shows an additional extension which is compatible with the MICADUS module but will exclusively be used for the mobile nodes: Outsourced to an additional SNoW[5] for autonomous path control using a LEGO Spybotics™ vehicle it interfaces a PWM motor controller to drive the differential gears[8].

[7] "Appropriate" regarding the benefits for the concrete application, but especially regarding the fact that we operate tiny but truly preemptive multitasking systems where – apart from each node's individual network address – even tasks must be (sub-)addressable and distinctively be notified about their individually received packets at minimum delay and with respect to their active priority (→ Section 4.3.2).

[8] Interconnected via I2C the second SNoW[5] node became necessary since executing both the localization algorithm and the fuzzy controller showed to be simply too complex for a single MSP430F1611 regarding its CPU, RAM, and ROM reserves.

### 10.2.2. Anchor Infrastructure and Deployment

The deployment of a technical system into an existing environment should exert as little disturbing influence on it as possible (→ LS2a[p218]) – for SNoW Bat this implies an adequate number and placement of the static anchor nodes. On the one hand, few devices mean a fast deployment, low (running) costs and energy consumption, reduced maintenance effort and environmental pollution due to e.g. radio transmissions. On the other hand, a certain minimal amount of anchors will be required to guarantee an area wide localization service coverage at sufficient precision and fault tolerance.

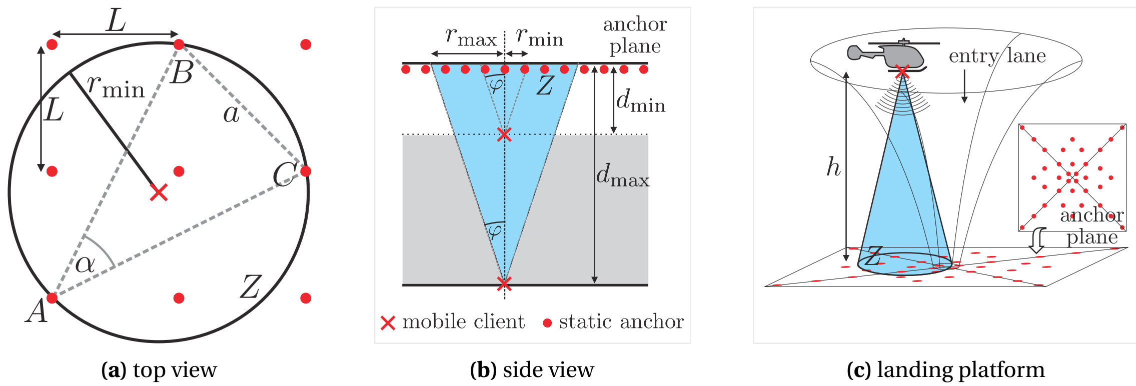
Although most position estimation algorithms need no special anchor alignment as long as each one knows its exact position in world coordinates and as long as it is assured that a mobile node can always measure a sufficient number of distances to different anchors, we intentionally rely on a grid pattern for three reasons:

1. The deployment can be simplified through prefabricated and standardized building units (like e.g. the ceiling panel from Figure 10.6b) with integrated sensor nodes.
2. The HashSlot data aggregation algorithm from Chapter 12 requires the anchors to be *roughly* aligned along a regular pattern to unfold its full potential.
3. The system calibration will be simplified [256][9] since each node knows its rough position in terms of the grid cell index (e.g. row and column).

Referring to Figure 10.5 we initially disclose our approach for finding an optimal anchor grid to guarantee a sufficient number of distance measurements for each position estimation process. During our considerations and experiments all anchors will be aligned to such a grid and mounted on (roughly) the same level – the so called *anchor plane* – at the ceiling above the observed space. As Figure 10.5c illustrates, it would also be possible to install the anchors on e.g. the floor – even with a varying grid constant if demanded by the application: In case of the landing platform as illustrated in Figure 10.5c the grid becomes finer towards the central landing point as a function of the helicopter's minimal height according to its entry lane. In any case the US transducers will be directed towards the serviced space and orthogonal to the anchor plane.

In order to find an appropriate grid constant $L$, we'll do some considerations about the room geometry first. Regarding the fact that at least $n+1$ distances are required to locate an object in $n$-dimensional space, these are of special importance as soon as clients might not only move in parallel to the anchor plane (e.g. in 2D on the floor) but also orthogonal to it (i.e. freely in 3D). While the maximum supported room height $h_{\text{sup}} := u \cdot \cos(\varphi)$ depends on the US transducer range $u$ and its beam angle $\varphi$ which are constant parameters for the applied hardware, the maximum distance $d_{\max} \leq h_{\text{sup}}$ of the mobile node from the anchor plane is implicitly limited. In contrast, the least possible distance $d_{\min} \leq d \leq d_{\max} \leq h_{\text{sup}}$ from a client to the anchor plane depends on the application demands, but is a much more important specification since it defines an upper bound for the grid constant $L$: In fact, the minimal coverage zone $Z$ of the US signal has a radius $r_{\min} := d_{\min} \cdot \tan(\varphi)$, and we nevertheless have to guarantee that at least

[9] Diploma thesis conducted in conjunction with this work.




(a) top view (b) side view (c) landing platform

**Figure 10.5.:** SNoW Bat room geometry and anchor node deployment analysis

four anchors are always within this (freely moving) circle. Thus, three grid points aligned like $A$, $B$ and $C$ in Figure 10.5a must always be located inside $Z$[10], or, in other words: The minimal coverage zone $Z$ must be at least as large as the circumcircle of the triangle $ABC$. Some simple trigonometric relations lead us to a simple equation for computing $L$ during the deployment stage: With

$$a = \sqrt{2 \cdot L^2} = \sqrt{2} \cdot L,$$

$$\left.\begin{array}{l} \sin\left(\frac{\alpha}{2}\right) = \frac{\frac{a}{2}}{\sqrt{L^2+4 \cdot L^2}} = \frac{1}{2} \cdot \sqrt{\frac{2}{5}} \\ \sin\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1}{2}(1-\cos(\alpha))} \end{array}\right\} \Rightarrow \cos(\alpha) = \frac{4}{5},$$

$$\sin(\alpha) = \sqrt{1-\cos^2\alpha} = \frac{3}{5} = 0.6,$$

and the just mentioned precondition

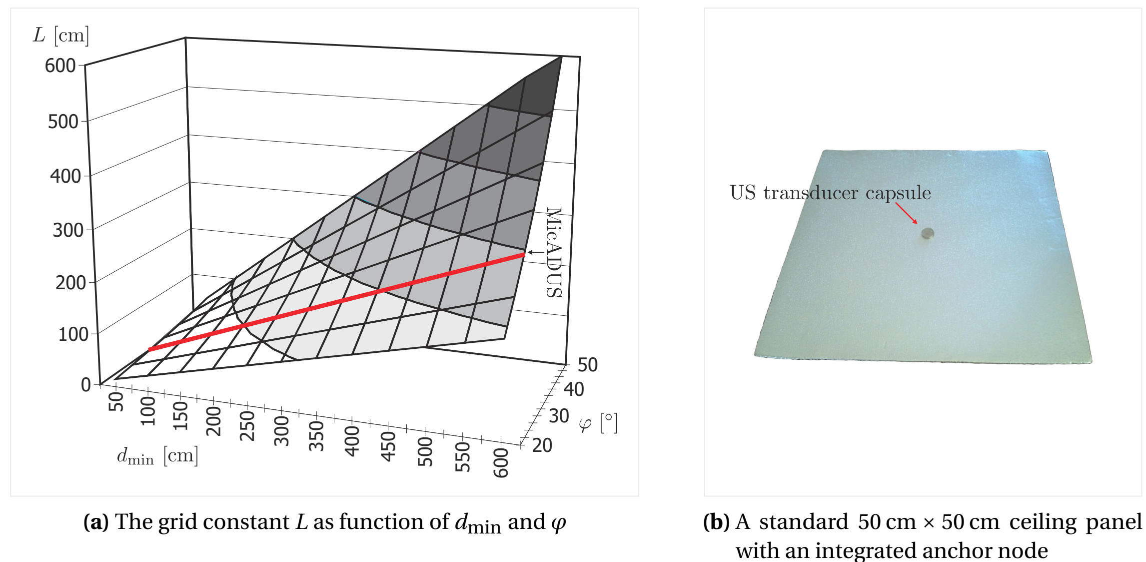
$$\underbrace{r_{min}}_{\text{(coverage zone)}} \overset{!}{\geq} \underbrace{\frac{a}{2 \cdot \sin(\alpha)} = \frac{\sqrt{2} \cdot L}{1.2}}_{\text{(circumcircle)}}$$

the maximal accepted grid constant computes as

$$L \leq \frac{1.2 \cdot r_{min}}{\sqrt{2}} = \frac{1.2 \cdot d_{min} \cdot \tan(\varphi)}{\sqrt{2}}. \tag{10.1}$$

Figure 10.6a gives an overview on various values. By arranging the anchors as described, it is possible to localize any mobile object within a distance between $d_{min}$ and $d_{max}$ from the anchor plane. Note, that fault tolerance improves implicitly with increasing distance from the ceiling as more anchors will be located within $Z$ then, and receive a mobile node's chirp to measure and return a distance. Another way to explicitly increase fault tolerance in case the client moves

[10]Although the coverage zone $Z$ will not exhibit the shape of a perfect circle under realistic conditions, we'll use this simplified model since it is close to the true outline and proved to be sufficiently precise when verifying our theoretical considerations during the real-world tests for both HashSlot and pVoted.




**(a)** The grid constant $L$ as function of $d_{\min}$ and $\varphi$

**(b)** A standard 50 cm × 50 cm ceiling panel with an integrated anchor node

**Figure 10.6.:** Room geometry considerations

close to the anchor plane is placing the anchors even more densely just where necessary.

Now, that $L$ is computable from the application-specific $d_{\min}$ and the hardware specific $\varphi$, it is also possible to calculate the whole system's maximum coverage area depending on the number of anchors. As reference the MICADUS hardware specifications are $\varphi = 30°$ and $u = 8$ m, leading to $h_{\sup} \approx 6.93$ m. For tracking a vehicle on the plain floor of a hall with $d_{\min} = d_{\max} = 5$ m an anchor grid with $L \leq 245$ cm would be required and an e.g. quadratic area of $(\lfloor\sqrt{64}\rfloor - 1)^2 \cdot L^2 \approx 294$ m$^2$ could be covered with the SNoW Bat service using 64 nodes. Within our 3D test setup as shown in Figure 10.1, $d_{\min} = 1$ m and $d_{\max} = 2$ m, and thus we require 45 anchors mounted at $L \approx 49$ cm for an area of 4 m × 2 m.

### 10.2.3. The Impact of the Data Aggregation on the Localization Frequency

In Section 10.2.4 we have already discussed the impact of the software design on the localization frequency, and pointed out that our preemptive and priority aware *Smart*OS kernel allows a considerable speedup through the partial interleaving of the four localization stages: With regard to data and control dependencies we serialized $P_2$ and $P_3$ in one high priority real-time task $T_{\text{US}}$ and optimized its reactivity. However, we still have to consider the impact of those two stages on the task's WCET since the data aggregation in particular involves hard-to-control node-to-node interaction, and will show to be *the* potential source of energy, time, and information loss if not managed carefully. In order to avoid unnecessarily reduced node lifetime, position update rates, precision, and accuracy through our HashSlot data aggregation protocol, we'll first evaluate the impact of this communication process on the total duration of a single localization iteration.

Let's assume we intend to use at least $k \geq n + 1$ distance measurements for a single $n$-dimensional position estimation. While $k$ may already include some extra DVs to compensate

for measurement noise and imprecision as discussed in Chapter 11, the quite noteworthy unreliability of some radio protocols is yet another factor to consider. With $\rho \in [0,1)$ being the expected packet loss rate (PLR) of the applied data aggregation protocol, we consequently have to request – either explicitly or implicitly – the distance vectors from at least $g \geq \lceil k \cdot (1-\rho)^{-1} \rceil$ anchors at the mobile client. Let's further assume that each distance vector contains $L_{\mathrm{DV}}$ bytes (including the payload and radio protocol overhead) and will be transmitted wirelessly with data rate $C_{\mathrm{DV}}$. On reception the client node requires some time $t_{\mathrm{DV}}^{\mathrm{CPU}}$ to fetch the radio packet, i.e. to read the radio transceiver's RX buffer and to re-enter RX mode. Since most sensor nodes employ just a single radio chip – and so does the SNoW[5] – this communication unit and the locally selected but globally effective radio channel represent exclusively shared resources. Consequently we can only receive one packet at a time, and the immediately successful transmission of a single DV will last for one so called *time slot* with the duration

$$\Delta_{\mathrm{SLOT}} := t_{\mathrm{DV}}^{\mathrm{HF}} + t_{\mathrm{DV}}^{\mathrm{CPU}} = \frac{L_{\mathrm{DV}}}{C_{\mathrm{DV}}} + t_{\mathrm{DV}}^{\mathrm{CPU}}. \tag{10.2}$$

According to these considerations and Figure 10.3, the entire data aggregation stage $P_3$ for the $g$ requested DVs is a serialized process, and takes at least the time

$$d_3 \geq \Phi(g) = g \cdot \Delta_{\mathrm{SLOT}}. \tag{10.3}$$

Equality in Eq. (10.3) will only be observed if all packets can be received successfully on the first attempt and in direct succession without any inter-packet spacing. However, both assumptions are rather unlikely for most communication protocols, and this raises two questions:

1. Can the theoretical optimum $d_3 = \Phi(g) = \Phi(k)$ for the data aggregation stage $P_3$ really be achieved? This is: Can we reduce $\rho \to 0$ while at the same time avoiding *any* additional temporal effort for coordinating the anchors' transmission schedules despite of the client's mobility and the varying subset of anchors within the freely moving US coverage zone?

2. If 1 is not possible, which timeout $t_{\mathrm{TO}} \geq \Phi(g)$ should be chosen to reasonably limit $P_3$ and proceed to $P_4$?

Although our deterministic HashSlot and HashSlot$^+$ algorithms will prove to achieve the optimal performance, we also want to compare some other protocols and thus accept a data aggregation timeout $\Delta_{\mathrm{TO}}$ of up to twice the minimal stage duration for these approaches:

$$d_3 \leq \Delta_{\mathrm{TO}} \leq 2 \cdot \Phi(g) = \Phi(2g)$$

Thereby the proportion $p(g)$ of the reply stage on the total execution time of the task $T_{\mathrm{US}}$ (i.e. without position estimation) is bounded by the number of requested packets $g$ on the one hand and by the given timeout $\Delta_{\mathrm{TO}}$ on the other hand:

$$\frac{g \cdot \Delta_{\mathrm{SLOT}}}{d_2 + g \cdot \Delta_{\mathrm{SLOT}}} \leq p(g) \leq \frac{\Delta_{\mathrm{TO}}}{d_2 + \Delta_{\mathrm{TO}}} \tag{10.4}$$

According to Amdahl's law undertaking considerable effort to be close to optimal or at least *not* to utilize the entire timeout is quite justified, since the data aggregation will commonly consume much more time than the measurement process itself; and this is true even though the ultrasound based distance measurement is already one of the slowest widely accepted techniques. While

$$d_2 := \Delta_{\mathrm{SYNC}} + \Delta_{\mathrm{TOF,max}} + \Delta_{\mathrm{DSP}}$$

is mainly fixed through physical laws and environmental circumstances like e.g. the room geometry it can not be optimized arbitrarily, and the only chance to improve the overall runtime time is to optimize the radio protocol. In fact, with decreasing duration $d_2$, e.g. through reducing $d_{\mathrm{max}}$ along with the time of flight or by even using entirely different measurement techniques (e.g. LASER or RSSI based), the impact of the wireless communication will grow even further.

**Case study.** Put into the context of the SNoW Bat system, each DV takes $L_{\mathrm{DV}} = 22\,\mathrm{B} + 19\,\mathrm{B} = 41\,\mathrm{B}$ according to Listing 10.2 (payload) and Figure 8.2[p196] (overhead). The radio transceiver operates at $C_{\mathrm{DV}} = 230\,\mathrm{kbit/s}$, and the DV processing time on the receiving SNoW[5] sensor nodes is $t_{\mathrm{DV}}^{\mathrm{CPU}} \approx 2.6\,\mathrm{ms}$. According to Eq. (10.2) each transmission takes $\Delta_{\mathrm{SLOT}} \approx 4\,\mathrm{ms}$, and we still seek to optimize the localization frequency $f_L$ by achieving $d_3 = \Phi(g) \approx g \cdot 4\,\mathrm{ms}$ for $g = k$ requested DVs with $\rho = 0$.

Regarding an entire localization process for the maximal distance $d_{\mathrm{max}} = d_{\mathrm{sup}} \approx 6.93\,\mathrm{m}$ between the clients and the anchor plane the duration of the measurement stage is $d_2 \approx 26.2\,\mathrm{ms}$. The blue graphs in Figure 10.7 show that returning the minimum of $n+1 = 4$ DVs back to the client will already consume between 38% in the best case and 55% if the timeout $\Delta_{\mathrm{TO}}$ limits $d_3$ – although we do not know if a sufficient number of DVs has really been received in the latter case! If we reduce the maximal distance to $d_{\mathrm{max}} = 2\,\mathrm{m}$ the red graphs reveal that this even worsens $p(g)$ to range between 57% and 73% as the measurement stage duration is also reduced to $d_2 \approx 11.8\,\mathrm{ms}$ due to a shorter $\Delta_{\mathrm{TOF,max}}$ of the US chirp.

Anticipating the measurement error characteristics from Figure 11.4b[p249] with about 60% "good" measurements around the central error and 40% side errors, some fault tolerance should not be avoided, and we will already demand for $k = \left\lceil \frac{n+1}{1-0.4} \right\rceil = 7$ DVs then. Adding another two DVs to compensate for e.g. node failures (which we cannot control) the data aggregation for $g = k+2 = 9$ will already take between 58% and 73% (for $d_{\mathrm{max}} = d_{\mathrm{sup}}$) and between 75% and 86% (for $d_{\mathrm{max}} = 2\,\mathrm{m}$) of the total time.

Eventually, the green graphs in Figure 10.7 indicate the maximal achievable localization frequency $f_{L,\mathrm{max}}$ when considering only the distance measurement task $T_{\mathrm{US}}$ while ignoring the execution time for the interleaved position estimation task $T_{\mathrm{PE}}$.

**Summary.** As we have seen in this section, the data aggregation stage $P_3$ in decentralized localization systems like SNoW Bat has significant influence on the maximum position estimation frequency $f_{L,\mathrm{max}}$. Apart from the quite obvious relevance for temporally fine-grained node tracking in various control systems, this core specification is particularly important for simultaneously supporting several mobile nodes: From the clients' point of view the anchors are

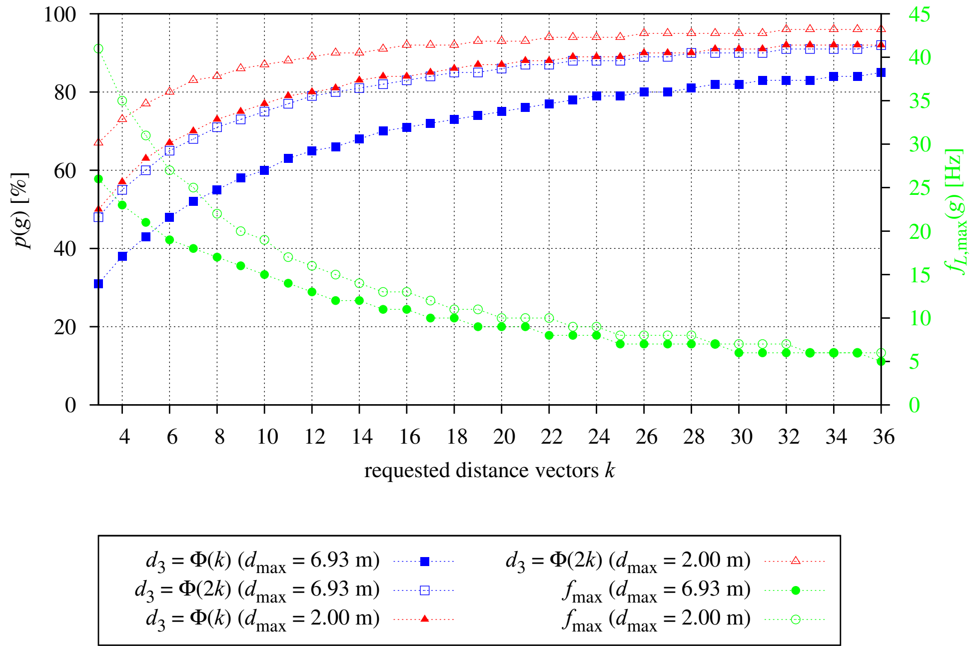



**Figure 10.7.:** Proportion $p(g)$ of the data aggregation time $d_3$ in the complete localization time $d_L^{\parallel}$

dynamically but exclusively shared short-term resources and the localization delay consequently reduces along with the localization speed. Thus, we will introduce the highly sophisticated HashSlot and HashSlot$^+$ protocols in Chapter 12 for optimizing the data aggregation stage with respect to our considerations from this section.

### 10.2.4. The Impact of the Software Design on the Localization Frequency

Having introduced the basic operation of SNoW Bat as a representative for ultrasound based localization systems, we'll initially address its theoretical performance regarding the localization frequency. Therefore we aim on parallelizing the execution of the stages $P_1 \ldots P_4$ on the mobile clients in order to minimize the duration $d_L$ of a single position estimation as well as the minimal period of successively conducted iterations, respectively. In this context, we'll consider the achievable localization frequency

$$f_L := \frac{1}{d_L} \leq f_{L,\max} := \frac{1}{d_{L,\min}} \tag{10.5}$$

with the help of Amdahl's law [6], and show how to optimize $f_L$ through an efficient utilization of the (preemptive) CPU and other (exclusive) resources under *Smart*OS.

Since the stages $P_4$ and $P_1$ are obviously data-dependent[11], and both stages $P_2$ and $P_3$ face demanding real-time requirements as well as potential conflicts regarding some exclusively shared resources (like e.g. radio, SPI, or the IRQ controller), their arbitrary parallelization is hard

[11] At least if we want to consider each position estimation ($P_4$) for the directly following prediction ($P_1$) during the next iteration.

to accomplish. Thus, we decided to combine the ultrasound measurement stages $P_2$ and $P_3$ in a high priority real-time task $T_{\mathrm{US}}$, and put the position estimation stages $P_1$ and $P_4$ into a lower priority task $T_{\mathrm{PE}}$ for pairwise parallelization.

While the mostly event-driven task $T_{\mathrm{US}}$ measures the sound propagation delays in stages $P_2$ and $P_3$ it will frequently suspend itself to grant a proper synchronization at the beginning of each measurement process and to wait for the DVs to arrive successively at the end. Consequently some task $T_{\mathrm{US}}$ specific self-suspensions will inevitably arise on the mobile node as illustrated in Figure 10.3. Though we won't go into detail here, it is important to note that these waiting delays during both stages $P_2$ and $P_3$ are mainly determined by physical laws and technical facts or realizations, and cannot be reduced arbitrarily (e.g. by "applying the optimal radio protocol"). In a strictly sequential implementation of the localization process however, they lead to unused idle periods and therefore provide great optimization potential.

In the following we denote the time periods where the CPU is busy during a stage $P_i$ as $b_i$, and the inevitable waiting times as $w_i$. With the just mentioned naïve sequential implementation of the localization process we would obviously obtain

$$d_L := \sum_{i=1}^{4} (b_i + w_i) = \sum_{i=1}^{4} d_i = d_{L,\max}. \tag{10.6}$$

Since $P_1$ and $P_4$ do obviously not exhibit any waiting times (i.e. $w_1 = w_4 = 0$), we can take advantage of the truly preemptive *Smart*OS scheduler, and parallelize them partially with $P_2$ and $P_3$ by interleaving the corresponding tasks $T_{\mathrm{PE}}$ and $T_{\mathrm{US}}$ as suitable. One important requirement to mention is that $T_{\mathrm{US}}$ must receive a truly higher base priority than $T_{\mathrm{PE}}$ ($P_{T_{\mathrm{US}}} > P_{T_{\mathrm{PE}}}$) since a suspended calculation can be resumed arbitrarily while, in contrast, violating a chirp emission time will result in measurement errors and processing DVs late can easily result in losing subsequent DVs. According to Figure 10.8a, which shows a compressed schematic of the interleaving, the resulting minimum period $d_L^{\|}$ of repeated position estimations computes as

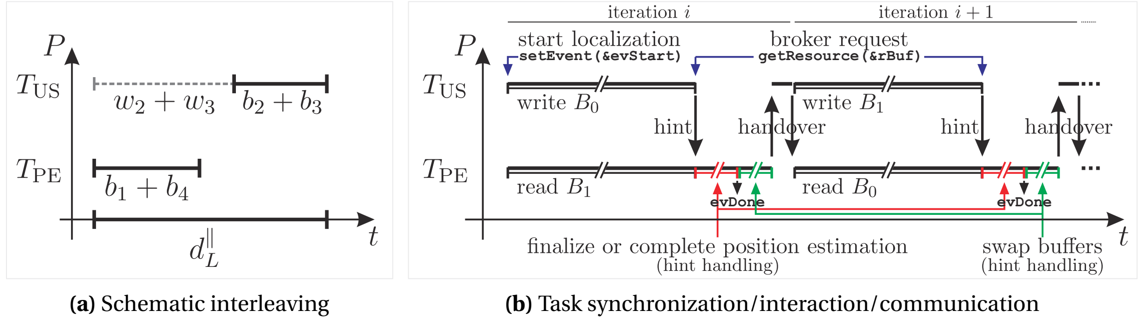
$$d_L^{\|} := b_2 + b_3 + \underbrace{\max\{w_2 + w_3, b_1 + b_4\}}_{:=\alpha_{\max}} = d_L - \underbrace{\min\{w_2 + w_3, b_1 + b_4\}}_{:=\alpha_{\min}}, \tag{10.7}$$

and the achievable speedup $s$ resulting solely from this parallelization finally computes as

$$s := \frac{d_L}{d_L^{\|}} = \frac{d_L}{d_L - \underbrace{\min\{w_2 + w_3, b_1 + b_4\}}_{\alpha_{\min}}} \geq 1. \tag{10.8}$$

According to Amdahl's law the primary aim should therefore be to optimize the limiting factors in $d_L^{\|}$, starting with the largest one. As soon as the optimization is done, and $\alpha_{\max}$ is minimal but still bounded by either the busy or waiting times (whichever are longer), one is still free to

- improve the position estimation by granting more CPU time as long as $b_1 + b_4 \leq w_2 + w_3$ holds, or
- extend the data aggregation or improve its reliability as long as $w_2 + w_3 \leq b_1 + b_4$ holds.




**(a)** Schematic interleaving

**(b)** Task synchronization/interaction/communication

**Figure 10.8.:** Parallelization of the localization stages $P_1 \dots P_4$ on a mobile client node

While both options can even be exhausted and traded dynamically at runtime the parallelization related speedup $s$ grows to its maximum if $\alpha_{\min} = \alpha_{\max} = \alpha$, i.e. if $w_2 + w_3 = b_1 + b4$ holds. In particular, the CPU load is 100% then since execution and waiting periods interleave perfectly[12]. On the other hand the parallelization of the four stages through the two tasks on a single core CPU will always restrict the speedup to

$$1 \leq s = \frac{b_2 + b_3 + 2 \cdot \alpha}{b_2 + b_3 + \alpha} \leq 2. \tag{10.9}$$

Although concrete implementation details about the just introduced stages and tasks will be deferred to the next sections, we'll already reveal some task interaction, communication, and synchronization concepts: Since the SNoW Bat position estimation subsystem can serve only one task at any given time, exclusive access to its public library functions is provided through an ordinary *Smart*OS system resource which must be allocated by the client task first[13]. Encapsulated in such a library function – which is always executed in the client task's context – a dedicated the *Smart*OS event triggers the ("driven") resumption of the currently ("self-preempted") waiting server task $T_{US}$ which will immediately start with stage $P_2$ of the position estimation process (→ `evStart` in Figure 10.8b).

Due to the high throughput of the HashSlot data aggregation protocol in $P_3$ and the low CPU performance in general, $T_{PE}$ will commonly take longer to complete and exhibit a higher WCET if all received DVs had to be processed thoroughly for the final position estimation. In consequence we will almost always encounter situations with $b_1 + b_4 > w_2 + w_3$, i.e. $T_{US}$ would commonly have to wait for $T_{PE}$ to complete. While one option is to simply accept this structural and computational imbalance (which might become quite extreme for a large number of DVs or computationally complex data and information therein), another option would be to stop the position estimation algorithm explicitly and on-demand as soon as the data aggregation stage has finished or the client task's timeout for the current iteration has been reached. In order to let this rather complex and once more event-driven task synchronization take place properly we need both a progressive position estimation algorithm (which would allow sporadic on-demand

[12] Apart from operating system overhead and additional load caused by other tasks or interrupts.

[13] Besides, this implicitly allows the priority aware sharing of this exclusive service through the priority inheritance protocol and DynamicHinting from Chapter 6.

requests for finalizing its computation almost arbitrarily), and the precise knowledge about the end of the data aggregation stage (which in turn requires perfect control and temporal predictability over the applied communication protocol to definitely identify the last TDMA slot even if the corresponding packet does not even arrive).

While both requirements were carefully considered during the design and implementation of the pVoted position estimation algorithm (→ Chapter 13) and the HashSlot data aggregation protocol (→ Chapter 12), the inter-task communication must also take care for the management of the local DV buffers with $T_{\mathrm{US}}$ being the producer (i.e. DV receiver) and $T_{\mathrm{PE}}$ being the consumer (i.e. DV processor): In fact, the task synchronization takes place via double-buffering: While $T_{\mathrm{US}}$ transfers incoming DVs into the buffer $B_r$ with $r \in \{0, 1\}$ the other task $T_{\mathrm{PE}}$ processes DVs from the previous iteration stored in the buffer $B_{(r+1) \mod 2}$. The mutual access is coordinated through a *Smart*OS resource as described in Section 4.3.7, and – as an additional benefit – we can once more take advantage of the DynamicHinting programming paradigm to accomplish the synchronization under real-time demands as illustrated in Figure 10.8b:

Having received and transferred the corresponding DVs for the current iteration into the currently assigned buffer $B_r$, $T_{\mathrm{US}}$ notifies the lower prioritized $T_{\mathrm{PE}}$ by requesting the `rBuf` resource which is commonly held by $T_{\mathrm{PE}}$. The purpose of `rBuf` is twofold in this context: First, `rBuf` protects the exclusive access to the two buffers which is managed by $T_{\mathrm{PE}}$, and second, `rBuf` acts like a *broker* resource comparable to the CoMem brokers from Chapter 7. Notified by the generated hint $T_{\mathrm{PE}}$ will – depending on the system configuration – either finalize or complete the currently executing position estimation algorithm before it

1. swaps the DV buffers,
2. temporarily releases the `rBuf` resource to resume $T_{\mathrm{US}}$ for the next measurement (handover)[14], and
3. starts over with processing the just collected DVs from the just swapped buffer.

In order to provide a non-blocking system service which can be started by any client task, $T_{\mathrm{US}}$ will eventually trigger an arbitrarily selectable event to signal this serviced task about each newly completed position estimation (e.g. `evDone` in Figure 10.8b)[15]. Stopping the service is always possible through a library function which simply instructs $T_{\mathrm{US}}$ to not request the broker (i.e. `rBuf`) any more, but to suspend itself and wait for another `evStart` event to occur.

Using this concept compared to the non-parallelized naïve implementation we managed to increase the localization frequency by $s \approx 81\%$ to $f_L^{\parallel} \approx 3$ Hz at least for those cases which will become relevant later (i.e. $7 \leq |A_m^R| \leq 8$ and a target accuracy of $e_L \leq 5$ mm for the position estimation). An acceleration beyond this factor would become possible only through a systematic improvement of the involved algorithms and their WCET. While Figure 14.2[p307] will finally reveal real-world results for $s$ and $f_L^{\parallel}$ we'll discuss these graphs for a more detailed analysis (including several side effects of the HashSlot and HashSlot$^+$ data aggregation protocols) in Section 13.3.

[14]Note that $T_{\mathrm{US}}$ will immediately return the broker after the handover (→ Listing 11.1[p243], lines 41/42).

[15]While the client passes its individual signaling event along with the position estimation command, it can finally query the result through yet another SNoW Bat API function.

## 10.3. Summary

This chapter introduced the general SNoW Bat operation principles and various design considerations including the hardware platform and the infrastructure deployment as well as the impact of both the wireless data aggregation and the software design on the localization frequency. Details on various potential approaches and solutions will be discussed next.

# 11. Ultrasound Distance Measurement: The Cut Algorithm

**Abstract**

In the last chapter we introduced the general SNoW Bat software architecture, and emphasized the special real-time requirements during the ultrasound based distance measurement stage $P_2$. Although we decided to assign a high base priority to the task $T_{\text{US}}$ in charge of sending and receiving the ultrasound chirp on both the mobile and the static nodes, respectively, we still have to consider the handling of various measurement related phenomena and problems which inevitably arise from physical conditions. In particular, we will refer to those issues which are caused by the attenuation of the acoustic waves[a] when traveling through the air, and to those caused by the irregular or at least angle dependent radiation power of directed ultrasound transducers as applied for the MICADUS boards. Finally, since measurement errors and imprecision can not be avoided entirely under real-world conditions, we will at least manage to obtain a special error characteristics which allows us to significantly simplify and improve the subsequent data fusion process in Chapter 13.

[a]From a physical point of view acoustic waves traveling through gases (or other compressible media) are longitudinal pressure waves with a certain oscillation frequency. The propagation characteristics is mainly affected by the media's temperature, motion, and viscosity. For air at 20 °C the speed of (ultra)sound is $v_{\text{US}} \approx 343\,\text{m/s}$.

## 11.1. Timing and Chirp Emission

Node synchronization and precisely timed signal emission at the client are the first details we want to address since these definitely represent *the* essential preconditions for determining the distance between two nodes or devices by measuring the time difference of arrival (TDoA) between two signals propagating at different speed from one device to the other. Since we are intentionally not interested in keeping all the nodes within a SNoW Bat system synchronized all the time[1], but can also not afford to initiate and accept intense communication efforts and an extended or even unpredictable temporal delay for synchronizing the nodes on-demand, we depend on finding a reliable mechanism which would allow us to "instantly" and "simultaneously" synchronize the subset $A_m^R \subseteq A$ of potentially replying anchor nodes by means of a single and unidirectional information transfer initiated by the mobile client $m$.[2] The nature inspired "thunder & lightning" principle has already been mentioned in Section 10.1.1: Sending a CAV right before *each* chirp is not only useful for announcing the chirp and for disseminating various configuration parameters from the client to the anchors, but the quite considerable clock drift between the nodes (→ Figure 5.6[p81]) can also be ignored then as long as it cannot accumulate to exceed the maximum resolution between the sync time $t_{\text{CAV}}$ and its last time-critical usage[3]. Since sending the CAV is accomplished on-demand (i.e. each time a localization process is triggered) through the *Smart*Net radio protocol from Section 8.1, the automatic tx/rx timestamping at both the sender and the receiver is guaranteed, and, based on these timestamps, we achieve a sufficiently good synchronization between the involved nodes based on $t_{\text{CAV}}^C$ and $t_{\text{CAV}}^A$ from Figure 10.3[p227], respectively.

Even though we already specified the reference time $t := t_{\text{CAV}} + \Delta_{\text{SYNC}}$ for both the client and the anchors we still have to ensure the precisely timed emission of the ultrasound chirp – i.e. with an imprecision within the range $[-\frac{1}{2}\,\mu\text{s}; +\frac{1}{2}\,\mu\text{s})$ according to Table 5.1[p74]. Therefore we once more rely on the dynamic self-calibration scheme for scheduling delays as presented in Section 5.4.1 and Listing 5.3[p82]: Lines 14 – 23 in Listing 11.1 show the corresponding realization within $T_{\text{US}}$ on the mobile node. The task also compensates the considerable but unpredictable influence of the system load on the task reactivity: Since the emission lateness will only affect the current position estimation, the actually measured distances can immediately be corrected by $\Delta_{\text{dist}}$ during stage $P_3$ as computed in Lines 25 – 29.

The chirp itself is generated by means of a precisely defined number of pulses of a rectangular digital 40 kHz signal which drives the ultrasound transmitter as depicted in Figure 11.2a, and enforces a deliberately shaped oscillation characteristic at the receiver as depicted in Figure 11.2b. This will turn out as a relevant detail for the precise determination of the chirp's rx time through our Cut algorithm.

[1] First, this would simply not be necessary but consume valuable energy, and second, we do not want to take influence on coexisting subsystems which might need to establish their individual time synchronization scheme.

[2] Letting an anchor take the role of the synchronizer would either require a dedicated node (which differs from the others) or a dynamic selection algorithm (which involves additional management efforts).

[3] For an expected clock drift of ≈ 20 ppm (→ Figure 5.3[p75]) we thus implemented the task $T_{\text{US}}$ to not exceed a WCET of 25 ms per iteration. The maximum expectable clock drift during one iteration ranges in $\pm 20\,\text{ppm} \cdot 25\,\text{ms} = \pm\frac{1}{2}\,\mu\text{s}$ then, and perfectly matches the *Smart*OS temporal resolution of 1 µs and the timestamping accuracy.

```
1  OS_TASKENTRY(tUS_Client) {
2
3    while (1) {
4
5      /* Wait infinitely for the evStart event to start a measurement process:
6         Will be triggered by an application task through the SNoW Bat API. */
7      clearEvent(&evStart);
8      waitEvent(&evStart);
9
10     /* Send a CAV via SmartNet and get the main sync time. */
11     if (batSendCAV(&CAVHandle) != 1) goto failed;
12     batData.t^C_CAV = CAVHandle.t_tx;   <······ the radio rx IRQ timestamp as logged by SmartNet
13
14     /* Schedule the chirp emission time (apply the self-calibration
15        scheme to compensate for any scheduling imprecision). */
16     batData.tChirpSched = batData.t^C_CAV + Δ_SYNC - batData.Δ_comp;
17
18     /* Send the US chirp and save the 'true' US emission time t^C_Chirp. */
19     if (sleepUntil(&(batData.tChirpSched)) == -1) goto failed;
20     usChirp(&(batData.t^C_Chirp));
21
22     /* Get the difference between the scheduled and the true chirp emission */
23     batData.Δ_comp = batData.t^C_Chirp - batData.tChirpSched;
24
25     /* Take the imprecision in obeying to Δ_SYNC to compensate for
26        distance measurement errors: Δ_dist will be added to each
27        successively received distance (DV preprocessing). */
28     Delay_t lateness = Δ_SYNC - (batData.t^C_Chirp - batData.t^C_CAV);
29     batData.Δ_dist = lateness * computeVSound(batData.localTemperature);
30
31     /* Schedule the DV reception via SmartNet.
32        The DVs will be processed through the position estimation task T_PE. */
33     batData.t^C_DA = batData.t^C_CAV + Δ_SYNC + Δ_TOF,max + Δ_DSP;
34     if (batReceiveDVs(batData.t^C_DA) < 0) goto failed;
35
36     /* Log the reference time for later processing through the application. */
37     batUser->timeChirp = batData.t^C_Chirp;
38
39     /* Notify the position estimation task T_PE about the
40        completion of the data aggregation (see Figure 10.8b) */
41     getResource(&rBuf);        // request the broker resource  ······> passes a hint to σ(rBuf) := T_PE
42     releaseResource(&rBuf);    // release the broker resource  ······> returns rBuf to T_PE
43
44     continue;
45 failed:
46     // some error handling...
47
48   }
49
50 }
```

**Listing 11.1:** The ultrasound task $T_{US}$ on the mobile client nodes.
The precise triggering of the chirp emission in lines 16 – 23 refers to the problem of "scheduling reaction times" (based on the previously captured CAV transmission timestamp) from Section 5.2 and applies the self-calibration scheme from Section 5.4.
Notifying a task through a broker resource in lines 41/42 refers to the technique from Section 7.5.1.

An adequate synchronization test can be accomplished by a trusted external observer (e.g. an oscilloscope during the development stage) which measures the delay between the first rectangular pulse for the chirp generation at the client, and an intentionally generated signal edge at the anchor which indicates its locally expected chirp emission time (→ red and blue channels in Figure 11.2a). While this delay should exhibit a symmetry around 0 *and* be bounded by $\pm\frac{1}{2}\mu s$ according to Chapter 5, a statistical analysis is omitted here since details on the reliability of our synchronization concept have already been presented in Section 5.3 of this work. Apart, yet another synchronization test will become possible at the beginning of the data aggregation stage $P_3$, and will be discussed thoroughly in Chapter 12 (→ Figures 12.11 and 12.12[p288]).

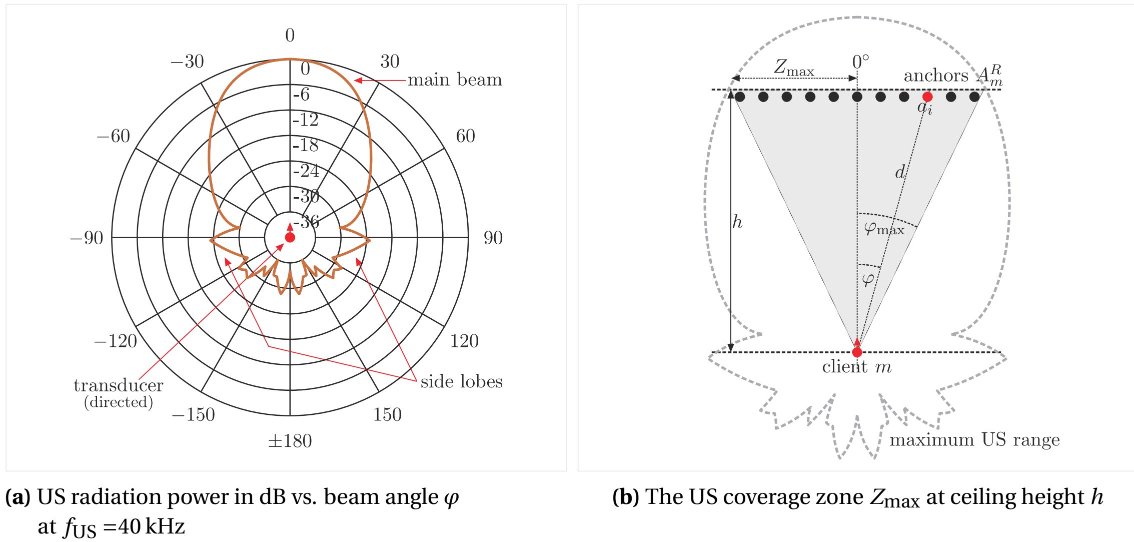
## 11.2. Digital Signal Processing

Having discussed the chirp emission by the client, the chirp detection at the anchors is the second detail we want to consider. Frequently underestimated, the challenge is to reliably identify the *beginning of the first* ultrasound wave front arriving at an anchor *and* to precisely capture the corresponding timestamp quickly and autonomously despite of the nodes' low CPU performance and unavailable hardware DSP support. Note that accidentally missing or adding $n$ complete US periods will definitely result in an individual distance measurement error of $n \cdot \lambda_{\mathrm{US}} = n \cdot v_{\mathrm{US}} \cdot f_{\mathrm{US}}^{-1} \approx n \cdot 8.6\,\mathrm{mm}$, and is likely to also affect the precision and accuracy of the position estimation algorithm during the sensor data fusion process in stage $P_4$.

Considering the rectangular generator signal from Figure 11.2a, one might expect a similarly shaped signal at the receiver. Unfortunately this is not the case, and, since both the transmitter and the receiver capsules exhibit a slow transient oscillation as soon as a voltage is applied or the wave reaches the membrane, we obtain a slowly rising shape as opposed in Figure 11.2b. In fact, this so called ramping is the main problem for most ultrasound based distance measurement systems as it is not constant regarding both the amplitude and the duration for any pair or set of capsules, but clearly depends on the distance $d$ and the angle $\varphi$ between the membranes[4]. Figures 11.1 and 11.3 illustrate these dependencies which are also discussed in [41][5].

A common approach to nevertheless capture the beginning of the incoming chirp at minimal effort is to (extremely) amplify the signal and to feed the resulting voltage into a so called capture/compare unit (as provided by most modern MCUs) which would trigger an interrupt as soon as a certain threshold voltage is reached. As an example Figure 11.2b shows the chirp oscillating around a carrier voltage offset of 1.44 V, and a trigger threshold of 1.65 V. The question remains how to reasonably select both the amplification factor and $\Delta V := V_{\mathrm{cap/comp}} - V_{\mathrm{carrier}}$. While a low amplification would result in a slow ramping and late triggering at large distances or angles (→ Figure 11.3b) and thus requires a small value for $\Delta V$, stronger amplifications would trigger earlier then (→ Figure 11.3a) but also increase the amplitude of the signal noise and consequently demand for larger $\Delta V$ values to not mistakenly interpret noise as signal. In our test beds this contradiction resulted in severe angle and distance related measurement errors

[4] Material and construction related issues are yet another factor which, however, won't be discussed here.
[5] Practical work conducted in conjunction with this work.




**(a)** US radiation power in dB vs. beam angle $\varphi$ at $f_{US}$ =40 kHz

**(b)** The US coverage zone $Z_{max}$ at ceiling height $h$

**Figure 11.1.:** US ranging depending on the node constellation and room geometry

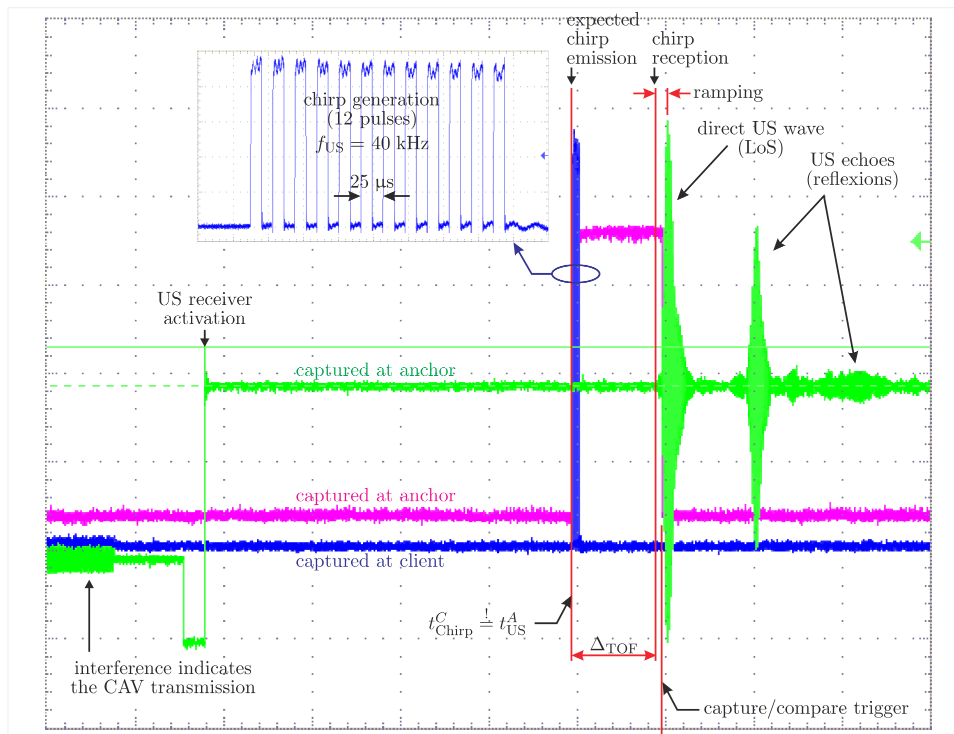
$e_d$ of up to 10 cm even for rather short distances of up to 2 m as depicted by the blue graphs in Figure 11.4a[6]. In particular it is not needed to emphasize that these errors cannot be easily compensated unless the relative position of the sender (towards the anchors) is known – an unattainable precondition since we precisely undertake the effort to obtain this still missing information in Stage $P_4$.

