Njulla and GreenLab: Programming Mote-Class Devices in the Context of a Testbed Designed for the Energy Harvesting Regime

Aslak Johansen aslj@itu.dk

PhD Dissertation

IT University of Copenhagen

Advisor: Philippe Bonnet phbo@itu.dk

March 14, 2014

Abstract

This dissertation is the conclusion of joint work between the IT University of Copenhagen (ITU) and DELTA, a danish advanced technology institute, in the context of DELTA's strategic action on services in wireless sensor networks (WSN) for long-term environmental monitoring.

Wireless sensor networks are composed of mote class devices: small computers with severely limited resources in terms of memory size, processing speed, storage size, wireless communication throughput, and power budget. In most application contexts, mote class devices are powered from batteries or even wall socket. However, for long term environmental monitoring, it is necessary to source their power from energy harvesters. In order to allow DELTA customers to experiment with applications adapted to the energy harvesting regime, I designed and implemented the GreenLab testbed.

I found TinyOS – the dominant framework for mote programming in the last decade – to be a bad fit for implementing such a testing facility. I revisited the design choices and found several issues – mainly regarding component system, arbitration and hardware abstractions – that hinders the ease of debugging as well as the applicability of the framework. During its evolution, TinyOS has built a component system so dense that the application programmers rarely grasp the implications of their choices, yet those implications often propagate across component boundaries.

As an alternative, I present Njulla, a mote programming framework designed with ease of debugging as the primary design principle. I take the opportunity to reflect on the ease of debugging, and what it means to the mote programmer.

Acknowledgments

The last three years have been a great experience. DELTA has kept revealing new facets to their gamut of expertise. At ITU I have gotten a new insight into the educational system and started developing theories about it. Both have provided me with new friends.

My advisor – Philippe Bonnet – allowed me to follow my own paths, nudging me back on track each time I had followed a tangent too far. Somehow he always managed to know what to tell me. Often the feedback would seem immediately obvious once received. It would be delivered in a way that made me understand the underlying reasoning. I could gain an understanding. During my time at ITU I have supervised several groups of master-level students. Being a student of Philippe's has made me a better advisor for them, and for myself. I am immensely grateful for this.

Thanks to Dušan Vučković for many good conversations and taking the time to elevate my understanding of electronics to a usable level.

Jesper Schaumburg-Müller – my father – has played a pivotal role in these years. He would often help me make sense of all my new experiences.

Also thanks to Geoffrey Challen for advising me during my stay abroad. He, Suzanna and Choo-Choo hosted me during the first part of this stay and proved to be excellent company.

The remainder of the list is long and complex. My family and friends all did their parts in less direct (but often equally constructive) ways. I am amazed at the magnitude of offered support and interest. It was appreciated.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Context

This dissertation is the conclusion of my work as a PhD student at the IT University of Copenhagen (ITU) and DELTA[1]. DELTA provided most of the funding as well as the problem, ITU provided the academic framework. My time was split accordingly, resulting in around 2/3 going to DELTA and 1/3 to ITU.

DELTA is a Danish company with 70 years of expertise in electronics, light and acoustics. It exists as a conglomerate of earlier and more specialized institutes. DELTA is mainly a test and consultancy house selling their knowledge; primarily to the danish industry. In particular, DELTA designs and performs electromagnetic compatibility (EMC) and highly accelerated lifetime tests (HALT).

Their deep roots in the Danish technological history has earned them the role of a GTS institute. This is a Danish construction wherein private companies are awarded contracts by the Ministry of Science, Technology and Innovation to perform research and disseminate the resulting knowledge. There are nine such institutes; each covering different areas of technology.