Our solution to the ramping problem takes an entirely different approach: Since the mobile sender emits a well-defined number of exactly 12 impulses during the chirp generation, the envelope curve of the sampled chirp at the receiving anchor exhibits an almost constant shape as shown for two different distances in Figure 11.3. Though the amplitude does still depend on the distance $d$ and the angle $\varphi$, the influence of both parameters fortunately proved to be comparable or hardly distinguishable, and the sampled curves are almost matching except for the voltage scaling in $y$ direction and the temporal shift in $x$ direction. Thus, our Cut algorithm samples the incoming chirp and applies a lightweight DSP to analyze the captured envelope characteristics regarding a constant *reference envelope* which has previously been recorded during the development of the SNoW Bat system[7]. Since the reference signal envelope has been captured at a well-known distance $d$ and $\varphi = 0°$, the distance for any captured chirp can be computed as follows (see Figures 10.3[p227] and 11.3b for identifiers):

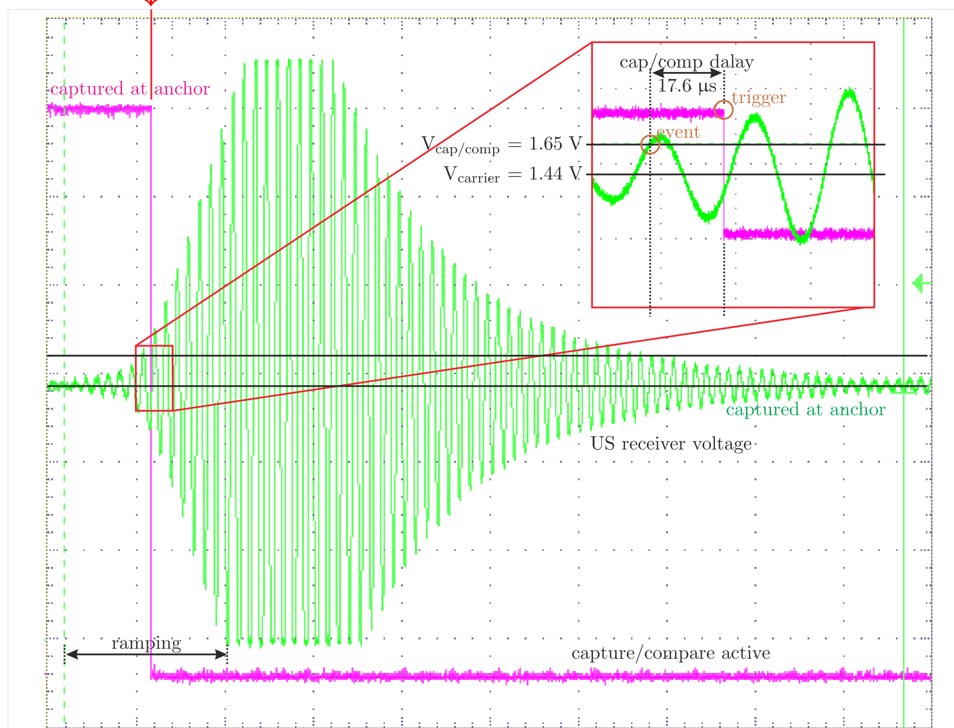
1. Sample each incoming chirp into a sufficiently large ring buffer: At the expected chirp emission time $t_{US}^A := t_{CAV}^A + \Delta_{SYNC}$ start the ADC and DMA peripherals, and wait for the capture/compare IRQ to indicate that the chirp has already arrived. Keep sampling for

[6]Since the applied MSP430 did only allow certain capture/compare voltages like e.g. 1.65 V = $\frac{1}{2} V_{CC}$ we selected $V_{carrier}$ = 1.44 V experimentally to obtain an adequate value of $\Delta V$ = 0.21 V for our Cut algorithm.

[7]Please note that clipping might cut the signal at the maximum sampling voltage $V_{CC}$ and removes valuable information which could otherwise be exploited to determine the maximal amplitude and to count back to the beginning of the chirp.

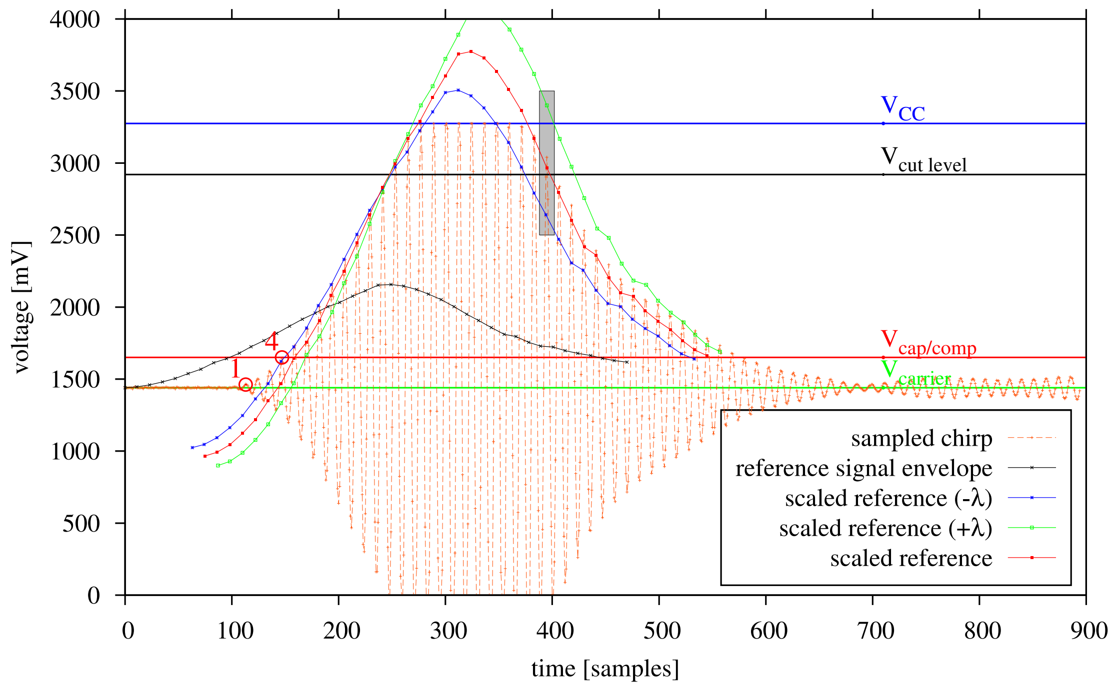



**(a)** Signal overview: Chirp tx and rx as seen by a global observer

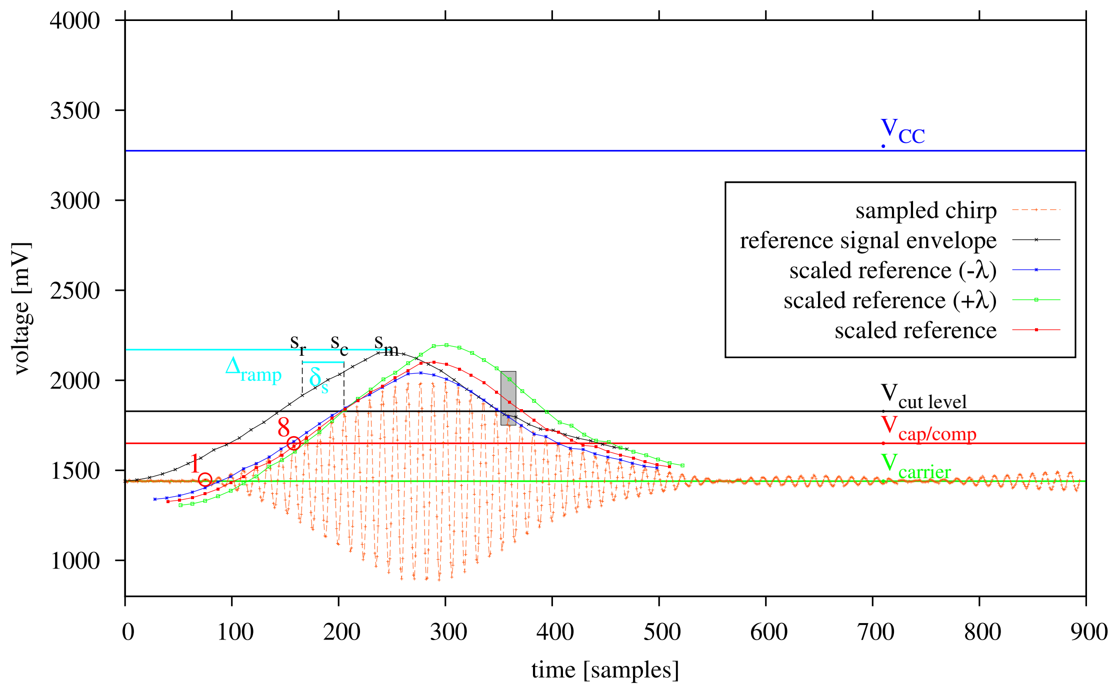



**(b)** Signal closeup: The deliberately shaped chirp and its detection via a capture/compare unit

**Figure 11.2.:** Ultrasound chirp generation (blue, rectangular signal), reception sampling (green, rx capsule), and detection (red, capture/compare)




**(a)** Short distance ($d$ = 1204 mm at $\varphi = 0°$):
Significant clipping but early capture/compare at 4th peak
Cut algorithm matches red line (no refinement)




**(b)** Long distance ($d$ = 1913 mm at $\varphi = 0°$):
No clipping but late capture/compare at 8th peak
Cut algorithm matches blue line (refined by $-1\lambda$)

**Figure 11.3.:** The Cut algorithm at short and long distance (small and large angles are comparable)

another 1.5 ms to make sure that the entire chirp has been captured but its beginning has not yet been overwritten cyclically. Save the timestamp $t_{\text{stop}}$ when deactivating the DMA and the corresponding sample index $s_{\text{stop}}$.

2. Determine the maximum gradient between two peaks of the sampled 40 kHz chirp signal, and use their average voltage as the so called *cut level* as illustrated in both Figures 11.3a and 11.3b. The sample index of the first one of those two peaks will be called $s_c$.

3. Count the number of sampled ultrasound amplitudes above the cut level and remove the same number of points beginning from the top of the reference envelope leaving the rightmost untouched point during the ramping as reference point. The sample index of this point will be called $s_r$. Consequently $\delta_s := s_c - s_r$ denotes the sample distance between the two peaks. Note, that considering the maximum gradient instead of the maximum peak will implicitly solve the clipping problem where the incoming signal voltage exceeds the ADC sampling range $[0\,\text{V}; V_{\text{CC}}]$ (→ Figure 11.3a).

4. Scale the reference envelope along the y-axis until the reference point voltage (i.e. the sample $s_r$) matches the cut level voltage, and shift it along the x-axis to match the captured signal's maximum gradient (i.e. the sample $s_c$). The anchor's absolute local start time of the chirp consequently computes as

$$t^A_{\text{Chirp}} = (s_m + \delta_s + s_{\text{stop}}) \cdot f^{-1}_{\text{DMA}} + t_{\text{stop}} - \Delta_{\text{ramp}} \tag{11.1}$$

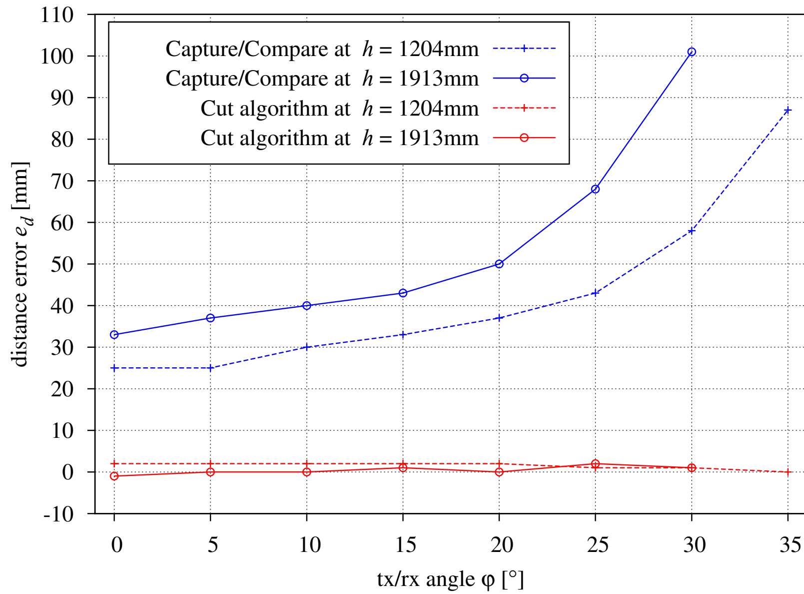
with $s_m$ being the known sample index of the largest amplitude in the reference envelope, and $\Delta_{\text{ramp}}$ being the known temporal delay between the beginning and the maximum of the reference chirp (i.e. the previously determined ramping duration).

5. In order to refine the just computed timestamp we shift the already matched reference curve by an additional $\pm 1\lambda_{\text{US}}$, and check for a proper match at the cut level during the fading of the chirp (see gray boxes in Figure 11.3). While the initial result (red graph) is almost perfect for Figure 11.2a, the blue graph in Figure 11.2b yields a better result then and $t^A_{\text{Chirp}}$ will be adjusted accordingly.

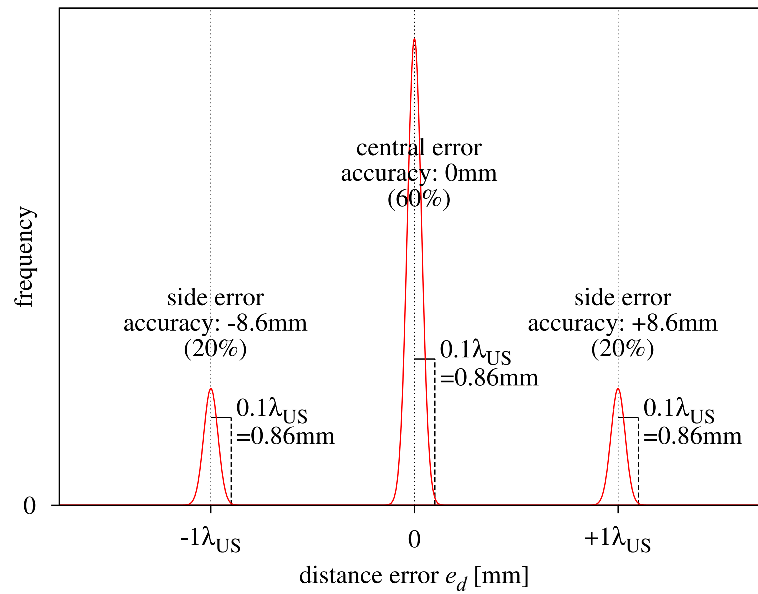
6. Recalling Figure 10.3[p227], the measured distance between the mobile client $m$ and the static anchor $a_i$ finally computes as

$$\tilde{d}_{m,a_i} = v_{\text{US}} \cdot \Delta_{\text{TOF}} = v_{\text{US}} \cdot (t^A_{\text{Chirp}} - t^A_{\text{US}}). \tag{11.2}$$

While the red graphs in Figure 11.4a show the almost distance and angle independent measurement error $e_d$ of our Cut approach in the average case, Figure 11.4b indicates the typical error distribution histogram and reveals an important fact for the pVoted position estimation algorithm based on it (→ Chapter 13): Opposed to the capture/compare method where the measurement error is almost uniformly distributed over the entire imprecision range, the Cut algorithm limits the error to three significant but quite narrow peaks with a distance of exactly $\pm 1\lambda_{\text{US}}$. Since each peak exhibits a normal distribution and a width of about $\pm 0.1\lambda_{\text{US}}$ they are




**(a)** Distance error $e_d$ vs. tx/rx angle $\varphi$ for various ceiling heights $h$




**(b)** Error distribution histogram using the Cut algorithm, $p(e_d \geq 10\,\text{cm}) \approx 0.1\%$

**Figure 11.4.:** Ultrasound distance measurement error characteristics at 20 °C and $f_{US} = 40\,\text{kHz}$ ($v_{US} = 343\,\text{m/s}$, $\lambda_{US} = 8.6\,\text{mm}$)

clearly separated and the emerging side errors can commonly be identified and removed during the final position estimation. According to our terminology from Figure 9.6[p216] this equals a precision of about ±0.86 mm and an accuracy of either close to 0 mm or ±8.6 mm. While various long-term tests within the real system revealed that the central error comprises about 60% of the measurements, the side errors comprise about 40% in total and are a result of imprecise curve matching during the refinement procedure just described. Finally we also receive a 0.1% chance for gross errors with $e_d \geq 10$ cm which result from arbitrarily delayed hardware interrupt processing[8] and the inevitably associated violation of real-time demands; these, however, can be neglected in practice.

## 11.3. Summary

In this section we presented a fast, precise, and quite accurate DSP technique for simultaneously determining the distance between one sensor node (the client) and an arbitrary number of further nodes (the anchors) based on deliberately shaped ultrasound chirps. In particular we managed to resolve the angle and distance dependency which frequently leads to severe problems in similar systems such as [92]. Regarding the design space demands from Section 9.2.3 we achieved to improve the localization performance and scalability (→ LS1, LS5[p218]) by reducing the measurement noise through clearly separating less accurate distances from more accurate ones, and by reducing the inter-node communication overhead to a single radio broadcast. The problem of efficiently collecting the distance information will be discussed in Chapter 12. Regarding the resource efficiency (→ LS2[p218]) we limited the hardware overhead to a simple ultrasound module on the anchors[9], and reduced the CPU and memory load to an extend which proved to be adequate for typical sensor nodes – at least in contrast to comparable but significantly more complex methods like curve fitting through e.g. cross correlation, or spline interpolation [41]. Safety and security issues (→ LS3[p218]) were intentionally omitted. The next obvious steps are to reliably and efficiently collect the measured distance information at the mobile client, and to process the aggregated data for obtaining a precise and accurate position estimation in a decentralized and autonomous manner.

[8]Most commonly these delays were caused by the non-interruptible *Smart*OS kernel mode.

[9]Note, that, though using RSSI based measurements would have been possible without further hardware, neither the precision nor accuracy would have been comparably good then. See [259] for details.

# 12. Efficient Data Aggregation: The HashSlot Algorithm

**Abstract**

The wireless collection of remotely emerging or generated data and information at a common sink is a well-known problem in the area of wireless sensor/actuator network research. The obvious challenge behind this so called *data aggregation* process is to reduce the *communication cost* by reasonably sharing the exclusive radio channel resource among the participating transmitters to optimize reliability, delay, predictability, throughput, and energy efficiency of all involved sensor nodes and the overall system. The less obvious challenge is to also reduce the *fusion cost* through an adequate traffic shaping and information limitation. In the special context of decentralized localization systems reliable many-to-one communication is a central but hardly addressed aspect: Compared to the position or location estimation itself, the involved communication effort and resulting information processing overhead regarding the just mentioned metrics is often ignored entirely.

We present the novel HashSlot TDMA protocol and its extension HashSlot$^+$ for the wireless but definitely collision-free aggregation of radio packets originating from multiple sources (anchors) at a common destination (client) within a constant and deterministic time. By semantically exploiting both statically available *and* dynamically – yet implicitly – emerging information from various sources, our approach exhibits the maximum achievable performance in terms of throughput, traffic volume, packet loss, and energy consumption for our special use case. Therefore HashSlot needs neither a central intelligence for generating and distributing transmission schedules nor any explicit coordination between the sensor nodes. Instead it relies on locally available information for self-organization to preserve a maximum of autonomy for each anchor. In addition, our approach offers various frequently requested features like selectable quality of service levels to dynamically limit the returned information to a “useful” amount at runtime without losing its just mentioned advantages. Thereby, both the spatial position estimation quality and the speed of the localization process can be adjusted and traded against fault tolerance and energy consumption.

## 12.1. Introduction

In Section 10.1.1 we have already introduced the general operation principles behind our decentralized SNoW Bat indoor localization system, and identified the four central stages $P_1$ – $P_4$ (position prediction, distance measurement, data aggregation, and position estimation) as quite typical within comparable approaches (like e.g. [91]). In Sections 10.2.3 and 10.2.4 we further discussed the impact of both the software architecture and the wireless communication on the overall localization performance and the position estimation update frequency[1]. This chapter will consequently focus on the demand for efficient measurement data aggregation in such decentralized approaches where a dedicated node – the self-localizing mobile client in our case – needs to reliably and quickly collect a sufficient and adequate amount of recently generated distance information from the infrastructure – represented by the pre-installed static anchors in our case – to eventually compute a position estimation.

Intense research in the general area of wireless communication schemes has already brought forth an almost countless number of protocols operating at various layers of the ISO/OSI model: Though intrinsically standardized, these layers are often reduced or merged for efficiency reasons regarding code, implementation, runtime, and data size overhead. While most of these protocols are general purpose approaches which strive to fulfill the various demands of many applications at best effort – occasionally with specific modifications – they do most commonly not achieve the optimal performance in the particular context. Only few protocols exploit the individual properties as well as the valuable emerging runtime information to become quality aware regarding the use case they were integrated in or adapted for[2]. Even worse, many protocols (e.g. token passing with fixed order) enforce their own philosophy, and require the application to adapt to their rules and laws. In the spirit of the already demanded aim for compositional software design from Section 1.2.1 this strategy might seem to be quite advisable at the first glance, since a multitude of different requirements must be fulfilled if initially independent – but nevertheless concurrently running(!) – software components need to be integrated on the same hardware platform[3]. Examples are mobile devices like e.g. the recently upcoming Smartphones which introduced far-reaching capabilities along with intense end-user interaction and customization with hard to predict characteristics [47]. However, for the special case of (autonomous and autarkic) wireless sensor nodes with limited energy budgets and considerable real-time demands, a second glance reveals various advantages of application-tailored communication protocols. Except for linking sensor nodes with other networks (→ e.g. [210]), where concrete protocols are often stipulated for compatibility reasons and to comply to industrial standards, specific solutions should not be underestimated since intensely exploiting the specific knowledge about the application's individual properties and the emerging runtime information will commonly help to achieve another essential goal from Section 1.2.1: Quality

[1] In fact, all systems from Table 9.1[p222] focus on hardware and algorithms for position estimation and the tracking of mobile objects, but hardly discuss wireless communication regarding the concerns mentioned previously.

[2] Apart it saves considerable efforts to apply well-established mechanisms instead of inventing new ones.

[3] Related resource management problems and solution strategies have already been discussed thoroughly in Part II of this work.

awareness through the semantic use of (dynamic) data and information.

For the special case of sensor data fusion in centralized indoor localization systems we thus developed the context-aware HashSlot data aggregation protocol with the goal to collect an adequate amount of information (this is not necessarily the maximum possible amount, though!) from a minimum of transmitted data. Though highly specialized, our approach removes almost all overhead in terms of time and energy consumption to the achievable minimum, and allows us to reduce both the communication cost *and* the fusion cost (as also demanded for e.g. routing algorithms in [189]).

## 12.2. Motivation and Requirements

WSN installations commonly consist of a more or less large number of sensor nodes, and, from time to time, a rather small but variable subset of nodes needs to transmit information to a common destination. Since – in the context of localization systems – the precision of the final position estimation does not only depend on the precision and accuracy but also on the amount and timeliness of acquired spatial measurements, it is important to guarantee the almost immediate availability of a certain minimum of information at a common sink.

Along with the accomplishment of various temporal and spatial measurements, and the position estimation itself, the sensor data aggregation can thus be considered as an equally relevant task in localization systems, and therefore deserves special attention[4]. While the main software/network co-design aspects of our HashSlot approach can already be found summarized in Figure 12.2, the accompanying challenges will be considered in this Section, and eventually lead to the definition of a design space for appropriate solution techniques and algorithms.

### 12.2.1. Existing Problems regarding the Communication Cost

**High radio load through quasi-simultaneous event detection.** Since distributed sensor networks are typically used to monitor environmental conditions and to instantly report observed changes, it is quite common that spatially distributed events will be detected almost simultaneously by different sensor nodes. Examples can be found in seismic surveillance, weather observation, and material surveillance systems [155]. For event-driven distributed systems in particular the locally derived information (which is associated with each relevant event) will also become available almost simultaneously, and the impending notification will lead to several timing and transmission related problems on the shared communication channel. In the context of a many-to-one data aggregation process transmitting on such a shared channel obviously needs sophisticated coordination to reduce (avoid) data loss through packet collision, i.e. radio interference, *and* to increase (maximize) the data throughput[5].

[4] Note that this chapter intentionally deals with *data aggregation* while *data fusion* will be merely considered but mainly addressed in the context of the position estimation algorithm pVoted in Chapter 13.

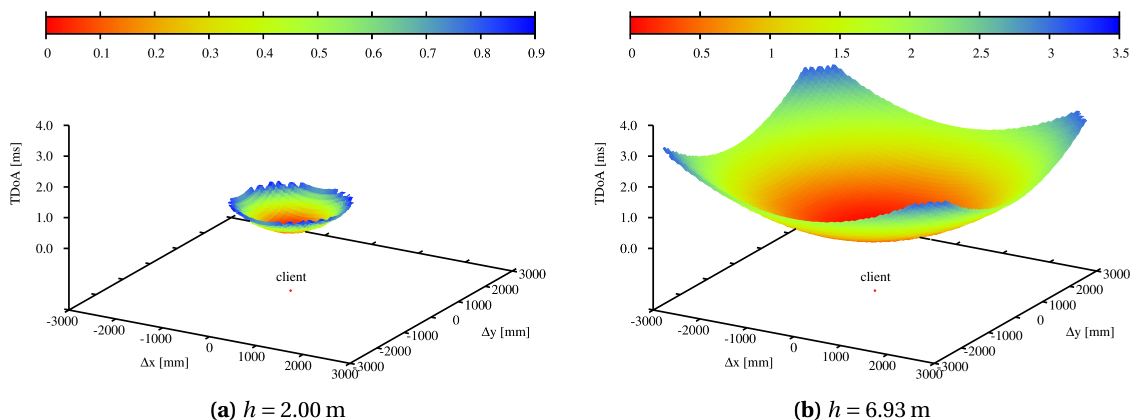
[5] According to the resource classification framework from Figure 3.2[p37] the radio channel in particular takes considerable physical (i.e. electromagnetic) influence on the transducer's surrounding, and thus is a dynamically as well as temporarily shared exclusive resource. This is a quite challenging problem since there is no central intelligence – comparable to e.g. the resource manager within an operating system kernel – which might grant access rights reliably or even protect the resource from unauthorized use.

For SNoW Bat the delay between the first and the last anchor's chirp detection – also denoted as the time difference of arrival $\Delta_{\mathrm{TDoA}}$ – depends on the current distance $d$ of the client from the anchor plane and the US emission angle $\varphi$. According to Figure 10.5b[p231] it is thus bounded through the length of the interval $I_{\Delta_{\mathrm{TOF}}} := \left[\frac{d}{v_{\mathrm{US}}}, \frac{d}{\cos\varphi \cdot v_{\mathrm{US}}}\right]$ after the chirp emission and results in $\Delta_{\mathrm{TDoA,max}} := \frac{d(1-\cos\varphi)}{v_{\mathrm{US}} \cdot \cos\varphi}$. With the specifications from the SNoW Bat system and an assumed $v_{\mathrm{US}} = 343\,\mathrm{m/s}$ we receive $I_{\Delta_{\mathrm{TOF}}} \approx [20.2\,\mathrm{ms}, 23.3\,\mathrm{ms}]$ for the largest distance $d = d_{\mathrm{sup}} \approx 6.93\,\mathrm{m}$, and $\Delta_{\mathrm{TDoA,max}} \approx 3.1\,\mathrm{ms}$; this time difference is notably shorter than the transmission and processing time for a single DV. Figure 12.1 shows the TDoA distribution for various ceiling heights.

**Information loss through sporadic radio unavailability.** Even if we manage to solve the just described problem, the sole achievement of collision-freedom is not sufficient by far since the receiving client requires some additional time to (pre-)process an incoming packet, i.e. to read the radio transceiver's RX buffer and to re-enter RX mode. Despite of our event-driven software design, sporadic resource competition (e.g. for the interconnection bus between the MCU and the radio device) might further delay or extend the packet handling, and will lead to variable periods of radio inattention then: Though invisible for the anchors, this phenomenon will cause additional information loss at the intended data sink since any subsequent packet won't be received even if it was scheduled properly to not collide. In case of SNoW Bat this preprocessing (mainly accomplished through *Smart*Net from Section 8.1) takes about 2.7 ms (measured), and we consequently have to avoid a shorter inter-packet spacing carefully.

**Energy, memory, and performance waste through greedy packet reception.** Receiving information wirelessly is a power consuming issue (→ Table 2.2[p27]) and should thus be reduced to a minimum whenever possible. In the context of intentionally triggered measurements with subsequent data collection, most approaches operate greedily, and try to receive and buffer every single reply packet. While this can rapidly lead to a significant energy and memory overkill if more packets then necessary are returned – a fact which won't become obvious before the end of the data fusion process(!) –, they also don't know when to stop listening since there is commonly no chance to reliably identify the last packet to come – there might still be another one pending. A solution would be to process each packet progressively, i.e. immediately after reception, and to determine if more information is required at all. Otherwise the reception can simply be stopped. At the same time it would be desirable to obtain a priori information for precisely predicting the total reply stage duration.

**Data aggregation interference through concurrently operating clients.** The last problem we want to consider explicitly refers to the protocol scalability with regard to the number of coexisting clients. As each client might potentially demand for an individually high localization frequency, the data aggregation stages might often overlap, and an appropriate interleaving strategy must be found. Once more, consulting a central master or using a (weighted) round robin scheme would be a solution. However, though this would keep up the data aggregation time itself, it would still share the system time among the clients and introduce additional




**(a)** $h = 2.00$ m **(b)** $h = 6.93$ m

**Figure 12.1.:** TDoA of the US chirp between the closest anchor (directly above the mobile node) and anchors at arbitrary positions $(\Delta x, \Delta y, h)$ for various ceiling heights $h$ and limited to $\varphi \leq 30°$

communication effort[6]. Beyond it would destroy our demand for a strictly self-organizing scheme, and introduce a single source of failure. Thus, another solution must consequently be found.

### 12.2.2. Feature Requests regarding the Data Fusion Cost

**Optimizing the data fusion cost through a reasonable transmission order.** The considered problems so far did mainly relate to the communication cost within the network of distributed systems. From the perspective of the individual local embedded system, which is finally responsible for the data fusion process, it is highly desirable to obtain this data ordered by decreasing information value. While it is almost impossible for each individual data source to evaluate its own quality without knowing the data from other sources in advance, this would nevertheless be a true benefit for most progressive (sensor) data fusion algorithms – at least for those, which, in the context of quality awareness, continuously monitor the changing outcome of their iteratively emerging results in order to stop as soon as either an evaluation function did reach a certain threshold or as a timeout has been reached.

Thus, an improvement for any data aggregation protocol would be to reorder the transmissions in a way to increase the probability for obtaining the maximal utility for the data fusion algorithm, i.e. to receive the most relevant (or “interesting”) data first. The problem however is to accomplish this feature *without* additional and unreasonable inter-communication cost[7] for determining (from an e.g. distributed selection algorithm) which sensor node may send first. Yet another reason for us to adapt the protocol to the application (rather than vice versa).

[6]Depending on the spatial extend of the localization system installation, even multi-hop communication might be required then (involving challenges like the hidden/exposed node problems). Fortunately, this is a problem which we do not need to consider for SNoW Bat.

[7]This would lead to increasing time and energy demands which we just intended to reduce.

**Reasonably limiting a surplus of information.** Being able to efficiently collect information even from a large number of sources is desirable indeed, however, aggregating too much information can also be inadequate – or at least useless – as soon as the data fusion algorithm runs into saturation and the result won't improve considerably from even more information. In order to reduce unnecessarily high traffic volumes, which would jam the radio channel while wasting time and energy for transmission and processing, it would be wise to a priori limit the number of transmissions. Comparable to the last point, the question will be how to forecast the amount of required or desired information, and how to select an adequate subset of sensors from those which have the relevant information available – again, an implicit selection would be preferable considering the additional effort for an explicit selection.

### 12.2.3. Feature Requests regarding the Network Maintenance

**Reliably detecting and identifying defective nodes.** Finding an optimal number and placement of sensor nodes to establish a useful infrastructure for a concrete application has already been indicated as a relevant and demanding optimization problem in Section 10.2.2. Even if a reasonable deployment was finally conducted to ensure a sufficient service coverage, the intentionally cheap hardware (→ Chapter 2) and the sometimes hazardous environmental conditions may lead to quite high node failure rates at runtime. Then, the network reliability and infrastructure maintenance would certainly benefit from communication protocols which do not only coordinate the actual data transmissions, but which can also detect *and* identify those nodes which do not transmit any more. Since it is commonly hard to determine (e.g. in a distributed manner) which node should participate in a certain case, this feature demands for a flexible topology control and situation awareness. Again, this is especially true in highly dynamic environments with partial node participation during a data aggregation stage.

### 12.2.4. The Protocol Design Space

Having introduced the most relevant problems and feature requests regarding the data aggregation in event-driven and distributed monitoring systems, even more specific demands for the special use case of decentralized (indoor) localization systems remain to be discussed in Section 12.6[8]. Nevertheless, we will next present the concrete protocol design space with respect to the WSAN optimization problem from Definition I.5[p7] and the general design space from Section 6.2.2[9]. The points marked with ■ (□) are (not) solved by HashSlot or HashSlot$^+$:

HS1 RUNTIME PERFORMANCE. *Minimize the data aggregation delay and optimize its duration*:

a) ■ Adjust the number of transmissions to
   - avoid more traffic than necessary
   - allow as much traffic as required

[8] These comprise traffic shaping and reasonable information selection as well as the reliable detection of defective nodes.

[9] If you feel like having a déjà vu when reading these lines, then this is partially true! In fact, the design space we have been using for the resource and memory managers in Chapters 6 and 7 comprised the same "main dimensions".

b) ■ Adjust the inter-transmission spacing to
   - minimize temporal overhead
   - reduce the impact of temporal radio unavailability at the receiver
   - allow for progressive information processing

c) ■ Allow the first transmission to take place with minimal delay after the event detection or information emergence

HS2 RESOURCE EFFICIENCY. *Maximize information exchange at minimum cost*:

a) ■ Local: Reduce energy consumption through unnecessary radio activation
b) ■ Global: Reduce administrative overhead for the channel access management
c) ■ Network: Avoid the need for retransmission and RTS/CTS strategies

HS3 SAFETY AND SECURITY. *Maximize reliability and minimize susceptibility*:

a) ■ Avoid packet loss, e.g. through collisions
b) ■ Provide a means to identify (or even locate) defective source nodes
c) □ Ensure data integrity, and prevent spoofing as well as unauthorized information sniffing

HS4 USABILITY. *Provide a means to achieve runtime quality awareness*:

a) ■ Support the deterministically adjustable duration of the data aggregation process
b) ■ Reflect changing system demands through dynamically adjustable information density
c) ■ Maximize information value and data utility through a reasonable transmission order

HS5 SCALABILITY. *Maximize compositionality*:

a) ■ Stay independent from the overall number of nodes
b) ■ Automatically compensate for node failures, and avoid unused channel reservations
c) □ Allow for coexisting radio protocols (on each node and within the network)

## 12.3. Related Work – Wireless Data Aggregation Protocols

The distributed character of wireless sensor/actuator networks and the associated problems concerning the radio communication led to the development of a myriad of data aggregation protocols regarding the just mentioned demands and feature requests from Section 12.2. Before we present the details about our novel HashSlot protocol, we will thus briefly outline some state of the art data aggregation approaches for coordinating numerous concurrently transmitting source nodes in event-driven WSAN applications. For instance, [284] proposes a generic approach which takes the temporal correlation in the sensor data into account: A prediction algorithm is executed on both the sources and the common sink, and a radio packet will only be sent if the source detects a significant derivation between the measured and the predicted value;

otherwise, the sink will implicitly use the predicted value. While this approach aims on reducing the network traffic and payload size in general, neither the actual medium access scheme nor the routing from the sources to the sink is discussed. Regarding these issues, [88] presents classifications and design challenges. Surveys on data collection algorithms for wireless sensor networks can be found in e.g. [177, 179, 295]. MAC protocols with special focus on mission critical WSN applications are discussed in [272].

The considered protocols can in general be divided into *multi-hop* and *single-hop* approaches with both *proactive* and *reactive* focus: Single-hop protocols target on providing a reliable MAC scheme to avoid radio interference and data loss within the communication radius of each directly involved node. In contrast, multi-hop protocols strive to extend this one-hop communication range through robust routing schemes which would allow to forward information over multiple intermediate nodes. While hybrid approaches can commonly be designed to better reflect the specific needs of the actual application, reliable single-hop MAC protocols are always required first when designing robust multi-hop routing protocols.

**Multi-hop (routing) protocols.** In the context of multi-hop protocols, the term proactive refers to the establishment of fixed communication paths between the sources and the destination [254]. Depending on the system dynamics and the link reliability, these fixed paths can either be established just once during the initiation stage of the distributed system, or periodically at the beginning of so called rounds. Reactive multi-hop protocols establish a communication path on-demand [137], i.e. for each single data aggregation, and are consequently more suitable for dynamic topologies or unstable links. In any case, however, most routing protocols try to establish a hierarchical topology where so called clusters of nodes are managed by dedicated cluster-heads which themselves form clusters, and so on. Depending on the just partial and dynamic node participation in most data aggregation scenarios, the actual cluster generation can thereby aim on e.g. balancing the fairness [101], ensuring the reliability [188], reflecting selective queries [162], or considering energy demands [149, 205, 323]. A prominent example for a purely proactive approach and round-based clustering is LEACH [123]: Meant for continuous environmental observations, it creates static routes for strictly periodical transmissions, and assumes that each node has always some data to send. Besides, TEEN [195] and EERP [286] refine the LEACH approach by also building the routing paths proactively, but using them reactively on-demand just in case a measured value exceeds a certain application-specific threshold.

**Single-hop (MAC) protocols.** In the context of single-hop protocols, the term proactive refers to regularly transmitting nodes (e.g. for the periodic data aggregation in continuous monitoring systems), and the term reactive applies to more event-driven scenarios (e.g. on-demand data aggregation in case of sporadic events). A two-class classification can be found in e.g. [316]: Proactive MAC protocols often apply (dynamic) time schedules (TDMA) to coordinate the channel access over long periods either autonomously as in e.g. [207], or through a master as in e.g. [58]. Reactive MAC protocols like e.g. Sift [143], Crankshaft [119], or B-MAC [228] often apply contention based techniques (CSMA) to check for a currently free radio channel

just when required, and rely on certain back-off strategies to potentially delay the transmissions. While the frequent clear channel assessment (CCA) consumes a significant amount of energy, back-off strategies are also hard to configure for achieving high reliability at low delay. In fact, collision-freedom can commonly not be guaranteed here due to the already mentioned switching delay (and temporary radio unavailability) between CCA and TX mode. "Optimized for reliability in scenarios with high node density and highly correlated event-driven data traffic", BPS-MAC (Back-off Preamble Sequential MAC) [156] overcomes this problem by introducing a preamble-based contention resolution: Each node sends a preamble of randomized length first, immediately switches to CCA mode thereafter, and only sends the actual payload if no other preamble is received then. While this method needs no central coordinator and supports dynamic topologies, it might require many retries for nodes with short preambles, and it is non-deterministic regarding the total duration of the data aggregation. In contrast, BitMAC [244] exploits a static star topology for generating a dynamic schedule based on periodic beacon packets: Each parent node coordinates the TDMA transmission of its children. Using parent specific radio channels to also avoid collisions in multi-hop networks, these children concurrently transmit one-bit allocation requests using OOK (on-off keying) modulation. Perfectly synchronized, these packets contain $n$ well-assigned bits for $n$ children, and intentionally interfere to result in a bitwise "or" on the radio channel. In turn the parent returns the allocation request through a broadcast, and thereby implicitly negotiates the schedule for the immediately following data transmission. Though collision-free, the number of nodes per parent must be known a priori, and dynamic topologies with mobile nodes are hard to support. The same is true for another integrative approach (MAC, topology control, and routing): Designed for reducing idle listening and overhearing in periodic data collection scenarios, Dozer [58] uses a tree-based network structure of arbitrary depth (the so called gathering tree) where parents schedule precise TDMA rendezvous times to their children.

### 12.3.1. Potential Candidates for SNoW Bat

Despite of the availability of the just presented wireless single-hop MAC protocols, these show significant deficits regarding our demands from Section 12.2. While each of them already addresses the problem of handling radio peak load in case of quasi-simultaneous event detection and radio transmissions, only BitMAC inherently supports concurrently operating data sinks. Avoiding information loss in case of sporadic radio unavailability would already demand for some adaptations to BitMAC, Dozer, and BPS-MAC, and detecting or even identifying defective node would only be possible in case the total number and ID of the initially available nodes is known. The demand for avoiding the need for greedy packet reception, for optimizing the transmission order, and for limiting a surplus of information is not considered at all.

Regarding the decentralized position estimation scenario in SNoW Bat, the most severe problem, however, is that the data sinks (i.e. the clients) themselves are mobile, and that the source node participation (i.e. the subset of anchors within the ultrasound coverage zone) during the data aggregation stage also depends on each client's individual position. Thus, generating a static TDMA schedule is not sufficiently flexible to reflect the actual anchor subset, and actively negotiating a new schedule through explicit node interaction for each data aggregation would

be too time-consuming. Thus, we omitted the usage of these protocols within SNoW Bat, but implemented five simple but widely accepted general purpose MAC protocols for comparison within our test beds. While two of them rely on synchronized TDMA time slots, just like HashSlot, the others are free to transmit arbitrarily, and only depend on the timestamp $t^{A}_{\text{Chirp}}$ when the US chirp has been detected (→ Figure 10.3[p227]). Though more sophisticated approaches might achieve better results, they would probably still perform somewhere between our selection of alternatives and HashSlot since the latter will prove to achieve the optimum in terms of throughput, traffic volume, packet loss, and energy consumption for our special use case.

For each one of our selected methods we'll also disclose the specific timeout $\Delta_{\text{TO}}$ we applied, and derive a lower as well as an upper bound for the time $\Delta_{\text{RX}}$ the client and each replying anchor will stay in radio receive mode. In contrast, the transmission time $\Delta_{\text{TX}}$ is fixed for the anchors as each DV was sent exactly once, and neither acknowledgments nor retransmissions were used. Consequently, we set $\Delta_{\text{TX}} = 0$ for the clients. The minimal time required for performing the *clear channel assessment* (CCA) is denoted as $t^{\text{CCA}}_{\text{DV}}$. Recall, that the reply stage in SNoW Bat always uses a dedicated and client specific radio channel for each mobile node. Thus, no interference with any other radio communication – e.g. from concurrently operating clients – needs to be expected as long as one dedicated return channel is available per client.

### 12.3.2. Non-slotted Methods

As non-slotted protocols don't need to synchronize to any special clock, but only to other nodes or the observed protocol traffic (if at all), their transmission might start in general as soon as the distance information is available. Additional delays between the packets will thus be required for collision avoidance, and can be dimensioned dynamically through various *back-off* strategies.

**CSMA/CA:** In carrier sense multiple access mode with collision avoidance (CSMA/CA), each anchor initially tries to start its DV transmission with minimal delay as soon as the distance measurement has completed, but performs the CCA right before actually entering tx mode to detect if the channel is currently busy. To avoid taking the risk of a collision and consequently corrupted data or lost information, it defers the transmission by a randomly chosen time and retries. Another option is to continuously perform the carrier sense (CS) until the channel becomes free, and to start the transmission with an additional (random) delay to avoid interference with other waiting nodes using the same strategy. Since most modern radio transceivers offer convenient CS/CCA hardware support, this protocol is fairly easy to implement[10]. Yet, it unfortunately involves several serious problems: First, collisions cannot be avoided entirely as the transition from CS/CCA to tx mode takes a small but certainly not negligible time in which the channel could get occupied unnoticed. Explicit considerations about this problem and a counteracting MAC scheme can be found in [156]; however, the solution presented there is non-deterministic and introduces significant delays which would jeopardize our goal to achieve

[10] While the dimensioning of the back-off delays is not trivial in general, the channel access depends on the CCA duration, and CSMA schemes will profit from future hardware improvements. In case of the CC1100 [67] radio transceiver configuration we use for the SNoW[5] nodes the CCA takes about 120 µs.

a high localization frequency. In the special case of SNoW Bat this effect is quite likely to be observed, as we have already discussed in Section 12.2.1. Second, the so called hidden and exposed node problems[11] [164, 258] are another issue in CSMA/CA based networks: Using RTS/CTS mechanisms or acknowledgments to safeguard the transmission would not only slow down the data aggregation, but it would generate even more traffic leading to an increased packet collision probability and energy consumption.

**Brute Force:** In brute force mode each anchor starts its DV transmission as soon as the distance measurement has completed and without performing any CCA first. This mode is for testing purposes only. While a quite short timeout $\Delta_{\text{TO}}$ can obviously be chosen, it will turn out as unreliable, since we have already discussed that the time difference of arrival (TDoA) of the chirp at the anchors is commonly less than the transmission time $\Delta_{\text{SLOT}}$ for a single DV. This fact will inevitably cause radio collisions, and no information at all should finally arrive at the client.

**Random Start + CSMA/CA:** For this approach the mobile client integrates the number of requested DVs $g$ into the CAV. In turn each anchor selects a random transmission delay $\Delta_{\text{DV}} \in [0; (\gamma \cdot g - 1) \cdot \Delta_{\text{SLOT}}]$ with $\gamma \in \mathbb{N}^+ \setminus \{0\}$ being a spreading factor which can be tuned to reduce the probability for collisions while stretching the reply stage duration and the applied timeout. For further collision avoidance each sender performs a CCA first, and delays the transmission if necessary.

**Summary.** Considering our protocol design space from Section 12.2 the approaches so far can not guarantee the successful transmission of a sufficient number of DVs. They also do not obey to synchronized transmission slots, and thus the mobile node must return to rx mode immediately after each received packet until the desired amount of information has been received or the timeout is reached. Another serious point to consider is, that valid packets might arrive in quick succession at the client, i.e. with almost no inter-packet spacing, and before the rx mode has been reactivated after the previous one. For SNoW Bat in particular the already discussed information loss through sporadic radio unavailability will also complicate the application of progressive position estimation algorithms as described in Chapter 13.

### 12.3.3. Slotted Methods

In contrast to the non-slotted protocols, the slotted approaches need to synchronize to defined slot boundaries, and thus require a well-defined reference time along with a precise time management. For SNoW Bat we can rely once more on the *Smart*OS timing concept and on the synchronized timestamp $t^A_{\text{CAV}}$ which was obtained previously through the chirp allocation vector, and which has already been used as a reference for the distance measurement in Chapter 11. According to Figure 10.3[p227], we'll start the data aggregation stage $P_3$ with the first return

[11] Hidden node problem: While two senders cannot hear each other – and thus cannot rely on CCA –, their packets interfere at a common destination node which is located within the transmission range of both source nodes. Exposed node problem: While one sender detects a busy channel during CCA and consequently delays itself, the transmission would yet be successful since its intended receiver is not within range of the detected signal.

| | $\Delta_{\text{TO}}$ | $\Delta_{\text{RX}}$ (anchor) | $\Delta_{\text{RX}}$ (client) |
|---|---|---|---|
| CSMA/CA | $\Delta_{\text{TOF,max}} + \gamma \cdot g \cdot \Delta_{\text{SLOT}}$ | $\left[\Delta_{\text{DV}}^{\text{CCA}}; \Delta_{\text{TO}}\right]$ | $\left[g \cdot t_{\text{DV}}^{\text{HF}}; \Delta_{\text{TO}}\right]$ |
| Random Start + CSMA/CA | $\Delta_{\text{TOF,max}} + \gamma \cdot g \cdot \Delta_{\text{SLOT}}$ | $\left[t_{\text{CCA}}; \Delta_{\text{TO}}\right]$ | $\left[g \cdot t_{\text{DV}}^{\text{HF}}; \Delta_{\text{TO}}\right]$ |
| Brute Force | $\Delta_{\text{TOF,max}} + t_{\text{DV}}^{\text{HF}}$ | 0 | $\Delta_{\text{TO}}$ |
| Random Slot | $\gamma \cdot g \cdot \Delta_{\text{SLOT}}$ | 0 | $\left[g \cdot t_{\text{DV}}^{\text{CCA}}; \gamma \cdot g \cdot t_{\text{DV}}^{\text{HF}}\right]$ |
| NodeID | $\|A\| \cdot \Delta_{\text{SLOT}}$ | 0 | $\left[g \cdot t_{\text{DV}}^{\text{HF}}; (\|A\| - g) \cdot t_{\text{DV}}^{\text{CCA}} + g \cdot t_{\text{DV}}^{\text{HF}}\right]$ |
| HashSlot | $\left\lceil\sqrt{g}\right\rceil^2 \cdot \Delta_{\text{SLOT}}$ | 0 | $\left\lceil\sqrt{g}\right\rceil^2 \cdot t_{\text{DV}}$ |

**Table 12.1.:** Comparison of various radio protocols for data aggregation in stage $P_3$

slot at $t_{\text{DA}}^{A} := t_{\text{CAV}}^{A} + \Delta_{\text{SYNC}} + \Delta_{\text{TOF,max}} + \Delta_{\text{DSP}}$ which should be identical for the mobile node $m$ and each replying anchor in $a_i \in A_m$. The oscilloscope snapshots in Figures 12.11 and 12.12[p288] already visualize this synchronicity.

**Random Slot:** Just like for the non-slotted Random Start method, the mobile node includes the number of desired DVs $g$ into the CAV. This time each anchor $a_i \in A_m^R$ selects a random slot $s(a_i) \in [0; \gamma \cdot g - 1]$ and transmits the DV at time $t_{\text{DV},a_i}^{A} := t_{\text{DA}}^{A} + s(a_i) \cdot \Delta_{\text{SLOT}}$. Like for random start, $\gamma \in \mathbb{N}^+ \setminus \{0\}$ can be tuned to provide more slots and to reduce the probability for collisions while expanding the timeout $\Delta_{\text{TO}}$ along with the data aggregation stage duration $d_3$.

**NodeID:** As node IDs in sensor networks are commonly available *and* unique, each anchor $a_i \in A_m^R$ can easily and reliably obtain a definitely collision-free return slot by simply using its ID $\text{ID}(a_i)$ as slot number. This way, the DV return time computes as $t_{\text{DV},a_i}^{A} := t_{\text{DA}}^{A} + \text{ID}(a_i) \cdot \Delta_{\text{SLOT}}$. In case we have strictly consecutive nodes IDs, this method can be organized to be comparable with *token passing* schemes (like e.g. [91]) where a dedicated node starts to transmit, and subsequent nodes add their information in a well-defined order[12], or to cable based installations (like e.g. [267] ) where a central instance successively queries each node for its most recent distance measurements. Nevertheless it might take up to $\text{ID}_{\max} - \text{ID}_{\min} \geq |A|$ slots, and one must always consider that the number of unused slots increases with the system size since $|A_m^R| \ll |A|$ will commonly hold and result in bad protocol scalability.

**Summary.** Considering our protocol design space from Section 12.2 an important advantage of slotted methods is the option to support progressive position estimation by selecting a sufficiently large inter-packet spacing through appropriate slot lengths. Additionally, slotted approaches allow the receiver to save valuable energy by activating the rx mode only for a short time $t_{\text{DV}}^{\text{CCA}}$ at the beginning of each slot[13]. Nevertheless it is an important optimization challenge to select an adequate number of slots *and* to keep the number of unused slots low.

[12] Note the emerging problem of token loss caused by node failures.

[13] Recall that in our particular case the receiver is an autarkically operating sensor node which commonly strives to save energy.

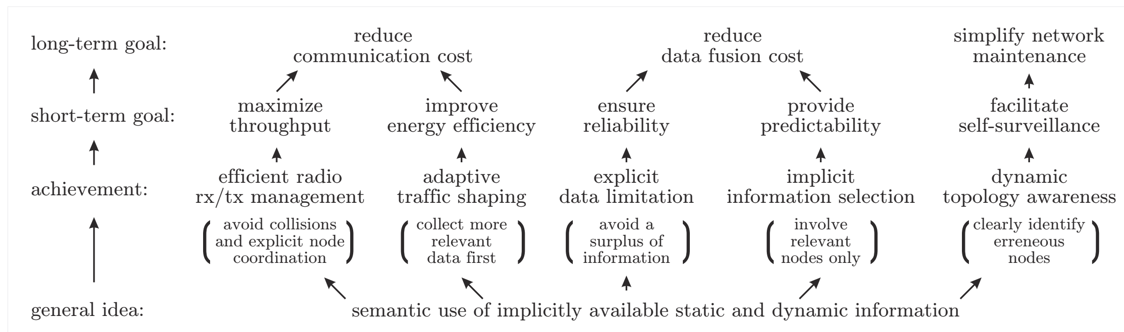



**Figure 12.2.:** The HashSlot software/network co-design: Long-term goals and how to achieve them through a reasonable and deterministic use of the shared radio channel

## 12.4. The HashSlot Protocol

The transmission of a sufficient number of distance vectors (DV) from the anchors to the mobile node is essential for a reliable and fault-tolerant location or position estimation. Yet, the required time must be kept short to achieve a high localization frequency, to save local as well as global resources, and to collaboratively avoid jamming on the shared radio channel. Though SNoW Bat was initially based on a simple CSMA/CA scheme, the performance loss during the data aggregation became unacceptable compared to the other stages, and we finally decided to design an application-tailored radio protocol according to the demands from Section 12.2: Except for HS3c and HS5c[p257], HashSlot addresses all design aspects and fulfills most of them. The underlying software/network co-design aspects are summarized in Figure 12.2.

This section addresses the basic computation of unique and tightly packed slot numbers for anchors $a_i \in A_m \subseteq A$ within a client's $m$ moving US coverage zone $Z_m$ as depicted in Figure 10.2[p226]. We will also show how situation specific QoS demands and varying environmental conditions can be considered algorithmically to achieve additional improvements. For fine-grained runtime adjustments further considerations will allow the a priori calculation of the network load, the overall data aggregation time, and the energy requirements.

**Classification and feature overview.** HashSlot is a single-hop data aggregation MAC protocol which semantically exploits some application-specific knowledge about the environment and the event locality principle (which can be observed in many event-driven WSN scenarios) to dynamically generate collision-free TDMA transmission schedules: For SNoW Bat it allows the quasi-optimal usage of the wireless communication medium regarding throughput, energy efficiency, reliability, and predictability[14]. Well aware of the precondition that anchors are mounted along a previously designed pattern with well-defined shape (→ Section 10.2.2), the slot calculation employs a mathematical *hash function* and draws advantage from the fact that only anchors within a common and contiguous area – namely the ultrasound coverage zone $Z$ – will attempt to transmit their just generated information towards a common destination.

[14] Though HashSlot can also be used for wired data aggregation (and where it could replace e.g. CSMA/CD approaches or round robin querying), we focus on wireless communication only.

The key behind the HashSlot algorithm is to let the entirely independently operating anchors combine the anyway available knowledge about their own location within the anchor grid as well as the most recent position estimation (if available) of the client they currently serve, to autonomously generate a definitely collision-free transmission slot without any further communication with other nodes or a central coordinator. This autonomy will in particular also guarantee the protocol's scalability regarding the number of nodes; in fact its performance is completely independent from the network size. The additional knowledge about the installation's ultrasound characteristics even allows to pack these slots tightly – i.e. in direct succession –, and to avoid undesired idle phases – so called *vacant slots* – on the communication channel. Yet, intentional and dynamically scalable delays between the transmissions make HashSlot particularly suitable for progressive information processing. Regarding the application's quality awareness and the clients' self-evaluation, our adaptive scheme also allows each client to autonomously and dynamically adjust the number of replying anchors for each localization process while preserving the just mentioned properties: Right after receiving a CAV, and based on the information included therein, each anchor determines algorithmically if it is requested to participate in the localization process, and otherwise omits the entire distance measurement immediately to quickly become re-available for other concurrently operation clients. It is worthwhile to note that the clients will never select certain anchors explicitly; instead they will just quantify the number of required DVs within the CAV – e.g. to trade the reply stage duration against the amount of information – without knowing which anchors will finally respond to this request. A reasonable DV transmission order will be generated automatically.

In order to react on node failures and other imponderabilities which might lead to vacant slots – for which the protocol is not reliable – the HashSlot$^+$ extension detects emerging and unforeseen dead times with minimal and mainly hardware dependent delay, and allows the remaining anchors to start their transmission early, but still strictly ordered. Though this feature requires the anchors to perform some kind of CCA at the beginning of each slot, additional supporting mechanisms like RTS/CTS or acknowledgments will never be required at all: Apart from the CAV broadcast the directed response communication will remain strictly unidirectional.

The last feature to consider at this point addresses the system maintenance in general: As the slot order computation can also be inverted by e.g. the client or a supervising system monitor to unambiguously obtain the location of the corresponding anchor, this property can finally be used to identify defective, faulty, or unreliable devices for e.g. recalibration, exchange, or repair.

### 12.4.1. Slot Calculation

**The general idea.** With respect to the often expected large scale of many wireless sensor networks, HashSlot is a self-organizing data aggregation approach in two regards: From the clients' perspective it allows to select and limit the amount of returned data and information implicitly, i.e. without explicit source selection but with respect to their dynamically varying requirements. From the anchors' perspective it allows to efficiently share the exclusive radio channel, i.e. without explicit coordination (and associated communication). The common goal of both intentions is to improve the quality of the subsequent data fusion process – i.e. the position estimation – which does not only depend on the precision and accuracy of the acquired

measurement results, but also on the amount, utility, and up-to-dateness of the contained data[15] – i.e. the information. To be successful the ability for self-organization must obviously be based on some specific information originating from both the clients and the anchors. Starting with the request for a position estimation (e.g. triggered by an arbitrary event and issued by the client's software), each client $m \in M$ embeds its current demands along with some timing specifications into the query for measurement results[16].

Received by an anchor $a_i \in A_m$ the client's information $I_m^C$ is enriched with locally available information $I_i^A$ and used as input for a hash function

$$\mathcal{H} : I_i^A \times I_m^C \mapsto S \subset \mathbb{N}_0^+. \tag{12.1}$$

As the resulting hash value $s_i \in S$ will finally be used as a TDMA slot number $s(a_i)$ to specify the sequence in which the transmissions will take place, it is highly desirable to obtain tightly packed values within a well-defined and minimal range. Regarding the number of requested DVs $g$, $|S| = g$ would be optimal. On the other hand, $\mathcal{H}$ must also be designed to always generate pairwise different values $s_i, s_j \in S$ $(s_i \neq s_j)$ for any two different anchors $a_i, a_j \in A_m$ $(a_i \neq a_j)$.

Since the CAV is simply broadcasted and the client cannot know which anchors will finally be involved in the measurement process, $I_m^C$ is always the same for each anchor. Nevertheless these anchors are not allowed to coordinate or communicate among each other (→ HS2b[p257]), and thus the key to success must obviously be embedded in each anchor's local information $I_i^A$: Though initially fixed, some situation specific information must be obtained through the semantic use of statically available and dynamically observed knowledge to enrich $I_i^A$ and to reflect the high system dynamics which inevitably results from the clients mobility.

**Realization.** Though we previously identified the almost simultaneous event detection in Section 12.2.1 as one of the most critical problems for many event-triggered WSN surveillance systems, we do exploit this very locality principle for our special use case by correlating the

- **static position**
  (i.e. the row and column index in the anchor grid, *static knowledge after deployment*)
- of each **involved anchor**
  (i.e. the ones which received the chirp, *implicit pre-selection through the US characteristics*)

to the

- **estimated spatial extent of the trigger event**
  (i.e. the size of the US coverage zone, *semantically derived by the anchors*)
- and the **client's QoS demands**
  (i.e. the total number $g$ of requested DVs, *explicitly adjusted by the client*).

[15]Note that the *amount* and *utility* of information is not necessarily the same: In the case of many localization algorithms even an arbitrary number if DVs is of no use if they originate from anchors at linear dependent positions (→ Section 12.6 and 13.3).

[16]While this query could be transmitted as a separate radio packet we include it WLOG into the CAV from Listing 10.1 as this packet will be broadcasted anyway for synchronization purposes during the stage $P_2$.

**The naïve method.** In Eq. (10.1) we calculated an upper bound for the grid constant $L$ depending on the minimal distance $d_{\text{min}}$ of the client from the anchor plane to guarantee that any circular US coverage zone $Z$ with minimal radius $r_{\text{min}}$ always includes at least four anchors for the distance measurement. As visualized in Figure 12.3 it also becomes apparent, that with increasing distance $d \in [d_{\text{min}}, d_{\text{max}}]$ a minimum square $Q$ around $Z$ always covers $n \in \mathbb{N}$ nodes aligned in $c \times c$ grid rows and columns:

$$4 \leq n \leq \left( \left\lfloor \frac{2 \cdot r_{\text{max}}}{L} \right\rfloor + 1 \right)^2 =: n_{\text{max}} \qquad 2 \leq c \leq \left( \left\lfloor \frac{2 \cdot r_{\text{max}}}{L} \right\rfloor + 1 \right) = c_{\text{max}} =: \Gamma \tag{12.2}$$

We will call $\Gamma := c_{\text{max}}$ the *standard grid module*, and initially reserve $n_{\text{max}}$ transmission slots for returning the absolute maximum number of $n_{\text{max}} = \Gamma^2$ of DVs to be expected.

Most important however, with Eq. (10.1) both $\Gamma \geq 3$ and $\Gamma \cdot L > 2 \cdot r_{\text{max}}$ always hold, and even the largest $Z$ possible, i.e. $Z_{\text{max}}$ with radius $r_{\text{max}}$, will always fit into a square of $n_{\text{max}} = \Gamma \times \Gamma$ anchors – independent from its overlay position. See the example in Figure 12.3a for $\Gamma = 6$ and $n_{\text{max}} = 6^2 = 36$.

The next step is to assign the reserved return slots to the involved anchors $A_m$ depending on their (geometric) grid position[17]. Considering the fact that the $z$-coordinate is almost constant for all anchors in the anchor plane, an arbitrary anchor $a_i \in A_m$ at absolute world coordinates $P_{a_i}(x|y|z)$ resides in column $C_x$ and row $C_y$ within the grid (→ Figure 12.4):

$$C_x := \left\lfloor \frac{x + \frac{L}{2}}{L} \right\rfloor \qquad C_y := \left\lfloor \frac{y + \frac{L}{2}}{L} \right\rfloor \tag{12.3}$$

From an algorithmic point of view, and as an important advantage for the infrastructure deployment, each anchor may thus safely be placed with an imprecision of about $\pm\frac{L}{2}$ in $x$ and $y$ direction around its intended exact grid point. Once computed, the cell index $(C_x, C_y)$ stays unique and constant for each anchor, and we can compute the associated hash values $H_x$ and $H_y$ by using the standard grid module $\Gamma$:

$$H_x := C_x \bmod \Gamma \qquad H_y := C_y \bmod \Gamma \tag{12.4}$$

The last step is to assign an exclusive return slot $s(a_i) \in [0; n_{\text{max}} - 1]$ to the anchor $a_i$:

$$s(a_i) = \mathscr{H}(I_i^A) := H_y \cdot \Gamma + H_x \tag{12.5}$$

According to Figure 10.3[p227], the resulting maximum amount of time $t_{n_{\text{max}}}$ which must finally be reserved for aggregating all DVs at the client bounds stage $P_3$ and is known a priori (through $d_{\text{max}}$ and $r_{\text{max}}$):

$$d_3 \leq t_{n_{\text{max}}} := n_{\text{max}} \cdot \Delta_{\text{SLOT}}. \tag{12.6}$$

While the naïve Eq. (12.5) does obviously not depend on any information $I_m^C$ from the client it *must* indeed be applied as long as no further information is available for optimization – e.g. dur-

[17] Additional optimization factors like the adjustable QoS level will be addressed later.

ing the first few localization processes. Nevertheless, the discretization of the anchor positions with respect to the anchor grid already resulted in sufficient anchor specific information $I_i^A$ to finally guarantee that any two different anchors $a_i \neq a_j$ with $s(a_i) = s(a_j)$ can never be located within the same $Z$ independent from the coverage zone's overlay position on the anchor grid. Thus, wherever a mobile client localizes itself, no two or more anchors will use the same return slot for their DVs (→ HS3a[p257]).

**Remaining deficiencies.** Though being provably collision-free so far, it is not always useful to reserve $n_{\max}$ slots (→ HS1a). As long as the mobile node is known (or limited) to move only in two dimensions *and* in parallel to the anchor plane (e.g. on the floor with $d_{\max} = d_{\min}$ resulting in $r_{\max} = r_{\min}$), both the grid constant $L$ and $n = n_{\max}$ would always be chosen perfectly through Eq. (10.1) and Eq. (12.2). But as soon as $d_{\max} \gtrsim d_{\min}$ (3D movement) we have to distinguish two cases:

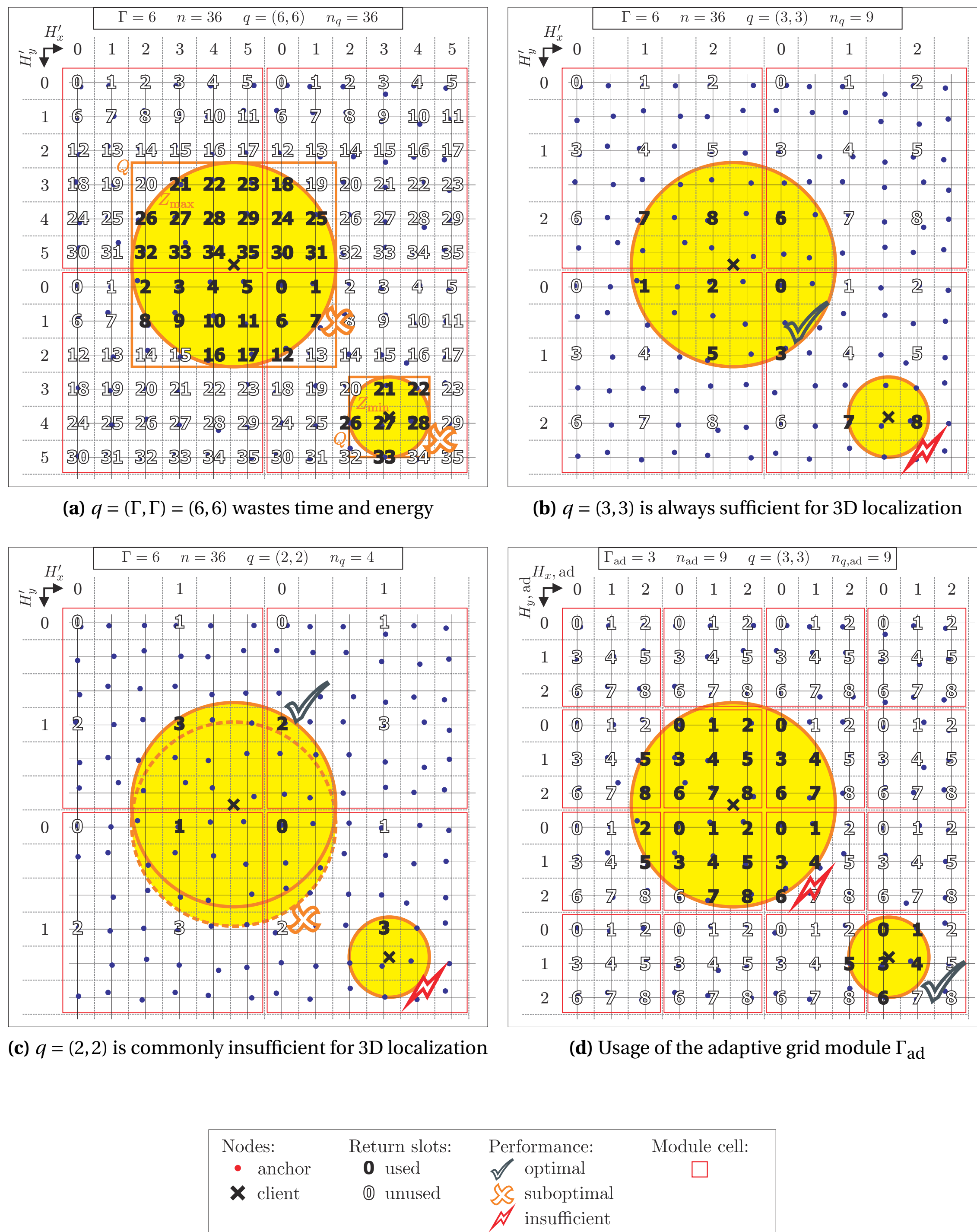
1. A client far away from the anchor plane may indeed receive almost $n_{\max}$ DVs, but in many cases the contained information significantly exceeds the amount which is required even for fault-tolerant position estimation. The large $Z$ in Figure 12.3a shows a scenario where 31 DVs would be collected within $d_3 = 36 \cdot \Delta_{\text{SLOT}}$ while 4 DVs might already be sufficient for 3D localization. A 900% surplus of information, and a waste of time and energy.
2. A client close to the anchor plane will still receive a sufficient number of DVs since $L$ was chosen accordingly, but only $n_{\min} = \left( \left\lfloor \frac{2 \cdot r_{\min}}{L} \right\rfloor + 1 \right)^2 \ll n_{\max}$ return slots may finally be used. The small $Z$ in Figure 12.3a shows a scenario where a sufficient number of only 6 DVs would be collected; yet it would still take $d_3 = 36 \cdot \Delta_{\text{SLOT}}$. A 600% waste of time.

This is where dynamic *quality of service* (QoS) requests may help to achieve a trade-off between time and energy consumption, and to limit the number of returned DVs for case 1. Case 2 can under certain circumstances be improved by the dynamic calculation of the so called *adaptive grid module* (AGM) $\Gamma_{\text{ad}}$. Remind, that any runtime optimization must always deal with the given static anchor grid since $L$ has already been fixed during the deployment stage and cannot be changed anymore.

**Improvement 1: Quality of service (QoS) selection.** The support for a discrete but dynamically selectable QoS level $q$ allows a client $m \in M$ to individually limit the subset $A_m^R \subseteq A_m$ of replying anchors within its US coverage zone $Z_m$. As part of $I_m^C$, $q := (q_x, q_y)$ is a tuple of natural numbers $q_x, q_y \in [1 \dots \Gamma]$ which is broadcasted along with the CAV and defines the number of anchor grid rows and columns to be involved[18]. Therefore $q$ also defines the maximal number $n_q$ of DVs to be expected at the client and the exact number $n_q$ of return slots to be reserved as

$$1 \leq n_q := q_x \cdot q_y \leq n_{\max}. \tag{12.7}$$

[18] Of course $q$ could easily be extended by $q_z$ to support three dimensions, e.g. grid layers, if required.




**(a)** $q = (\Gamma, \Gamma) = (6, 6)$ wastes time and energy

**(b)** $q = (3, 3)$ is always sufficient for 3D localization

**(c)** $q = (2, 2)$ is commonly insufficient for 3D localization

**(d)** Usage of the adaptive grid module $\Gamma_{\text{ad}}$

**Figure 12.3.:** Return slot assignment within the smallest / largest supported coverage zone at various QoS and AGM levels

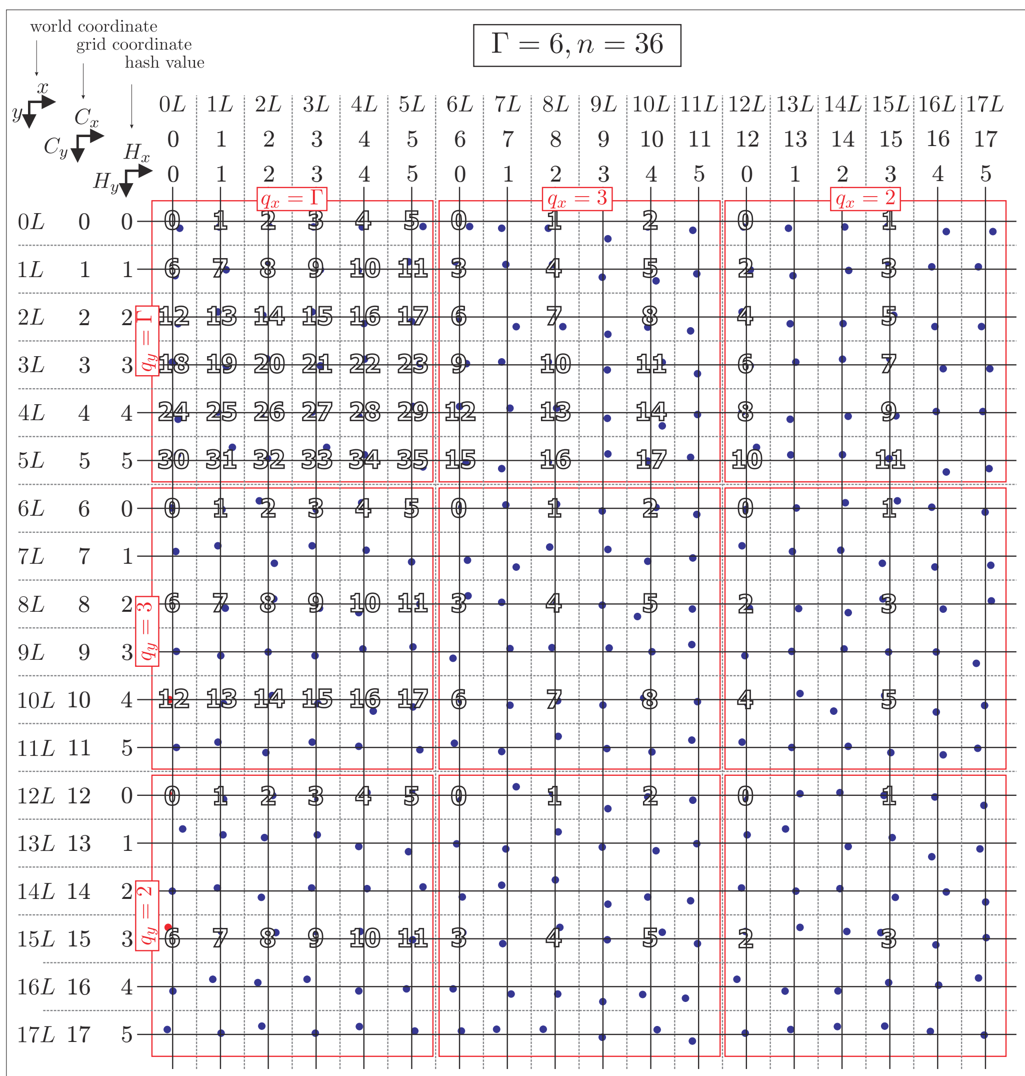



**Figure 12.4.:** Slot number calculation based on of various QoS levels $q = (q_x, q_y)$

The required time to transmit the DVs computes analoguous to Eq. (12.6) as

$$t_{n_q} := n_q \cdot \Delta_{\text{SLOT}} \leq t_{n_{\max}}. \tag{12.8}$$

While Figure 12.4 gives an overview on various QoS settings for $\Gamma = 6$, both Figures 12.3b and 12.3c show concrete application examples for the QoS level[19].

In order to implicitly select "appropriate" anchors, and to limit the number of reserved return slots to $n_q$ (→ HS1a), we have to rearrange the transmissions more tightly over time by scaling the hash values from Eq. (12.4) with $\frac{q}{\Gamma}$ first:

$$H'_x = \left\lfloor H_x \cdot \frac{q_x}{\Gamma} \right\rfloor \qquad H'_y = \left\lfloor H_y \cdot \frac{q_y}{\Gamma} \right\rfloor \tag{12.9}$$

Next, it is required to avoid radio collisions by properly selecting exactly those nodes which may return their DVs. To retain its autonomy (→ HS2b) each node determines this privilege independently from the others, and will only return a DV if

$$w := \bigwedge_{v \in \{x,y\}} \Big( \big( H_v \cdot q_v \big) \bmod \Gamma < q_v \Big) \tag{12.10}$$

is true. This results in a uniform sub-grid, and the corresponding slot $s(a_i)$ for an anchor $a_i \in A_m$ will be computed analogous to Eq. (12.5) as

$$s(a_i) = \mathscr{H}(I_i^A, I_m^C) := \begin{cases} H'_y \cdot q_x + H'_x & \text{if } w = \text{true} \\ \text{none} & \text{otherwise} \end{cases} \qquad \in [0; n_q - 1] \cup \{\epsilon\}, \tag{12.11}$$

and generates the subset of replying anchors $A_m^R \subseteq A_m$.

Yet, there remains the question for the *optimal* QoS level $q_{\text{opt}}$ if the distance $d \in [d_l, d_u] \subseteq [d_{\min}, d_{\max}]$ from the mobile object to the anchor plane (or at least the lower bound $d_l$) is known (→ Figure 12.5). This might be feasible for clients that move only in parallel to the anchor plane or for those which can estimate $d_l$ from prior localizations. Assuming once more a circular $Z$ we'll give a solution for $q_{\text{opt},x} = q_{\text{opt},y}$: With

$$\begin{aligned} r_{d_l} &\overset{\text{Fig. (10.5)}}{:=} d_l \cdot \tan(\varphi) && (\geq r_{\min}) \\ L_{\text{opt}} &\overset{\text{Eq. (10.1)}}{:=} \frac{1.2 \cdot r_{d_l}}{\sqrt{2}} && (\geq L, \text{ fixed!}) \\ \Gamma_{\text{opt}} &\overset{\text{Eq. (12.2)}}{:=} \left\lfloor \frac{2 \cdot r_{d_l}}{L_{\text{opt}}} \right\rfloor + 1 = 3 && (\leq \Gamma) \end{aligned}$$

[19] Remind that as visualized in Figure 10.5a[p231], though 1 is the minimum QoS level per dimension, selecting $q_x, q_y < 3$ might result in less than 4 DVs for the position estimation depending on the overlay position of $Z$.

we can easily calculate

$$q_{\text{opt},x} = q_{\text{opt},y} := \left\lceil \frac{L \cdot \Gamma}{L_{\text{opt}}} \right\rceil = \left\lceil \frac{r_{\min}}{r_{d_l}} \cdot \Gamma \right\rceil \quad (\leq \Gamma). \tag{12.12}$$

In case $r_{d_l} = r_{\min}$ this selects the maximum QoS level $q_{\text{opt}} = (\Gamma, \Gamma)$ which is really required then since we must not exclude any row or column. The small US coverage zone in Figure 12.3a gives an example. In case $r_{d_l} = r_{\max}$

$$\begin{aligned} q_{\text{opt},x} = q_{\text{opt},y} &\overset{\text{Eq. (12.12)}}{=} \left\lceil \frac{r_{\min}}{r_{\max}} \cdot \Gamma \right\rceil \overset{\text{Eq. (12.2)}}{=} \left\lceil \frac{r_{\min}}{r_{\max}} \cdot \left( \left\lfloor \frac{2 \cdot r_{\max}}{L} \right\rfloor + 1 \right) \right\rceil \\ &\geq \left\lceil \frac{r_{\min}}{r_{\max}} \cdot \frac{2 \cdot r_{\max}}{L} \right\rceil \overset{\text{Eq. (10.1)}}{=} \left\lceil \frac{2 \cdot \sqrt{2}}{1.2} \right\rceil = 3 \end{aligned}$$

as expected. Therefore it is proven that $q_{\text{opt},x/y} \in [3; \Gamma]$, and that at least 9 slots will always be reserved to hopefully collect at least 4 DVs at the client. The large US coverage zone in Figure 12.3b gives an example.

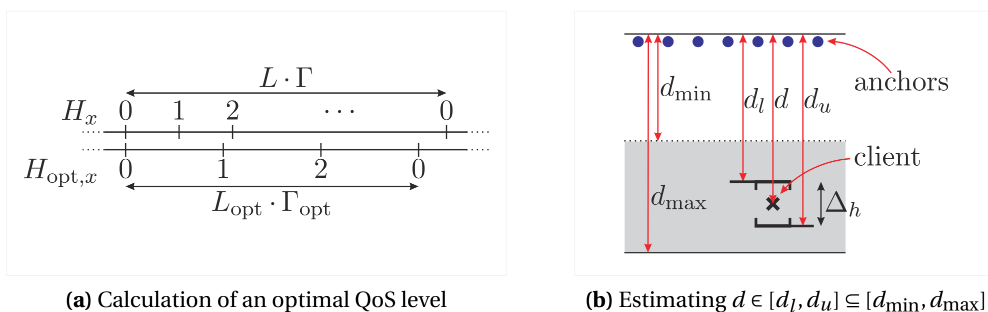
**Improvement 2: The adaptive grid module (AGM).** As both Figures 12.3b and 12.3c show, adapting the QoS level is no adequate option for small US coverage zones, since leaving out whole rows or columns is likely to result in (too) few returned DVs – at least in comparison to the number of reserved slots. To still obtain sufficient DVs and information in a short time despite of a small $Z$ and a large grid module $\Gamma$ we initially keep all anchors $a_i \in A_m$ involved, but temporarily adjust the grid module to the expected size of the US coverage zone. Therefore we also require the client to predict its distance $d \in [d_l, d_u] \subseteq [d_{\min}, d_{\max}]$ from the anchor plane (or at least the upper bound $d_u$) as depicted in Figure 12.5b.

Having broadcasted $d_u$ along with the CAV the client $m$ enables each anchor $a_i \in A_m$ to autonomously compute the corresponding adaptive grid module $\Gamma_{\text{ad}} \leq \Gamma$ for the current localization procedure. Obtaining $\Gamma_{\text{ad}}$, $s(a_i)_{\text{ad}}$, and $t_{\text{ad}}$ works analogous as before:

$$\begin{aligned} r_{d_u} &\overset{\text{Fig. (10.5)}}{:=} d_u \cdot \tan(\varphi) && (\leq r_{\max}) \\ n_{d_u} &\overset{\text{Eq. (12.2)}}{:=} \Gamma_{\text{ad}}^2 = \left( \left\lfloor \frac{2 \cdot r_{d_u}}{L} \right\rfloor + 1 \right)^2 && (\leq n_{\max}) \end{aligned} \tag{12.13}$$

$$\begin{aligned} \forall_{v \in \{x,y\}} : H_{v,\text{ad}} &\overset{\text{Eq. (12.4)}}{:=} C_v \bmod \Gamma_{\text{ad}} && (\leq H_v) \\ s(a_i)_{\text{ad}} &\overset{\text{Eq. (12.5)}}{:=} H_{y,\text{ad}} \cdot \Gamma_{\text{ad}} + H_{x,\text{ad}} && (\in [0; n_{d_u} - 1]) \\ t_{\text{ad}} &\overset{\text{Eq. (12.6)}}{:=} n_{d_u} \cdot \Delta_{\text{SLOT}} && (\leq t_{n_{\max}}) \end{aligned} \tag{12.14}$$

The small $Z$ in Figure 12.3d exemplifies that the adaptive grid module allows both, a reasonable reduction of reserved return slots as well as their extremely tight packing. In contrast, the large




**(a)** Calculation of an optimal QoS level

**(b)** Estimating $d \in [d_l, d_u] \subseteq [d_{min}, d_{max}]$

**Figure 12.5.:** QoS related considerations

$Z$ in Figure 12.3d shows, that underestimating $d_u$ must be avoided carefully (i.e. the client is farer away from the anchor plane than expected) as it would inevitably result in an adaptive grid module $\Gamma_{ad}$ which is too small for the US coverage zone; in consequence valuable information would be lost through multiply assigned slot numbers and colliding DV transmissions.

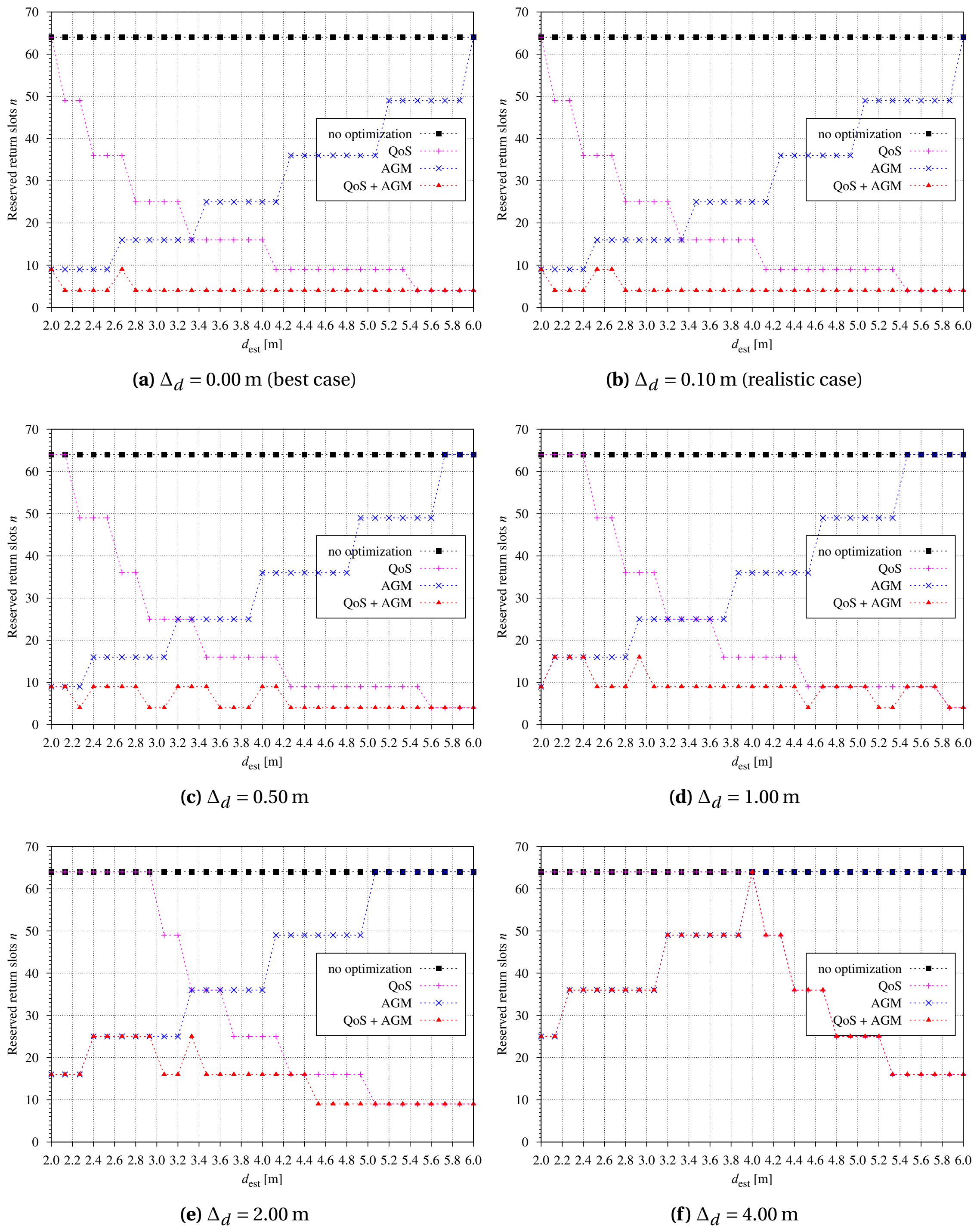
**Improvement 3: Combining QoS and AGM.** As depicted in Figure 12.6 this mixture reveals the most effective method for adapting the data aggregation to a freely moving client's current distance $d$ from the anchor plane: If both an upper and a lower bound for $d \in [d_l, d_u] \subseteq [d_{min}, d_{max}]$ can be estimated, it is obviously possible to compute $\Gamma_{ad}$ from $d_u$ via Eq. (12.13) first, and $q_{opt}$ from $d_l$ and $\Gamma_{ad}$ via Eq. (12.12) thereafter. The idea is to temporarily assume a "modified" $d_{min}$ and $d_{max}$ for obtaining a new imaginary $L$ and $\Gamma$. As these values are fixed since the anchor deployment, the combination of $\Gamma_{ad}$ and $q_{opt}$ finds the most time saving solution for the number of return slots $n_d$ with respect to the fixed anchor grid constant $L$ and the uncertainty in $d$:

$$n_d \overset{\text{Eq. (12.7)}}{:=} q_{opt,x} \cdot q_{opt,y} \qquad t_d \overset{\text{Eq. (12.6)}}{:=} n_d \cdot \Delta_{SLOT} \tag{12.15}$$

Since the resulting efficiency and performance of HashSlot will show to improve with decreasing uncertainty $\Delta_d := (d_u - d_l)$, this is a good motivation for optimizing the 3D precision and accuracy of the applied localization algorithm according to Figure 9.6[p216]. It is also worthwhile to note, that computing $q_{opt}$ from $\Gamma_{ad}$ reaches its maximum value $q_{opt,max}$ at $d \in [d_{min}, d_{min} + \Delta_d]$ for any given uncertainty $\Delta_d$. Unlike $\Gamma$ and $n$ in Eq. (12.2), the value of $q_{opt,max}$ becomes independent from $d_{max}$ and $r_{max}$, respectively:

$$q_{opt,x/y} \overset{\text{Eq. (12.12)}}{=} \left\lceil \frac{d_{min}}{d_l} \cdot \underbrace{\left( \left\lfloor \frac{2 \cdot \sqrt{2}}{1.2} \cdot \frac{d_u}{d_{min}} \right\rfloor + 1 \right)}_{\Gamma_{ad}} \right\rceil \leq q_{opt,max,x/y} = \left\lfloor \frac{2 \cdot \sqrt{2}}{1.2} \cdot \frac{d_{min} + \Delta_d}{d_{min}} \right\rfloor + 1 \tag{12.16}$$

This way, combining QoS plus AGM always delivers an autonomously and precisely computable number of required return slots with $n_{d,max} \overset{\text{Eq. (12.16)}}{\propto} (d_{min} + \Delta_d)^2$. Without optimizations,




**(a)** $\Delta_d = 0.00$ m (best case)

**(b)** $\Delta_d = 0.10$ m (realistic case)

**(c)** $\Delta_d = 0.50$ m

**(d)** $\Delta_d = 1.00$ m

**(e)** $\Delta_d = 2.00$ m

**(f)** $\Delta_d = 4.00$ m

**Figure 12.6.:** The number of reserved DV slots for different optimization strategies and uncertainties for $d \in [d_l, d_u]$ with $\Delta_d = d_u - d_l$ (System setup: $d_{\min} = 2$ m, $d_{\max} = 6$ m $\Rightarrow \Gamma = 8$, $n_{\max} = 64$)

$n_{\max} \overset{\text{Eq. (12.2)}}{\propto} d^2_{\max}$ would always hold instead. Figure 12.6 shows concrete examples for the number of return slots with respect to different optimization strategies and uncertainty ranges $\Delta_d$ for $d$. In particular, the graphs show that combining QoS plus AGM will always reserve less slots than a single optimization does.

**Choosing the optimal configuration for SNoW Bat.** If a mobile client wants to estimate its own position, it first tries to predict the upper and lower bound $d_u, d_l$ for $d$ at the instant $t^C_{\text{Chirp}}$ of the intended chirp emission. While $\Gamma_{\text{ad}}$ is ultimately fixed through $d_u$ and Eq. (12.13) to avoid colliding slots, $q_x, q_y \in [q_{\text{opt}}, \Gamma_{\text{ad}}]$ can still be chosen arbitrarily to reflect the application's current requirements regarding the number of desired DVs $k \leq q_x \cdot q_y$. If WLOG we assume $q_x = q_y$ for the sake of simplicity the number of reserved slots will always be quadratic, and thus $g = \left\lceil \sqrt{k} \right\rceil^2$ DVs should finally be requested by using

$$q_x = q_y := \min\left\{\max\left\{q_{\text{opt}}, \left\lceil \sqrt{k} \right\rceil\right\}, \Gamma_{\text{ad}}\right\} \in \left[q_{\text{opt}}; \Gamma_{\text{ad}}\right]. \tag{12.17}$$

In case $d$ cannot be estimated at all, e.g. due to the lack of a meaningful position estimation history to predict a movement path from, we have to revert to worst case conditions and assume $d_l = d_{\min}, d_u = d_{\max}$, and $k = n_{\max} = \Gamma^2$ to sacrifice speed for reliability. Apart from the first few position estimations of a newly activated client, this strategy can also be pursued if the last position estimation has been considered as unreliable[20], or if significantly less DVs than expected have been received. While both symptoms often cause each other and build up over several iterations, another reason for the latter can be observed if the client moves close to a border of the anchor grid where the ultrasound signal covers only a few anchors.

To definitely avoid colliding slots through different calculations of $\Gamma_{\text{ad}}$ at the anchors soever, the client computes this essential value, includes it in $I^C_m$, and broadcasts it along with $q$ within the CAV from Listing 10.1[p228].

## 12.4.2. Transmission Time Calculation and Slot Boundary Compliance

Now, as each anchor is able to autonomously calculate a collision-free TDMA slot number for returning its DV to the client it currently serves, it still has to determine the corresponding start time of its own slot, and to comply to the boundaries. While the quality of the distance measurement depends on the synchronization between a client $m$ and the anchors $a_i \in A_m$, the data aggregation in stage $P_3$ also requires the anchors to be synchronized. Based on the timestamp $t^A_{\text{CAV}}$, for which we already showed to be sufficiently precise in Section 11.1, we define $t^A_{DA} = t^A_{\text{CAV}} + d_2$ as the start time of both $P_3$ and the first slot. Thus

$$t^A_{\text{DV},a_i} := t^A_{\text{DA}} + s(a_i) \cdot \Delta_{\text{SLOT}}. \tag{12.18}$$

[20] Of course this requires the position estimation algorithm or some postprocessor to support such an "reliability indicator". See pVoted in Chapter 13 for an example.

```
1  Time_t tDV =  tDA + slot * ΔSLOT -
2                tEmisComp;
3  if (sleepUntil(&tDV) == -1) goto failed;    // delay the DV transmission
4  if (SmartNET_SendPacket(&DVHandle) != 0)
5    goto failed;                              // immediate send through SmartNet
6  if (waitEventFor(&evDVSent, ΔSLOT) != 1) {  // error handling:
7    SmartNET_StopRXTX(&DVHandle);             // do not violate the TDMA slot
8    goto failed;
9  }                              precisely captured by SmartNet during DV tx
10 tEmisComp = DVHandle.RXTXTime - tDV;        // compute lateness
```

**Listing 12.1:** Anchor node self-calibration for the precisely timed emission of distance vectors

We have already learned form Chapter 5 that the scheduling of reaction times is not trivial. As sending the DV is a timed reaction on receiving the CAV, we once more apply the self-calibration technique from Section 5.4.1. The implementation relies on *Smart*Net, and an excerpt can be found in Listing 12.1.