In late 2009, DELTA was awarded a 3 year contract to offer services in wireless sensor networks (WSN) for long-term environmental monitoring. Danish SMEs[2] were the main target. It was decided to power the sensor nodes through ambient energy harvesting to eliminate the need for battery replacement. For this purpose solar, thermal and vibrational harvesters were to be considered. The focus was on an energy harvesting regime where the production is smaller than the sleep consumption of the hardware. An energy storage is thus introduced and sub-

[1] `http://www.madebydelta.com`

[2] SME is an abbreviation for small and medium-sized enterprises.

jected to charge-discharge cycles, often with extremely asymmetric components. Depending on the specific setup it may take weeks to charge and perhaps a second to discharge. The discharge is what powers the sensor node. To be cost-effective these harvesters would need to be scaled to produce just enough power. This lead to a need for precise testing facility, a testbed. Due to the environmental link of the harvesters this testbed needed to be located outdoors. So the overall question for my work was: *How should a testbed be designed, implemented and operated so that a range of different danish SMEs can experiment with long term data acquisition infrastructures based on energy harvesting solutions?*

1.2 Data Acquisition Infrastructure

In order to put this work in context, let us look back in the history of data acquisition. In the old days, those who wanted to understand a greater part of the world they lived in would make observations and scribble them down. Over time these records would provide them – as well as those who followed – with a body of evidence significant enough to formulate models of the observed phenomena.

Metrics are based on comparison with yardsticks. Temperature used to be metered using a fluid with a known coefficient of expansion, weight by comparison to agreed upon reference weights. Contraptions performing these comparisons with easily observable results are known as instruments and they produce the data we are so keen to acquire.

Field biologists would hike to their lake of interest and spend considerable time and effort creating what is essentially a time series dataset. Their tools were instruments for measuring their phenomena of interest (time being one of them), pen and paper. The process required much labor and was associated with multiple sources of potential errors; the instrument had to be *sampled*, the value written down, later read and finally used in calculations. The process was flawed, but the results were valuable, and the planning and execution did not require much theoretical background beyond that of the phenomena in focus.

The labor requirement was mitigated by the introduction of dataloggers. In the beginning strips were used for storage, either for stamped values or for graphs. Both removed some human error from the process. The graphs limited these to deployment errors but didn't result in a raw dataset. The relative ease of post-processing was traded off for a visual result that everybody could relate to. Over time the dataloggers have evolved with technology and today we can store digitally, sometimes even in open formats. This eliminated the need for transcribing the logs to a processable format. Because dataloggers operate autonomously they need to be tailored to a specific sensor or sensor interface. As a result today a wide gamut of dataloggers are in use to cater for the varying requirements of the sensors in use. Human errors are reserved mainly to deployment, but also result from the added complexities of having to deal with a layer of abstraction on top of the instruments sensor.

Over time variations over the datalogger theme evolved. Some added wireless capabilities, and some allowed for configuration of how and when to sample. While each of these variations have their place today, some have reached a state where they can be seen as regular small computers with programmable logic, storage and communication. The feature space and number of potential clever applications have exploded. They can be networked to make macroscopes; pseudoinstruments covering multiple sensor sites[67, 63]. They can perform adaptive sampling based on real-time processing of sensor readings[10]. They can monitor their own health and report on potential problems[3]. As such, a well thought out network of sensors may carry little to no maintenance and processing penalty. This shift opens up for much denser deployments given a low enough per-node price. Research has thus primarily been based on what has become known as mote-class devices[3]. These are computers of severely limited resources on all accounts. The constraints include memory size, processing speed, storage size, wireless communication speed and reliability, and power budget. They can be assembled to form wireless sensor networks.

1.3 Problem

The field biologist used to need little or no knowledge of her instruments to deploy them in the field. Likewise, facility management is used to deploying sensors or data loggers that they know nothing about to verify the quality of the indoor climate. With wireless sensor networks, the situation is significantly different. The sophistication of the instrumentation platform makes it hard to envisage a turnkey solution. So far, all sensor network deployments have required significant manpower in terms of design, programming and debugging (both at hardware and software levels)[4].