### 12.4.3. Summary

From this section we learned that the HashSlot based TDMA slot scheduling algorithm does not only facilitate definitely collision-free (→ HS3a[p257]) and tightly packed (→ HS1b) wireless data aggregation, but also that the overall data throughput can carefully be adjusted (→ HS4b) against the amount of requested information (→ HS1a) – a considerable fact which consequently helps to control the performance of many subsequent sensor data fusion processes, e.g. the position estimation in our case. Nevertheless it relies on an entirely autonomous slot calculation scheme to avoid any coordination cost between the anchors (→ HS2), and provides a deterministic data aggregation duration for the clients (→ HS4a)[21]. In summary, HashSlot commonly scales with the number of requested reply packets $k$ – and sometimes with $n_{\max}$ in case no supplementary information is available – but it is always entirely independent from the network size and the number of anchors $|A|$ (→ HS5a).

## 12.5. A first Real-World Test Bed and Performance Analysis

For a meaningful performance evaluation under real-world conditions we set up a SNoW Bat test bed comprising $|A| = 36$ anchors and a single client, and did let HashSlot compete against the communication protocols from Section 12.3 regarding reliability, speed, and energy consumption.

**Direct measurements.** For certain selected combinations of radio protocol and subsets of replying anchors $A_m^R \subseteq A$ with $|A_m^R| \in \{4, 9, 16, 25, 36\}$ we observed 400 data aggregation stages

[21] A closer look to the equations within this section shows that simple integer arithmetic is sufficient for all computations, and allows an efficient implementation even on typically weak sensor node MCUs (→ Chapter 2).

and accomplished the following measurements at the client[22]:

a) **The average packet loss rate** $\Lambda(|A_m^R|)$ is the standard metric, as it indicates the percentage of DVs which did not arrive at the client $m$ (→ Figure 12.7a).

b) **The average sufficiency rate** $\Omega(|A_m^R|)$ evaluates the protocol reliability, as it indicates the percentage of data aggregation stages where at least $\frac{\pi}{4} \cdot |A_m^R|$ DVs did arrive at client $m$ (→ Figure 12.7b). The reason for using the factor $\frac{\pi}{4}$ will be discussed in Section 12.7. Considering the already discussed fact that a certain minimum of DVs must be made available at the client to obtain a "good" position estimation, this metric also describes the number of localization attempts which must be made to finally succeed once.

c) **The average reply stage duration** $d_3(|A_m^R|)$ evaluates the protocol speed, as it indicates the time it took to either receive at least $\frac{\pi}{4} \cdot |A_m^R|$ DVs at the client $m$ or until the timeout had been reached (→ Figure 12.7c).

d) **The average RX mode duration** $t_{RX}(|A_m^R|)$ evaluates the protocol's energy consumption, as it indicates the time the client $m$ had to keep the radio receive mode active (→ Figure 12.7d).

For these measurements we did intentionally focus on the client, since the energy supply for the anchors within the infrastructure is most commonly less critical compared to the e.g. battery powered mobile nodes. Additionally we were particularly interested in the maximum temporal resolution the client could possibly achieve when tracing its own movement path[23].

Regarding the test bed results from Figures 12.7a – d, the Brute Force method did indeed perform as badly as expected, and impressively verified the packet collision problem caused by the quasi-simultaneous event detection from Section 12.2: ≈ 100% packet loss rate at ≈ 0% sufficiency rate. Conversely, the NodeID based schedule achieved ≈ 0% packet loss rate and ≈ 100% sufficiency rate; yet it is tremendously slow (even for the low $ID_{max} = 69$ we used), and the radio rx time in particular depends on the anchor constellation and ID distribution within the current $Z$.[24]

Considering the Random Slot approach the packet loss rate is still low but almost constant at ≈ 10%; however, the much more interesting average sufficiency rate rapidly decreases from $\Omega(4) \approx 90\%$ to $\Omega(9) \approx 43\%$. At least it achieved an almost linear reply stage duration and rx time – possibly caused by the uniform distribution of the DV emission over the timeout range.

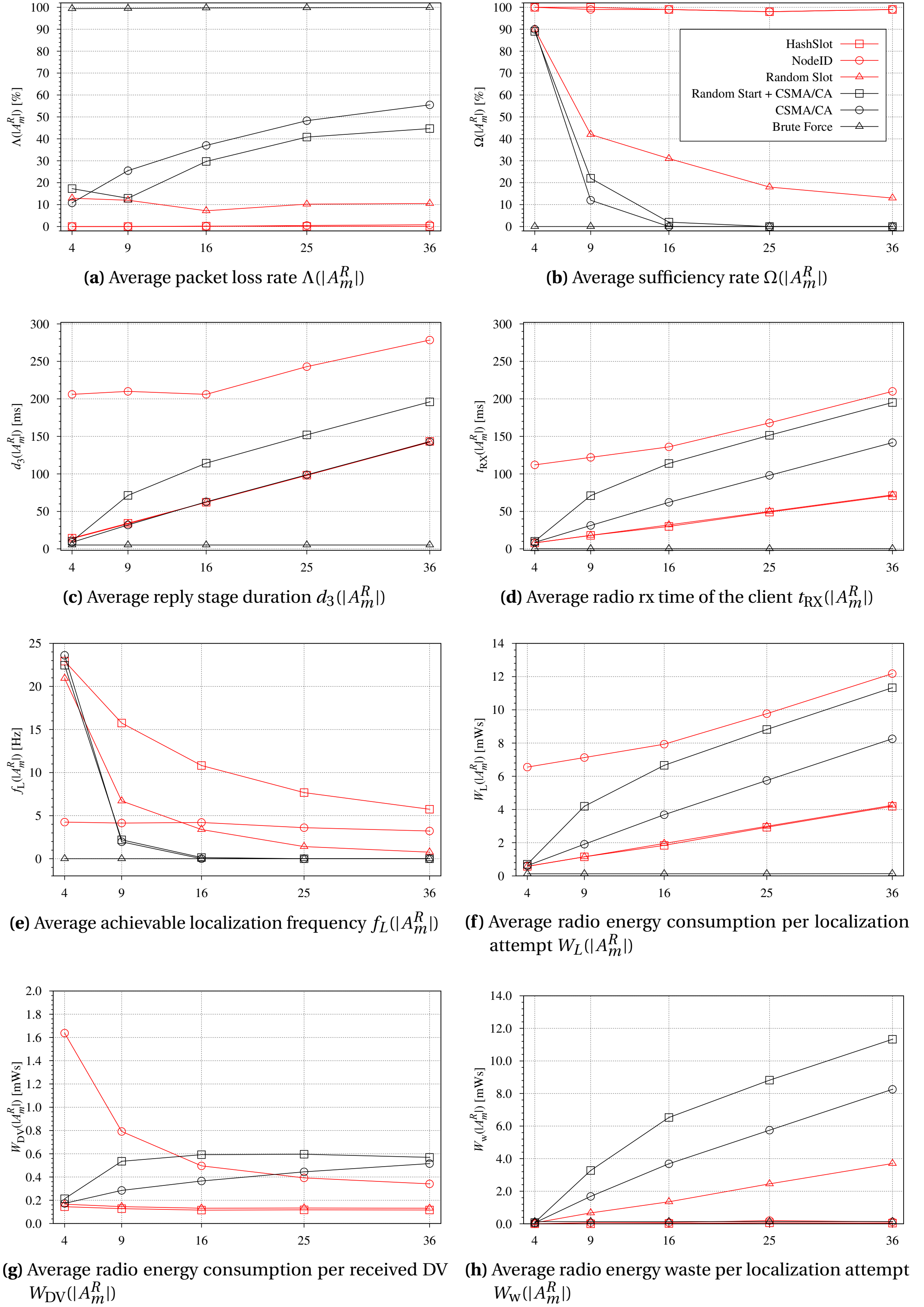
Looking at the non-slotted methods, Random Slot + CSMA/CA exhibits an increasing packet loss rate depending on the number of involved anchors, and consequently achieves a lower average sufficiency rate of $\Omega(9) \approx 22\%$. Also, it takes significantly more time for both $d_3$ and $t_{RX}$.

The pure CSMA/CA protocol performs even worse. Possibly due to the fact that the anchors did their CCA almost simultaneously, and could consequently not observe each others transmissions,

[22] Two further sniffer nodes were configured to also record the radio traffic for validating the correct operation of the client.

[23] The durations $d_3$ and $t_{RX}$ were measured using the *Smart*OS timing functionality.

[24] The larger $ID_{min}$ the longer it takes to receive the first DV. Beyond, the IDs are commonly not tightly packed.




**(a)** Average packet loss rate $\Lambda(|A_m^R|)$

**(b)** Average sufficiency rate $\Omega(|A_m^R|)$

**(c)** Average reply stage duration $d_3(|A_m^R|)$

**(d)** Average radio rx time of the client $t_{\mathrm{RX}}(|A_m^R|)$

**(e)** Average achievable localization frequency $f_L(|A_m^R|)$

**(f)** Average radio energy consumption per localization attempt $W_L(|A_m^R|)$

**(g)** Average radio energy consumption per received DV $W_{\mathrm{DV}}(|A_m^R|)$

**(h)** Average radio energy waste per localization attempt $W_{\mathrm{W}}(|A_m^R|)$

**Figure 12.7.:** Measurement results (a – d) and derived metrics (e – h) from the real-world test bed for selected counts of anchors $|A_m^R|$ in a client's $m$ US coverage zone

the quite long switching delay from rx to tx mode led to numerous packet collisions and a low average sufficiency rate of $\Omega(9) \approx 12\%$ for the relevant case. Yet, both $d_3$ and $t_{\mathrm{RX}}$ were almost linear.

By semantically exploiting the information about the clients predicted spatial position, HashSlot achieved almost perfect collision-freedom and an average sufficiency rate of $\approx 100\%$ independent from the involved anchor subset. Comparable to the NodeID method in this regard, the tight slot packing also reduced $d_3$ and $t_{\mathrm{RX}}$ to the theoretically achievable minimum. Note that though these two results might seem to be comparable to Random Slot in Figures 12.7c and 12.7d, this is only the case for this particular test bed setup where we used the spreading factor $\gamma = 1$. While using $\gamma = 2$ or $\gamma = 3$ resulted in just a slightly higher sufficiency rate then, both values $d_3$ and $t_{\mathrm{RX}}$ did double or triple then. Of course they remained constant for HashSlot.

**Derived metrics.** Relating the just presented measurement results to the SNoW[5] energy consumption characteristics from Table 2.2[p27] allowed us to derive some more interesting values. Though we still focus on the client, we also considered the energy required for transmitting the synchronizing CAV in stage $P_2$, but explicitly omitted the energy required by the MCU or the ultrasonic hardware to strictly focus on the radio protocol. The results are visualized in Figures 12.7e – 12.7h:

e) **The achievable localization frequency** when operating only on a sufficient number of DVs:

$$f_L(|A_m^R|) := \frac{1}{d_1 + d_2 + d_3(|A_m^R|) + d_4} \cdot \Omega(|A_m^R|)$$

f) **The average radio energy consumption per localization attempt** independent from success of failure:

$$W_L(|A_m^R|) := t_{\mathrm{RX}}(|A_m^R|) \cdot P_{\mathrm{RX}} + t_{\mathrm{TX}} \cdot P_{\mathrm{TX}}$$

g) **The average radio energy consumption per received DV**:

$$W_{\mathrm{DV}}(|A_m^R|) := \frac{W_L(|A_m^R|)}{|A_m^R| \cdot \left(1 - \Lambda(|A_m^R|)\right)}$$

h) **The average radio energy waste per localization attempt** i.e. the invested energy for finally canceled position estimations, due to an insufficient number of received DVs:

$$W_w(|A_m^R|) := W_L(|A_m^R|) \cdot \left(1 - \Omega(|A_m^R|)\right)$$

Once more it becomes obvious that the tightly packed and collision-free slots facilitate a comparably higher localization frequency for HashSlot (→ Figure 12.7e)[25]. In fact, our new approach requires virtually the same amount of energy per localization attempt and DV reception than

[25]Note that the trend is not linear since the other factors, i.e. $d_1, d_2, d_4$ were considered to be constant for this evaluation.

Random Slot (→ Figures 12.7f, 12.7g), but it wastes significantly less energy due to almost no canceled localizations (→ Figure 12.7h).

As expected, the most remarkable result is, that HashSlot is independent from the total number of anchors $|A|$, and even achieves almost constant values for the average packet loss rate, sufficiency rate, energy consumption per DV, and energy waste per localization attempt when considering the subset of replying anchors $A_m^R \subseteq A$. Just the achievable localization time and frequency $f_L$ as well as the total energy requirement is not constant but depends on the number of received DVs. Though we can already state that the theoretically expected characteristics of the basic HashSlot approach were indeed verifiable within our real-world test bed, there are still some more problems to be considered in the next section.

## 12.6. Further Improvements for Real-World and Application-Specific Requirements

Until now we have discussed the general requirements and design considerations for event-triggered data aggregation protocols in Section 12.2, and presented the HashSlot approach for the special case of ultrasound based distance measurement in Section 12.4. A test bed for evaluating the pure protocol performance regarding our primary goal to reduce the communication cost was presented in Section 12.5, and already showed the benefits of semantically exploiting some dynamically emerging runtime information for optimizing the transmission schedule of the involved anchors. Yet, receiving DVs efficiently and reliably is not our only objective; regarding the fusion cost and some maintenance issues for indoor localization systems in particular HashSlot has more to offer:

**Network maintenance: Locating defective anchors through inverted slot calculation.** For the test bed in Section 12.5 we assumed that each anchor operates perfectly reliable; in fact, a hand-picked set of sensor nodes from our pool guaranteed failure-free DV transmissions. This certainty however, it not self-evident under real-world conditions. In fact, TDMA slots will be observed vacant if anchors fail to send for whatever reason. Even if sporadic malfunctions might be tolerable for many applications (e.g. if the number of requested packets can be adjusted to largely compensate for such problems), permanent local failures might rapidly lead to the unreliability of the overall system installation, and will commonly demand for repair and maintenance as indicated in Section 12.2.3. While the specific impact of unused slots on the protocol performance will be discussed and largely attenuated by the HashSlot$^+$ extension in Section 12.7, we'll next present a method for determining the anchor grid coordinate related to an arbitrary (unused) slot number (→ HS3b[p257]). While at least one DV must have been received successfully, either before or after the vacant slot, no more information than already available will be required to let the client (or any other observer) implement this feature:

Let $D_i := (x_i, y_i, s(a_i))$ be the relevant information within a successfully received DV (→ Listing 10.2[p228]) from an anchor $a_i \in A_m^R$ at absolute world position $(x_i, y_i)$, and let $s(a_j) \neq s(a_i)$ be the

unused slot number related to another anchor $a_j \in A_m$ at world coordinate $(x_j, y_j)$.[26] The missing DV is represented by $D_j := (x_j, y_j, s(a_j))$, and we are interested in calculating $(x_j, y_j)$ or at least the grid coordinate $(C_{x_j}, C_{y_j})$ from $s(a_j)$ (observed), $D_i$, and $q$ (given).

For any slot number $u < q_x \cdot q_y$ Eq. (12.5) allows to compute

$$H_x(u) := u \bmod q_x \qquad \text{and} \qquad H_y(u) := \left\lfloor \frac{u}{q_y} \right\rfloor. \tag{12.19}$$

Thus, the relative distance in grid cells between $a_i$ and $a_j$ is

$$\Delta H_x(a_i, a_j) := H_x(s(a_j)) - H_x(s(a_i)) \qquad \text{and} \qquad \Delta H_y(a_i, a_j) := H_y(s(a_j)) - H_y(s(a_i)). \tag{12.20}$$

According to Eq. (12.3) we also obtain an initial tuple for the cell coordinate $(\tilde{C}_{x_j}, \tilde{C}_{y_j})$ as

$$\tilde{C}_{x_j} := C_{x_i} + \left\lfloor \Delta H_x(a_i, a_j) \cdot \frac{\Gamma}{q_x} \right\rfloor \qquad \text{and} \qquad \tilde{C}_{y_j} := C_{y_i} + \left\lfloor \Delta H_y(a_i, a_j) \cdot \frac{\Gamma}{q_y} \right\rfloor. \tag{12.21}$$

A problem with this result is that $(\tilde{C}_{x_j}, \tilde{C}_{y_j})$ will always relate to an anchor $a_j$ within the same *module cell* than $a_i$. Let's take a look at the US coverage zone in Figure 12.8 for an example: Imagine a DV $D_i = (4\,\text{m}, 6\,\text{m}, 2)$ has been received from $a_i$ at $(C_{x_i}, C_{y_i}) = (4, 6)$ during slot $s(a_i) = 2$. Slot 3 has been assigned to $a_j$, but was observed vacant at the client. With $q = (3, 3)$ we obtain

$$\Delta H_x(a_i, a_j) = (3 \bmod 3) - (2 \bmod 3) = -2 \qquad \text{and} \qquad \Delta H_y(a_i, a_j) = \left\lfloor \frac{3}{3} \right\rfloor - \left\lfloor \frac{2}{3} \right\rfloor = 1. \tag{12.22}$$

With $\Gamma = 6$ we also receive

$$\tilde{C}_{x_j} = 4 + \left\lfloor -2 \cdot \frac{6}{3} \right\rfloor = 0 \qquad \text{and} \qquad \tilde{C}_{y_j} = 6 + \left\lfloor 1 \cdot \frac{6}{3} \right\rfloor = 8. \tag{12.23}$$

While the initial result $\tilde{C}_j = (0, 8)$ indicates an anchor $a_h$ with $s(a_h) = 3$ within the same module cell than $a_i$, the true anchor $a_j$ with $s(a_j) = 3$ resides within an adjacent module cell[27] with

$$C_{x_j} \in \Theta_x := \left\{ \tilde{C}_{x_j} - \Gamma, \tilde{C}_{x_j}, \tilde{C}_{x_j} + \Gamma \right\} \qquad \text{and} \qquad C_{y_j} \in \Theta_y := \left\{ \tilde{C}_{y_j} - \Gamma, \tilde{C}_{y_j}, \tilde{C}_{y_j} + \Gamma \right\}. \tag{12.24}$$

A disambiguation is easily possible by considering the set $R$ of some more received DVs:

$$\exists_{k \in R} : C_{x_k} \in \Theta_x \Rightarrow C_{x_j} := C_{x_k} \qquad \text{and} \qquad \exists_{k \in R} : C_{y_k} \in \Theta_y \Rightarrow C_{y_j} := C_{y_k} \tag{12.25}$$

For our example from Figure 12.8 we obtain $\Theta_x = \{-6, 0, 6\}$ and $\Theta_y = \{2, 8, 14\}$, and since we also received the anchors for slot 5 in grid row 8 and for slot 6 in grid column 6 we know that

[26] Note that even if $s(a_i)$ is not explicitly included in the DV payload it can easily be calculated according to Eq. (12.18) and Figure 10.3[p227] from the rx timestamp at the client: $s(a_i) := \left\lfloor \frac{t^C_{\text{DV},a_i} - t^C_{\text{DA}}}{\Delta_{\text{SLOT}}} \right\rfloor$.

[27] The proof follows directly from the selection of $\Gamma$ in Eq. (12.2).

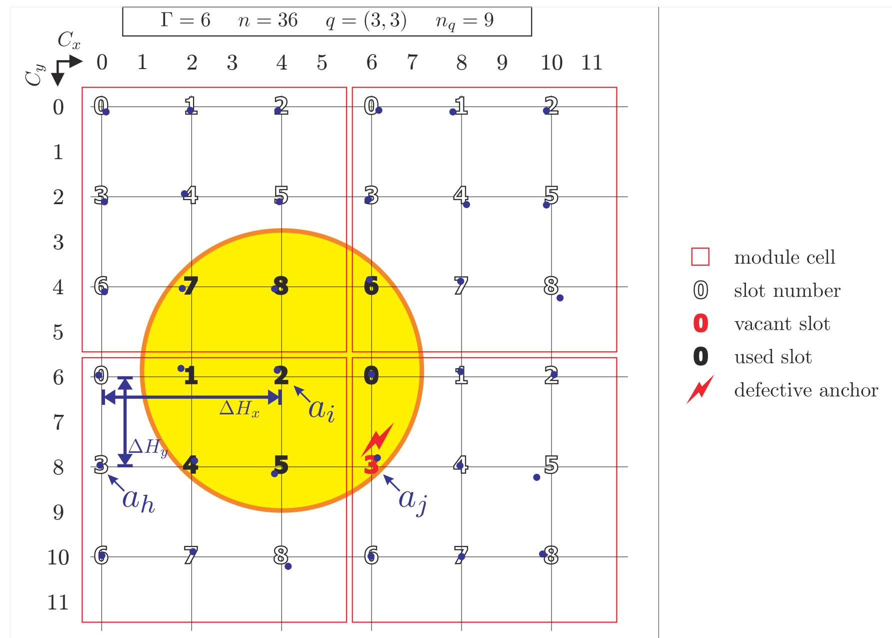


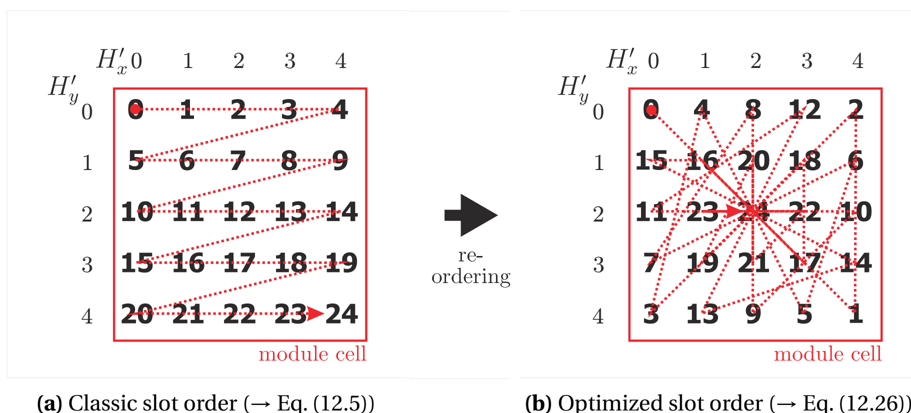

**Figure 12.8.:** Locating defective anchors through inverted slot calculation

the missing DV relates to $a_j$ at $C_j = (6,8)$. Finally the (defective?) node $a_j$ can be found at the absolute world position $(6L, 8L)$ with $L$ being the initially defined grid constant.

**Accelerating the data fusion: Improving the DV utility through slot reordering.** As already suggested in Section 12.2.2, it can be a valuable benefit for many progressive data fusion processes to receive the most relevant or valuable data first. Though it might appear like a detail for HashSlot itself, a slight modification to the hash function reorders the DV slots in a way to obtain a significantly higher utility for the data fusion process in the context of the considered position estimation algorithm from Chapter 13 (→ HS4c[p257]).

Without anticipating the details of pVoted or any other approach, there is the inevitable question about how to determine the "quality" of each DV a priori from a "zero knowledge decision" and without any coordinating anchor interaction. Apart, the specific DV utility might not only depend on the directly contained information, but also on various relations between the DVs – these however might not become visible unless the DVs have been gathered and processed.

Fortunately we can once more profit from the anchor grid structure we proposed for SNoW Bat, and extract valuable information for the special case of localization and position estimation algorithms. Since most of them are based on solving linear equations or maximum likelihood estimations, they require the anchors to be located at non-collinear positions (→ Section 13.3 for details). While this is obviously not always attainable within a quasi-rectangular anchor grid, Eq. (12.5) even generates a so called Z-order as depicted in Figure 12.9a where anchors at $q_x$ linear dependent positions transmit first. In other words, the larger $q_x$ the longer it will take until the first numerically reliable position estimation can start.




**(a)** Classic slot order (→ Eq. (12.5)) **(b)** Optimized slot order (→ Eq. (12.26))

**Figure 12.9.:** HashSlot Slot number reordering for improved reception of non-collinear DVs ($q_x = q_y = 5$)

Obtaining a more appropriate schedule through consistent slot reordering can be accomplished in various ways. With the client's initially broadcasted information $I_m^C$ containing the grid module and QoS level to be used as globally available protocol configuration, we'll describe the distributed computation of just one example permutation as depicted in Figure 12.9b: Beginning at the border of a module cell and proceeding to its center, the anchors at the 4 corners will transmit first, followed by their clockwise neighbors. From Eq. (12.9) each anchor knows its individual $H'_x, H'_y$ values, and, if we assume WLOG equal QoS values $q_{xy} := q_x = q_y$ for simplicity reasons, the new slot number $s^*(a_i)$ can still be computed consistently and autonomously by each anchor $a_i \in A_m^R$:

$$
\begin{aligned}
V &:= \min\left\{H'_x, H'_y, q_{xy} - H'_x - 1, q_{xy} - H'_y - 1\right\} \\
U &:= q_{xy} - 1 - V; \\
B &:= \sum_{j=1}^{V} 4\cdot(q_{xy} + 1 - 2j)
\end{aligned}
$$

$$
s^*(a_i) := \begin{cases} 4\cdot(H'_x - V) + 0 + B & \text{if } (H'_y = V \wedge H'_x \neq U) \\ 4\cdot(U - H'_x) + 1 + B & \text{if } (H'_y = U \wedge H'_x \neq V) \\ 4\cdot(H'_y - V) + 2 + B & \text{if } (H'_x = U \wedge H'_x \neq U) \\ 4\cdot(U - H'_x) + 3 + B & \text{if } (H'_x = V \wedge H'_x \neq V) \\ B & \text{if } (V = U) \end{cases} \tag{12.26}
$$

**Traffic Shaping: Adjusting the transmission rate to a varying data fusion complexity.** Another problem with progressive (sensor) data fusion algorithms is that they often require more time the more information has already been received at the processing node. Using TDMA slots of fixed order can easily reflect this demand by dynamically expanding the basic slot duration $\Delta_{\text{SLOT}}$ according to some simple scaling factor or a complex calculation (→ HS1b);

depending on the slot number the additional information required therefore can be both static or dynamic, e.g. integrated into the information request. For tiny embedded systems in particular, where dynamic memory allocation is possible (→ Chapter 7) but often avoided for performance and real-time reasons, this technique even helps to avoid the memory intensive buffering of information[28], but allows to process the data packets completely as they arrive.

As an example, a linear scaling factor $\alpha$ for extending the basic slot length $\Delta_{\text{SLOT}}$ might be included into the CAV to instruct each anchor $a_i \in A_m^R$ to override Eq. (12.18), and to compute its individual DV emission time as

$$t^A_{\text{DV},a_i} := \begin{cases} t^A_{\text{DA}} & \text{if } s(a_i) = 0 \\ t^A_{\text{DV},a_{i-1}} + \alpha^{s(a_i)-1} \cdot \Delta_{\text{SLOT}} & \text{if } s(a_i) \neq 0 \end{cases}. \tag{12.27}$$

**Scalability: Supporting concurrently operating clients through individual frequencies.** This suggestion is less an improvement for scheduling the anchors which currently serve a common client, but an option to perform several data aggregation processes simultaneously. As a DV is always meant for a single client, and a client $m$ will not intend to receive DVs from other anchors $a_i \notin A_m$ than covered by its own US coverage zone $Z_m$, the data aggregation processes are entirely independent from each other[29]. Thus, we are not constrained to temporal communication interleaving (as we were for transmitting associated DVs), but can benefit from the dynamic channel switching capability of most modern radio transceivers[30] as already depicted in Figure 10.3[p227]: While the CAV must be transmitted using a common radio frequency – the so called *control channel* – to reach a broad number of anchors, the DVs will simply be returned using a client specific frequency – the so called *return channel* – as specified in the CAV (→ Listing 10.1). Using this FDMA-like (frequency division multiple access) approach implicitly avoids mutual disturbance.

The transmission of the CAV however is not covered by the HashSlot protocol, and must be managed by other approaches. Client specific requirements (e.g. on the localization frequency and fairness) are also likely to be relevant here and demand for similarly reliable techniques. Options are e.g. static round robin, or dynamic desynchronization [207, 208].

## 12.7. The HashSlot$^+$ Extension

Up to now the basic HashSlot approach was designed to let autonomously operating anchor nodes compute their individual, collision-free, and tightly packed TDMA slots without any mutual interaction or explicit coordination with other potentially interfering systems. The thereby avoided coordination overhead reduced both the time and energy requirements to a

[28] It also removes the need for a reasonable dimensioning of the data buffer itself, which is commonly hard and can easily lead to memory waste through internal fragmentation of the valuable RAM memory (→ Section 7.2.1).

[29] Note, that though uncritical for the data aggregation, two or even more overlapping US coverage zones might be a severe problem for the distance measurement process: If the chirps interfere, both the signal shapes and the source matching can be damaged.

[30] The CC1100 applied on the SNoW[5] supports 255 channels (→ Section 10.2.1).

minimum demanded for the pure DV transmission and aggregation[31].

However the resulting performance can only be retained under realistic conditions if each reserved slot will really be used and the reserved time will not pass by, i.e. if (1.) each slot *is* assigned to an anchor, and (2.) if *each* anchor for which a slot has been assigned will indeed participate in the data aggregation process. In the last section we already presented a method to identify anchor positions by their corresponding slot number (e.g. for maintenance reasons); in this Section we'll extend the credulous HashSlot algorithm by an additional observation scheme which accepts some additional energy effort to detect vacant slots as they appear, and dynamically contracts the data aggregation stage by letting subsequently scheduled anchors transmit early to fill up valuable dead time on the radio channel. The so called HashSlot$^+$ *slot compression* will nevertheless keep the general operation from Section 12.4 and the presented improvements from Section 12.6 valid since, once fixed, the anchors' slot order itself remains untouched – in fact, retaining the order is an inevitable precondition to know which anchor might start early without causing a radio collision.

**Background.** The reason for vacant slots can originate from various imponderabilities which can neither be predicted nor considered by the hash function or emission time calculation due to their sporadic nature. While defective anchors are an obvious problem, another issue are geometric discrepancies between the US coverage zone $Z$, i.e. the area covered by the event to be detected, and the shape of the anchor deployment pattern. For our rectangular (or quadratic) anchor grid HashSlot always assigns $q_x \cdot q_y$ slots to anchors spanning a rectangle $Q \supseteq Z$, and the protocol performance increases the better $Z$ matches $Q$ since anchors within the area $Q \backslash Z$ won't return a DV but waste the reserved slot time $\Delta_{\text{SLOT}}$ (→ Figure 12.3 for several examples).

Unfortunately the ultrasound transducers emit a conical beam (→ Figure 11.1[p245]), and generate a circular $Z$ which moves freely over the anchor plane. Thus no anchor pattern will ever be perfect[32], and we have to find another feasible way to compensate for this structural performance loss. By the way, it is hard to quantify the number of anchors to be expected in $Q \backslash Z$: The underlying *lattice point problem* [132, 151, 221] which addresses the question of how many lattice points $N(r)$ are located within a freely placed circular area with radius $r$ is hard to solve for the general case, and even for a known center position of $Z$ there is no closed formula. Though iterative equations can easily be found[33] and an approximation is given through $N(r) = \pi r^2 + O(r^\lambda)$ with $\frac{1}{2} \leq \lambda \leq \frac{46}{73}$ (Huxley [132]), these calculations presume a perfect grid without any irregularities stemming from the anchor deployment.

For reduced complexity we simply found the fraction $\frac{\pi r^2}{4r^2} = \frac{\pi}{4} \approx 78.5\%$ to be sufficiently precise, and consequently expect $\phi \approx 21.5\%$ of vacant slots in average.

[31] Note that in particular entering rx mode is avoided for the anchors whereas activating tx mode is avoided for the clients during the data aggregation stage $P_3$.

[32] Maybe a honeycombed pattern would be slightly more suitable, however the anchor deployment and calibration effort would increase then, and so would the overall number of anchors.

[33] For the specific *Gaussian circle problem*, where the center of $Z$ matches a lattice point, $N(r) = 1 + 4\lfloor r \rfloor + 4\sum_{i=1}^{\lfloor r \rfloor} \left\lfloor \sqrt{r^2 - i^2} \right\rfloor$ according to [127].

```
1  int8_t HSPlus_send(uint8_t ownSlot) {
2    uint8_t received = 0, vacant = 0, nextSlot = 0;
3    Exception_t ex;
4
5    TRY {
6      while (nextSlot <= ownSlot) {
7        uint8_t c      = ownSlot - nextSlot;
8        Time_t  tNext  = t^A_DA + received * Δ_SLOT + vacant * τ;
9        Time_t  tDL    = tNext += c * τ;
10
11       if (sleepUntil(&tNext) == -1) throw EX_DH;       // handle early wakeup
12       if (c == 0) return sendDV();                      // regular send
13       if (carrierSenseUntil(&tRX, &tDL) == 1) {        // foreign DV detected
14         uint8_t v = (tRX - tNext) / τ;                  // number of vacant slots
15         received  += 1;                                  // count the received DV
16         vacant    += v;                                  // count the missed DVs
17         nextSlot  += v + 1;                              // next slot to be expected
18       } else {                                           // deadline reached
19         return sendDV();
20       }
21     }
22   } CATCH (ex) {                                         // dynamic hinting
23     handleFailure();
24     return 0;
25   }
26 }
```

**Listing 12.2:** The DV emission and slot compression algorithm for HashSlot$^+$

**Approach.** Our solution to the problem of vacant slots (→ HS5b) – for whatever reason these might emerge – requires to put the anchors $A^R_m$ in rx mode beginning from the start of the data aggregation stage $P_3$ at $t^A_{\text{DA}}$ and ending with their individual DV transmission. In between they listen to the protocol traffic caused by anchors with lower slot number, and without processing the passively received DVs they count the number of used and unused slots to compute the next true slot boundary. To reliably identify an unused slot we require each participating anchor $a_i \in A^R_m$ to transmit its DV within a time window $\left[t^A_{\text{DV},a_i} : t^A_{\text{DV},a_i} + \tau\right]$ with $\tau < \Delta_{\text{SLOT}}$. Though real-time-critical, this demand is no problem due to the general SNoW Bat software design using *Smart*OS, and the task priority selection in particular as discussed in Section 10.2.4.

The idea is formalized in Listing 12.2: Having calculated its own slot number $s(a_i)$ each anchor $a_i \in A^R_m$ iterates until finished (L6), and updates the number of slots $c$ to let pass by until itself may send (L7). It also determines the adapted time $t_{\text{NEXT}}$ of the next slot (L8), and – since it is not yet known then whether the expected $c$ anchors will send indeed – it also computes its individual deadline $t_{\text{DL}}$ based on $t_{\text{NEXT}}$ (L9) to resume at the latest, i.e. if no foreign transmission has been detected, and to eventually emit its own DV early. Then it suspends itself until the next slot begins (L11), and sends immediately if it is its turn, i.e. if $c = 0$; otherwise it switches to rx mode and performs a carrier sense (L13) until it either receives a foreign DV (from another anchor with lower slot number) or the deadline is reached (if the intermediate anchors did not send). In case of the deadline (L18) the anchor may send early, and we are done. Otherwise (L14 et seq.) we have to count 1 observed DV as well as $v$ vacant slots for this iteration; the number of the next slot to be expected is also updated accordingly before we start over. Figure 12.10

compares HashSlot and HashSlot$^+$ visually, and illustrates the slot compression; Table 12.2 gives the corresponding values for the involved anchors. In particular both nodes scheduled for slot 3 and 6 reached their deadline while waiting for a DV and started their own transmission right thereafter.

As the duration of a vacant slot is defined as $\tau \leq \Delta_{\text{SLOT}}$, the duration $d_3^+$ of the data aggregation stage $P_3$ under HashSlot$^+$ is no longer constant for a given number of requested DVs $n = q_x \cdot q_y$, but depends on the number $m = |A_m^R| \leq n$ of truly replying anchors:

$$d_3^+ := \underbrace{m \cdot \Delta_{\text{SLOT}} + (n-m) \cdot \tau}_{\text{HashSlot}^+} \quad \leq \quad d_3 := \underbrace{n \cdot \Delta_{\text{SLOT}}}_{\text{HashSlot}} \tag{12.28}$$

For the client it is also possible to dynamically update the maximum remaining duration of the data aggregation stage based on its initially selected QoS level. Even the worst case – where no anchor replies at all – will become obvious if no DV has been received after $t_{\text{DA}}^C + n \cdot \tau$.

**Improvements and consequences.** If we imagine an average ratio $\phi$ of non-participating anchors and assume a relationship $\tau := \alpha \cdot \Delta_{\text{SLOT}}$ with $\alpha \in [0;1]$, the temporal improvement of HashSlot$^+$ compared to the conventional HashSlot approach computes as

$$1 - \frac{(1-\phi) \cdot \Delta_{\text{SLOT}} + \phi \cdot (\alpha \cdot \Delta_{\text{SLOT}})}{\Delta_{\text{SLOT}}} = \phi(1-\alpha). \tag{12.29}$$

Regarding our initial consideration about the expected loss of $\phi \approx 21.5\%$ of slots due to the discrepancy between the circular $Z$ and the rectangular grid, the example from Figure 12.10 with an arbitrarily selected $\tau = \frac{1}{5} \cdot \Delta_{\text{SLOT}}$ would reduce $d_3$ by $0.215 \cdot 0.8 = 17.2\%$. A result which is only about 4% worse than the optimum. While $\alpha = 0$ would obviously be optimal, the minimal value of this scaling factor and the resulting $\tau$ depends on the real-time capability of the anchors to start the CCA or DV transmission right at the beginning of each slot. An adequate self-calibration scheme has already been proposed in Section 5.4.1 and applied for HashSlot in Listing 12.1.

A consequence which results from this temporal improvement is the increased energy demand caused by the newly introduced carrier sense. However it is limited to the anchors, and even depends on their individual slot number (the lower the less) as well as on the total number of requested DVs. In the worst case where only the last slot is used, the corresponding anchor has to listen for a period of at most $\tau \cdot (q_x \cdot q_y - 1)$. For the best case where each slot is used, the additional energy demand once more reduces the closer towards 0 the better each anchor manages to perfectly schedule the DV transmission to the beginning of its slot.

Though detecting and compressing vacant slots is the key to the HashSlot$^+$ performance improvements, it also introduces a severe weakness in case of unreliable hardware or software: If an anchor misses a foreign DV and considers the corresponding slot as vacant it becomes temporally misaligned to the other anchors which did receive the DV. Eventually it might cause a packet collision when sending its own DV too early then. Yet, during our real-world tests (using *Smart*Net from Section 8.1 as MAC layer in charge of entering rx and tx mode under real-time demands) this problem occurred too seldom to justify taking explicit counter-measures.

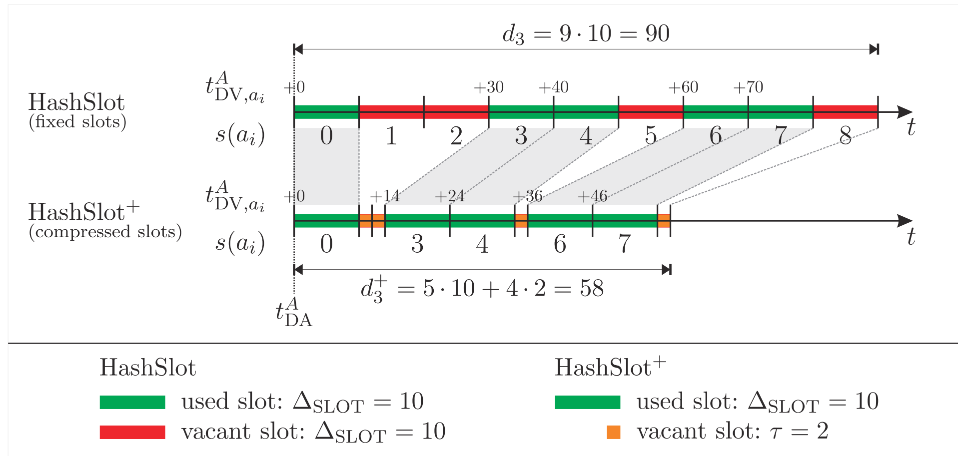



**Figure 12.10.:** Example for slot compression under HashSlot+ (→ Table 12.2, Listing 12.2)

| anchor @ slot | iteration | c | tNext | tDL | tRX | $t^A_{\mathrm{DV}}$ | L | received | vacant | nextSlot |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | **0** | 0 | 0 | | **0** | | 0 | 0 | 0 |
| **3** | 1 | 3 | 0 | 6 | 0 | | 0 | 1 | 0 | 1 |
| | 2 | 2 | 10 | 14 | **DL** | **14** | | | | |
| **4** | 1 | 4 | 0 | 8 | 0 | | 0 | 1 | 0 | 1 |
| | 2 | 3 | 10 | 16 | 14 | | 2 | 2 | 2 | 4 |
| | 3 | **0** | 24 | 24 | | **24** | | | | |
| **6** | 1 | 6 | 0 | 12 | 0 | | 0 | 1 | 0 | 1 |
| | 2 | 5 | 10 | 20 | 14 | | 2 | 2 | 2 | 4 |
| | 3 | 2 | 24 | 28 | 24 | | 0 | 3 | 2 | 5 |
| | 4 | 1 | 34 | 36 | **DL** | **36** | | | | |
| **7** | 1 | 7 | 0 | 14 | 0 | | 0 | 1 | 0 | 1 |
| | 2 | 6 | 10 | 22 | 14 | | 2 | 2 | 2 | 4 |
| | 3 | 3 | 24 | 30 | 24 | | 0 | 3 | 2 | 5 |
| | 4 | 2 | 34 | 38 | 36 | | 1 | 4 | 3 | 7 |
| | 5 | **0** | 46 | 46 | | **46** | | | | |

**Table 12.2.:** Example for slot compression under HashSlot+ (→ Figure 12.10, Listing 12.2)

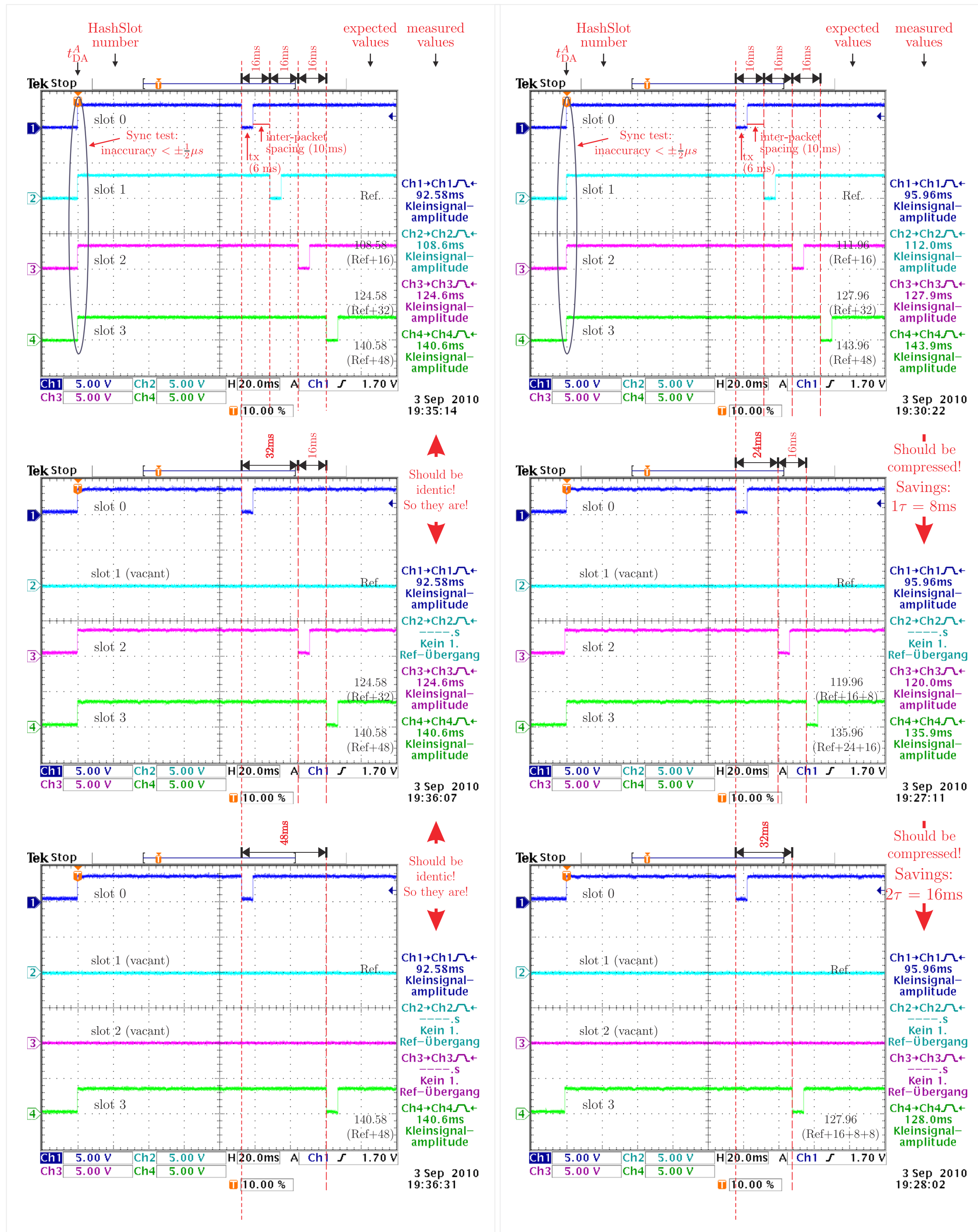



**Figure 12.11.:** Vacant slots under HashSlot: Slots 0 – 3, $\Delta_{\mathrm{SLOT}} = 16\,\mathrm{ms}$

**Figure 12.12.:** Vacant slots under HashSlot+: Slots 0 – 3, $\Delta_{\mathrm{SLOT}} = 16\,\mathrm{ms}$, $\tau = \frac{1}{2}\Delta_{\mathrm{SLOT}} = 8\,\mathrm{ms}$

**Real-world feasibility test.** A real-world comparison between HashSlot and HashSlot$^+$ is depicted in Figures 12.11 and 12.12: The scope snapshots show the protocol operation for 4 selected anchors with assigned slots 0 – 3 and $\Delta_{\text{SLOT}} = 16$ ms. As already demanded in our SNoW Bat reference Figure 10.3[p227], each snapshot indicates the node synchronicity at the beginning $t^A_{\text{DA}}$ of the data aggregation stage through a rising signal edge[34]. Another interesting detail is the precisely met minimal inter-packet spacing of 10 ms following each transmission indicated by a 6 ms low signal level at the beginning of each slot. For the topmost snapshot in particular – where all anchors emit their DV – the high temporal precision complies to our demand for granting the client a sufficient amount of time for progressive packet processing without prophylactic buffering. In fact both HashSlot and HashSlot$^+$ induce the same timing characteristics and demonstrate the deterministic behavior of our scheduling scheme. Considering the second snapshot in each figure depicts the slot compression under HashSlot$^+$: With $\tau = \frac{1}{2} \cdot \Delta_{\text{SLOT}} = 8$ ms we intentionally deactivated the anchor assigned to slot 1 and indeed observed a temporal shift of the 2nd slot by $\Delta_{\text{SLOT}} - \tau = 8$ ms for HashSlot$^+$, which resulted in 24 ms to elapse between the first two DVs compared to $2 \cdot \Delta_{\text{SLOT}} = 32$ ms under HashSlot. Additionally deactivating the anchor for slot 2 amplified this phenomenon to require $\Delta_{\text{SLOT}} + 2 \cdot \tau = 32$ ms under HashSlot$^+$ compared to $3 \cdot \Delta_{\text{SLOT}} = 48$ ms for HashSlot as depicted by the last snapshots in both figures.