[3]While the origin of the term mote class is hard to track, the term mote was coined by Kris Pister in the Smart Dust project proposal [52], back in 1997. The importance of classes of computer systems was theorized by Gordon Bell [5], in 2007. Gordon Bell mentions the wireless sensor network class, which we now refer to as mote class.

[4]The programming culture has not been standing still while sensor networks have evolved. There has been a general trend of continuous lowering of the perceived level of programming languages. Today within most fields of computer science C is considered a low-level language. It didn't use to be. While electrical engineers are likely to continue regarding assembly language high-level computer scientists and – even more so – their students are on the move. This has to be considered when designing systems that are going to require maintenance for years to come.

TinyOS[44] emerged ten years ago as a programming framework for mote class devices and quickly became dominant in the wireless sensor network research community. Arch Rock was founded to commercialize turnkey instrumentation solutions based on mote-class devices programmed with TinyOS. While Arch Rock achieved very significant engineering results, it did not succeed in providing a one size fits all mote-based instrumentation infrastructure. The Arch Rock team has been integrated in the Smart Grid division of Cisco and thus develops solutions tailored to a single domain.

Still, the illusion of universality lives well in large parts of the sensor network research community. It is not rare to read papers where it is assumed that (i) mote-class devices form a uniform class of devices for wireless sensor networks (or the Internet of Things) represented by Telos-B or Epic designs, or (ii) that TinyOS is an operating system well suited for mote-class devices.

So, *should we rely on a TinyOS based testbed for DELTA's customers?* Or put in a more general context: *Is TinyOS a universal framework for programming any form of sensor network applications on top of mote-class devices?* Equally important: *How to efficiently support rapid prototyping for a range of different solutions for energy-havesting based sensor networks?* In other words: *How to make it efficient and easy to deploy and reprogram mote-class devices in the context of a testbed designed for the energy harvesting regime?* And in the first place: *How to characterize the energy harvesting regime for long term environmental monitoring?* These are the questions at the heart of this dissertation. More specifically, I focused on the following two problems:

- ***Programming Framework for Mote-Class Devices***: Is TinyOS well suited for programming any complex sensor network applications on any form of mote-class device? More interestingly, what distinguishes the sensor network regimes where TinyOS is a good fit from those where it is not? In general, what are the dependencies between hardware and operating system design? What are the dependencies between the application requirements and the mote design space? What are the specific requirements from the energy harvesting regime?

- ***A testbed for the energy harvesting regime***: A requirement from DELTA was that the testbed should make it easy to rapidly prototype new applications. I chose to focus on two complementary aspects of rapid prototyping on sensor networks: (1) the ease of debugging and (2) the ease of mote reprogramming:

    1. For a system to be easy to debug it needs to appear (and be) simple

In five years time how easy will it be to find a graduate student qualified to maintain complex C code? What can we do to lower the burden of code adoption?

to the developer. This simplicity depends on system complexity and observability. The complexity of a system depends on the involved components and the connections between those. Sometimes the complexity can be lowered by defining subsystems encapsulating – and isolating – subgraphs of this component graph. However, as long as the understanding of the whole depends on the understanding of a specific component, complexity is dependent on the observability of the internals of this component. The concept of subsystems trades observability for simplicity at the risk of providing a false sense of low complexity. Good tools can help a developer make sense of complexity, thus shifting the tradeoff. Examples include visualizations and execution traces.

2. Reprogramming a mote is the act of programming it by means of logic implemented on the mote itself. This is in contrast to the situation where a JTAG or similar programmer is used to stream an image to mote ROM. The ability to perform reprogramming allows us to perform remote updates of the software running on motes without the requirement of a linked programmer. This extends the solution space for experimentation on motes deployed in a remote testbed.

1.4 Approach

It is important to recall that both the problem I tackled during my thesis, and the approach I took were heavily shaped by the requirements and the constraints imposed by DELTA. My role there was to design and implement the wireless sensor network testbed. It was feared that instrumentation would affect the operation of experiments to such a degree that the results would no longer be representative of the reality. The approach was thus to limit the amount of instrumentation to a bare minimum. I designed GreenLab[35] – a prototype testbed – and implemented an initial version of it, relying on TinyOS for programming the mote class devices. Due to the combination of a dependency on Blip[31] and the use of a MSP430f1611 microcontroller (the first design was based on the Epic[16]), there was very little memory headroom on the devices. This caused DELTA to design a new board based on a different microcontroller of the same MSP family, i.e. the MSP430f5437, and I began porting TinyOS for the new board.

This task turned out to be significantly more time consuming than expected. The problems with TinyOS - along with the expectations of ports to a range of different platforms in the future – caused DELTA to look for other options. As a result DELTA changed strategy and I began to work on a new framework called **Njulla** to replace TinyOS for DELTAs purposes. The main areas of focus were portability, ease of debugging and performance. The goal was to reach a tradeoff which was better suited for the project than TinyOS had been. This goal was assumed to be within reach due to the narrower operational space associated with energy harvesting of sub-sleep power levels. This is not the story of the new framework, but it is the story of the thoughts and design decisions that went into it. The Njulla framework for DELTAs motes was built with ease of debugging as the leading design principle. The framework was designed, implemented, tested and used for experiments. In each of these phases we have learnt lessons that are reported in this manuscript.

I also designed GreenLab, a wireless sensor network testbed for the energy harvesting regime. What distinguishes this from other testbed scenarios is the degree to which experiment results are fragile regarding external influences on the power budget and – due to the long charge-discharge cycles – the lesser significance of the temporal overhead of the test apparatus. The first incarnation of GreenLab was implemented on top of TinyOS[35]. After the departure from TinyOS I reimagined GreenLab with ease of debugging in mind and implemented it on top of Njulla. How the ease of debugging fits into the testbed is described in this manuscript.

The method used is taken from experimental computer science[13]. Throughout the design, implementation and experimental work that constitute the core of this thesis, I have formulated hypotheses, and then directed effort at trying to confirm or falsify them qualitatively, or quantitatively when appropriate. Experimental apparatus is constructed around the system under test (e.g., the Njulla framework or the GreenLab testbed) and experiments are performed to observe the behavior of that apparatus.

The evaluation is rooted in the problem: Allowing SMEs to easily prototype energy harvesting sensor networks. For Njulla this is done qualitatively for portability and ease of debugging, and quantitatively for performance relating to the typical sub-sleep energy harvesting scenario: Boot from full off state, sample a sensor and send the resulting value over a radio link. To reveal less context-specific properties of the performance I also evaluate low-level pin and serial operation. For GreenLab I measure individual components involved in the operation of the testbed: Upload of image, switch of image and download of produced data. I evaluate this in the light of the typical use case of the testbed.

1.5 Contribution

The contribution of this dissertation mainly falls into three categories:

1. **TinyOS critique**. Based on my experiences porting TinyOS, and the experience of my research group at the IT University, I revisit the space of sensor network design regimes to distinguish those where TinyOS is a good fit from those where it is not. The lessons learned offer a new angle on the evolution and the impact of TinyOS, and some insights about the design of future network embedded operating systems aiming at large scale adoption.

2. **Design of the Njulla Framework and the GreenLab testbed**. With the design of Njulla, I took the option of a simple programming framework that could be easily ported and easily understood by new developers. Part of my contribution is the systematic description of the dependencies between hardware design and programming framework design. The goal of the testbed is to allow for experimentation on sub-sleep energy harvesting motes. With Greenlab, the key design decision was thus to limit interference with the power budget imposed by the energy harvesters. Accordingly, limiting the overhead of instrumentation was a key design parameter. This was accomplished by avoiding the use of a backchannel (e.g., an ethernet cable or a second mote).