Another remarkable detail in the snapshots is the enormous stability in the measured values in general. If we recall the SNoW$^5$ MCU speed of 8 MHz and the resolution of 1 µs for the *Smart*OS system time it is quite astonishing to continuously observe absolutely identical values for the different system setups under HashSlot in Figure 12.11 – even though the temporal resolution of the oscilloscope had to be chosen relatively low (i.e. 10 µs) to capture all relevant signal events. Notably the high complexity of the observed distributed system of autonomously operating and self-synchronizing sensor nodes involves device serial dispersion, hardware imponderabilities, radio communication, event-driven multitasking, dynamic resource sharing issues, timestamp capturing, and various complex algorithms in general. Nevertheless the measured values differ only slightly from the expected values – even for the increased temporal dynamics under HashSlot$^+$ in Figure 12.12.

## 12.8. Summary

In this chapter we identified the challenge of wireless data aggregation as a major performance issue in many WSN/WSAN applications, and emphasized its special relevance for optimizing decentralized localization systems like SNoW Bat. After the introduction of an extensive problem-specific design space in Section 12.2.4 we presented the basic HashSlot data aggregation protocol and its extension HashSlot$^+$ for solving all but two denoted demands.

Based on the distributed calculation of TDMA slot numbers for each anchor it solves, among other issues, the problem of sporadically high radio load upon quasi-simultaneous event detection in event-triggered systems. The key to success in reducing the pure communication cost is to semantically exploit static information (known to the static anchors) as well as dynamic

[34] Though not visible here, the synchronization test revealed an inaccuracy of $\leq \frac{1}{2}$ µs as expected from Chapter 5.

information (provided by the mobile client and observed from the environment) to obtain collision-free and tightly packed transmission slots without explicit communication or even coordination between the involved nodes. In fact, this renders HashSlot highly scalable and entirely independent from the overall number of nodes: While the total and deterministic duration of the data aggregation stage can always be predicted by the client, additional features like dynamic quality of service (QoS) selection refines the approach by reasonable limiting the amount of returned data, and allows an optimal adaption to changing environmental conditions and the application's demands. Regarding the demand for reduced data fusion cost, several extensions allow to adjust e.g. the inter-transmission spacing for simplified progressive information processing, and to reorder the packets regarding their projected utility for the data fusion algorithm.

Considering node failure HashSlot$^{+}$ manages to detect vacant slots as these appear, and dynamically reduces the data aggregation stage duration close to the optimum; inverting the slot selection equations at the client even allows to locate the position of (defective) devices from (vacant) slot numbers (e.g. for simplified network maintenance).

As an advantage for the quite complex SNoW Bat software design, both the implementation as well as usage of the presented approach is quite simple. Based on the already discussed MAC protocol *Smart*Net from Section 8.1 it integrates seamlessly into the concurrently running software components.

As indicated in this chapter's introduction, tailoring the radio protocol to the application meant some effort at first, but proved to result in exceptionally high performance and energy savings as shown in the test bed results (almost optimal speed and minimal packet loss rate). Nevertheless, using the general HashSlot idea for other application scenarios is still possible. Though the central hash function must probably be adapted to the specific demands, even the assignment of e.g. radio channels or spread codes for FDMA or CDMA based protocols instead of slots for TDMA schedules is quite conceivable.

# 13. Progressive Position Estimation: The pVoted Algorithm


**Abstract**

As we have seen in the introduction to this part of the work, localization and position estimation algorithms form a vast topic in wireless sensor networking, and hence it is not surprising that quite a number of such data fusion approaches do already exist.

Most commonly, a certain amount of environmental information (e.g. distances, angles, or neighborhood relations) is gathered first and then processed at once. However, especially in highly dynamic environments where severe real-time requirements, energy constraints, and reliability demands meet, progressive algorithms can significantly increase the overall system performance with respect to various metrics.

Taking advantage of the previous chapters, this chapter introduces the novel pVoted algorithm for accurate and robust 3D position estimation despite of noisy (i.e. imprecise) and sometimes even faulty (i.e. inaccurate) distance measurements. Our decentralized approach allows mobile sensor nodes to localize themselves autonomously from progressively gathered distance information (the “p” in pVoted is for progressive), and it is optimized for the dynamic adjustment of speed, precision, and energy consumption. Regarding our demand for quality awareness and self-evaluation from Section 1.2.1 this allows each node to adjust its preferences at runtime according to its current requirements and available resources. This includes the limitation of the required (RAM) memory as well as the calculation of an individual quality indicator for each estimation.

Beyond the already completed discussion of the HashSlot goals to reduce the communication cost and to simplify the network maintenance (→ Figure 12.2[p263]), the remaining goal to reduce the data fusion cost while still obtaining “good” position estimations will be further addressed in this chapter.

## 13.1. Introduction

If we consider the general WSN process flow from Figure 9.2[p211] in the context of decentralized localization systems, this part of the work so far did already cover the data generation and preprocessing through the Cut DSP algorithm (→ Chapter 11), as well as the data aggregation through HashSlot (→ Chapter 12). The last step to go in order to obtain usable position information from the observed environmental conditions is the data fusion process.

In this regard several preliminary considerations about how to integrate this step into the final software design have already been made in Section 10.2.4, and various preparations regarding the quality and quantity of the distance measurements as well as the order and delay of their transmission through the anchors have been made in Sections 11.2, 12.4, 12.6, and 12.7, respectively. In order to reasonably exploit these deliberately created arrangements we developed the pVoted position estimation algorithm: Optimized for various common system properties it completes the philosophy of hardware/software/network co-design to finally reduce the data fusion cost while retaining a “good” quality and performance for the position estimations.

However, despite of the obvious relevance of location or position estimation algorithms for any localization system, we will intentionally limit ourselves to the integration related part of pVoted, i.e. the mutual influences between the algorithm and the remaining system. Regarding the mathematical aspects only a brief overview will be given, since the required theoretical and mathematical background to understand every detail about the myriads of position estimation approaches would demand for an entirely independent thesis. Also, these details are not directly related to the scope of “time-critical system design considerations for sensor/actuator systems”. Nevertheless, and for the sake of completeness, we will show how pVoted as a representative for similar (progressive) approaches can benefit from the previously presented techniques and paradigms.

Though pVoted offers a multitude of configuration options to adjust the data fusion process to the specific properties of the environment and the measurement error characteristics, we’ll thus highlight just the most central concepts without analyzing all specific (mutual) effects of each parameter thoroughly. Based on a single configuration set – which is optimized for the SNoW Bat system installation from Chapter 10 –, we’ll yet compare some selected algorithms with respect to the speed, precision, and accuracy.

## 13.2. The pVoted Position Estimation Algorithm

In contrast to most ranging based position estimation algorithms which rely on maximum likelihood estimators, force models, or the solving of linear equation systems to reduce the derivation between the measured distances and the ones calculated between the anchor positions and the position estimation, pVoted calculates several potential solutions from received DV triplets and compares these for consistency. The idea is illustrated in Figure 13.1: Each anchor’s DV describes a sphere with the anchor position representing the center (of the ultrasound receiver capsule), and the measured distance (to the client’s sender capsule) representing the radius.

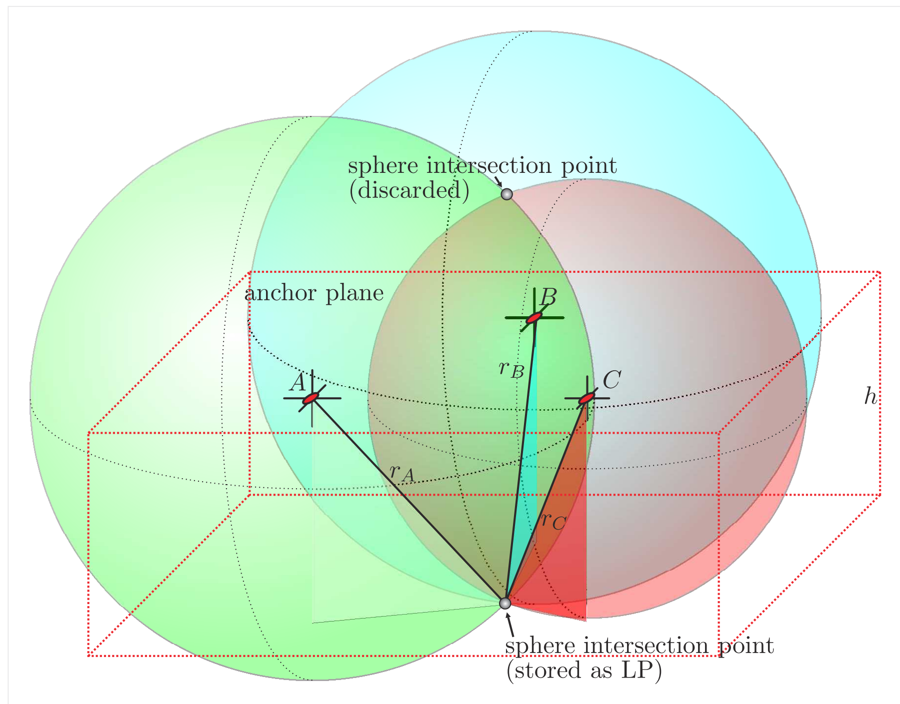



**Figure 13.1.:** Intersecting a triplet of spheres based on distance measurements between three anchors $A, B, C$ and one client (due to measurement errors, the resulting location point (LP) needs not necessarily match the client's true position $p_m$)

Intersecting a triplet of such spheres will always result in either 0, 1, or 2 intersection points. Since we assume the anchors to be placed in a common anchor plane[1] only the last solution scenario is relevant for us, and we discard the intersection point above the anchor plane; the other one will be stored as so called *location point* (LP), and represents *one* potential solution for the current position estimation. This way pVoted can start its computation as soon as the third DV has been received, and immediately receives an "early result". Please note, that when using the HashSlot TDMA slot reordering scheme from Section 12.6, the first three DVs will definitely describe spheres at non-collinear center positions – a relevant advantage for many localization algorithms which cannot operate otherwise (like e.g. trilateration using linear equations), or which would at least suffer from numerical instability (like e.g. WCL or pVoted) then.

To progressively obtain more LPs from each successively received DV, pVoted does not only store the LPs but also the relevant DV information (i.e. the sphere's center and radius). Apart, it generates up to $\binom{i-1}{2}$ new LPs for the $i$-th received DV. Listing 13.1 gives an outline on the DV processing function. Obviously, the data fusion process will take longer with each received DV, and finally, for $g$ received DVs, $\binom{g}{3}$ DV triplet intersections must be computed to receive and store up to $\binom{g}{3}$ LPs[2]. If we however recall the feature of the HashSlot data aggregation protocol to limit the number $g$ of sending anchors using a QoS request, and to progressively extend the slot lengths after each successful transmission, we see once more that the foresighted and

[1] Minor deviations in the z-coordinate can easily be compensated on-the-fly as described in [242].
[2] With some optimization within our code this involves $\sum_{i=1}^{g}(i-1)$ sphere intersections.

```
1  int processDV(dv) {
2    s = createSphere(dv);          // create new sphere from just received DV
3
4    foreach (triplet t of spheres containing s) {
5      q1 = intersectTriplet(t);    // new LP (Eq.(13.2), Eq.(13.14))
6      foreach (previously created LP q2) {
7        if (d(q1, q2) <= δ) {      // mutual voting (Eq.(13.4))
8          vote(q1, q2);            // updates consistencies, voter count, and WCL
9                                   // information (Eq.(13.5) -- Eq.(13.9))
10         qb = updateBestLP(q1,q2); // select by max. consistency
11       }
12     }
13     storeLP(q1);  // if necessary, deletes the least consistent LP first
14                   // to free memory
15   }
16
17   storeSphere(s); // if necessary, deletes the worst sphere first to free memory
18                   // i.e. the one which is involved in the least voted DVs
19
20   if (α(qb) ≥ 2/3 && β(qb) ≥ 2/3)  // classify by (Eq.(13.11), Eq.(13.12))
21       return 1;  // indicate good quality of qb (stop data aggregation)
22   else return 0;  // indicate bad quality of qb (wait for more DVs)
23 }
```

**Listing 13.1:** The pVoted core algorithm for processing a just received DV (pseudocode)

application-specific protocol design from Chapter 12 turns out as a great benefit now[3].

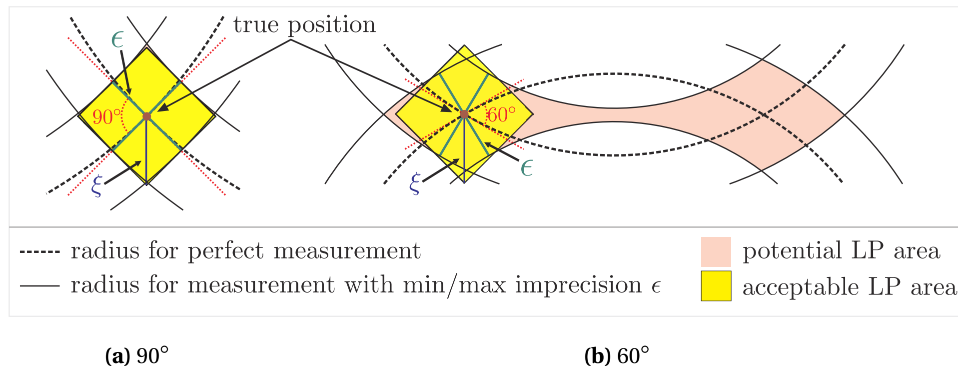
Nevertheless, regarding the low CPU performance and the severely restricted RAM of typical sensor nodes[4] the resource-related and computational effort would initially still be too complex to store all DVs and LPs – especially if we consider that error-prone DVs will also increase the data fusion cost. For example, $g = 9$ received DVs would already demand for 36 sphere intersections resulting in up to $\binom{9}{3} = 84$ LPs, and require $9 \cdot 8\,\text{B} + 84 \cdot 6\,\text{B} = 576\,\text{B}$ just for the spatial information[5]. While we can accelerate the computations by storing intermediate results for subsequent DVs at even more memory consumption(!), another approach showed to be much better for reducing both costs:

The idea is to simply limit the maximum number of storable DVs and LPs. If we take the sharp separation of central and side errors from Figure 11.4b[p249] into account, we can let the LPs vote for each other to evaluate their trustworthiness, and delete the least "consistent" LPs as well as the corresponding DVs as soon as the available memory is exhausted. This frees space and saves CPU time for the next DV processing. While a careful LP ranking is essential then to not mistakenly delete good data but keep the bad instead, the consistency will also be used for selecting one LP for the final position estimation. Since it might still happen that "good" data must be deleted during the data aggregation process – e.g. in case of many received DVs compared to the memory quotas – it is important to transfer the relevant information to the remaining LPs first. Following Listing 13.1 this is achieved as follows:

[3] Of course it would always be possible with any protocol to simply stop after the $g$-th DV, but the ones received so far would probably not be distributed comparably uniform within the US coverage zone. Besides, a waste of time and (exclusive) anchors resources would occur if these stood involved unnecessarily long.

[4] … which already meant considerable resource sharing efforts through dynamic memory management to integrate both the SNoW Ghost remote maintenance system and the Cut DSP algorithm (→ Section 7.6.2).

[5] While we assume 16 bit values for x/y/z-coordinates and distances at millimeter resolution, the true memory demand of our implementation is 2610 B and includes some additional runtime information.




**(a)** 90° **(b)** 60°

**Figure 13.2.:** The impact of the distance measurement imprecision $\epsilon$ on the potential position of intersection points (2D visualization).

**Triplet intersection analysis for precision evaluation.** Intersecting three spheres (→ Line 5) with a distance measurement imprecision of up to $\pm\epsilon$ in each radius reduces the expectable precision and accuracy of the intersection point the less the closer the angles between the tangents in this intersection point approach 90°. A 2D example for two circles is illustrated in Figure 13.2: For $\varphi = 60°$ in Figure 13.2b even a slight imprecision can potentially move an intersection point – and the corresponding LP in particular(!) – significantly further away from the true position than exactly the same imprecision would do for $\varphi = 90°$ in Figure 13.2a. Projected to the 3D case, the intersection of two spheres $s_1, s_2$ with an intersection angle $\varphi$ receives an initial precision trust

$$P(\varphi_{s_1,s_2}) := \begin{cases} \frac{1}{90} \cdot \varphi_{s_1,s_2} & \text{if } \varphi_{s_1,s_2} \in [0°, 90°] \\ -\frac{1}{90} \cdot \varphi_{s_1,s_2} + 2 & \text{if } \varphi_{s_1,s_2} \in (90°, 180°] \end{cases} \in [0,1] \tag{13.1}$$

as visualized in Figure 13.3. While we considered various alternatives for Eq. (13.1) which will not be discussed here, this one achieved the best results throughout our tests. Accordingly, each newly created LP $q$ (which in turn results from intersecting two sphere intersections from a triplet $(s_1, s_2, s_3)$ with $s_1$ being the latest sphere) receives the precision trust

$$P(q) := \frac{1}{2}\Big(P(\varphi_{s_1,s_2}) + P(\varphi_{s_1,s_3})\Big). \tag{13.2}$$

Taking also $P(\varphi_{s_2,s_3})$ into account would be intuitive, but was intentionally avoided for performance reasons and since it did not affect the quality of the result notably. Another issue which became apparent during our tests is, that anchors at larger distance from each other within the US coverage zone tend to result in more trusted intersection angles[6]. Thus it is yet another benefit of HashSlot that reordering the transmission slots will schedule anchors opposing each other within their module cell first (→ Figure 12.9[p282]).

[6]While it should thus be optimal to use US transmitters with an emission angle of about 90° (→ Figure 11.1[p245]) our transceivers did unfortunately not allow us to approve this under real-world conditions.

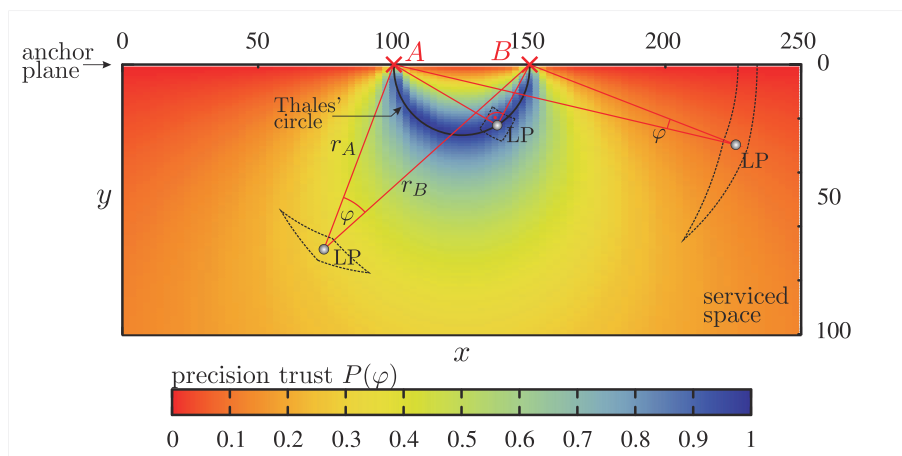



**Figure 13.3.:** The precision trust distribution for two anchors $A, B$.
Referring to Figure 13.2 the three depicted intersection examples also exhibit the potential LP areas: The "more quadratic" the larger the precision trust $P(\varphi)$.

**Mutual LP voting for accuracy evaluation and on-line data reduction.** Having assigned a precision trust $P(q)$ for each newly created LP $q$, the next step is the actual voting (→ Line 6). Once more we take the measurement error characteristics from Figure 11.4b[p249] into account: If we imagine a 3D version of Figure 13.2a, it becomes obvious that even for this optimal constellation with $\varphi = 90°$, and a maximum imprecision of $\pm\epsilon$ for three intersecting spheres from the same accuracy category (either central or side error), the maximal imprecision $\xi$ which we have to accept for the final position estimation computes as

$$\xi := \sqrt{3\epsilon^2} = \sqrt{3}\epsilon. \tag{13.3}$$

Since we always assume at least a small distance measurement error we also assume $\xi > 0$. Thus, a "good" LP must never be more than $\xi$ away from the true position, and the maximum distance $d \in \mathbb{R}_0^+$ we accept between two LPs $q_1$ and $q_2$ to vote for each other (→ Line 7 et seq.) is limited by

$$\delta := 2\xi = 2\epsilon\sqrt{3}. \tag{13.4}$$

Evaluating pairwise LP distances $d$ will not only create clouds of LPs voting for each other, but it is also intended to separate "good" LPs (which emerged from accurate distance measurements around the central error) from inaccurate ones (which resulted from side errors). Figure 13.4 shows an example scenario from our evaluation environment.

From the overall set of LPs we still have to select the most likely one within the most likely cloud for our final position estimation. Therefore the voting process successively counts each LP's voters $v(q)$ and adds a distance $d$ dependent attraction bonus $s(d)$ to the consistency $C(q)$ of each one of the two LPs $q_1, q_2$ it currently compares:

$$C(q_1) := C(q_1) + L(q_2) \cdot s(d) \quad \text{and} \quad C(q_2) := C(q_2) + L(q_1) \cdot s(d) \tag{13.5}$$

$$v(q_1) := v(q_1) + 1 \quad \text{and} \quad v(q_2) := v(q_2) + 1 \tag{13.6}$$

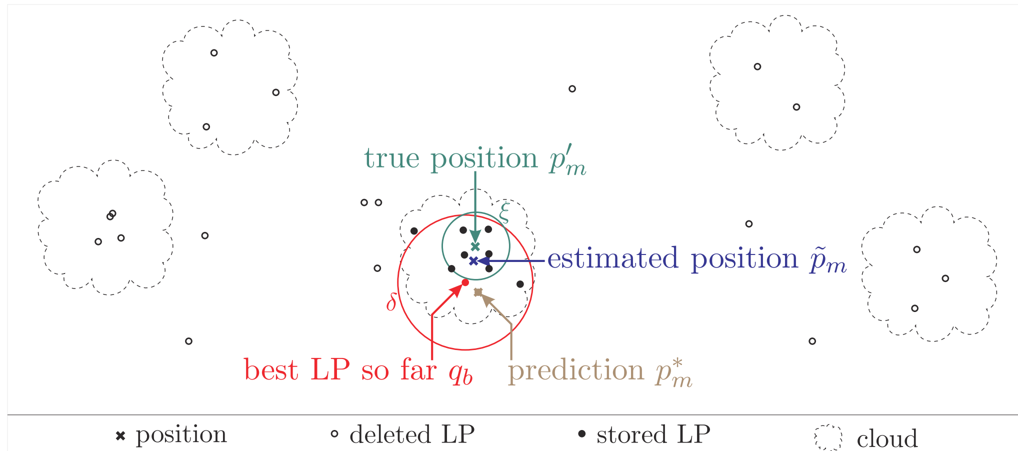



**Figure 13.4.:** A potential pVoted scenario: The LPs within the red circle of radius $\delta$ did vote for the best LP so far, and the final position estimation is indeed within the green circle of radius $\xi$ representing the tolerable distance error.

While the initial values for both $v(q)$ and $C(q)$ is 0, $L(q)$ is an LP's location trust which adjusts the attraction between two LPs and remains to be discussed later in the context of position prediction – for now we simply assume $L(q) = 1$. The attraction $s(d)$ represents a linear score with 1 indicating the maximum tolerated distance (i.e. $d \overset{\text{Eq. (13.4)}}{=} \delta$), and 2 indicating a perfect match (i.e. $d = 0$):

$$s(d) := -\frac{d}{\delta} + 2 \in [1;2] \tag{13.7}$$

Keeping our goal to implement a progressive scheme in mind, the most consistent LP $q_b$ so far will be marked and updated immediately after each voting (→ Line 10). Also, each newly created LP will be stored (→ Line 13). In case the available memory therefore is exhausted, the least consistent one will be removed for on-line data reduction (not information reduction as described next). It is similar for saving the spheres: The ones which contributed to the least voted LPs will be removed first (→ Line 17).

**WCL for position estimation.** Apart from the accuracy related information, pVoted also accumulates some spatial information for each LP $q$ to simplify and accelerate the final position estimation using an adapted weighted centroid localization (WCL) over the most consistent LP $q_b$ and its surrounding cloud of supporting voters. The idea is to move the coordinate of $q_b$ component-wise according to the weighted distances of its voters in range $\delta$.

However, we have to avoid the need to identify, access, and process these voters once more then, since, apart from their possibly large number, some of them might already have been deleted. Thus, we also accumulate the component-wise weights $W_{x/y/z}$ for the two LPs $q_1$ and $q_2$ with Euclidean distance $d = ||(d_x, d_y, d_z)|| \le \delta$ voting for each other:

$$\begin{aligned} W_\omega(q_1) &:= W_\omega(q_1) + d_\omega \cdot P(q_2) \cdot s(d) \quad \text{and} \\ W_\omega(q_2) &:= W_\omega(q_2) + d_\omega \cdot P(q_1) \cdot s(d) \quad \text{with} \quad \omega \in \{x, y, z\} \end{aligned} \tag{13.8}$$

While the mutually imposed weight obviously depends on the precision trust $P(q)$ of each voter, a divider with initial value 0 must also be updated incrementally:

$$\begin{aligned} \text{divider}(q_1) &:= \text{divider}(q_1) + P(q_2) \cdot s(d) \\ \text{divider}(q_2) &:= \text{divider}(q_2) + P(q_1) \cdot s(d) \end{aligned} \tag{13.9}$$

In the end, i.e. as the data aggregation stops for whatever reason (→ Section 10.2.4 for examples), the final position estimation $\tilde{p}_m$ for a client $m$ will be computed from just the currently most consistent LP $q_b$ at position $(q_{b,x}, q_{b,y}, q_{b,z})$:

$$\tilde{p}_m := (m_x, m_y, m_z) \quad \text{with} \quad m_\omega := q_{b,\omega} + \frac{W_\omega(q_b)}{\text{divider}(q_b)} \quad \text{for} \quad \omega \in \{x, y, z\} \tag{13.10}$$

**Result classification for early termination.** Considering the ultrasound measurement error distribution from Figure 11.4b[p249] we can easily predict the expected number of accurate DVs from the total number of received DVs. From these values we can in turn compute the expectable number of voters $V_{\text{exp}}$ as well as the maximum obtainable number of voters $V_{\max}$ for a "good" LP[7]. These values can be used to classify the LP $q$ and the final position estimation $\tilde{p}_m$ resulting thereof as

$$\alpha(\tilde{p}_m) = \alpha(q) := \begin{cases} \frac{v(q)}{2 \cdot V_{\text{exp}}} & \in [0; \frac{1}{2}) \text{ for } v(q) < V_{\text{exp}} \\ \frac{v(q) - V_{\text{exp}}}{2 \cdot (V_{\max} - V_{\text{exp}})} + \frac{1}{2} & \in [\frac{1}{2}; 1] \text{ for } v(q) \geq V_{\text{exp}} \end{cases} \tag{13.11}$$

and

$$\beta(\tilde{p}_m) = \beta(q) := \frac{\text{divider}(q)}{2 \cdot v(q)} \in [0; 1]. \tag{13.12}$$

This means: The larger $\alpha(\tilde{p}_m)$ the more accuracy can be expected from the position estimation, and the larger $\beta(\tilde{p}_m)$ the more precision can be expected. During our tests values below $\frac{1}{3}$ for both metrics commonly indicated "bad" position estimations; these should neither be used for subsequent position predictions nor by the application which uses the localization service. In contrast, we stopped the data aggregation as soon as both values reached $\frac{2}{3}$ since this empirically indicated the result to be sufficiently "good", i.e. within its tolerable distance error $\xi$.

**Position prediction.** pVoted maintains a history of up to 3 most recent position estimations $\tilde{p}_i$ with $i \in \{1, 2, 3\}$[8], and $\tilde{p}_1$ being the latest. Moreover, it assigns a tristate marker $\eta_i$ for each one to indicate if itself did match with its own prediction $p_i^*$:

$$\eta_i := \begin{cases} 1 & \text{if } \|\tilde{p}_i - p_i^*\| \leq \xi \\ 0 & \text{if } \|\tilde{p}_i - p_i^*\| > \xi \\ -1 & \text{if no prediction } p_i^* \text{ was available} \end{cases} \tag{13.13}$$

[7] While details on the formulas will be omitted here, we use a lookup table for up to 6 DVs. This proved to be absolutely sufficient for our error characteristics.

[8] Note that the maximum tolerable age of a historic estimation must be adjusted to the client's speed and its likeliness to change its direction.

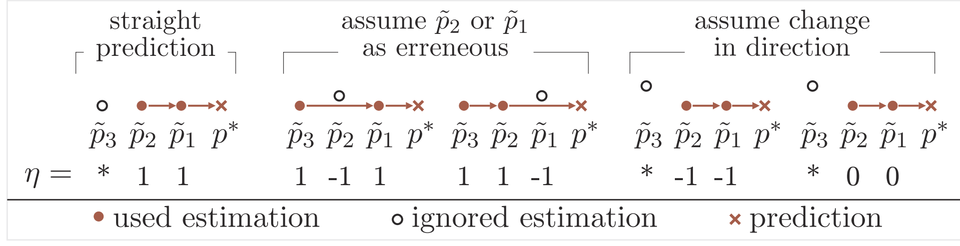



**Figure 13.5.:** Position prediction based on three historic estimations $\tilde{p}_1, \tilde{p}_2, \tilde{p}_3$

As illustrated in Figure 13.5 the new prediction $p^*$ for the current position estimation computes as linear interpolation over a rule based selection of two historic estimations: If $\eta_1 \geq 0$ use $\tilde{p}_1$ and the next $\tilde{p}_i$ with $i \in \{2,3\}$ and $\eta_i \geq 0$; this eliminates one potential error in either $\tilde{p}_2$ or $\tilde{p}_3$. If $\eta_1 = \eta_2 \leq 0$ assume a change in the client's movement direction and nevertheless use $\tilde{p}_1, \tilde{p}_2$. If $\eta_1 < 1$ but $\eta_2 = \eta_3 = 1$ assume $\tilde{p}_1$ as erroneous and use $\tilde{p}_2, \tilde{p}_3$. In any other case (some won't occur), the prediction is omitted entirely.

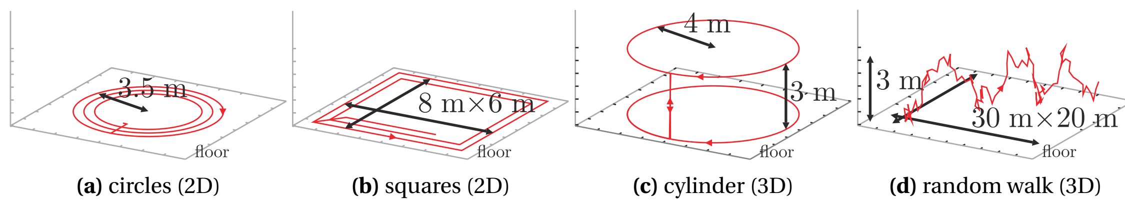
Once computed, the prediction $p^*$ itself will according to Eq. (13.11) and Eq. (13.12) simply be considered as the first LP $q^*$. While $q^*$ will never be deleted, its consistency $C(q^*)$ and precision trust $P(q^*)$ will simply be computed from the average $\alpha$ and $\beta$ values of the used historic estimations. Apart, $q^*$ will initially be assumed as the best LP so far (i.e. $q_b := q^*$), and the prediction attracts other LPs within its cloud more than ordinary LPs do. Thus, as applied in Eq. (13.5), the location trust $L(q) \in [0;1]$ for each newly created LP $q$ computes as

$$L(q) := \begin{cases} 1 & \text{if } q_i = q^* \text{ or if no prediction available} \\ \frac{2}{C(q^*) \cdot v(q^*)} & \text{otherwise} \end{cases} . \tag{13.14}$$

While this decreases the impact of ordinary LPs along with an increasing expected "quality" of $q^*$, a bad prediction $p^*$ will still be outperformed as the initially most consistent LP during the progressive DV processing. Beyond, this history based evaluation of the future even smooths captured paths for tracking applications.

## 13.3. Evaluation

During the just presented description of the most central pVoted operation steps we have already anticipated to expect various synergetic benefits between the progressive voting approach, the data aggregation protocol design, and the software architecture. Though we avoided the discussion of general mathematical details on position estimation algorithms (since this would go beyond the scope of this work for the reasons given in Section 13.1), we will instead focus on analyzing the mutual influences between these interacting concepts, and present some empirical studies to support the validity of our assumptions from the previous chapters. Namely, these are:




**(a)** circles (2D) **(b)** squares (2D) **(c)** cylinder (3D) **(d)** random walk (3D)

**Figure 13.6.:** SNoW Bat test bed evaluation paths (ceiling height $h = 7$ m, $d_{\min} = 2.8$ m, $L = 1.3$ m)

- The general pVoted position estimation performance (under various HashSlot configurations) in comparison to some other selected approaches.
- The impact of both the DV slot reordering from Section 12.6 *and* the result classification scheme from Section 13.2 on both the pVoted precision and the localization speed.

**The analysis framework.** For a convenient comparison of various system configurations and position estimation algorithms, we developed a tailored observation framework for desktop PCs with support for both purely simulation based analysis *and* hardware-in-the-loop operation. Implemented in Java it interfaces a *Smart*Net compatible SNoW[5] sniffer node as depicted in Figure 10.2[p226] to passively or actively participate in the network's radio communication. This way it gains easy access to the CAV and DV packets, and can once more supervise the reliability of the HashSlot protocol as well as the temporal progression of most SNoW Bat operation stages $P_1$ – $P_4$ from Section 10.1.1[9]. Most important for this evaluation, the wirelessly gathered information (→ Listings 10.1, 10.2[p228]) allowed to obtain the real-world anchor positions, and to compare the mobile client's position estimation results of the native *Smart*OS implementation (using integer arithmetic and truly limited memory reserves) to the Java implementation (using floating point arithmetic and quasi “unlimited” memory reserves). Yet, the problem with this approach is that neither the simulator nor the real-world installation has perfect knowledge about the client's current position – especially in case of a moving client. Beside we were always limited to the rather small test installation from Figure 10.1[p225].

As a consequence, the analysis framework was extended to also simulate arbitrarily deployed virtual anchors and moving clients within 3D environments of arbitrary size: An emulation for the ultrasound distance measurement is based on the error characteristics which we obtained from using our Cut algorithm in Chapter 11 as real-world reference model (→ Figure 11.4b[p249]). A HashSlot emulation calculates the TDMA slot schedule for the anchors within the projected US coverage zone as well as the corresponding DV processing order at a freely moving virtual client.

For each position estimation iteration, either the sniffed DVs or the ones which were created by the simulator are successively forwarded to the available position estimation algorithms. Thus we achieved a perfectly identical information base for all range based data fusion approaches under test:

[9] As a side effect, the sniffer node also served as gateway for the SNoW Ghost remote-management subsystem from Section 8.2.

- Centroid Localization (CL), Weighted Centroid Localization (WCL), and MinMax (also known as BoundingBox) were tested, but won't be presented here due to their comparably bad performance (see [259] for an evaluation in the context of RSSI based localization).
- Multilateration (ML) was tested in two modes: While the regular implementation was provided with simply all received DVs, the "oracle" implementation received and processed only the accurate distance measurements, i.e. those within the central error in Figure 11.4b[p249]. Although the oracle method is obviously not applicable in real-world systems, it achieves almost completely error-free results and served as a challenging reference competitor for pVoted.
- Trilateration with Kalman filtering according to Eckert [91] was finally used to compare an approach from another real-world installation.
- pVoted – of course.

Apart from the pVoted and Eckert's trilateration approach an overview on the remaining algorithms can be found in [259][10].

**Analysis I: Simulation.** For our first analysis we tracked a mobile node along four traces within a virtual industrial hall of size 30 m×20 m×7 m. While three paths were fixed for spatial repeatability under varying HashSlot and pVoted configurations (→ Figures 13.6a – 13.6c), one path follows a random walk for increased diversity regarding environmental constellations (→ Figures 13.6d). The anchors at the ceiling were installed according to the demanded grid structure from Section 10.2.2 and Eq. (10.1) to grant a 99% probability for receiving at least 4 accurate measurements during each position estimation: With a uniformly distributed placement deviation of $\pm 10\% \cdot L$ in both the $x$ and $y$ direction they were nevertheless assumed to be perfectly calibrated. While this is initially acceptable for this test bed, the impact of less precisely calibrated anchors as well as a corresponding self-calibration approach for SNoW Bat is discussed in [256][11] but won't be applied here.

For pVoted only, the already mentioned memory saving strategy was incorporated by reserving space for up to 6 spheres (DVs) and up to 10 LPs before we had to delete the least consistent ones to gain space for new data. Figure 13.4 once more gives an example for a potential LP constellation within this test bed. For all algorithms mentioned above, Figure 13.7a shows the rooted mean square errors (RMSE)

$$\sqrt{\frac{1}{n} \cdot \sum_{i=0}^{n-1} \left\| p'_{m_i} - \tilde{p}_{m_i} \right\|^2} \tag{13.15}$$

between the true client positions $p'_{m_i}$ and the position estimation $\tilde{p}_{m_i}$ over $n = 1000$ iterations.

Two bars are presented for pVoted: While the red bar represents the results over all $n$ iterations, the pink bar (pVoted*) refers to only those results which were classified as "good" by the algorithm

[10] Diploma thesis conducted in conjunction with this work.
[11] Diploma thesis conducted in conjunction with this work.

itself; i.e. those estimations for which the algorithm assumed a precision below $2 \cdot \xi \overset{\text{Eq. (13.3)}}{=} 2\epsilon \cdot \sqrt{3} \approx 2.98\,\text{mm}$. The true percentage of position estimations within this tolerable bound is also given as numbers for each algorithm within this Figure, and identifies pVoted as quite comparable to the oracle multilateration. Note that the RMSE values become worse for the 3D paths since moving closer to the ceiling involves less anchors and information for the data fusion process. Regarding pVoted's quality self-awareness Figure 13.7b shows the corresponding binary evaluation of the pVoted classifier [325]: The good “correct classification” and “positive predictive values” (PPV) raise the hope that using consistency thresholds can reliably be used for terminating the data aggregation process early without losing too much precision and accuracy[12].

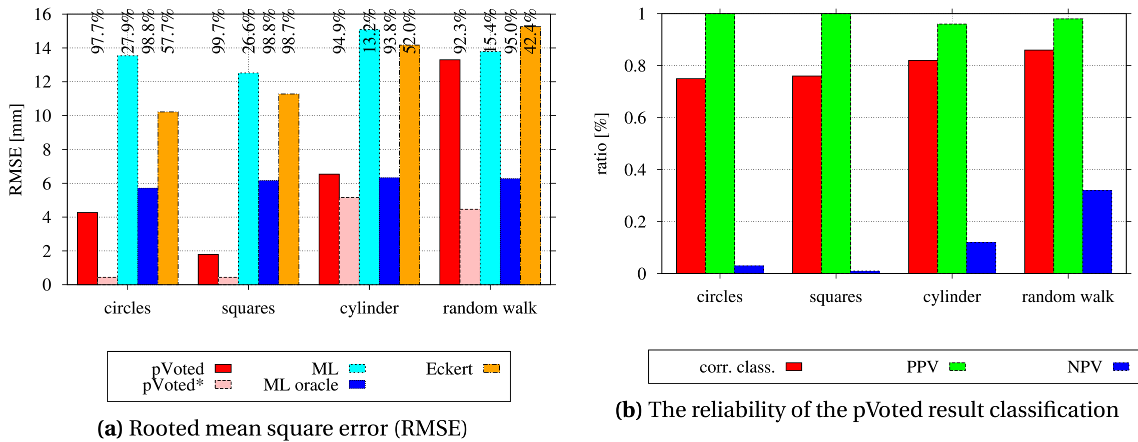
For pVoted only Figure 13.8b illustrates these savings by comparing the average number of potentially available DVs to the average number of discarded DVs per position estimation iteration. Apart, it also supports the already expected influence of the TDMA slot schedule and DV processing order: Using slot reordering according to Section 12.6 was intended to obtain the more relevant DVs first, and indeed manages to cope with less DVs; especially for the “circles” and “squares” paths where more DVs can potentially be aggregated due to the client's larger distance from the anchor plane. Even more interesting, Figure 13.8a shows that reordering really maintains a higher precision for pVoted (red bars) than doing without (pink bars). This influence is obviously less significant for the other schemes which always process all DVs and do not attempt to ignore some of them entirely. On the other hand, ignoring DVs under pVoted also results in reduced CPU load and improved data aggregation speed as already discussed in Section 10.2.4.

Up to now we did not consider the impact of the HashSlot QoS option. As we have just seen, this is acceptable for position estimation algorithms which are able to stop the data aggregation process according to a dynamic classification scheme; for the others, however, explicitly requesting an “adequate” number of DVs remains the only option to tune the performance. Comparable to Figure 13.7 without any QoS limitation, Figure 13.9 show the RMSE and classification evaluation for $g = 16$ requested DVs. Though the RMSE increases slightly for the “circles” and “squares” paths due to the reduced amount of information (Figure 13.8b certifies about 20 potentially available DVs without the QoS limitation), the other paths are less significantly affected since the average number of available DVs remained almost constant.

**Analysis II: Real-World.** Following the Java implementation, pVoted was also implemented in C[13] for execution on real SNoW[5] hardware within our SNoW Bat setup from Figure 10.1[p225]. This allowed us to verify the expected impact of the hardware/software/network co-design as discussed in e.g. Section 10.2.4 and Chapter 12, as well as the position estimation error under real-world conditions. Since however the overall system quality and performance depends on the intense interaction between the various techniques, concepts, and paradigms from earlier

[12]The comparably low “negative predictive values” (NPV) are quite acceptable since classifying sufficiently precise estimations as “bad” every now and then is less critical than vice versa.

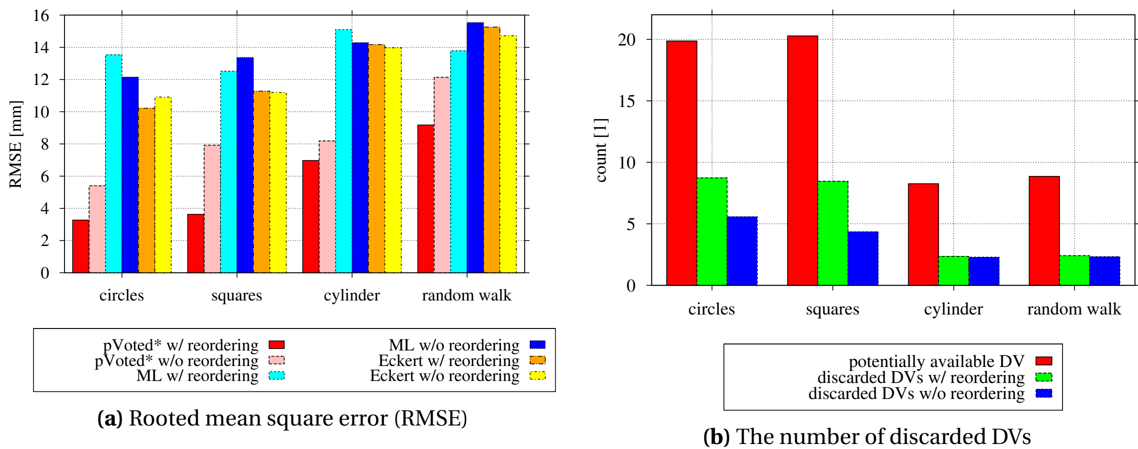
[13]Though the realization as *Smart*OS library is quite straight forward, dynamic hints must be handled according to Figure 10.8b[p237].




**(a)** Rooted mean square error (RMSE)

**(b)** The reliability of the pVoted result classification

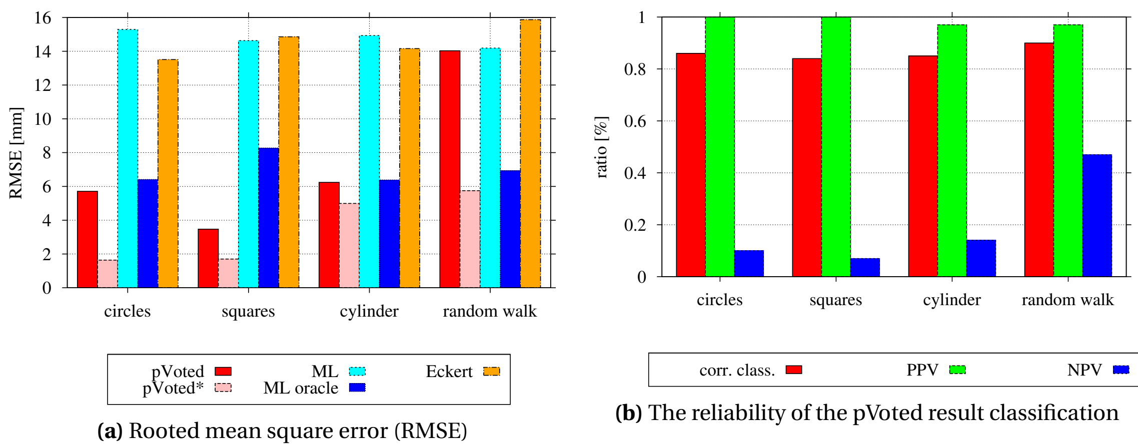
**Figure 13.7.:** Position estimation results for the paths from Figure 13.6 (average over 1000 iterations) (HashSlot: QoS = max, slot reordering enabled — pVoted: early termination disabled)




**(a)** Rooted mean square error (RMSE)

**(b)** The number of discarded DVs

**Figure 13.8.:** The impact of the HashSlot TDMA slot reordering scheme on the data aggregation speed (average over 1000 iterations) (HashSlot: QoS = max — pVoted: early termination enabled)




**(a)** Rooted mean square error (RMSE)

**(b)** The reliability of the pVoted result classification

**Figure 13.9.:** Position estimation results for the paths from Figure 13.6 (average over 1000 iterations) (HashSlot: QoS = 16, TDMA slot reordering enabled — pVoted: early termination disabled)

chapters, we'll defer the discussion of the "combined" results to the conclusion Chapter 14. Yet, as an early summary we can already anticipate that the entire system of distributed nodes performed sufficiently fast and reliable to generate about 2.5 position estimations per second at an average spatial error of about 10 mm: Also limited to storing up to 6 DVs and 10 LPs for saving memory and CPU load (just like we did for the simulative analysis), the final position estimation results between the Java and the C version processing the same captured distance information differed by about 2 mm caused by rounding and the discrepancies between floating point and integer arithmetic. The remaining differences in the spatial results will be discussed next.

## 13.4. Summary

This chapter introduced the pVoted position estimation algorithm as a representative for progressive data fusion approaches. Based on successively collected distance measurements, potential solutions are calculated *and* classified immediately, i.e. with each incoming data packet, through mutual voting: Taking the (known) distance measurement error characteristic into account, this allows to reliably distinguish "good" from "bad" information, and to filter the latter immediately for reducing the data fusion cost in term of memory demands and CPU load on the typically weak and resource constrained sensor node hardware.