3. **Reflections on the ease of debugging**. The ease of debugging comes down to the effort required – when presented with an anomaly – to track down the cause of this anomaly, come up with a solution for removing it and implement that solution. The main component of this is the process of gaining enough perspective to reason about the cause. In this thesis, I reflect on the elements of the programming framework design that make it easier to avoid anomalies, and to detect and correct those anomalies that cannot be avoided.

1.6 Thesis Structure

This dissertation is organized in two parts: The state of the art in Part I and the contribution in Part II.

The state of the art contains a description of mote-class hardware in Chapter 3. The goal is to give the reader a deep understanding of the characteristics of mote-class devices. In Chapter 4, I then review the characteristics of mote-class devices programming. I identify a set of trade-offs in the design of such programming framework and discuss the design decisions taken by two existing frameworks: TinyOS and Contiki[14]. In Chapter 5, I quickly review existing testbed solutions and focus on the reprogramming solutions that have been described in the literature. I conclude this part with a summary of the design decisions that DELTA had to take in terms of mote hardware, mote programming framework as well as testbed.

The contribution part is composed of a critique of TinyOS in Chapter 8, followed by the description of the Njulla programming framework in Chapter 9 and of the Greenlab testbed in Chapter 10. Last but not least, I discuss the lessons learned in terms of ease of debugging in Chapter 11.

The dissertation is concluded in section 12.

Part I

State of the Art

Chapter 2

Energy Harvesting

Energy harvesting – or scavenging – is the process of converting ambient energy into electrical energy that can be used for powering electronics. Electrical energy can be extracted from light (photovoltaic panels), temperature differences (peltier element), vibration (piezoelectric effect, electromagnetic generators), radio waves (antenna) and biological processes of plants[2, 29] (pH). Depending on the context one can choose to include the charge and kinetic energy emitted by isotopes[39] and microbial fuel cells[32]. The goal is to create perpetual deployments.

When used to power motes the output of a harvester can be classified based on its magnitude relative to the operational modes of the mote:

- ***Plentiful production*** The harvester generally produces more power than the mote – as well as any attached sensors or actuators – can consume. The mote can stay active over extended periods.
- ***Sub-active production*** The harvester produces less power than the mote consumes when active, but more than it consumes in sleep mode. The mote needs an energy storage and has to enter a sleep mode (or power off) to keep the storage from emptying.
- ***Sub-sleep production*** The harvester produces less power than the mote consumes while sleeping. The mote needs an energy storage and has to power off to keep the storage from emptying. Due to the low output of the harvester and the inefficiencies of the energy path it can take weeks to collect enough energy to do something useful.

This dissertation is mainly dealing with energy harvester setups that have sub-sleep production. Focusing on such energy harvesters was a decision from DELTA, on which I had no influence, and that I had to comply with. To support this,

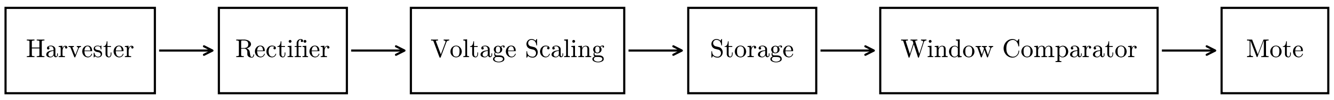


Figure 2.1: The generic energy path of an energy harvesting node.

electronics is added to supply power in bursts. This results in the whole system operating in charge-discharge cycles. Low-power modes can not keep the mote from running out of energy; only delay the cycle. Instead the mote needs to go through the boot process every cycle.

Figure 2.1 illustrates the energy path of an energy harvesting node. The harvester generates electrical power. Depending on the harvester used, this may need to be rectified. The voltage is then scaled to fit the operational envelope of the mote and stored. The operational state of the mote is controlled through a window comparator. Each of these steps will be briefly described in the coming sections.