As already expected in the previous Chapter 12, where we dealt with the general concepts of wireless data aggregation in lateration based indoor localization systems, the pVoted test beds showed once more that is is essential (or at least beneficial) to obtain the most "useful" distance information first: In combination with the progressive filtering and "quality" classification this can be exploited for terminating the data fusion process early, and to save even more CPU time and energy in particular.

Compared to most naïve approaches, were information is collected arbitrarily and greedily (maybe just bounded by a certain deadline or packet count limitation), the combination of

- DV transmission reordering through HashSlot,
- solution filtering through mutual voting, and the
- early termination through continuous result classifications

did not only satisfy our long-term goals to reduce both the data aggregation *and* the data fusion cost (→ Figure 9.2), but it also managed to reliably keep the position estimation error low. In this concern, pVoted always achieved overall results which are definitely comparable to the "oracle" multilateration approach. If we limit the evaluation to those values which were considered as "good" by the algorithm itself – and these make up about 98% of the overall position estimations – we even obtain significantly better results as depicted in Figures 13.7a and 13.9a. Thus, we can finally state that incorporating (quality) self-awareness as already demanded in Figure 1.7[p14] proofed to be a true benefit for our specific application, and will probably be a good advice for any time-critical system – at least if it employs severely constrained hardware.

# 14. Real-World Evaluation: The complete SNoW Bat System Installation

This chapter evaluates the composition of the previously introduced techniques, concepts, and paradigms from within this work by means of a comprehensive real-world analysis[1]. Based on the SNoW Bat installation from Figure 10.1[p225] the presentation of real-world benchmark results completes our initially self-defined demand for testing each presented part of the work within a real environment. The installation comprises a grid of $9 \times 5 = 45$ static anchors (grid constant $L = 0.25$ m) mounted 2 m above the ground, and 1 mobile client moving freely in 3 dimensions with a minimum distance of $d_{\text{min}} = 1$ m from the anchor plane. With respect to the title of this work we pay special attention to the temporal performance as mainly defined by the hardware/software/network co-design, but also present some related spatial performance results for the sake of completeness. While the general operation principles of SNoW Bat have already been introduced in Chapter 10, Figure 14.1 summarizes once more the complete execution flow spanning over the stages $P_1$ - $P_4$ as defined in Section 10.1.

[1]Specific benchmarks can be found within the corresponding chapters of this work.

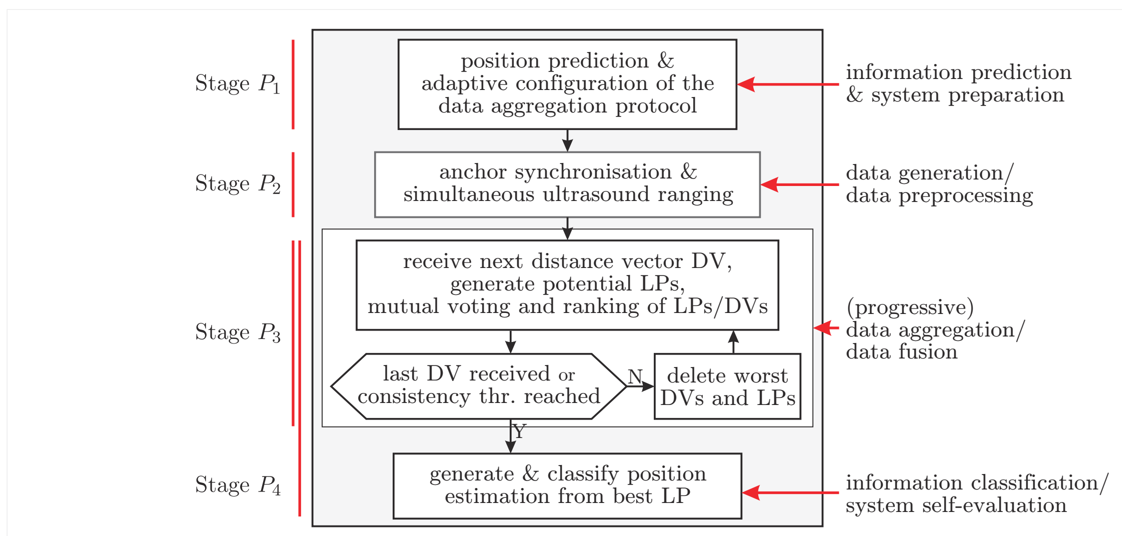



**Figure 14.1.:** The SNoW Bat execution flow with regard to the specific operation stages from Figure 10.3[p227] and the general WSN process flow from Figure 9.2[p211]

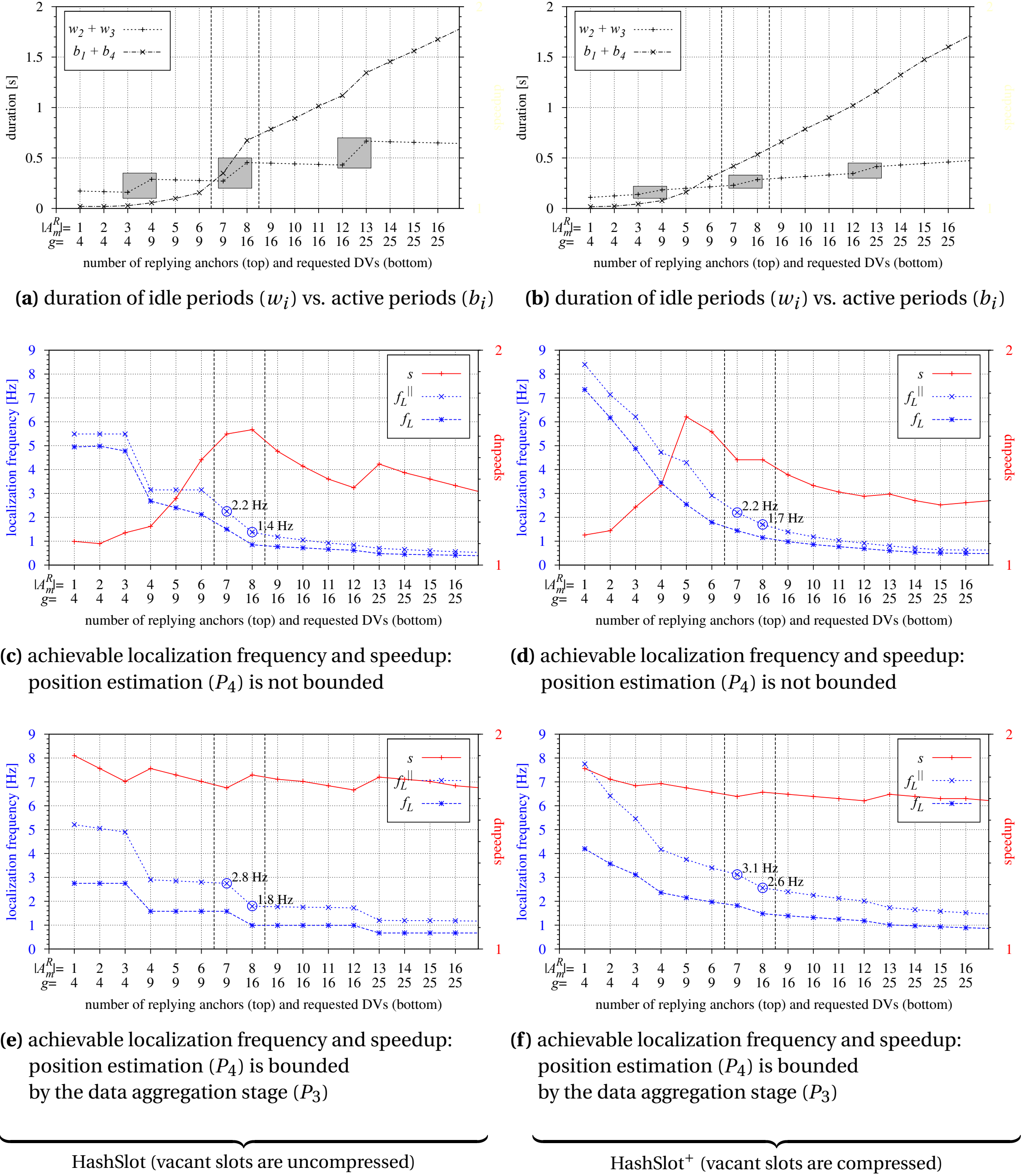
## 14.1. Temporal Performance

Figure 14.2 gives the timing results for various system configurations at the mobile client[2]: Regarding our software design optimization attempts from Section 10.2.4 to parallelize the position estimation task $T_{PE}$ and the ultrasound task $T_{US}$, both Figures 14.2a and 14.2b compare both tasks' total active and idle periods when using either HashSlot or HashSlot$^+$ for the data aggregation: While for an equal number $g$ of received DVs the data fusion process (i.e., $b_1 + b_4$) can obviously not benefit from the TDMA slot compression (in fact, the CPU load for pVoted remains constant), the idle times (i.e., $w_2 + w_3$) for waiting on the DVs were significantly reduced through HashSlot$^+$. In particular, this also explains the clear steps in Figure 14.2a for those cases where the fraction of used ($|A_m^R|$) and reserved ($g$) slots is low, i.e. when we switch to the next power-of-two number for the QoS level. This phenomenon is almost completely smoothed out in Figure 14.2b where vacant slots are significantly shorter. On the other hand, since the two tasks $T_{US}$ and $T_{PE}$ are intended to operate interleaved, one might think that pVoted can profit from increased radio silence between two DVs to process previously collected information and (maybe) terminate early or be more precise. Even though this is not wrong per se, we do luckily not depend on such occasional coincidences, but profit once more from the traffic shaping feature of HashSlot and HashSlot$^+$ which adjusts the DV transmission rate of the anchors to the varying data fusion complexity of the client as described in Section 12.6: In fact, the TDMA slot length will grow progressively to provide sufficient time for the DV processing[3].

Keeping this option in mind, we can continue to compare the achieved localization frequencies $f_L$ and $f_L^{\|}$ for the serialized and the parallelized task execution: Figures 14.2c and 14.2d refer to a setup where the position estimation stage is not bounded, i.e. where pVoted "runs to completion" until either all DVs have been processed or until the consistency threshold has been reached to terminate early. Figures 14.2e and 14.2f refer to a setup where the data fusion is additionally bounded by the deterministic duration of the corresponding data aggregation stage. This means the hint in Figure 10.8b[p237] is ignored for the first and accepted for the latter setup: While the relatively large idle times under HashSlot still result in clearly visible frequency steps when changing the number of requested DVs $g$, HashSlot$^+$ continues to reliably compensate this effect and consequently achieves an increased overall localization frequency. Limiting the data fusion duration to the data aggregation duration increases the localization frequency even further, and makes the entire localization process deterministic – a highly appreciated property in time-critical systems which is clearly reflected by the quite linear frequency graphs for constant values of $g$ in Figures 14.2e and 14.2f. The effect is even better reflected by the increased and more steady speedup $s$ (in comparison to the unbounded data fusion) which we gain from evaluating the benefit of the task parallelization. Even though Eq. (10.9)[p237] predicted $s = 2$ in the optimum case, various system overhead (e.g. caused by the operating system) and

[2] Evaluating the static anchors was intentionally omitted here since the client's operation is much more complex and time-critical.

[3] Note that HashSlot – which does not observe the radio channel to detect foreign DVs or vacant slots – will increase the TDMA slot length for *every* slot, while HashSlot$^+$ – which observes the radio channel to detect and reduce vacant slots – will only increase the slot length for *truly used* TDMA slots. The applied scaling factor has been chosen empirically to match the pVoted runtime demands within our test bed.




**(a)** duration of idle periods ($w_i$) vs. active periods ($b_i$)

**(b)** duration of idle periods ($w_i$) vs. active periods ($b_i$)

**(c)** achievable localization frequency and speedup: position estimation ($P_4$) is not bounded

**(d)** achievable localization frequency and speedup: position estimation ($P_4$) is not bounded

**(e)** achievable localization frequency and speedup: position estimation ($P_4$) is bounded by the data aggregation stage ($P_3$)

**(f)** achievable localization frequency and speedup: position estimation ($P_4$) is bounded by the data aggregation stage ($P_3$)

HashSlot (vacant slots are uncompressed) HashSlot$^+$ (vacant slots are compressed)

**Figure 14.2.:** Serial vs. parallel execution of the distance measurement task $T_{\text{US}}$ and the position estimation task $T_{\text{PE}}$ using HashSlot and HashSlot$^+$ for data aggregation. Relevant scenarios which resulted in sufficiently "fast" and "good" position estimations are enclosed between the vertical lines. Reminder:
$g$ is the number of requested DVs and reserved TDMA slots
$|A_m^R| \leq g$ is the number of replying anchors and used TDMA slots

configuration imperfection (e.g. caused by the lattice point problem as described in Section 12.7) reduces this theoretical gain to $s \approx 1.7$ in Figure 14.2f. Nevertheless, we see once more the enormous advantages an appropriate hardware/software/network co-design can offer, and use the setup from Figure 14.2f with an average localization frequency of $f_L^{\parallel} \approx 2.85$ Hz for the relevant scenarios when analyzing the system's spatial performance quality next.

## 14.2. Spatial Performance

As already mentioned in the previous chapter about pVoted, the achievable position error for a moving client was hard to evaluate under real-world conditions since we had no reliable "reference position estimation system" available for a meaningful comparison. Thus we put a static client at various manually determined positions on the floor and on an elevated platform. For both resulting distances from the anchor plane, i.e. 1913 mm and 1204 mm, respectively[4], we logged the position prediction over $n = 1000$ iterations, and observed two phenomena:

1. For a newly calibrated infrastructure of anchors, we obtained an average position error of about 9 mm – virtually independent from the client's position.
2. About one week after the calibration – though we carefully avoided to touch the installation – the average error had already increased to about 30 mm.

How can this be explained? First, comparing the simulatively obtained position errors of significantly less than 9 mm which we received for the just analyzed virtual industrial hall (→ Figure 13.7[p303]) to these real-world results, shows once more that realistic conditions are hard to represent by a system model: Irregularities in the node hardware (e.g. concerning oscillators and ultrasound capsules) and within the infrastructure deployment itself (e.g. concerning the anchors' intended but hard to preserve common $z$-coordinate for spanning a perfect anchor plane) introduces various sources of imponderabilities as already anticipated in Section 2[pviii]. Nevertheless, achieving a 3D position accuracy of about 10 mm under real-world conditions and with low-cost off-the-shelf sensor node hardware can be considered as acceptable. At least it is a significant improvement over many other ultrasound based systems (→ Table 9.1[p222]). This is particularly true if we also take the localization frequency of about 2.85 Hz into account.

Second, regarding the deteriorating results after one week even minor vibrations, e.g. from people passing by, obviously exerted a significant effect on the rather unsteady metal frame. Using more solid anchor suspensions (e.g. by means of fixed ceiling panels as depicted in Figure 10.6b[p232]) might be one (untested) solution. Relying on continuous self-recalibration (as demanded anyway in the context of self-x techniques in Figure 1.7[p14]) might be another approach. Though this seems to be an essential requirement for counteracting the truly serious problem of keeping long-term installations operational, it will not be discussed within this work; in fact, it already *has been* discussed for SNoW Bat in [256][5].

More detailed spatial performance evaluations regarding the SNoW Bat real-world installation will also not be considered here: Implementation specific precision and accuracy issues have

[4]These distances have already been used for evaluating the Cut algorithm from Chapter 11 (→ Figure 11.3[p247]).
[5]Diploma thesis conducted in conjunction with this work.

already been discussed in [242][6], and a study about the SNoW Bat client tracking capabilities were presented in the context of an autonomous car steering system using fuzzy control and periodic position estimations in [191][7].

[6]Bachelor thesis conducted in conjunction with this work.
[7]Diploma thesis conducted in conjunction with this work.

# Part IV.

# Summary & Conclusion

»Wissenschaft ist dort, wo diejenigen, die als Wissenschaftler angesehen werden, nach allgemein als wissenschaftlich anerkannten Kriterien forschend arbeiten.«

*(Helmut Seiffert, German philosopher)*

# 15. Summary, Outlook and Conclusion

This final chapter of the thesis summarizes the scientific contributions of the various addressed topics. Considering theory and practice, we will also suggest future work and potential research activities to overcome still remaining limitations in the context of concrete application scenarios, and to benefit from emerging possibilities as well as from technological advances to be expected.

## 15.1. Scientific Contributions

According to the title of this work, we present a number of significant "advances in distributed real-time sensor/actuator operation", ranging from sensor node hardware over embedded real-time operating systems to wireless communication issues and concrete application design considerations for indoor localization. As already announced in the introduction, the diversity of the addressed research areas was intentionally accepted to gain a deep understanding of the underlying problems and mutual side effects, and to finally provide far-reaching concepts and paradigms for creating reasonable hardware/software/network co-designs in complex embedded systems. While Part I specifies the related demands, Part II and Part III contain the central contribution of this work with respect to the proposed WSN/WSAN research directions from Section 1.2.1 (→ Figure 1.7[p14]):

**Regarding the hardware aspect**, Chapter 2 initially introduces the **SNoW$^5$** sensor node as versatile base for subsequent test beds and for real-world installations.

**Regarding the operating systems design**, Chapter 4 introduces ***Smart*OS** as an entirely novel solution for implementing preemptive multitasking systems under both hard and soft real-time demands in resource constrained embedded systems. Apart from the support for quite "common" features (like events, semaphores, inter-task-communication, etc.), its central scientific contribution aims on facilitating compositional software designs through four unprecedented dynamic concepts for time management, resource management, and error handling in the area of reactive task systems:

1. Chapter 5, time management: The sophisticated integration of time into the *Smart*OS kernel allows an automatic and unified capturing of event timestamps, as well as the precise scheduling and invocation of reactions with perfectly symmetric temporal error intervals around the true event occurrence or intended reaction time, respectively. In the average case, the temporal error reduces to 0. Furthermore, providing the underlying temporal semantics to the application layer enables the implementation of highly reactive software and precisely predictable system behavior (e.g. for environmental interactions or time synchronization).

2. Chapter 6, resource management: DynamicHinting is an entirely novel concept for sharing arbitrary but strictly exclusive resources (like e.g. peripheral devices or data structures) under real-time conditions in open multitasking environments. Based on extending classic priority inheritance techniques, DynamicHinting features a collaborative approach: Resource conflicts are detected at runtime by the resource manager, and indicated through so called "dynamic hints" to exactly those spurious tasks which currently block other higher priority tasks. Thereby, DynamicHinting applies a generous resource assignment policy where starving low priority tasks is avoided entirely. The associated novel programming paradigm consequently allows for on-demand resource hand-overs: Self-controlled by each individual task, so called time-utility-functions can be used to decide from application specific contracts and from the blocking task's own requirements, about whether to follow or ignore a hint. As a major contribution (compared to all other embedded operating systems we know about), bounded priority inversions can be resolved on-demand to reliably reflect hard real-time demands among concurrently running tasks *without* forced resource revocations or the prophylactic maintenance of so called safe states.

3. Chapter 7, memory management: The CoMem approach is based on DynamicHinting, and provides lock-free and truly dynamic heap management for assembling even memory-demanding subsystems on memory-constrained hardware. As an entirely novel feature in embedded systems design, the application of optional contracts and suitable time-utility-functions for releasing allocated memory blocks on-demand even guarantees worst case allocation times for time-critical requests.

4. Chapter 4, exception handling: The native support for task-specific exception handling is another unknown concept in most embedded operating system kernels. As otherwise only known from higher level programming languages, *Smart*OS exceptions are provided as an improvement to clearly separate the task logic from the error handling. In the special context of dynamic resource management, mapping dynamic hints to exceptions even allows to synchronize sporadically and asynchronously emerging resource conflict situations to the execution flow of any spurious task. This way, the indication delay is only bounded by the overhead for a single task context switch, and increases the performance of both DynamicHinting and CoMem, respectively.

**Regarding the system design and communication aspect**, and to further demonstrate the conceptual advantages and practical suitability of the previously presented techniques, Chapter 8 closes Part II of this work with a first idea on how to combine *Smart*OS, DynamicHinting, and CoMem to realize both the wireless MAC protocol *Smart*Net and the remote-management system SNoW Ghost.

Part III of this work thoroughly addresses various aspects of ultrasound based indoor localization systems. Apart from discussing initial design considerations on the general WSAN process flow (→ Figure 9.2[p211]), we continue to apply the building blocks from Part II for developing **SNoW Bat** as a real-world example for a complex, distributed, and time-critical sensor/actuator system. While the underlying system architecture is described in Chapter 10, the central scien-

tific contribution of Part III covers three novel concepts for co-designing efficiently interacting data generation, data aggregation, and data fusion schemes:

1. Chapter 11, data generation: The Cut algorithm for ultrasound based distance measurement makes use of a lightweight DSP algorithm and the *Smart*OS timestamping capability to simultaneously determine several spatial distances between static anchors within the infrastructure and freely moving clients within the serviced area. As a specific contribution, we managed to create a special error characteristics, which allows us to significantly improve the subsequent data fusion process.

2. Chapter 12, data aggregation: HashSlot is a radio-based TDMA MAC protocol for single-hop data aggregation in reactive networks. The scientific objective is twofold:

   Regarding general communication issues, HashSlot is designed for gathering data on-demand from statically deployed sensor nodes at a mobile sink. Without the need for any explicit coordination or communication between the nodes, the distributed slot schedule calculation is accomplished by each source autonomously: The process relies on a hash function which exploits both implicitly available and explicitly provided information (from the data sink) to guarantee collision-free and tightly packed transmissions even in case of sporadically high radio loads. As often required for event-driven WSN systems with dynamic topology, the schedule is calculated for each data aggregation stage. Nevertheless it scales well, and provably achieves the optimum in terms of response delay, energy consumption, data throughput, and transmission reliability. Using HashSlot$^+$ even compensates for real-world imponderabilities, like node failures, by compressing vacant slots on-demand.

   Regarding the hardware/software/network co-design, the hash function can be adapted to reflect the demands of the actual data fusion process by a suitable transmission order. In case of real-time demands, QoS parameters also allow to precisely configure and predict the duration of the entire data aggregation process, and successively increasing slot lengths facilitate the use of progressive data fusion algorithms with increasing execution times. For a simplified network maintenance, using invertible hash functions finally allows the detection and location of defective nodes from their (vacant) slot numbers.

3. Chapter 13, data fusion: The pVoted position estimation algorithm provides an entirely novel approach for a progressive fusion of the just collected distance information. As final contribution regarding our WSAN research directions from Section 1.2.1, it takes the ultrasound measurement characteristics and a history based position prediction into account to develop a special quality self-awareness: Before each iteration, pVoted allows to configure both the data generation and the data aggregation to reflect the current demands of the mobile sink. During the actual position estimation process, the position estimation filters bad measurements by letting intermediately computed results vote for or against each other. It also evaluates the preliminary and successively improving results to maybe terminate the data algorithm early in case of a sufficiently “good” score.

## 15.2. Future Work

Potential future work in the context of this thesis comprises various options:

**Regarding (indoor) localization systems**, it is always desirable to further improve the precision and accuracy. Within our particular concept, improving the quality awareness for more reliable break conditions during the position estimation process would be another goal. Apart, HashSlot is only optimal for the special anchor pattern we have been using so far. Extending the approach and its hash function to maybe arbitrary deployments or even entirely different purposes would also be an interesting challenge.

**Regarding the demands of event-driven systems**, the implementation of the presented timestamping concept in hardware would be desirable. Since tick counters are available anyway in many CPU architectures, individual timestamps could be taken automatically and simultaneously for each accepted IRQ. In particular, this would overcome the handler latency problem for cascading or tightly consecutive interrupts.

**Regarding the general embedded systems design**, the still increasing demands on the capabilities and features of mobile networked devices impose a strong effect on the evolution of the related hardware and software. An exceptionally interesting point for the conception of reactive open systems is the evolution of multi-core or even many-core processors: These architectures do not only increase the computational power by allowing the truly parallel execution of program code, but they also promise to reduce the overall energy consumption of the CPU.

For many applications, however, these advantages are largely defeated as long as developers are not able to overcome the problem of sharing resources efficiently among the cores. Though numerous virtualization techniques are already available (e.g. [153, 247, 310]), they commonly rely on a (rather strict) isolation between the software subsystems, and assign resources statically and exclusively to the available cores [102, 124].

Here, we see the most promising research direction for extending this work: As already stated, *Smart*OS is not limited to tiny and resource constrained sensor nodes, but it also proved to be suitable for more generic computer systems. DynamicHinting in particular might establish an entirely novel option for sharing resources even across core borders: An early FPGA based realization of the approach already manages the resource access in hardware, and uses special IRQs to pass hints between the cores. While advanced metrics for deciding on hints must be applied then, a first and promising evaluation of the resulting hierarchical three-layer co-design (hardware hints to notify cores, operating system hints to notify tasks, and application based resource conflict handling) can be found in [30].

## 15.3. Conclusion

This work presents various novel advances in distributed real-time sensor/actuator systems operation, ranging from operating systems over communication to application design concepts.

Apart from the theoretical evaluation of the presented approaches, techniques, and paradigms, we also fulfilled our initially self-imposed demand to also prove their feasibility and suitability

under real-world conditions. Therefore, we implemented them on real hardware, and built the indoor localization system SNoW Bat which integrates quite complex subsystems. Though time-critical, these subsystems concurrently accomplish memory intense measurements, computationally complex position estimation, energy efficient wireless communication, and reliable remote maintenance.

The resulting composition revealed various real-time and resource sharing problems. While trying to solve these, we found our initial thesis confirmed: Both novel programming paradigms *and* co-designed approaches are definitely required to properly and reasonably coordinate the typically limited resources of sensor nodes or general embedded systems at runtime – especially when operating in highly dynamic environments.

In this regard, *Smart*OS, DynamicHinting, CoMem (systems level), and HashSlot (network level) can be considered as the most central contributions of this work. In combination with the kernel's temporal semantics and exception handling concept (application level), we provided a strategy to achieve and retain an exceptionally high task and system reactivity when sharing exclusive resources in open multitasking environments. Within our special use case, our localization algorithm pVoted achieved a position estimation accuracy of 1 cm or better, and an update frequency of about 3 Hz using cheap and low power SNoW[5] hardware with off-the-shelf components only.

# Part V.

# Appendix

"I love it when a plan comes together!"

---

*(Colonel John 'Hannibal' Smith, The A-Team)*

# Appendix A.

# *Smart*OS API and Data Type Reference

## Data Type Reference

Data types within the public *Smart*OS kernel API refer to tasks, time, mutexes, events, resources, and interrupt processing. These OS objects can be declared via macros from Section A. The non-public kernel data is not visible to the application code.

As depicted in Figure A.1, the RAM memory is organized by the linker to contain a well structured arrangement for control blocks, and stack areas. Furthermore, special sections for task entry functions, the remote update service, and other information data are reserved within the ROM memory. While this special alignment is hidden from application code, it provides an efficient access to various data by simple pointer arithmetic within the kernel.

***Smart*OS Basic Data Types** — data structure: various (–)

```
typedef uint8_t   TaskId_t;     // task IDs
typedef TaskId_t  Tasklist_t;   // task IDs
typedef uint8_t   Priority_t;   // task priorities

typedef uint64_t  Time_t;       // time in us (64 Bit unsigned)
typedef uint32_t  Delay_t;      // delays in us (32 Bit unsigned)
typedef int64_t   TimeDiff_t;   // time differences in us (32 Bit signed)

typedef int       Mutex_t;      // Mutex data type
```

***Smart*OS Event Control Block (ECB)** — data structure: `Event_t` (1 W)

```
typedef struct {
  char        value;       // 0 for unset, 1 for set
  Tasklist_t  waitqueue;   // head of the priority queue for waiting tasks
} Event_t;
```

**_Smart_OS Resource Control Block (RCB)** data structure: `Resource_t` (6 W)

```
typedef struct {
  Event_t   release_event; // the event connected to this resource
  TaskId_t  owner;         // the owner task (NIL_ID for not allocated)
  uint8_t   flags;         // flags for Dynamic Hinting
  char      *name;         // resource name (for debugging only)
  int (*fInit)(void);      // resource initialization (before scheduler start)
  int (*fGet)(const Time_t *deadline); // called during first allocation
  int (*fRelease)(void);   // called during last deallocation
} Resource_t;
```

**_Smart_OS IRQ Handler Control Block (ICB)** data structure: `IRQHandler_t` (2 W)

```
typedef struct {
  void (*handler)(unsigned int);  // the handler function
  unsigned int argument;          // the argument for the handler
} IRQHandler_t;
```

**_Smart_OS Exceptions** data structure: `EC_t` (2 B + 1 W)

```
jmp_buf                *env;    // the execution context (see setjmp.h)
char                   caught;  // caught indicator (0/1)
volatile unsigned char id;      // the exception identifier
```

**_Smart_OS Task Control Block (TCB)** data structure: `Task_t` (30 W)

```
typedef struct {
  int reg_context[REG_CONTEXT_SIZE];  // register context
  void          (*dhHandler)(void);   // the dynamic hint handler
  Resource_t    *hint;           // the actual hint (NULL for none)
  Resource_t    *awaits;         // pointer to the awaited resource (NULL=none)
  char          *name;           // task name (for debugging only)
  Tasklist_t    *memberlist;     // head of prio. queue this task belongs to
  Priority_t    priority;        // the task's active priority
  Priority_t    priority_base;   // the task's base priority
  TaskId_t      id;              // the task's id
  uint8_t       flags;           // flags for Dynamic Hinting

  unsigned int  *stackarea;      // address of the last word on the stack
  TaskId_t      next_timeout;    // next task in the timeout queue (or NIL_ID)
  TaskId_t      next;            // next task in any priority queue (or NIL_ID)
  Time_t        time_out;        // next timeout of this task
  void          (*entry)(void);  // the task entry function on system start.
  Priority_t    hintPrio;        // receive hints only when blocking tasks
                                 // above this priority threshold φ (Section 6.5.4)
  EC_t          ec;              // the task-specific exception context
} Task_t;
```

**SmartOS Dynamic Kernel Data (RAM)** data structure: various (18 B + 5 W)

```
/* timing related */
Time_t            timeline;            // 8B the current system time
unsigned char     timeout_queue_dirty; // 1B head of timeout queue has changed
Tasklist_t        timeout_queue;       // 1B the head of the timeout queue

/* task related */
Tasklist_t        ready_tasks;      // 1B ID of the first task in ready queue
Task_t            *running_task;    // 1W the currently running task's TCB
TaskId_t          running_task_id;  // 1B the currently running task's ID
uint16_t          deadlockCount;    // 2B total number of detected deadlocks
                                    // since system start

/* misc */
EC_t              *current_ec       // 1W the current exception context
unsigned int      os_RTFlags;       // 1W runtime flags (init./running/mode)
unsigned int      __hwirq_number;   // 1W IRQ number (<200) or syscall address
unsigned int      __hwirq_TS;       // 1W timer counter when entering an IRQ
Delay_t           __hwirq_COMP;     // 4B the IRQ time compensation value.
```

**SmartOS Static Kernel Data (ROM)** data structure: various (4 B)

```
uint8_t const     task_count;      // 1B the total number of declared tasks
uint8_t const     resource_count;  // 1B the total number of declared resources
uint16_t const    os_CTFlags;      // 2B kernel compile time flags
```

**SmartOS OS Objects Control Blocks** data structure: various (–)

```
Task_t        tasks[];                // RAM: task_count * sizeof(Task_t)
                                      //       the array of TCBs
Resource_t    resources[];            // RAM: resource_count * sizeof(Resource_t)
                                      //      the array of RCBs
IRQHandler_t  __os_irq_table[];       // ROM: |HardIRQs| * sizeof(IRQHandler_t)
                                      //     the hardware IRQ table
IRQHandler_t  __os_softirq_table[];   // ROM: |SoftIRQs| * sizeof(IRQHandler_t)
                                      //     the software IRQ table
```

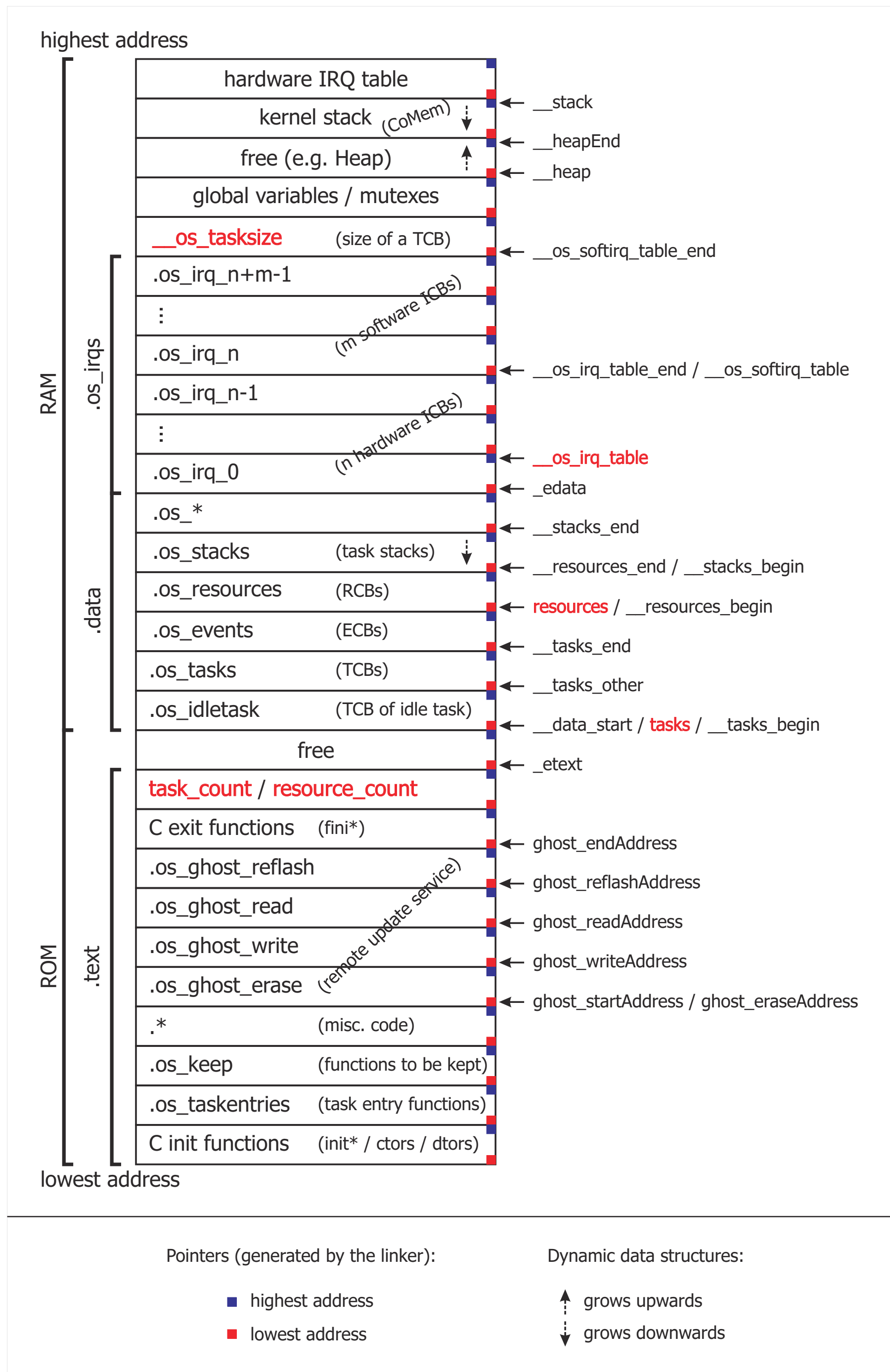



**Figure A.1.:** The *Smart*OS memory layout for the MSP430F1611 MCU

## API Function Reference

Function names within the *Smart*OS kernel API should be self-explaining in the context of their use throughout this work. However, we use the following markers: (D) for macros which declare OS objects, (I) for macros which import OS objects, and (S) for functions which will finally invoke a syscall. For these functions, the call graphs and the partially automatic parameterization is shown in Figure A.2. The subsequent syscall execution flow is depicted in Figure 4.5[p66].

**_Smart_OS Kernel** — data structure: – (40 B)

**Functions:**

```
void os_run(void);
char os_isRunning(void);
char os_isInitialized(void);
unsigned int os_inKernelMode(void);
unsigned int os_inTaskMode(void);
uint16_t os_getCompileTimeFlags(void);
```

**_Smart_OS Time** — data structure: `Time_t` (8 W)

**Functions:**

```
void getCurrentTime(Time_t *t);
void getIRQTime(Time_t *t);
void os_setIRQTimeCompensation(Delay_t comp);
```

**_Smart_OS Interrupts** — data structure: `IRQHandler_t` (2 W)

**Declaration:**

```
OS_DECLARE_IRQ_HANDLER(int irqno, *handler, int argument);(D)
```

**_Smart_OS Tasks and Priorities** — data structure: `Task_t` (30 W)

**Declaration:**

```
OS_DECLARE_TASK(name, int stackSize, Priority_t priority);(D)
OS_DECLARE_TASK_DH(name, hHandler, int stacksize, Priority_t priority);(D)
OS_DECLARE_USER_IDLE_TASK(entry, int stacksize);(D)
```

**Functions:**

```
TaskId_t os_getMyTaskID();
unsigned int os_check_stack(Task_t *task);
unsigned int os_check_my_stack(void);
void yield(void); (S)void setBasePriority(Task_t *task, Priority_t
priority);(S)
void setMyBasePriority(Priority_t priority);
Priority_t getBasePriority(Task_t *task);
Priority_t getMyBasePriority();
Priority_t getMyCurrentPriority();
```

**_Smart_OS Mutexes** data structure: `Mutex_t` (1 W)

**Declaration:**

```
OS_DECLARE_MUTEX(name);(D)
OS_DECLARE_SET_MUTEX(name);(D)
OS_IMPORT_MUTEX(name);(I)
```

**Functions:**

```
int CAS(Mutex_t *addr, int expVal, int newVal);(S)
int TAS(Mutex_t *addr);
int TAC(Mutex_t *addr);
int TST(Mutex_t *addr);
void CLR(Mutex_t *addr);
```

**_Smart_OS Events** data structure: `Event_t` (1 W)

**Declaration:**

```
OS_DECLARE_EVENT(name);(D)
OS_DECLARE_ACTIVE_EVENT(name);(D)
OS_IMPORT_EVENT(name);(I)
```

**Functions:**

```
void setEvent(Event_t *event);(S)
int waitEventFor(Event_t *event, Delay_t timeout);
int waitEventUntil(Event_t *event, Time_t *deadline);(S)
int clearEvent(Event_t *event);(S)
int waitEvent(Event_t *event);(S)
int sleep(Delay_t timeout);(S)
int sleepUntil(Time_t *deadline);(S)
int checkEvent(Event_t* event);
```

**_Smart_OS Resources** data structure: `Resource_t` (6 W)

**Declaration:**

```
OS_DECLARE_RESOURCE(name);(D)
OS_DECLARE_RESOURCE_EXT(name, f *fInit, f *fGet, f *fRelease);(D)
OS_IMPORT_RESOURCE(name);(I)
```

**Functions:**

```
void releaseResource(Resource_t *resource);(S)
int getResourceUntil(Resource_t *resource, Time_t *deadline);(S)
int getResourceFor(Resource_t *resource, Delay_t timeout);
int getResource(Resource_t *resource);
int testResource(Resource_t *resource);
int isResourceOwner(TaskId_t taskID, Resource_t *resource);

see Section 6.5.4 for DynamicHinting related functions
```

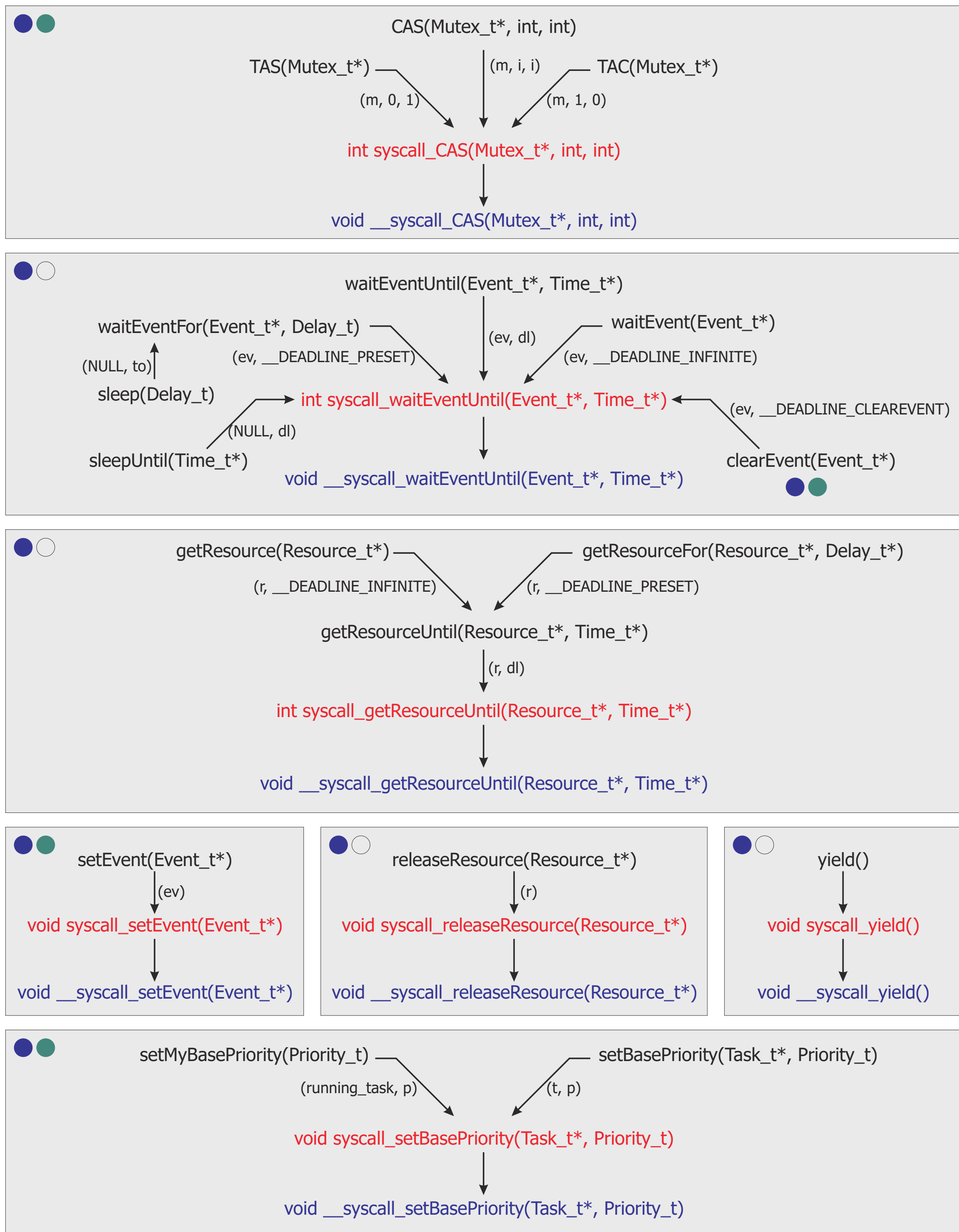



**Figure A.2.:** *Smart*OS kernel functions and syscalls (self-suspending functions support an additional parameter $\varphi$ for DynamicHinting as described in Section 6.5.4.)

# Appendix B.

# SNoW$^5$ Schematics and Layout

This appendix contains the circuit schematics (Figures B.1 – B.3) and PCB layouts (Figure B.4) for the SNoW$^5$ sensor node as described in Section 2.2. Since SNoW$^5$ was designed for usability and flexibility in research and education, its design propagates a trade-off between size (85 mm × 50 mm) and easy expandability through a complete set of connectors. We'll just give a short overview here. See Chapter 2.2 for some design considerations and [31, 34] for a detailed technical description of the SNoW$^5$ platform.

**Connectors** are grouped in I/O ports and aligned in a 2.54 mm raster as commonly specified for standard breadboards in rapid prototyping. Since the ports provide access to each I/O pin of all mounted devices, their observation by measurement equipment (e.g. oscilloscopes or logic analyzers) is simplified.

**Power** can be either directly supplied by any 1.8 – 3.3 V source or by any 4 – 20 V source via a reverse voltage protected DC/DC regulator. The direct and regulated voltage is also forwarded to the header ports via `DC_T` and `VCC`, respectively. Three connectors (`NC1, NC2, NC3`) are freely available for propagating additional signals or voltages across expansion boards. To save energy, e.g. when battery powered, the power indicator LED is only active in combination with the DC/DC regulator and the RS232 level converter can be switched off manually.

**Programming and debugging** the device is done via the service port which offers access to the MCU's JTAG (IEEE1149.1) and BSL interface, while, at the same time, it can be used for direct power supply. Apart from the hardware supported programming, there is also an option for *Smart*OS based over-the-air software updates by using the SNoW Ghost remote maintenance system [20] as described in Section 8.2.

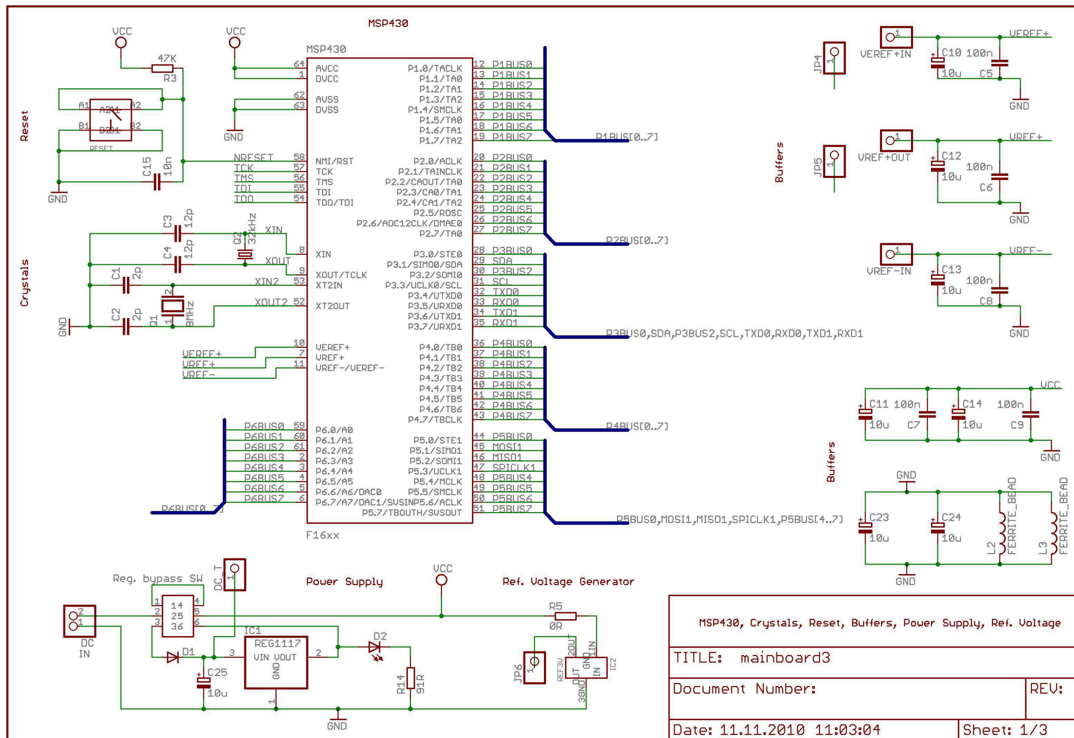


**Figure B.1.:** SNoW$^5$ schematics: MCU and power supply

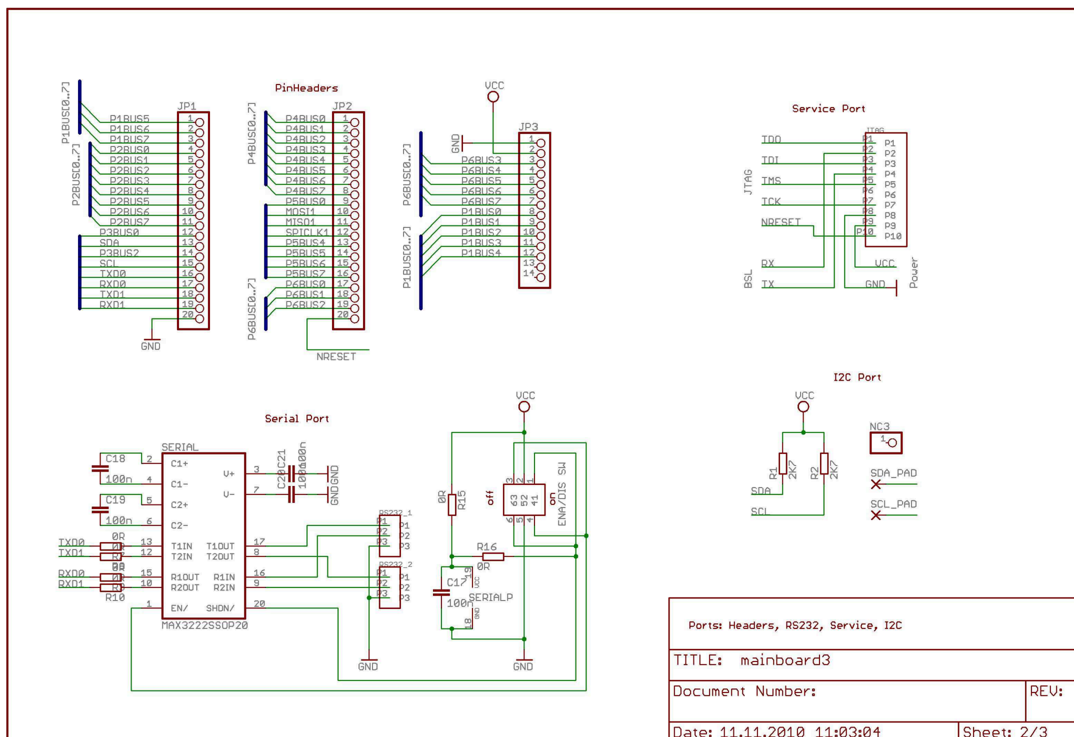


**Figure B.2.:** SNoW$^5$ schematics: I/O header ports

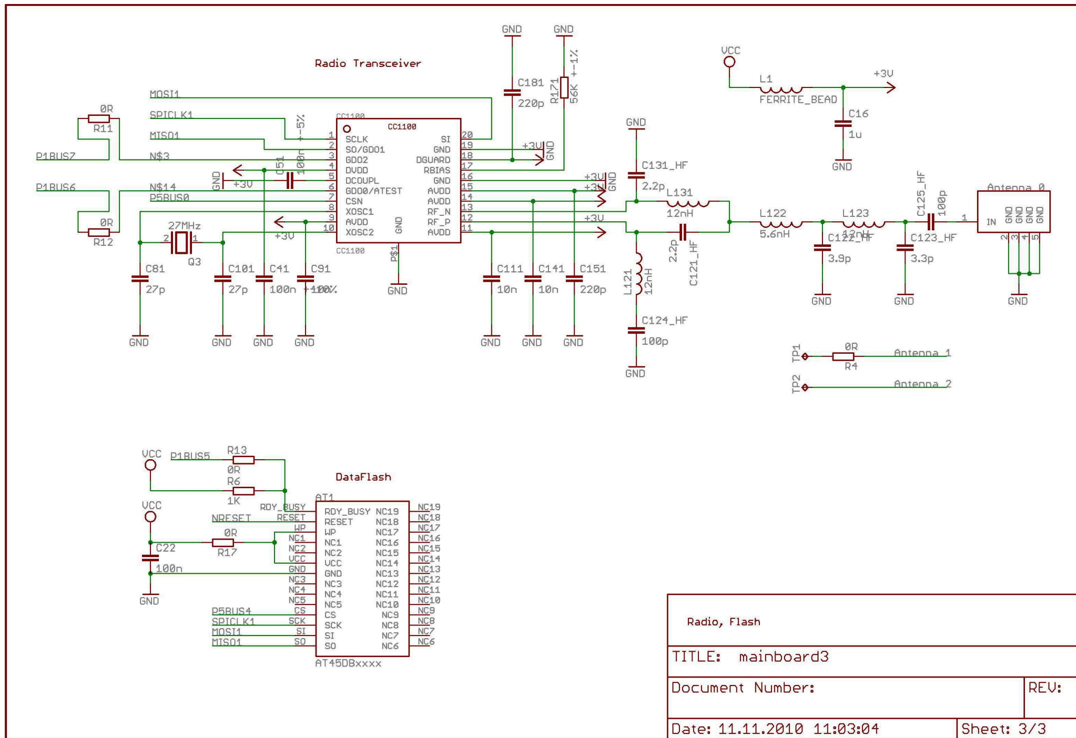


**Figure B.3.:** SNoW[5] schematics: Radio transceiver and data flash

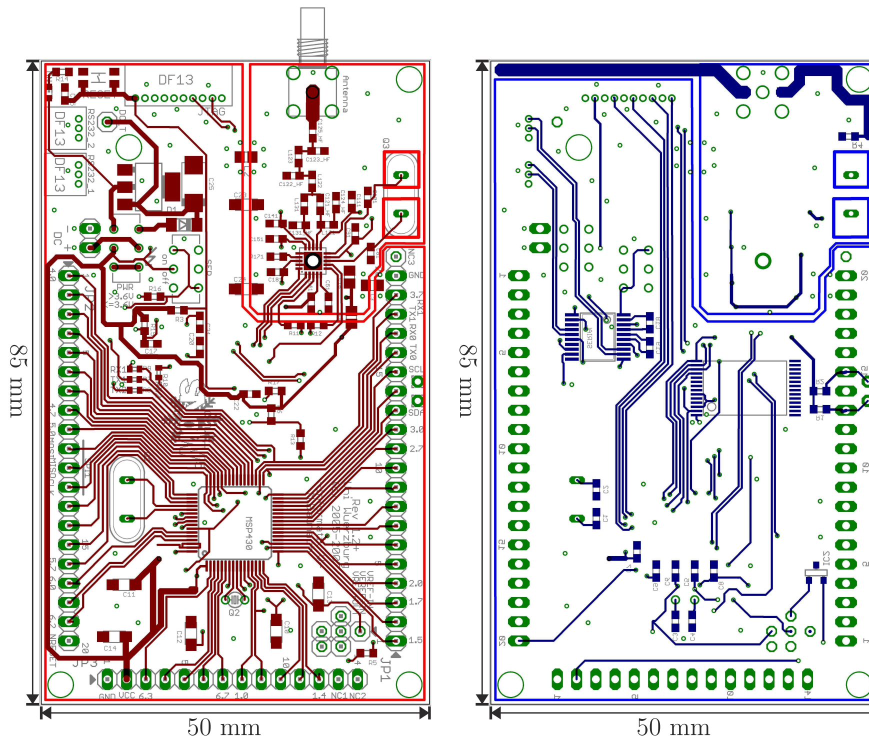


**Figure B.4.:** The SNoW[5] layout: Top (left) and bottom (right) side at original size

# Part VI.

# Lists and Indexes

«Je ne cherche pas. Je trouve.»

---

*(Pablo Picasso,*
*Spanish painter)*

# Bibliography

This bibliography avoids URLs whenever possible. For those citations where links could not be avoided, their availability has been checked on June 16, 2012.


[1] Tarek F. Abdelzaher, Leonidas J. Guibas, and Matt Welsh, editors. **6th International Conference on Information Processing in Sensor Networks (IPSN 2007)**, Cambridge, MA, USA, 2007. ACM.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. **MANTIS: System Support for MultimodAl NeTworks of In-Situ Sensors**. In *2nd ACM international conference on Wireless sensor networks and applications (WSNA 2003)*, pages 50–59, New York, NY, USA, 2003. ACM. ISBN 1-58113-764-8.

[3] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. **Generic Virtual Memory Management for Operating System Kernels**. In *Symposium in Operating System Principles*, pages 123–136, Litchfield Park AZ (USA), December 1989. ACM.

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. **Wireless Sensor Networks: A Survey**. *Computer Networks*, 38(4):393–422, March 2002.

[5] Michael Allen, Sebnem Baydere, Gurhan Kucuk, and Elena Gaura. **Evaluation of Localization Algorithms**. In *Localization Algorithms and Strategies for Wireless Sensor Networks* Mao and Fidan [196], pages 1–23. ISBN 9781605663968.

[6] Gene M. Amdahl. **Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities**. In *AFIPS '67: Spring joint computer conference*, pages 483–485, New York, NY, USA, April 1967. ACM.

[7] Waleed Ammar, Ahmed ElDawy, and Moustafa Youssef. **Secure Localization in Wireless Sensor Networks: A Survey**. *CoRR*, abs/1004.3164, 2010.

[8] Isaac Amundson and Xenofon D. Koutsoukos. **A Survey on Localization for Mobile Wireless Sensor Networks**. In *2nd international conference on Mobile entity localization and tracking in GPS-less environments (MELT 2009)*, pages 235–254, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Hsu Kahn And, V. S. Hsu, J. M. Kahn, and K. S. J. Pister. **Wireless Communications for Smart Dust**. In *Electronics Research Laboratory Technical Memorandum M98/2*, 1998.

[10] Christian Appold. **Using State Symmetries to Speed up Symmetry Reduction in Model Checking**. In *9th International Workshop on Symmetry and Constraint Satisfaction Problems (SymCon 2009)*, September 2009.

[11] Christian Appold. **Reliable Model Checking for WSNs**. In *8. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* Tec [277].

[12] Christian Appold, Marcel Baunach, and Clemens Mühlberger. **The HaDes X CPU**. Technical report, Chair of Computer Science, University of Würzburg, 2010.

[13] **16-megabit 2.5-volt Only or 2.7-volt Only DataFlash AT45DB161B**. Atmel Corp., San Jose (USA), October 2004. URL `http://www.atmel.com/dyn/resources/prod_documents/doc2224.pdf`.

[14] Martin Atzmueller, Dominik Benz, Stephan Doerfel, Andreas Hotho, Robert Jäschke, Bjoern Elmar Macek, Folke Mitzlaff, Christoph Scholz, and Gerd Stumme. **Enhancing Social Interactions at Conferences**. *it - Information Technology*, 53(3):101–107, May 2011. ISSN 16112776. doi: 10.1524/itit.2011.0631.

[15] N. Audsley, R. Gao, A. Patil, and P. Usher. **Efficient OS Resource Management for Distributed Embedded Real-Time Systems**. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[16] Various Authors. **10 Emerging Technologies That Will Change the World**. *MIT Technology Review*, February 2003.

[17] Susmit Bagchi. **Nano-kKernel: A Dynamically Reconfigurable Kernel for WSN**. In *1st International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (MOBILWARE 2008)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[18] Paramvir Bahl and Venkata N. Padmanabhan. **RADAR: An In-Building RF-Based User Location and Tracking System**. In *19th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, pages 775–784, 2000.

[19] Theodore P. Baker. **A Stack-Based Resource Allocation Policy for Realtime Processes**. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[20] Marcel Baunach. **Ghost: Software and Configuration Distribution for Wireless Sensor/Actor Networks**. In Ritter et al. [246], pages 81–84. Technical Report B 08-12.

[21] Marcel Baunach. **Speed, Reliability and Energy Efficiency of HashSlot Communication in WSN Based Localization Systems**. In Verdone [299], pages 74–89. ISBN 978-3-540-77689-5.

[22] Marcel Baunach. **Dynamic Hinting: Real-Time Resource Management in Wireless Sensor/Actor Networks**. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009)*, pages 31–40. IEEE Computer Society, 2009. ISBN 978-0-7695-3787-0.

[23] Marcel Baunach. **Priority aware Resource Management for Real-Time Operation in Wireless Sensor/Actor Networks**. In *8. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* Tec [277].

[24] Marcel Baunach. **Collaborative Memory Management for Time-Critical Sensor/Actor Systems**. In Reiner Kolla, Marcel Baunach, Clemens Mühlberger, and Christian Appold, editors, *9. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, Wuerzburg, Germany, September 2010. Julius-Maximilians-University, Institute of Computer Science.

[25] Marcel Baunach. **CoMem: Cooperative Memory Management for Real-Time**

**Operation within Reactive Sensor/Actor Networks**. In *18th International Conference on Real-Time and Network Systems (RTNS 2010)*, November 2010.

[26] Marcel Baunach. **Collaborative Memory Management for Reactive Sensor/Actor Systems**. In *5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2010)*, Denver, Colorado, USA, October 2010.

[27] Marcel Baunach. **pVoted: A Progressive On-Line Algorithm for Robust Real-Time Localization and Tracking in spite of Faulty Distance Information**. In *1st International Conference on Indoor Positioning and Indoor Navigation (IPIN 2010)* ETH [96].

[28] Marcel Baunach. **Dynamic Hinting: Collaborative Real-Time Resource Management for Reactive Embedded Systems**. *Journal of Systems Architecture (JSA)*, 57:799–814, October 2011. ISSN 1383-7621. doi: 10.1016/j.sysarc.2011.07.001. URL `http://www.sciencedirect.com/science/article/pii/S1383762111000890`.

[29] Marcel Baunach. **CoMem: Collaborative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks**. *Real-Time Systems*, 48:75–100, August 2012. ISSN 0922-6443. doi: 10.1007/s11241-011-9136-7. URL `http://dx.doi.org/10.1007/s11241-011-9136-7`.

[30] Marcel Baunach. **Towards Collaborative Resource Sharing under Real-Time Conditions in Multitasking and Multicore Environments**. In *17th IEEE International Conference on Emerging Technology & Factory Automation (ETFA 2012)*. IEEE Computer Society, September 2012. to be published.

[31] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. **SNoW$^5$: A Versatile Ultra Low Power Modular Node for Wireless Ad Hoc Sensor Networking**. In Pedro José Marrón, editor, *5. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, pages 55–59, Stuttgart, July 2006. Institut für Parallele und Verteilte Systeme.

[32] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. **Beyond Theory: Development of a Real World Localization Application as Low Power WSN**. In *32nd IEEE Conference on Local Computer Networks (LCN 2007)*, pages 872–884, Washington, DC, USA, October 2007. IEEE Computer Society. ISBN 0-7695-3000-1.

[33] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. **A Method for Self-Organizing Communication in WSN Based Localization Systems: HashSlot**. In *32nd IEEE Conference on Local Computer Networks (LCN 2007)*, pages 825–832, Washington, DC, USA, October 2007. IEEE Computer Society. ISBN 0-7695-3000-1. SenseApp 2007, Dublin (Ireland).

[34] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. **SNoW$^5$: A Modular Platform for Sophisticated Real-Time Wireless Sensor Networking**. Technical Report 399, Institute for Computer Science, University of Würzburg, January 2007.

[35] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. **SNoW Bat: A high precise WSN based Location System**. Technical Report 424, Institute for Computer Science, University of Würzburg, May 2007.

[36] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. **Introduction to a Small Modular Adept Real-Time Operating System**. In *6. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*. RWTH Aachen University, July 2007.

[37] Marcel Baunach, Clemens Mühlberger, and Christian Appold. **Enabling Real-Time in WSN Applications**. In Dirk Pesch and Sajal Das, editors, *6th European Conference on Wireless Sensor Networks (EWSN 2009)*, pages 37–38, Cork, February 2009. Department of Computer Science, University College Cork.

[38] Marcel Baunach, Clemens Mühlberger, Christian Appold, Martin Schröder, and Florian Füller. **Analysis of Radio Signal Parameters for Calibrating RSSI Localization Systems**. Technical Report 455, Institute for Computer Science, University of Würzburg, March 2009.

[39] Ramon Bauza, Javier Gozalvez, and Joaquin Sanchez-Soriano. **Road Traffic Congestion Detection Through Cooperative Vehicle-to-Vehicle Communications**. In *5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2010)*, Denver, Colorado, USA, October 2010.

[40] Carter Bays. **A Comparison of Next-Fit, First-Fit, and Best-Fit**. *Commun. ACM*, 20(3):191–192, 1977. ISSN 0001-0782.

[41] Martin Becker. **Ankunftszeitpunktbestimmung von Ultraschallsignalen**. Practical work, University of Würzburg, Würzburg, July 2009.

[42] M. Ben-Ari. **Principles of Concurrent and Distributed Programming**. Addison-Wesley, 2nd edition, 2006. ISBN 9780321312839.

[43] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. **Hoard: A Scalable Memory Allocator for Multithreaded Applications**. Technical report, University of Texas at Austin, Austin, TX, USA, 2000.

[44] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. **Reconsidering Custom Memory Allocation**. In *17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2002)*, pages 1–12, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1.

[45] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. **Prototyping Wireless Sensor Network Applications with BTnodes**. In *1st European Workshop on Sensor Networks (EWSN 2004)*, volume 2920 of *LNCS*, pages 323–338. Springer, Berlin, January 2004.

[46] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. **MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms**. *Mobile Networks & Applications (MONET)*, 10(4):563–579, 2005. ISSN 1383-469X.

[47] D. Bimschas, H. Hasemann, M. Hauswirth, M. Karnstedt, O. Kleine, A. Kröller, M. Leggieri, R. Mietz, A. Passant, D. Pfisterer, K. Römer, and C. Truong. **Semantic-Service Provisioning for the Internet of Things**. In *Workshop on Semantic Services for the Internet of Things (SSIT 2011)*, 2011.

[48] Pratik Biswas, Tzu chen Liang, Kim chuan Toh, Ta chung Wang, and Yinyu Ye. **Semidefinite Programming Approaches for Sensor Network Localization with Noisy Distance Measurements**. *IEEE Transactions on Automation Science and Engineering*, 3:2006, 2006.

[49] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. **Handbook on Scheduling: Models and Methods for Advanced Planning**. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540280464.

[50] Luís M. Borges, Fernando J. Velez, and António S. Lebres. **Taxonomy for Wireless Sensor Networks Services Characterisation and Classification**. In *7th Conference on Telecomunications (ConfTele 2009)*. IEEE Computer Society, May 2009.

[51] Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. **Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?** In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE Computer Society, April 2008. ISBN 978-0-7695-3146-5.

[52] Jürgen Bregenzer. **SmartOS versus TinyOS – Ein Vergleich zweier Betriebssysteme für Sensorknoten**. Diploma thesis, University of Würzburg, Würzburg, December 2007.

[53] Cyril Brignone, Tim Connors, Geoff Lyon, and Salil Pradhan. **SmartLOCUS: An Autonomous, Self-Assembling Sensor Network for Indoor Asset and Systems Management**. Technical report, Mobile and Media Systems Laboratory, HP Laboratories Palo Alto, June 2005.

[54] Paul Brna. **Models of Collaboration**. Web site `http://homepages.inf.ed.ac.uk/pbrna/papers/bcs98paper/bcs98.html`, 1998.

[55] Nirupama Bulusu. **Introduction to Wireless Sensor Networks**. In Bulusu and Jha [57], chapter 1, pages 1–19.

[56] Nirupama Bulusu. **Localization**. In Bulusu and Jha [57], chapter 4, pages 45–57.

[57] Nirupama Bulusu and Sanjay Jha, editors. **Wireless Sensor Networks - A Systems Perspective**. Artech House, 2005.

[58] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. **Dozer: Ultra-Low Power Data Gathering in Sensor Networks**. In Abdelzaher et al. [1], pages 450–459.

[59] M. Burton and P. Brna. **Clarissa: An Exploration of Collaboration through Agent-Based Dialogue Games**. In P. Brna, A. Paiva, and J.A. Self, editors, *European Conference on Artificial Intelligence in Education*, pages 393–400. Edicões Colibri, Lisbon, 1996.

[60] Giorgio C. Buttazzo. **Real-Time Scheduling and Resource Management**. In Leep et al. [172]. ISBN 1584886781.

[61] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. **The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks**. In *7th international conference on Information processing in sensor networks (IPSN 2008)*, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3157-1.

[62] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. **RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks**. In *6th international conference on Information processing in sensor networks (IPSN 2007)*, pages 148–157. ACM, 2007. ISBN 978-1-59593-638-X.

[63] Albert M. K. Cheng and James Ras. **The Implementation of the Priority Ceiling Protocol in Ada-2005**. *Ada Lett.*, XXVII (1):24–39, 2007. ISSN 1094-3641.

[64] Chih chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. **SOS: A dynamic operating system for sensor networks**. In *3rd International Conference on Mobile Systems, Applications, And Services (MobiSys 2005)*. ACM Press, 2005.

[65] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. **Analysing Memory Resource Bounds for Low-Level Programs**. In *7th international symposium on Memory management (ISMM 2008)*, pages 151–160, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7.

[66] Octav Chipara, Chenyang Lu, and Thomas C. Bailey Gruia-Catalin Roman. **Reliable Clinical Monitoring using Wireless Sensor Networks: Experiences in a Step-Down Hospital Unit**. In *8th ACM Conference on Embedded Networked Sensor Systems (SenSys 2010)*, pages 155–168, New York, NY, USA, 2010. ACM.

[67] **CC1000 Single Chip Very Low Power RF Transceiver**. Chipcon AS, Oslo (Norway), 2005. URL `http://www.ti.com/lit/gpn/cc1000`.

[68] Chee-Yee Chong and S.P. Kumar. **Sensor Networks: Evolution, Opportunities, and Challenges**. *Proceedings of the IEEE*, 91(8):1247 – 1256, August 2003.

[69] Rui Chu, Lin Gu, Yunhao Liu, Mo Li, and Xicheng Lu. **Versatile Stack Management for Multitasking Sensor Networks**. In *30th International Conference on Distributed Computing Systems (ICDCS 2010)*, pages 388–397, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4059-7.

[70] E. G. Coffman, M. Elphick, and A. Shoshani. **System Deadlocks**. *ACM Computing Surveys*, Vol. 3:67–78, 1971. ISSN 0360-0300.

[71] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. **Dependent Types for Low-Level Programming**. In *In European Symposium on Programming*, 2007.

[72] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. **Efficient Memory Safety for TinyOS**. In *5th International Conference on Embedded Net-*

*worked Sensor Systems (SenSys 2007)*, 2007.

[73] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri, editors. **Scheduling in Real-Time Systems**. Wiley, 2002. ISBN 978-0-470-84766-4.

[74] **MICA2 Wireless Measurement System**. Crossbow Technology Inc., San Jose (USA), 2005.

[75] R. Davis, F. Foote, J. Anderson, and E. Mikhail. **Surveying, Theory and Practice**. McGraw-Hill, 6th edition, 1981.

[76] Pradip De, Yonghe Liu, and Sajal K. Das. **Energy-Efficient Reprogramming of a Swarm of Mobile Sensors**. *IEEE Trans. Mob. Comput.*, 9(5):703–718, 2010.

[77] Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. **DESYNC: Self-Organizing Desynchronization and TDMA on Wireless Sensor Networks**. In Abdelzaher et al. [1], pages 11–20.

[78] Ashay Dhamdhere, Hao Chen, Alexander Kurusingal, Vijay Sivaraman, and Alison Burdett. **Experiments with Wireless Sensor Networks for Real-Time Athlete Monitoring**. In *5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2010)*, Denver, Colorado, USA, October 2010.

[79] Edsger W. Dijkstra. **The mathematics behind the Banker's Algorithm**. In *Selected Writings on Computing: A Personal Perspective*, pages 308–312. Springer-Verlag, 1982.

[80] Eric Dilger and Marcel Baunach. **Quasi-Präemptive Ressourcenverwaltung in Hardware**. Technical report, Institute for Computer Science, University of Würzburg, 2011.

[81] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. **Heap Data Allocation to Scratch-Pad Memory in Embedded Systems**. *Journal of Embedded Computing*, 1:2005, 2005.

[82] Falko Dressler. **Self-Organization in Sensor and Actor Networks**. John Wiley & Sons, December 2007. ISBN 978-0-470-02820-9.

[83] Cormac Duffy, Utz Roedig, John Herbert, and Cormac J. Sreenan. **Adding Preemption to TinyOS**. In *4th Workshop on Embedded Networked Sensors (EmNets 2007)*, pages 88–92, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-694-3.

[84] Adam Dunkels. **Programming Memory-Constrained Networked Embedded Systems**. Phd thesis, Swedish Institute of Computer Science (SICS), Stockholm, Sweden, February 2007.

[85] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. **Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors**. In *29th Annual IEEE International Conference on Local Computer Networks (LCN 2004)*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2260-2.

[86] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. **Using Protothreads for Sensor Node Programming**. In *Workshop on RealWorld Wireless Sensor Networks (REALWSN 2005)*, 2005.

[87] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. **Protothreads:**

**Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems**. In *4th International Conference on Embedded Networked Sensor Systems (SenSys 2006)*, pages 29–42, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3.

[88] A. K. Dwivedi and O. P. Vyas. **Network Layer Protocols for Wireless Sensor Networks: Existing Classifications and Design Challenges**. *International Journal of Computer Applications*, 8(12): 30–34, October 2010.

[89] A. K. Dwivedi, M. K. Tiwari, and O. P. Vyas. **Operating Systems for Tiny Networked Sensors - A Survey**. *International Journal of Recent Trends in Engineering*, 1(2): 152–157, May 2009.

[90] Arvind Easwaran and Björn Andersson. **Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling**. In Theodore P. Baker, editor, *IEEE Real-Time Systems Symposium*, pages 377–386. IEEE Computer Society, 2009. ISBN 978-0-7695-3875-4.

[91] Juergen Eckert, Falko Dressler, and Reinhard German. **An Indoor Localization Framework for Four-rotor Flying Robots Using Low-power Sensor Nodes**. Technical report, University of Erlangen, Department of Computer Science 7, February 2009.

[92] Juergen Eckert, Falko Dressler, and Reinhard German. **Real-time Indoor Localization Support for Four-rotor Flying Robots using Sensor Nodes**. In *IEEE International Workshop on Robotic and Sensors Environments (ROSE 2009)*, pages 23–28, Lecco, Italy, November 2009. IEEE.

[93] Andreas Engel. **SNoW-RA: Reap and Allot – Energieneutraler Betrieb des SNoW5-Sensorknotens durch Gewinnen und Verteilen von Solarenergie**. Diploma thesis, University of Würzburg, Würzburg, July 2009.

[94] D. R. Engler, M. F. Kaashoek, and J. O'Toole. **Exokernel: An Operating System Architecture for Application-Level Resource Management**. In *15th ACM symposium on Operating systems principles (SOSP 1995)*, pages 251–266, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4.

[95] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. **Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks**. In *26th IEEE International Real-Time Systems Symposium (RTSS 2005)*, pages 256–265, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7.

[96] **1st International Conference on Indoor Positioning and Indoor Navigation (IPIN 2010)**, September 2010. ETH Zurich.

[97] Patrick Fakesch. **Methoden zur fehlertoleranten Datenhaltung in Sensornetzen**. Diploma thesis, University of Würzburg, Würzburg, August 2010.

[98] Muhammad Omer Farooq, Sadia Aziz, and Abdul Basit Dogar. **State of the Art in Wireless Sensor Networks Operating Systems: A Survey**. In Tai-Hoon Kim, Young-Hoon Lee, Byeong Ho Kang, and Dominik Slezak, editors, *FGIT*, volume 6485 of *LNCS*, pages 616–631. Springer, December 2010. ISBN 978-3-642-17568-8.

[99] Jessica Feng, Farinaz Koushanfar, and Miodrag Potkonjak. **Sensor Network Architecture**. In Ilyas and Mahgoub [134], chapter 12, pages 1–19.

[100] Nathan Fisher, James H. Anderson, and Sanjoy Baruah. **Task Partitioning upon Memory-Constrained Multiprocessors**. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005)*, pages 416–421, Washington, DC, USA, 2005. IEEE Computer Society.

[101] Patrik Floréen, Petteri Kaski, Jukka Kohonen, and Pekka Orponen. **Exact and Approximate Balanced Data Gathering in Energy-Constrained Sensor Networks**. *Theoretical Computer Science*, 344(1):30–46, November 2005. ISSN 0304-3975.

[102] Johan Fornaeus. **Device Hypervisors**. In *47th Design Automation Conference (DAC 2010)*, pages 114–119, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5. doi: 10.1145/1837274.1837305. URL `http://doi.acm.org/10.1145/1837274.1837305`.

[103] Keir Fraser and Tim Harris. **Concurrent Programming without Locks**. *ACM Trans. Comput. Syst.*, 25(2):5, 2007. ISSN 0734-2071.

[104] Free Software Foundation. **The GNU C Library**. Web site `http://www.gnu.org/s/libc/`, 2011.

[105] FU Berlin. **ScatterWeb - ESB**, 2011. URL `http://cst.mi.fu-berlin.de/projects/ScatterWeb/`.

[106] Yasuhiro Fukuju, Masateru Minami, Hiroyuki Morikawa, and Tomonori Aoyama. **DOLPHIN: An Autonomous Indoor Positioning System in Ubiquitous Computing Environment**. In *IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES 2003)*, page 53, Washington, DC, USA, 2003. IEEE Computer Society.

[107] Yasuhiro Fukuju, Masateru Minami, Kazuki Hirasawa, Shigeaki Yokoyama, Moriyuki Mizumachi, Hiroyuki Morikawa, and Tomonori Aoyama. **DOLPHIN: A Practical Approach for Implementing a Fully Distributed Indoor Ultrasonic Positioning System**. In *UbiComp 2004: Ubiquitous Computing (LNCS 3205)*, pages 347–365, Berlin, 2004. Springer.

[108] Elena Gaura, Michael Allen, Lewis Girod, James Brusey, and Geoffrey Werner-Challen, editors. **Wireless Sensor Networks: Deployments and Design Frameworks**. Springer Press, 2010. ISBN 1441958339.

[109] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. **The nesC Language: A Holistic Approach to Networked Embedded Systems**. In *ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI 2003)*, pages 1–11, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5.

[110] Eduard Glatz. **Betriebssysteme – Grundlagen, Konzepte, Systemprogrammierung**. dpunkt Verlag, 2nd edition, 2010. ISBN 978-3-89864-678-9.

[111] Wolfram Gloger. **The ptmalloc homepage**. Web `http://www.malloc.de`, 2010.

[112] James Gosling, Bill Joy, and Guy L. Jr. Steele. **The Java Language Specification**. Addison-Wesley, 3rd edition, 2005. ISBN 0321246780.

[113] Mohammed G. Gouda, Yi-Wu Han, E. Douglas Jensen, Wesley D. Johnson, and Richard Y. Kain. **Radar Scheduling: Section 1, The Scheduling Problem**. *Distributed Data Processing Technology, Chapter 3*, IV, 1977.

[114] H. P. Grice. **Logic and Conversation**. In P. Cole and J. L. Morgan, editors, *Syntax and Semantics: Vol. 3: Speech Acts*, pages 41–58, San Diego, CA, 1975. Academic Press.

[115] Tom Gross and Ljlwdo Qirupdwlrq (qylurqphqwv. **Supporting Collaboration and Cooperation in digital Information Environment**, 1998.

[116] Lin Gu and John A. Stankovic. **t-kernel: A Translative OS Kernel for Wireless Sensor Networks**. Technical Report UVA CS TR CS-2005-09, University of Virginia, June 2005.

[117] Lin Gu and John A. Stankovic. **t-kernel: Providing Reliable OS Support to Wireless Sensor Networks**. In *4th international conference on Embedded networked sensor systems (SenSys 2006)*, pages 1–14, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3.

[118] Simon Gyula, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. **Sensor Network-Based Countersniper System**. In *2nd International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 1–12, New York, NY, USA, November 2004. ACM Press.

[119] G. P. Halkes and K. G. Langendoen. **Crankshaft: An Energy-Efficient MAC-Protocol for Dense Wireless Sensor Networks**. In Koen Langendoen and Thiemo Voigt, editors, *4th European Conference on Wireless Sensor Networks (EWSN 2007)*, volume 4373 of *LNCS*. Springer, January 2007. ISBN 3-540-69829-9.

[120] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. **A Dynamic Operating System for Sensor Nodes**. In *3rd international conference on Mobile Systems, applications, and services (MobiSys 2005)*, pages 163–176, New York, NY, USA, 2005. ACM. ISBN 1-931971-31-5.

[121] Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava. **Sensor Network Software Update Management: A Survey**. *Journal of Network Management*, 15:283–294, 2005.

[122] Jer Hayes, Stephen Beirne, King-Tong Lau, and Dermot Diamond. **Evaluation of a Low Cost Wireless Chemical Sensor Network for Environmental Monitoring**. In *IEEE Sensors 2008*. IEEE Computer Society, October 2008.

[123] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. **Energy-Efficient Communication Protocol for Wireless Microsensor Networks**. In *33rd Hawaii International Conference on System Sciences (HICSS 2000)*, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0493-0.

[124] Gernot Heiser. **The Role of Virtualization in Embedded Systems**. In *1st Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, April 2008.

[125] **Fundamentals Of Quartz Oscillators**. Hewlett Packard, May 1997.

[126] Jeffrey Hightower and Gaetano Borriello. **A Survey and Taxonomy of Location Systems for Ubiquitous Computing**. Technical Report UW-CSE 01-08-03, IEEE Computer, August 2001.

[127] David Hilbert and S. Cohn-Vossen. **Geometry and the Imagination**. American Mathematical Society, October 1999. ISBN 0821819984.

[128] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. **System Architecture Directions for Networked Sensors**. *SIGPLAN Not.*, 35(11):93–104, 2000. ISSN 0362-1340.

[129] Jason Lester Hill. **System Architecture for Wireless Sensor Networks**. Phd thesis, University of California, Berkeley, 2003. Adviser-Culler, David E.

[130] Dominic Hillenbrand. **A Flexible Design Space Exploration Platform for Wireless Sensor Networks**. Phd thesis, Fakultät für Informatik - Karlsruher Institut für Technologie KIT, 2010.

[131] Jonathan W. Hui and David E. Culler. **The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale**. In John A. Stankovic, Anish Arora, and Ramesh Govindan, editors, *SenSys*, pages 81–94. ACM, November 2004. ISBN 1-58113-879-2.

[132] M. N. Huxley. **Corrigenda: Exponential Sums and Lattice Points II**. *Proc. London Math. Soc.*, s3-68(2):264–, 1994.

[133] **1003.1-2008 Portable Operating System Interface (POSIX) Base Specifications, Issue 7**. IEEE, December 2008. URL `http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm`.

[134] Mohammad Ilyas and Imad Mahgoub, editors. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. CRC Press, 2005.

[135] National Imagery and Mapping Agency (NIMA). **Department of Defense World Geodetic System 1984**. Technical Report 8350.2, 3rd edition, National Imagery and Mapping Agency (NIMA), January 2000.

[136] Information Sciences Institute, University of Southern California. **The Network Simulator - ns-2**. Web site `http://isi.edu/nsnam/ns/`, 2011.

[137] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. **Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks**. In *6th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom 2000)*, pages 56–67. ACM, 2000.

[138] IPIN2011. **2nd International Conference on Indoor Positioning and Indoor Navigation (IPIN 2011)**, September 2011.

[139] **ISO/ANSI Norm ISO-8652:1995/AMD 1:2007**. ISO/ANSI, 2007.

[140] **ISO/IEC standard 7498-1:1994 – Open Systems Interconnection**. ISO/IEC, 1994.

[141] **ISO/IEC standard 14882:2003 – Programming Languages C++**. ISO/IEC, 2003.

[142] Kenji Ito, Noriyoshi Suzuki, Satoshi Makido, and Hiroaki Hayashi. **Periodic Broadcast Type Timing Reservation MAC Protocol for Inter-Vehicle Communications**. In *28th IEEE conference on Global telecommunications (GLOBECOM 2009)*, pages 714–719, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-4147-1.

[143] Kyle Jamieson, Hari Balakrishnan, and Y.C. Tay. **Sift: a MAC Protocol for Event-Driven Wireless Sensor Networks**. In Römer et al. [250]. ISBN 3-540-32158-6.

[144] E. Douglas Jensen, C. Douglas Locke, and Hideyuki Tokuda. **A Time-Driven Scheduling Model for Real-Time Operating Systems**. In *IEEE Real-Time Systems Symposium*, pages 112–122. IEEE Computer Society, December 1985.

[145] Jaein Jeong. **Incremental Network Programming for Wireless Sensors**. In *1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004)*, pages 25–33, 2004.

[146] Mark S. Johnstone and Paul R. Wilson. **The Memory Fragmentation Problem: Solved?** In *1st international symposium on Memory management (ISMM 1998)*, pages 26–36, New York, NY, USA, 1998. ACM. ISBN 1-58113-114-3.

[147] Mike Jones. **What Happened on Mars?**, 1997. URL `http://www.ece.cmu.edu/~raj/mars.html`.

[148] Joseph M. Kahn, Randy Howard Katz, and Kristofer S. J. Pister. **Emerging Challenges: Mobile Networking for Smart Dust**. *Journal of Communications and Networks*, 2:188–196, 2000.

[149] Dionisis Kandris, Panagiotis Tsioumas, Anthony Tzes, George Nikolakopoulos, and Dimitrios D. Vergados. **Power Conservation through Energy Efficient Routing in Wireless Sensor Networks**. *Sensors*, 9(9):7320–7342, 2009. ISSN 1424-8220. doi: 10.3390/s90907320.

[150] Rabindra P. Kar. **Implementing the Rhealstone Real-Time Benchmark**. http://www.drdobbs.com/184408332, 1990. URL `http://www.drdobbs.com/184408332`.

[151] H. B. Keller. **Numerical Studies of the Gauss Lattice Problem**. Technical Report CRPC-TR97699, Center for Research on Parallel Computation, Houston, January 1997.

[152] Kavi Kumar Khedo and R. K. Subramanian. **A Service-Oriented Component-Based Middleware Architecture For Wireless Sensor Networks**. *International Journal of Computer Science and Network Security*, 9(3):174–182, March 2009.

[153] Jan Kiszka. **Towards Linux as a Real-Time Hypervisor**. In *11th Real Time Linux Workshop*, 2009.

[154] Thomas Kleffel. **Rechnergestützte Analyse und Visualisierung des Speicherbedarfs von Multiasking-Systemen für**

**Kleinstrechner**. Diploma thesis, University of Würzburg, Würzburg, May 2007.

[155] Alexander Klein. **Performance Issues of MAC and Routing Protocols in Wireless Sensor Networks**. Phd thesis, University of Würzburg, November 2010.

[156] Alexander Klein. **BPS-MAC: Backoff Preamble Based MAC Protocol with Sequential Contention Resolution**. In *4th International Conference on Multiple Access Communications (MACOM 2011)*, pages 39–50, 2011. ISBN 978-3-642-23794-2.

[157] Kevin Klues, Vlado Handziski, David Culler, David Gay, Phillip Levis, Chenyang Lu, and Adam Wolisz. **Dynamic Resource Management in a Static Network Operating System**. Technical Report WUCSE-2006-56, Washington University in St. Louis, November 2006.

[158] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. **TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS**. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys 2009)*, pages 127–140, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2.

[159] Donald E. Knuth. **The Art of Computer Programming**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[160] Joel Koshy. **Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks**. In *2nd European Workshop on Wireless Sensor Networks*, pages 354–365. IEEE Press, 2005.

[161] Nathan Krislock and Henry Wolkowicz. **Explicit Sensor Network Localization Using Semidefinite Representations and Clique Reductions**. Technical report, Department of Combinatorics and Optimization, University of Waterloo, 2009.

[162] Lars Kulik, Egemen Tanin, and Muhammad Umer. **Efficient Data Collection and Selective Queries in Sensor Networks**. In Silvia Nittel, Alexandros Labrinidis, and Anthony Stefanidis, editors, *GeoSensor Networks*, pages 25–44. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-79995-5.

[163] Ram Kumar, Eddie Kohler, and Mani Srivastava. **Harbor: Software-Based Memory Protection for Sensor Nodes**. In *In ACM IPSN*, pages 340–349, 2007.

[164] Sunil Kumar, Vineet S. Raghavan, and Jing Deng. **Medium Access Control Protocols for ad hoc Wireless Networks: A Survey**. *Ad Hoc Networks*, 4:326–358, May 2006. ISSN 1570-8705. doi: 10.1016/j.adhoc.2004.10.001.

[165] Mauri Kuorilehto, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen. **SensorOS: A New Operating System for Time Critical WSN Applications.** In Stamatis Vassiliadis, Mladen Berekovic, and Timo D. Hämäläinen, editors, *SAMOS Workshop on Systems, Architectures, Modeling, and Simulation*, volume 4599 of *LNCS*. Springer, July 2007. ISBN 978-3-540-73622-6.

[166] Manish Kushwaha, Isaac Amundson, Xenofon Koutsoukos, Sandeep Neema,

and Janos Sztipanovits. **OASiS: A Programming Framework for Service-Oriented Sensor Networks**. In *In IEEE/Create-Net COMSWARE 2007*, 2007.

[167] Tor-Inge Kvaksrud. **Range Measurements in an Open Field Environment**. Texas Instruments Inc., 2008. [Online]. Available: http://www.ti.com/litv/pdf/swra169a.

[168] Oh-Heum Kwon and Ha-Joo Song. **Localization through Map Stitching in Wireless Sensor Networks**. *IEEE Transactions on Parallel and Distributed Systems*, 19:93–105, 2008. ISSN 1045-9219.

[169] Rafael Lajara, José Pelegrí-Sebastiá, and Juan J. Perez Solano. **Power Consumption Analysis of Operating Systems for Wireless Sensor Networks**. *Sensors*, 10 (6):5809–5826, 2010. ISSN 1424-8220.

[170] Doug Lea. **A Memory Allocator**, 2010. URL `http://g.oswego.edu/dl/html/malloc.html`.

[171] Edward A. Lee. **Cyber Physical Systems: Design Challenges**. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, January 2008.

[172] Insup Leep, Joseph Y-T. Leung, and Sang H. Son, editors. **Handbook of Real-Time and Embedded Systems**. CRC Press, 2007. ISBN 1584886781.

[173] Philip Levis and David Culler. **Maté: A Tiny Virtual Machine for Sensor Networks**. In *10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X)*, pages 85–95, 2002.

[174] Philip Levis and David Gay. **TinyOS Programming**. Cambridge University Press, New York, NY, USA, 2009. ISBN 0521896061, 9780521896061.

[175] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. **TinyOS: An Operating System for Sensor Networks**. In *Ambient Intelligence*. Springer Verlag, 2004.

[176] Peng Li, Binoy Ravindran, and E. Douglas Jensen. **Adaptive Time-Critical Resource Management Using Time/Utility Functions: Past, Present, and Future**. *Computer Software and Applications Conference*, 2:12–13, 2004. ISSN 0730-3157.

[177] Xu Li, Kaiyuan Lu, Nicola Santoro, Isabelle Simplot-Ryl, and Ivan Stojmenovic. **Alternative Date Gathering Schemes for Wireless Sensor Networks**. In *International Conference on Relations, Orders and Graphs: Interaction with Computer Science (ROGICS 2008)*, pages 577–586, Mahdia, Tunisie, 2008.

[178] Chieh-Jan Mike Liang, Razvan Musaloiu-Elefteri, and Andreas Terzis. **Typhoon: A Reliable Data Dissemination Protocol for Wireless Sensor Networks**. In Verdone [299], pages 268–285. ISBN 978-3-540-77689-5.

[179] Stephanie Lindsey, Cauligi Raghavendra, and Krishna M. Sivalingam. **Data Gathering Algorithms in Sensor Networks Using Energy Metrics**. *IEEE Transactions on Parallel and Distributed Systems*, 13:924–935, 2002.

[180] C. L. Liu and James W. Layland. **Scheduling Algorithms for Multiprogramming**

**in a Hard-Real-Time Environment**. *Journal of the ACM*, 20:46–61, January 1973. ISSN 0004-5411.

[181] Hui Liu, Houshang Darabi, Pat Banerjee, and Jing Liu. **Survey of Wireless Indoor Positioning Techniques and Systems**. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(6):1067–1080, November 2007.

[182] Ting Liu and Margaret Martonosi. **Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems**. In *9th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2003)*, pages 107–118. ACM Press, 2003.

[183] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. **Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet**. In *2nd international conference on Mobile systems, applications, and services (MobiSys 2004)*, pages 256–269, New York, NY, USA, 2004. ACM. ISBN 1-58113-793-1.

[184] Doug Locke. **Priority Inheritance: The Real Story**, July 2002. URL `http://thing1.linuxdevices.com/articles/AT5698775833.html`.

[185] Cristina V. Lopes, Amir Haghighat, Atri Mandal, Tony Givargis, and Pierre Baldi. **Localization of Off-The-Shelf Mobile Devices using Audible Sound: Architectures, Protocols and Performance Assessment**. *SIGMOBILE Mob. Comput. Commun. Rev.*, 10(2):38–50, 2006. ISSN 1559-1662.

[186] Konrad Lorincz and Matt Welsh. **A Robust, Decentralized Approach to RF-Based Location Tracking**. Technical Report TR-04-04, Harvard University, 2004.

[187] Ravenbrook Ltd. **The Memory Management Reference**. Web site `http://www.memorymanagement.org/`, 2010.

[188] Mingming Lu and Jie Wu. **Utility-Based Data-Gathering in Wireless Sensor Networks with Unstable Links**. In *9th international conference on Distributed Computing and Networking (ICDCN 2008)*, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[189] Hong Luo, Jun Luo, Yonghe Liu, and Sajal K. Das. **Adaptive Data Fusion for Energy Efficient Routing in Wireless Sensor Networks**. *IEEE Trans. on Comp*, 55, 2006.

[190] Gerald Lutter. **An AT54DB161B file system for YaOS**. Department of Computer Engineering, University of Wuerzburg, Würzburg, Germany, 2006.

[191] Gerald Lutter. **Bahnregelung eines Fahrzeugs auf Basis eines schnellen Lokalisationssystems**. Diploma thesis, University of Würzburg, Würzburg, November 2008.

[192] Dimitrios Lymberopoulos, Quentin Lindsey, and Andreas Savvides. **An Empirical Characterization of Radio Signal Strength Variability in 3-D IEEE 802.15.4 Networks Using Monopole Antennas**. In Römer et al. [250], pages 326–341. ISBN 3-540-32158-6.

[193] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. **Wireless Sensor Networks for**

**Habitat Monitoring**. In *1st ACM international workshop on Wireless sensor networks and applications (WSNA 2002)*, pages 88–97, New York, NY, USA, 2002. ACM. ISBN 1-58113-589-0.

[194] Rajib Mall. **Real-Time Systems: Theory and Practice**. Pearson Education, 2007. ISBN 9788131700693.

[195] Arati Manjeshwar and Dharma P. Agrawal. **TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks**. *International Parallel and Distributed Processing Symposium*, 3:30189a+, 2001. ISSN 1530-2075.

[196] Guoqiang Mao and Baris Fidan. **Localization Algorithms and Strategies for Wireless Sensor Networks**. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2009. ISBN 9781605663968.

[197] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. **Flexcup: A flexible and efficient Code Update Mechanism for Sensor Networks**. In *3rd European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 212–227, 2006.

[198] Kirk Martinez, Jane K. Hart, and Royan Ong. **Deploying a Wireless Sensor Network in Iceland**. In *3rd International Conference on GeoSensor Networks (GSN 2009)*, pages 131–137, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02902-8.

[199] M. Masmano, I. Ripoll, A. Crespo, and J. Real. **TLSF: A New Dynamic Memory Allocator for Real-Time Systems**. In *Euromicro Conference on Real-Time Systems*, pages 79–86, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[200] Miguel Masmano, Ismael Ripoll, Patricia Balbastre, and Alfons Crespo. **A Constant-Time Dynamic Storage Allocator for Real-Time Systems**. *Real-Time Systems*, 40(2):149–179, 2008. ISSN 0922-6443.

[201] Friedemann Mattern. **Vom Verschwinden des Computers - Die Vision des Ubiquitous Computing**. In Friedemann Mattern, editor, *Total vernetzt*, pages 1–41. Springer-Verlag, 2003. ISBN 3-540-00213-8.

[202] Rainer Mautz. **The Challenges of Indoor Environments and Specification on some Alternative Positioning Systems**. In *6th Workshop on Positioning, Navigation and Communication 2009 (WPNC 2009)*, 2009. doi: 10.3846/1392-1541.2008.34.66-70.

[203] Rainer Mautz. **Overview of Current Indoor Positioning Systems**. *Geodesy and Cartography*, 35(1):18–22, 2009. doi: 10.3846/1392-1541.2009.35.18-22.

[204] Rainer Mautz and Sebastian Tilch. **Survey of Optical Indoor Positioning Systems**. In IPIN2011 [138].

[205] Davide Merico and Roberto Bisiani. **Contexta-NET. A Data-Gathering system that supports Situation-Aware Applications: A Healthcare Example**. In IPIN2011 [138].

[206] Clemens Mühlberger. **Energetic and Temporal Analysis of a Desynchronized TDMA Protocol for WSNs**. In Institut für Telematik, editor, *8. GI/ITG*

*KuVS Fachgespräch Drahtlose Sensornetze*, pages 59–62, Hamburg, Germany, August 2009. Technische Universität Hamburg-Harburg, Institute of Telematics.

[207] Clemens Mühlberger. **Integrating Time-Stamped Synchronization into a Periodical MAC Protocol - Problems and Experiences**. Technical Report 477, Institute for Computer Science, University of Würzburg, December 2010.

[208] Clemens Mühlberger. **Desynchronization in Multi-Hop Topologies: A Challenge**. In Reiner Kolla, editor, *9. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, pages 21–24, Würzburg, Germany, September 2010. Institute for Computer Science, University of Würzburg. urn:nbn:de:bvb:20-opus-51106.

[209] Clemens Mühlberger and Marcel Baunach. **Tab WoNS: Calibration Approach for WSN based Ultrasound Localization Systems**. In Ritter et al. [246], pages 41–44. Technical Report B 08-12.

[210] Clemens Mühlberger and Tobias Schäfer. **SuperG: A Multi-Radio Architecture to interconnect multiple Wireless Sensor Networks**. In *10. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* Uni [293].

[211] Maged M. Michael. **Scalable Lock-Free Dynamic Memory Allocation**. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004)*, pages 35–46. ACM, 2004. ISBN 1-58113-807-5.

[212] Hong Min, Sangho Yi, Yookun Cho, and Jiman Hong. **An Efficient Dynamic Memory Allocator for Sensor Operating Systems**. In *ACM Symposium on Applied Computing (SAC 2007)*, pages 1159–1164, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4.

[213] Linda K. Moore. **Emergency Communications: The Future of 911**. *Library of Congress*, June 2009.

[214] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. **FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks**. In Verdone [299], pages 286–304. ISBN 978-3-540-77689-5.

[215] Waqaas Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. **Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks**. In *International Conference on Communications (ICC)*, Cape Town, South Africa, May 2010.

[216] David Muñoz, Frantz Bouchereau Lara, Cesar Vargas, and Rogerio Enriquez-Caldera. **Position Location Techniques and Applications**. Academic Press, San Diego, CA, 2009. ISBN 0123743532.

[217] Akira Mutazono, Masashi Sugano, and Masayuki Murata. **Frog Call-Inspired Self-Organizing Anti-Phase Synchronization for Wireless Sensor Networks**. In *2nd International Workshop on Nonlinear Dynamics and Synchronization (INDS 2009)*, pages 81–88, Klagenfurt (Austria), July 2009.

[218] Siri Namtvedt. **Design Note DN506 - CC1100 GDO Pin Usage**. Technical report, Texas Instruments Inc., October 2007. URL `http://focus.ti.com/lit/an/swra121a/swra121a.pdf`.

[219] Asis Nasipuri, Robert Cox, James M. Conrad, Luke Van der Zel, Bienvenido

Rodriguez, and Ralph McKosky. **Design Considerations for a Large-Scale Wireless Sensor Network for Substation Monitoring**. In *5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2010)*, Denver, Colorado, USA, October 2010.

[220] Tomasz Naumowicz, Robin Freeman, Holly Kirk, Ben Dean, Martin Calsyn, Achim Liers, Alexander Braendle, Tim Guilford, and Jochen Schiller. **Wireless Sensor Network for Habitat Monitoring on Skomer Island**. In *5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2010)*, Denver, Colorado, USA, October 2010.

[221] Werner Georg Nowak. **Lattice Points in a Circle and Divisors in Arithmetic Progressions**. *Manuscripta Mathematica*, 49(2):195–205, 1984. doi: 10.1007/BF01168751.

[222] Sotaro Ohara, Makoto Suzuki, Shunsuke Saruwatari, and Hiroyuki Morikawa. **A Prototype of a Multi-core Wireless Sensor Node for Reducing Power Consumption**. In *International Symposium on Applications and the Internet (SAINT 2008)*, pages 369–372, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3297-4.

[223] Diego Ongaro, Alan L. Cox, and Scott Rixner. **Scheduling I/O in Virtual Machine Monitors**. In *4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 1–10, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4.

[224] Amitangshu Pal. **Localization Algorithms in Wireless Sensor Networks - Current Approaches and Future Challenges**. *Network Protocols and Algorithms*, 2(1), 2010.

[225] Javier Palafox-Albarrán, Reiner Jederman, and Walter Lang. **Prediction of Temperature inside a refrigerated Container in the Presence of Perishable Goods**. In *7th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2010)*, 2010.

[226] Santosh Pandey and Prathima Agrawal. **A Survey on Localization Techniques for Wireless Networks**. *Journal of the Chinese Institute of Engineers*, 27(7): 1125–1148, 2006.

[227] Jim Partan and Jim Kurose Brian Neil Levine. **A Survey of Practical Issues in Underwater Networks**. In *The ACM International Workshop on UnderWater Networks (WUWNet 2006)*, pages 17–24, 2006.

[228] Joseph Polastre, Jason Hill, and David Culler. **Versatile Low Power Media Access for Wireless Sensor Networks**. In *SenSys04*, pages 95–107, Baltimore, MD, November 2004. B-MAC.

[229] Joseph Polastre, Robert Szewczyk, and David Culler. **Telos: Enabling Ultra-Low Power Wireless Research**. In *4th International Conference on Information Processing in Sensor Networks*, 2005.

[230] Edmond Prakash, Jim Wood, Baihua Li, Michael Clarke, George Smith, and Keith Yates. **Games Technology: Console Architectures, Game Engines and Invisible Interaction**. In *4th International*

*Conference on Computer Games, Multimedia and Allied Technology (CGAT 2011)*, April 2011.

[231] Ramjee Prasad and Marina Ruggieri. **Applied Satellite Navigation Using GPS, GALILEO, and Augmentation Systems**. Artech House, 2005. ISBN 978-1580538145.

[232] Helena Preiß, Sebastian Lempert, Christopher Kaffenberger, and Alexander Pflaum. **Wireless Sensor Network-Based Asset Management for Mobile Measurement Devices**. In *10. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* Uni [293].

[233] Nissanka B. Priyantha, Allen K.L. Miu, Hari Balakrishnan, and Seth Teller. **The Cricket Compass for Context-Aware Mobile Applications**. In *7th annual international conference on Mobile computing and networking (MobiCom 2001)*, pages 1–14, New York, NY, USA, July 2001. ACM Press.

[234] Nissanka Bodhi Priyantha. **The Cricket Indoor Location System**. Phd thesis, Massachusetts Institute of Technology, June 2005.

[235] The FreeBSD Project. **FreeBSD**. Web site `http://www.freebsd.org/`, 2011.

[236] Isabelle Puaut. **Real-Time Performance of Dynamic Memory Allocation Algorithms**. In *14th Euromicro Conference on Real-Time Systems (ECRTS 2002)*, pages 41–49. IEEE Computer Society, June 2002.

[237] Kimmo E. E. Raatikainen. **Operating System Issues in Future End-User Systems**. In *16th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2005)*. IEEE Computer Society, September 2005.

[238] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. **Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems**. In *SPIE/ACM Conference on Multimedia Computing and Networking*, pages 150–164, 1998.

[239] Theodore Rappaport. **Wireless Communications: Principles and Practice**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130422320.

[240] Adi Mallikarjuna V. Reddy, A.V.U. Phani Kumar, D. Janakiram, and G. Ashok Kumar. **Wireless Sensor Network Operating Systems - A Survey**. *International Journal on Sensor Networks*, 5(4):236–255, 2009. ISSN 1748-1279.

[241] Niels Reijers and Koen Langendoen. **Efficient Code Distribution in Wireless Sensor Networks**. In *2nd ACM international conference on Wireless sensor networks and applications (WSNA 2003)*, pages 60–67, New York, NY, USA, 2003. ACM. ISBN 1-58113-764-8.

[242] Tobias Reusing. **Implementierung und Analyse eines WSN gestützten Lokalisationsalgorithmus auf Basis des SNoW5 Sensorknotens**. Bachelor thesis, University of Würzburg, Würzburg, June 2010.

[243] **TR1000 916.50 MHz Hybrid Transceiver**. RF Monolithics, 2008.

[244] Matthias Ringwald and Kay Roemer. **BitMAC: A Deterministic, Collision-Free, and Robust MAC Protocol for Sensor**

**Networks**. In *2nd European Conference on Wireless Sensor Networks (EWSN 2005)*, pages 57–69, Istanbul, Turkey, January 2005.

[245] Ismael Ripoll, Patricia Balbastre, Miguel Masmano, Alfons Crespo, and Alan Burns. **Contract Based Management of the Memory Resource**. In *17th International Conference on Real-Time and Network Systems (RTNS 2009)*, pages 115–124, 2009.

[246] Hartmut Ritter, Kirsten Terfloth, Georg Wittenburg, and Jochen Schiller, editors. **7. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze**, Berlin, Germany, September 2008. Freie Universität Berlin, Institute of Computer Science. Technical Report B 08-12.

[247] Wind River. **The Wind River Hypervisor**. Web site `http://windriver.com/products/hypervisor/`, 2012.

[248] Kay Römer. **Discovery of Frequent Distributed Event Patterns in Sensor Networks**. In Verdone [299], pages 106–124. ISBN 978-3-540-77689-5.

[249] Kay Römer and Friedemann Mattern. **The Design Space of Wireless Sensor Networks**. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[250] Kay Römer, Holger Karl, and Friedemann Mattern, editors. **3rd European Workshop on Wireless Sensor Networks (EWSN 2006), Zurich**, volume 3868 of *LNCS*, February 2006. Springer. ISBN 3-540-32158-6.

[251] J.M. Robson. **Worst Case Fragmentation of First Fit and Best Fit Storage Allocation Strategies**. *The Computer Journal*, 20(3):242–244, 1977.

[252] Daniel Ronzani. **The Battle of Concepts: Ubiquitous Computing, Pervasive Computing and Ambient Intelligence in Mass Media**. *Ubiquitous Computing And Communication Journal UbiCC*, 4(2), May 2009. ISSN 1992-8424.

[253] Jeremy Roschelle and Stephanie D. Teasley. **The Construction of Shared Knowledge in Collaborative Problem Solving**. In Claire O'Malley, editor, *Computer-Supported Collaborative Learning*, pages 69–97. Springer, Berlin, 1995.

[254] E. M. Royer and Chai-Keong Toh. **A review of current routing protocols for ad hoc mobile wireless networks**. *IEEE Personal Communications*, 6(2):46–55, April 1999.

[255] Luis Ruiz-Garcia, Pilar Barreiro, Jose Ignacio Robla, and Loredana Lunadei. **Testing ZigBee Motes for Monitoring Refrigerated Vegetable Transportation under Real Conditions**. *Sensors*, 10: 4968–4982, 2010. ISSN 1424-8220.

[256] Armin Runge. **Verfahren zur Selbstkalibrierung von Lokalisationssystemen basierend auf drahtlosen Sensornetzen**. Diploma thesis, University of Würzburg, Würzburg, November 2010.

[257] Armin Runge and Marcel Baunach. **Precise Self-Calibration of Ultrasound Based Indoor Localization Systems**. In IPIN2011 [138].

[258] M. J. Saeed, M. Merabti, and R. J. Askwith. **Hidden and Exposed Nodes and Medium Access Control in Wireless Ad-Hoc Networks**. In *7th Annual PG Symposium on the Convergance of Telecommunications, Networking and*

*Broadcasting (PGNet 2006)*, June 2006. ISBN 1902560139.

[259] Martin Schröder. **Indoor- und Outdoor-Lokalisation mittels empfangener Signalstärke in Drahtlosen Sensornetzen**. Diploma thesis, University of Würzburg, Würzburg, April 2009.

[260] Timo Schröder, Tobias Pilsak, and Jan Luiken ter Haseborg. **Entwicklung eines Sensornetzwerk für den Einsatz in Schiffsmaschinenräumen**. In *8. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* Tec [277].

[261] Herbert Schweinzer and Mohammad Syafrudin. **LOSNUS: An Ultrasonic System Enabling High Accuracy and Secure TDoA Locating of Numerous Devices**. In *1st International Conference on Indoor Positioning and Indoor Navigation (IPIN 2010)* ETH [96].

[262] L. Sha, R. Rajkumar, and J. P. Lehoczky. **Priority Inheritance Protocols: An Approach to Real-Time Synchronization**. *IEEE Transactions on Computers*, 39(9): 1175–1185, 1990. ISSN 0018-9340.

[263] Yi Shang, Wheeler Ruml, Ying Zhang, and Markus P. J. Fromherz. **Localization from mere Connectivity**. In *4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc 2003)*, pages 201–212, New York, NY, USA, 2003. ACM. ISBN 1-58113-684-6.

[264] Oliver Sharma, Jonathan Lewis, Alice Miller, Al Dearle, Dharini Balasubramaniam, Ron Morrison, and Joe Sventek. **Towards Verifying Correctness of Wireless Sensor Network Applications Using Insense and Spin**. In *16th International SPIN Workshop on Model Checking Software*, pages 223–240, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02651-5.

[265] Anthony Man-Cho So and Yinyu Ye. **Theory of Semidefinite Programming for Sensor Network Localization**. In *16th annual ACM-SIAM symposium on Discrete algorithms (SODA 2005)*, pages 405–414, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. ISBN 0-89871-585-7.

[266] Andrey Somov, Ivan Minakov, Alena Simalatsar, Giorgio Fontana, and Roberto Passerone. **A Methodology for Power Consumption Evaluation of Wireless Sensor Networks**. In *14th IEEE international conference on Emerging technologies & factory automation (ETFA 2009)*, pages 1326–1333, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-2727-7.

[267] Lei Song and Yongcai Wang. **3D Accurate Location Stream Tracking and Recognition Using an Ultrasound Localization System**. In IPIN2011 [138].

[268] Gerhard. F. Spitzer and Herbert. F. Schweinzer. **LOSNUS: Ultrasonic indoor locating for numerous static and mobile devices.** In *7th Workshop on Positioning, Navigation and Communication (WPNC 2010)*, pages 277–283. IEEE, March 2010.

[269] Stig Støa and Ilangko Balasingham. **Periodic-MAC: Improving MAC Protocols for Wireless Biomedical Sensor Networks through Implicit Synchronization**. InTech, January 2011. ISBN 978-953-307-475-7.

[270] John A. Stankovic and Krithi Ramamritham. **A Reflective Architecture for Real-Time Operating Systems**. In *Advances in Real-Time Systems*, pages 23–38. Prentice-Hall, Inc., 1995. ISBN 0-13-083348-7.

[271] **Multithreaded Programming Guide**. Sun Microsystems Inc., Santa Clara (USA), September 2008.

[272] Petcharat Suriyachai, Utz Roedig, and Andrew Scott. **A Survey of MAC Protocols for Mission-Critical Applications in Wireless Sensor Networks**. *IEEE Communications Surveys & Tutorials*, 99: 1–25, 2011. ISSN 1553-877X.

[273] Swedish Institute of Computer Science. **MSPsim - a Java-based simulator of MSP430 sensor network platforms**. Web site `http://www.sics.se/project/mspsim`, 2011.

[274] Nazifa Tahir and Graham Brooker. **Recent Developments and Recommendations for Improving Harmonic Radar Tracking System**. In *IEEE Symposium on Industrial Electronics and Applications (ISIEA 2010)*, October 2010.

[275] Majid Alkaee Taleghan, Amirhosein Taherkordi, Mohsen Sharifi, and Tai-Hoon Kim. **A Survey of System Software for Wireless Sensor Networks**. *Future Generation Communication and Networking*, 2:402–407, 2007.

[276] Andrew S. Tanenbaum. **Modern Operating Systems**. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2008. ISBN 9780136006633.

[277] **8. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze**, August 2009. Technical University Hamburg-Harburg (TUHH).

[278] Guodong Teng, Kougen Zheng, and Wei Dong. **SDMA: A Simulation-Driven Dynamic Memory Allocator for Wireless Sensor Networks**. In *International Conference on Sensor Technologies and Applications*, Los Alamitos, CA, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3330-8.

[279] **MSP430x1xx Family User's Guide**. Texas Instruments Inc., 2006.

[280] **MSP430x161x Mixed Signal Microcontroller**. Texas Instruments Inc., Dallas (USA), August 2006. URL `http://focus.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=msp430f1611`.

[281] **CC1100 Single Chip Low Cost Low Power RF Transceiver**. Texas Instruments Inc., Dallas (USA), 2009. URL `http://www.ti.com/lit/gpn/cc1100`.

[282] **CC2520 2.4 GHz IEEE 802.15.4 / ZigBee RF Transceiver**. Texas Instruments Inc., Dallas (USA), 2009. URL `http://www.ti.com/lit/gpn/cc2520`.

[283] **CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver**. Texas Instruments Inc., Dallas (USA), 2010. URL `http://www.ti.com/lit/gpn/cc2420`.

[284] C. Tharini and P. Vanaja Ranjan. **An Efficient Data Gathering Scheme for Wireless Sensor Networks**. *European Journal of Scientific Research*, 43(1):148–155, 2010.

[285] The OMNeT++ Community. **The OMNeT++ Network Simulation Framework**. Web site http://www.omnetpp.org, 2011.

[286] Abdullah Erdal Tümer and Mesut Gündüz. **Energy-Efficient and Fast Data Gathering Protocols for Indoor Wireless Sensor Networks**. *Sensors*, 10 (9):8054–8069, 2010. ISSN 1424-8220. doi: 10.3390/s100908054.

[287] H. Tomiyama, S. Honda, and H. Takada. **Real-Time Operating Systems for Multicore Embedded Systems**. In *International SoC Design Conference (ISOCC 2008)*, November 2008.

[288] Eric Trumpler and Richard Han. **A Systematic Framework for Evolving TinyOS**. In *3rd Workshop on Embedded Networked Sensors*, pages 61–65, 2006.

[289] Yu-Chee Tseng, Chi-Fu Huang, and Sheng-Po Kuo. **Positioning and Location Tracking in Wireless Sensor Networks**. In Ilyas and Mahgoub [134], chapter 21, pages 1–13.

[290] Alan M. Turing. **On Computable Numbers, with an Application to the Entscheidungsproblem**. *The London Mathematical Society*, 2(42):230–265, 1936.

[291] UC Berkeley. **TinyOS**. Web site http://code.google.com/p/tinyos-main/, 2011.

[292] Steve Underwood. **mspgcc - A port of the GNU tools to the Texas Instruments MSP430 microcontrollers**, 2003. URL http://mspgcc.sourceforge.net/.

[293] **10. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze**, September 2011. Universität Paderborn.

[294] University of Cambridge. **Nemesis – An Operating System with Principles**. Web site http://www.cl.cam.ac.uk/research/srg/netos/old-projects/nemesis/, 2011.

[295] M. Vahabi, M. F. A. Rasid, R. S. A. R. Abdulllah, and M. H. F. Ghazvini. **Adaptive Data Collection Algorithm for Wireless Sensor Networks**. *Journal of Computer Science and Network security (IJCSNS)*, 8: 125–132, June 2008.

[296] Martijn van den Heuvel, Reinder J. Bril, and Moris Behnam. **Extending an HSF-enabled Open Source Real-Time Operating System with Resource Sharing**. In *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010.

[297] Lodewijk F.W. van Hoesel, Stefan O. Dulman, Paul J.M. Havinga, and Harry J. Kip. **Design of a Low-Power Testbed for Wireless Sensor Networks and Verification**. Internal Report 41407, CTIT, University of Twente, 2003.

[298] Steven J. Vaughan-Nichols. **Game-Console Makers Battle over Motion-Sensitive Controllers**. *Computer*, 42:13–15, 2009. ISSN 0018-9162.

[299] Roberto Verdone, editor. **5th European Conference on Wireless Sensor Networks (EWSN 2008)**, volume 4913 of *LNCS*, Bologna, January 2008. Springer. ISBN 978-3-540-77689-5.

[300] Freddy López Villafuerte, Jochen Schiller, Ernesto Tapia, Marte Ramírez, and

Erik Valdemar. **Evaluating Parameters for Localization in Wireless Sensor Netwroks: A Survey**. In *4th International Congress on Electronics and Biomedical Engineering, Computer Science and Informatics (CONCIBE 2008)*, Guadalajara, Mexico, May 2008.

[301] Lucas Francisco Wanner, Arliones Stevert Hoeller Junior, Fauze Valério Polpeta, and Antônio Augusto Fröhlich. **Operating System Support for Handling Heterogeneity in Wireless Sensor Networks**. In *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005)*. IEEE, September 2005.

[302] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. **The Active Badge Location System**. *ACM Transactions on Information Systems*, 10:91–102, January 1992. ISSN 1046-8188.

[303] Andrew Martin Robert Ward. **Sensor-Driven Computing**. Phd thesis, University of Cambridge, 1998.

[304] Andy Ward, Alan Jones, and Andy Hopper. **A New Location Technique for the Active Office**. *IEEE Personal Comm.*, 4 (5):42–47, October 1997.

[305] Mark Weiser. **Some Computer Science Issues in Ubiquitous Computing**. *Commun. ACM*, 36(7):74–84, 1993.

[306] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. **Dynamic Storage Allocation: A Survey and Critical Review**. In *International Workshop on Memory Management*, pages 1–116. Springer-Verlag, 1995.

[307] Windriver. **VxWorks Operating System**. Web site `http://www.windriver.com/products/vxworks/`, 2011.

[308] Georg Wittenburg, Norman Dziengel, Christian Wartenburger, and Jochen Schiller. **A System for Distributed Event Detection in Wireless Sensor Networks**. In *9th ACM/IEEE International Conference on Information Processing in Sensor Networks (ISPN2010)*, IPSN '10, pages 94–104, New York, NY, USA, 2010. ISBN 978-1-60558-988-6.

[309] Jian Wu, Shenfang Yuan, Genyuan Zhou, Sai Ji, Zilong Wang, and Yang Wang. **Design and Evaluation of a Wireless Sensor Network Based Aircraft Strength Testing System**. *Sensors*, 9(6):4195–4210, 2009. ISSN 1424-8220.

[310] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher D. Gill. **RT-Xen: Towards Real-Time Hypervisor Scheduling in XEN**. In *11th International Conference on Embedded Software (EMSOFT 2011)*, pages 39–48. ACM, October 2011.

[311] Peng Xie and Jun-Hong Cui. **R-MAC: An Energy-Efficient MAC Protocol for Underwater Sensor Networks**. In *International Conference on Wireless Algorithms, Systems and Applications (WASA 2007)*, pages 187–198, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2981-X.

[312] **Spartan-3AN FPGA Family Data Sheet**. Xilinx Inc., 2009.

[313] Yuyan Xue, Byrav Ramamurthy, and Mark Burbach. **A Two-Tier Wireless Sensor Network Infrastructure for Large-Scale Real-Time Groundwater Monitoring**. In *5th IEEE International Work-*

*shop on Practical Issues in Building Sensor Network Applications (SenseApp 2010)*, Denver, Colorado, USA, October 2010.

[314] Li Yanbing, Miodrag Potkonjak, and Wayne Wolf. **Real-Time Operating Systems for Embedded Computing**. In *ICCD*, pages 388–392, 1997.

[315] Mark Yarvis and Wei Ye. **Tiered Architectures in Sensor Networks**. In Ilyas and Mahgoub [134], chapter 13, pages 1–22.

[316] Kiran Kumar Yedavalli. **On Location Support and One-Hop Data Collection in Wireless Sensor Networks**. Phd thesis, University of Southern California, May 2007.

[317] Sangho Yi, Hong Min, Junyoung Heo, Boncheol Gu, Yookun Cho, Jiman Hong, Hyukjun Oh, and Byunghun Song. *XMAS***: An eXtraordinary Memory Allocation Scheme for Resource-Constrained Sensor Operating Systems**. In Jiannong Cao, Ivan Stojmenovic, Xiaohua Jia, and Sajal K. Das, editors, *MSN*, volume 4325 of *LNCS*, pages 760–769. Springer, December 2006. ISBN 3-540-49932-6.

[318] Sangho Yi, Hong Min, Seungwoo Lee, Yeongkwun Kim, and Injoo Jeong. **Sesame: Space-Efficient Stack Allocation Mechanism for Multi-Threaded Sensor Operating Systems**. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *SAC*, pages 1201–1202. ACM, March 2007. ISBN 1-59593-480-4.

[319] Sangho Yi, Seungwoo Lee, Yookun Cho, and Jiman Hong. **Sesame-P: Memory Pool-Based Dynamic Stack Management for Sensor Operating Systems**. In Sotiris E. Nikoletseas, Bogdan S. Chlebus, David B. Johnson, and Bhaskar Krishnamachari, editors, *DCOSS*, volume 5067 of *LNCS*, pages 544–549. Springer, June 2008. ISBN 978-3-540-69169-3.

[320] Sangho Yi, Hong Min, Yookun Cho, and Jiman Hong. **Molecule: An Adaptive Dynamic Reconfiguration Scheme for Sensor Operating Systems**. *Elsevier Computer Communications*, 31(4):699–707, 2008.

[321] Özalp Babaoğlu, Keith Marzullo, and Fred B. Schneider. **A Formalization of Priority Inversion**. *Real-Time Systems*, 5 (4):285–303, 1993. ISSN 0922-6443.

[322] Eli Zelkha, Brian Epstein, Simon Birrell, and Clark Dodsworth. **From Devices to "Ambient Intelligence"**. In *Digital Living Room Conference*, June 1998.

[323] Hüseyin Özgür Tan and Ibrahim Körpeoglu. **Power Efficient Data Gathering and Aggregation in Wireless Sensor Networks**. *SIGMOD Record*, 32(4):66–71, December 2003. ISSN 0163-5808.

[324] Junhui Zhao and Yongcai Wang. **Autonomous Ultrasonic Indoor Tracking System**. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2008)*, pages 532–539, Washington, DC, USA, 2008. IEEE Computer Society.

[325] X. H. Zhou, N. A. Obuchowski, and D. M. McClish. **Statistical Methods in Diagnostic Medicine**. Wiley & Sons, 2002. ISBN 9780471347729.

# List of Figures


1 This work at a glance ........ xii

1.1 Schematic of hierarchical control systems in wireless sensor/actuator networks ..... 5
1.2 Characterization of wireless sensor/actuator nodes and networks........ 7
1.3 The design space and classification of wireless sensor/actuator networks ........ 9
1.4 A selection of relevant research disciplines for the WSAN domain ........ 10
1.5 The Gartner Hype Cycle and priority matrix for selected emerging technologies ..... 11
1.6 Why real-time matters ........ 13
1.7 WSN/WSAN research directions, benefits, and aims........ 14

2.1 Example for a network of heterogeneous nodes ........ 22
2.2 Sensor nodes: Schematic and examples........ 23
2.3 The SNoW$^5$ platform overview ........ 24
2.4 Block diagram of the Texas Instruments MSP430F1611 MCU ........ 25
2.5 The SNoW$^5$ radio transceiver current consumption at various base frequencies...... 26
2.6 The SNoW$^5$ mainboard (approximately in original size) ........ 30
2.7 The SNoW$^5$ sensor node with various extension boards........ 31
2.8 The SNoW$^5$ sensor node equipped with various extension boards........ 31

3.1 Reference architecture for sensor node and embedded systems hardware and software layers........ 34
3.2 Resource classification framework: In general and under *Smart*OS........ 37
3.3 Selected classification features of embedded operating systems kernels ........ 39
3.4 Comparison of process organization and implementation approaches........ 41

4.1 The central concepts of the *Smart*OS philosophy ........ 49
4.2 Example for priority and timeout queues in *Smart*OS ........ 52
4.3 Execution of the `setEvent` function when called as IRQ handler ........ 65
4.4 Execution of an IRQ handler for I/O port demultiplexing........ 65
4.5 The *Smart*OS kernel mode execution flow depending on its trigger ........ 66
4.6 Execution time of the `yield` function when called from within a task........ 67
4.7 Interrupt processing overhead for various interrupt frequencies $f_{\mathrm{IRQ}}$ ........ 67
4.8 Execution time of the system timer ISR (unloaded system, idle task only)........ 68
4.9 Frequency and duty cycle of the timer IRQ (unloaded system, idle task only) ........ 68

5.1 Error in the state vector of a node $n$ regarding a 2D position $\boldsymbol{x}$ and time $t$ ........ 71

5.2 Emergence of timing errors for the naïve discretization of time .......... 73
5.3 Temperature sensitivity of typical HC49 quartz crystals .......... 75
5.4 IRQ timestamp acquisition under *Smart*OS .......... 77
5.5 The Ping Pong test bed for evaluating the timing precision of *Smart*OS .......... 80
5.6 Clock drift for three node pairs .......... 81
5.7 The node timing error after 100 s as measured by each node .......... 85

6.1 Colliding resource requirement between two interfering tasks .......... 92
6.2 Main challenges for many system service design problems .......... 93
6.3 Types of priority inversions .......... 96
6.4 Task state transitions under *Smart*OS .......... 103
6.5 Examples for Resource-Await-Queues .......... 106
6.6 Example for priority inheritance and DynamicHinting .......... 108
6.7 DynamicHinting examples .......... 112
6.8 Hint processing strategies .......... 114
6.9 Various shapes of time-utility-functions and behavior functions .......... 117
6.10 Stack layout and evolution when resuming a task .......... 122
6.11 Stack layout evolution when throwing an exception from within a HintHandler .......... 123
6.12 Streaming test: Atomic fixed-length packets vs. DynamicHinting .......... 128
6.13 The dining philosopher's problem .......... 130
6.14 The dining philosophers test bed under PIP .......... 134
6.15 The dining philosophers test bed under PCP .......... 135
6.16 The dining philosophers test bed: Conditioning PIP and PCP .......... 136

7.1 Dynamic memory management: From system design to runtime failure .......... 142
7.2 Persisting memory allocation failures for various implementations of `malloc` .......... 147
7.3 Related work for dynamic memory management systems .......... 148
7.4 Dynamic memory management examples .......... 151
7.5 Pool layouts for dynamic memory management .......... 152
7.6 Delayed memory allocation .......... 157
7.7 Critical scenarios during dynamic memory management .......... 158
7.8 API and call hierarchy for the CoMem `malloc` functions .......... 166
7.9 Various memory allocation flows related to the source code from Listing 7.4 .......... 168
7.10 CoMem deadlock freedom and async-signal safety .......... 172
7.11 Memory allocation contracts under CoMem .......... 174
7.12 The CoMem heap organization for RT blocks and co-located non-RT blocks .......... 175
7.13 Examples for heap fragmentation penalties .......... 179
7.14 CoMem test bed task state diagrams .......... 181
7.15 CoMem stress test results for various policies and heap sizes .......... 182
7.16 CoMem stress test results for various policies and heap sizes .......... 184

8.1 SNoW Bat communication subsystems and task interactions for a mobile client .......... 195
8.2 The *Smart*Net packet layout .......... 196

8.3 SNoW Ghost in action ... 199
8.4 SNoW Ghost update flow and performance evaluation ... 201

9.1 Localization systems: Range vs. accuracy (based on [202]) ... 210
9.2 The WSN/WSAN process flow: From environmental conditions to usable information 211
9.3 Measuring distances based on signal propagation delays ... 212
9.4 The radio signal reflection model ... 213
9.5 Wireless measurement techniques for localization systems ... 215
9.6 Accuracy vs. precision ... 216
9.7 The SNoW Bat localization system: An overview on Part III ... 220

10.1 The SNoW Bat test installation ... 225
10.2 Schematic design of the SNoW Bat indoor localization system ... 226
10.3 The SNoW Bat localization process ... 227
10.4 The SNoW Bat task system for mobile and anchor nodes ... 229
10.5 SNoW Bat room geometry and anchor node deployment analysis ... 231
10.6 Room geometry considerations ... 232
10.7 Proportion of the data aggregation time in the complete localization time ... 235
10.8 Parallelization of the localization stages $P_1 \ldots P_4$ on a mobile client node ... 237

11.1 US ranging depending on the node constellation and room geometry ... 245
11.2 Ultrasound chirp generation, reception sampling, and detection ... 246
11.3 The Cut algorithm at short and long distance ... 247
11.4 Ultrasound distance measurement error characteristics ... 249

12.1 TDoA of the US chirp for various anchor distances ... 255
12.2 The HashSlot software/network co-design ... 263
12.3 Return slot assignment at various QoS and AGM levels ... 268
12.4 Slot number calculation based on of various QoS levels ... 269
12.5 QoS related considerations ... 272
12.6 The number of reserved DV slots for different optimization strategies ... 273
12.7 Measurement results and derived metrics from the real-world test bed ... 277
12.8 Locating defective anchors through inverted slot calculation ... 281
12.9 HashSlot Slot number reordering for improved reception of non-collinear DVs ... 282
12.10 Example for slot compression under HashSlot$^+$ ... 287
12.11 Vacant slots under HashSlot ... 288
12.12 Vacant slots under HashSlot$^+$ ... 288

13.1 Intersecting a triplet of spheres ... 293
13.2 The impact of the distance measurement imprecision ... 295
13.3 The precision trust distribution for two anchors $A, B$ ... 296
13.4 A potential pVoted scenario ... 297
13.5 Position prediction based on three historic estimations $\tilde{p}_1, \tilde{p}_2, \tilde{p}_3$ ... 299

13.6 SNoW Bat test bed evaluation paths (ceiling height $h = 7$ m, $d_{min} = 2.8$ m, $L = 1.3$ m) 300
13.7 pVoted benchmark results ........ 303
13.8 pVoted benchmark results ........ 303
13.9 pVoted benchmark results ........ 303

14.1 The SNoW Bat execution flow ........ 305
14.2 Serial vs. parallel execution of the SNoW Bat tasks ........ 307

A.1 The *Smart*OS memory layout for the MSP430F1611 MCU ........ 324
A.2 *Smart*OS kernel functions and syscalls ........ 327

B.1 SNoW$^5$ schematics: MCU and power supply ........ 330
B.2 SNoW$^5$ schematics: I/O header ports ........ 330
B.3 SNoW$^5$ schematics: Radio transceiver and data flash ........ 331
B.4 The SNoW$^5$ layout at original size ........ 331

# List of Tables


2.1 Sensor node design constraints and design aspects ..... 21
2.2 Current consumption of various SNoW[5] sensor node components in mA ..... 27
2.3 Comparison of some selected wireless sensor nodes ..... 29

3.1 Comparison of WSAN operating system kernels ..... 46

4.1 Comparison between mutexes, events, and resources under *Smart*OS ..... 56
4.2 *Smart*OS Rhealstone ($m_i$) and other benchmark results ..... 61
4.3 Maximal code and data sizes for the *Smart*OS kernel and various modules ..... 64

5.1 Error intervals for different discretization techniques (system time resolution: $\lambda_C$) ..... 74
5.2 Drift calculation for $\Delta_c = 12$ ..... 84
5.3 Drift calculation for $\Delta_c = 16$ ..... 84

6.1 A comparison of common synchronization protocols with priority inheritance ..... 99
6.2 Property matrix for the presented hint reception methods ..... 116

7.1 Memory allocation delays for $t_{\mathrm{RC}}$ within SNoW Bat ..... 190

9.1 Comparison of various sound based indoor localization systems ..... 222

12.1 Comparison of various radio protocols for data aggregation ..... 262
12.2 Example for slot compression under HashSlot$^+$ ..... 287

# List of Listings


4.1 *Smart*OS `main()` function and interrupt handling example . . . . . 51
4.2 Mutex usage under *Smart*OS . . . . . 53
4.3 Periodic tasks under *Smart*OS . . . . . 54
4.4 The producer/consumer problem in *Smart*OS: Buffer handling . . . . . 57
4.5 Resource usage under *Smart*OS . . . . . 58
4.6 Exception handling examples: A comparison . . . . . 60
4.7 Benchmarking under *Smart*OS . . . . . 63

5.1 Timestamping within the kernel ISR for an IRQ $e$ . . . . . 79
5.2 Delayed timestamp calculation for the last interrupt . . . . . 79
5.3 The timing test bench: Trigger task and IRQ handlers . . . . . 82

6.1 *Smart*OS functions for the DynamicHinting API . . . . . 115
6.2 Combining DynamicHinting with *Smart*OS exceptions . . . . . 118
6.3 getResource(...): Main loop for processing the affected RAQ . . . . . 120
6.4 Timeout handling: Processing of $A(t)$ if $t$'s request for resource $r$ times out . . . . 121
6.5 Streaming test in operation mode 1 (DMA): Use Early Wakeup to react on hints . 126
6.6 Streaming test in operation mode 2 (DSP): Use a HintHandler to react on hints . 127

7.1 The CoMem `MCB_t` data structure for Memory Control Blocks (MCB) . . . . . 163
7.2 The CoMem memory control block (MCB) declaration and initialization . . . . . 164
7.3 The CoMem management resource (protection) and event (signaling) . . . . . 164
7.4 The CoMem API/collaborator functions . . . . . 169
7.5 Hint handling for the CoMem stress test . . . . . 183
7.6 The SNoW Bat ultrasound chirp detection subsystem . . . . . 188
7.7 The SNoW Bat remote-management subsystem SNoW Ghost . . . . . 189

10.1 The SNoW Bat Chirp Allocation Vector (CAV) data structure . . . . . 228
10.2 The SNoW Bat Distance Vector (DV) data structure . . . . . 228

11.1 The ultrasound task $T_{US}$ on the mobile client nodes . . . . . 243

12.1 Anchor node self-calibration for the precisely timed emission of distance vectors 275
12.2 The DV emission and slot compression algorithm for HashSlot$^+$ . . . . . 285

13.1 The pVoted core algorithm for processing a just received DV (pseudocode) . . . . 294

# Abbreviations

| | |
|---|---|
| ABI | Application Binary Interface |
| ACK | Acknowledgement |
| ADC | Analog to Digital Converter |
| AGM | Adaptive Grid Module |
| AM | Active Mode |
| AP | Atomic Packets |
| API | Application Programming Interface |
| ARI | Avoidance Related Inversion |
| ARR | Avoidance Related Rejection |
| ASAP | As Soon As Possible |
| AoA | Angle Of Arrival |
| BC | Best Case |
| BE | Best Effort |
| BPI | Bounded Priority Inversion |
| BSL | BootStrap Loader |
| C64 | Commodore 64 |
| CA | Collision Avoidance |
| CAN | Controller Area Network |
| CAS | Compare And Swap |
| CAV | Chirp Allocation Vector |
| CCA | Clear Channel Assessment |
| CD | Collision Detection |
| CDMA | Code Division Multiple Access |
| CL | Centroid Localization |
| COTS | Commercial Off-The-Shelf |
| CPS | Cyber Physical Systems |
| CPU | Central Processing Unit |
| CS | Carrier Sense |
| CSMA | Carrier Sense Multiple Access |
| CoO | Cell Of Origin |
| DA | Data Aggregation |
| DAC | Digital to Analog Converter |
| DARPA | Defense Advanced Research Projects Agency |
| DC | Direct Current |
| DCF77 | D=Deutschland (Germany), C=long wave signal, F=Frankfurt, 77=frequency: 77.5 kHz |
| DCO | Digitally Controlled Oscillator |
| DECT | Digital Enhanced Cordless Telecommunications |
| DH | Dynamic Hinting |
| DHH | Dynamic Hint Handler |
| DIN | Deutsche Industrie Norm |
| DL | DeadLine |
| DM | Dynamic Memory |
| DMA | Direct Memory Access |
| DSN | Distributed Sensor Network |
| DSP | Digital Signal Processing |
| DV | Distance Vector |
| ECB | Event Control Block |
| EDF | Earliest Deadline First |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| EERP | Energy Efficient Routing Protocol |
| EGNOS | European Geostationary Navigation Overlay Service |
| EMMA | Embedded Middleware in Mobility Applications |

| | |
|---|---|
| EQ | Explicit Querying |
| ES | Embedded System |
| ESB | Embedded Sensor Board |
| EW | Early Wakeup |
| FCFS | First-Come, First-Served |
| FDMA | Frequency Division Multiple Access |
| FIFO | First In First Out |
| FP6 | Sixth Framework Programme |
| FPGA | Field Programmable Gate Array |
| FSK | Frequency Shift Keying |
| FSPM | Free Space Propagation Model |
| GCC | GNU Compiler Collection |
| GFSK | Gaussian Frequency Shift Keying |
| GNSS | Global Navigation Satellite System |
| GNU | GNU's Not Unix |
| GPS | Global Positioning System |
| GSM | Global System for Mobile communications |
| HAL | Hardware Abstraction Layer |
| HH | Hint Handler |
| HIL | Hardware-In-the-Loop |
| HLP | Highest Locker Protocol |
| HS | HashSlot |
| I2C | Inter-Integrated Circuit |
| IBM | International Business Machines |
| IC | Integrated Circuit |
| ICB | IRQ handler Control Block |
| ID | IDentifier |
| IEEE | Institute of Electrical and Electronics Engineers |
| IP | Internet Protocol |
| IPC | Inter-Process Communication |
| IRI | Inheritance Related Inversion |
| IRQ | Interrupt ReQuest |
| IRS | Inheritance Related Starvation |
| ISA | Instruction Set Architecture |
| ISM | Industrial, Scientific and Medical band |
| ISO | International Organization for Standardization |
| ISR | Interrupt Service Routine |
| ITC | Inter-Task Communication |
| JTAG | Joint Test Action Group |
| LAN | Local Area Network |
| LASER | Light Amplification by Stimulated Emission of Radiation |
| LBS | Location Based Service |
| LEACH | Low Energy Adaptive Clustering Hierarchy |
| LED | Light Emitting Diode |
| LIFO | Last In First Out |
| LP | Location Point |
| LQI | Link Quality Indicator |
| LoS | Line of Sight |
| MAC | Media Access Control |
| MCB | Memory Control Block |
| MCL | Memory Control List |
| MCLK | Main CLocK |
| MCU | Microcontroller Unit |
| MDS | MultiDimensional Scaling |
| ME | Median Error |
| MEMS | MicroElectroMechanical Systems |
| ML | Multilateration |
| MMS | Multi Media System |
| MMU | Memory Management Unit |
| MPU | Memory Protection Unit |
| MSAS | Multi-functional Satellite Augmentation System |
| MSE | Mean/Meadian Squared Error |
| MSP430 | Mixed Signal Processor 430 |
| NPP | Non Preemption Protocol |
| NPV | Negative Predictive Value |
| OOK | On-Off Keying |
| OS | Operating System |

| | |
|---|---|
| OSEK | Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug |
| OSI | Open Systems Interconnection |
| PC | Program Counter |
| PCB | Printed Circuit Board |
| PCP | Priority Ceiling Protocol |
| PCPE | Priority Ceiling Protocol Emulation |
| PHY | PHYsical layer |
| PIP | Priority Inheritance Protocol |
| PLR | Packet Loss Rate |
| POSIX | Portable Operating System Interface |
| PPV | Positive Predictive Value |
| PWM | Pulse Width Modulation |
| QPSK | Quadrature Phase-Shift Keying |
| QoS | Quality of Service |
| RADAR | RAdio Detection And Ranging |
| RAM | Random Access Memory |
| RAQ | Resource-Await-Queue |
| RCB | Resource Control Block |
| RET | RETurn (from subroutine) |
| RETI | RETurn from Interrupt |
| RF | Radio Frequency |
| RFID | Radio Frequency IDentification |
| RISC | Reduced Instruction Set Computer |
| RM | Rate Monotonic |
| RMS | Rate Monotonic Scheduling |
| RMSE | Root Mean Square Error |
| ROM | Read Only Memory |
| RR | Round Robin |
| RS232 | Recommended Standard 232 |
| RSSI | Received Signal Strength Indicator |
| RT | Real-Time |
| RTC | Real-Time Clock |
| RTS | Ready To Send |
| RTSJ | Real Time Specification for Java |
| RToF | Roundtrip Time of Flight |
| RX | receive |
| SDP | SemiDefinite Programming |
| SH2A | SuperH 2A |
| SHPER | Scaling Hierarchical Power Efficient Routing |
| SN | Sensor Network |
| SNoW5 | Sensor Node of Würzburg 5 |
| SPI | Serial Peripheral Interface |
| SRAM | Static Random Access Memory |
| SRP | Stack Resource Policy |
| STM | Software Transactional Memory |
| TAC | Test And Clear |
| TAS | Test And Set |
| TCB | Task Control Block |
| TCP | Transmission Control Protocol |
| TDMA | Time Division Multiple Access |
| TDoA | Time Difference of Arrival |
| TEEN | Threshold sensitive Energy Efficient Network protocol |
| TMC | Traffic Message Channel |
| TOF | Time Of Flight |
| TUF | Time-Utility-Function |
| TX | transmission |
| ToA | Time of Arrival |
| ToF | Time of Flight |
| UMTS | Universal Mobile Telecommunications System |
| UPI | Unbounded Priority Inversions |
| USA | United States of America |
| USB | Universal Serial Bus |
| VCC | Voltage of the Common Collector |
| VM | Virtual Machine |
| WAAS | Wide Area Augmentation System |
| WC | Worst Case |
| WCAT | Worst Case Allocation Time |

| | |
|---|---|
| WCET | Worst Case Execution Time |
| WCL | Weighted Centroid Localization |
| WCRT | Worst Case Response Time |
| WDT | Watch Dog Timer |
| WGS84 | World Geodetic System |
| WLAN | Wireless Local Area Network |
| WLOG | Without Loss Of Generality |
| WSAN | Wireless Sensor/Actuator Network |
| WSN | Wireless Sensor Network |
| YaOS | Yet another Operating System |

*«Sed fugit interea, fugit inreparabile tempus, singula dum capti circumuectamur amore.»*

*(Publius Vergilius Maro, Georgics 3.284)*