2.1 Harvester

Transducers are used for harvesting ambient energy. Some produce a direct current while others produce an alternating one. Photovoltaic arrays generates a direct current. The configuration of the array determines the balance between current and voltage: connecting cells in series will multiply to voltage while connecting them in parallel will multiply the current. Some electromagnetic generators are essentially magnets suspended in a spring and wrapped in a coil. When momentum causes the magnet to oscillate, the coil observe a change in magnetic flux. This change induces an electric current which alternates with the oscillations.

Some sources are relatively stable (e.g., pH in a tree) while others depend on prevailing conditions (e.g., solar and wind). This brings a new set of challenges, mainly (i) how do you take advantage of significant in-network variations in available energy?, and (ii) If the cycle of each node in the network is highly dependent on the local conditions, how do you synchronize nodes?

A network of solar powered nodes could be programmed to route around shadows in order to shift loads to nodes with excess energy. UCLA are doing this with their Heliomote[54]. They are working on sub-active power production.

One solution to the synchronization problem is to operate in a star topology or have a backbone of nodes on stable power. Prabal Dutta's group has uses a similar approach where energy harvesting leaf nodes uses a real-time clock to define communication windows and use these if they have enough power[72].

2.2 Rectifier

For harvesters producing alternating current, a rectifier is needed. Without the charging will average to zero. For harvesters producing a direct current, the rectifier should be avoided.

2.3 Voltage Scaling

Some harvesters produce very small voltages and others produce high ones. These need to be scaled (DC-DC converted) so that they can be used to charge the storage to a sensible level. This level is within the operating conditions of the mote. This problem has drawn a significant amount of research from the perspective of electrical engineering.

Generally speaking, the solutions are using switching converters based on capacitors or inductors. The switched capacitor solutions are called step up/down converters. A step-up converter works by charging capacitors in a parallel configuration, reconfiguring them to a serial configuration for discharge and the starting over again. The step-down variant reverses the operation. Buck and boost are two popular basic topologies of inductor-based converters.

The operating efficiency of a photovoltaic array depends not only on the input conditions (temperature, light spectrum and -angle) but also the output conditions (the applied load). By adjusting the load it is possible to locate an optimal power point. This is the voltage/current combination resulting in the highest amount of power being transferred from the storage to the mote. However, the maximum power point depends on the input conditions as well. The situation is similar for other harvesters. Most of the research being done relates to tracking the maximum power point of harvesters, and scaling the voltage accordingly.

2.4 Storage

Rechargeable batteries have are subject to physical deterioration, which is triggered by recharge cycles. Although they are typically rated at a few hundred charge cycles, they can handle many more shallow cycles. For long-time deployments this is still considered a problem. Batteries also react negatively to low temperatures. Lithium batteries require 4.2V in order to charge. If the mote can be run on half of that, then it may be efficient to scale down the voltage between the storage and the mote.

Supercapacitors are an option. They are large-capacity capacitors at a comparatively small size. However, the leakage current is dependent to the energy level. This is a continuous flow of current to ground through the capacitor and represents the charging level needed to keep the charge.

The Prometheus[34] system uses two energy storages. The first one is a super capacitor and it is used to take the frequent charge cycles. The second is a Lithium battery used as a backup. Logic in software determines when to power the mote through the primary storage and when to power it through the secondary one. It also determines when to charge the secondary storage from the primary, thus limiting the charge cycles seen by the battery and extending its life. Prometheus is working on sub-active power production.

2.5 Window Comparator

The output of the window comparator depends not only on the input but also on the current state. This characteristic is known as hysteresis. The window has a high and a low threshold. If the input is above the high threshold then the state is 'on'. If the input is below the low threshold then the state is 'off'. If the input is between the thresholds then the state stays the same. The output mirrors the state so that the mote is turned on when the state is 'on'.

The input to the window comparator is the voltage level of the storage. The two thresholds obviously need to be within the operational range of the mote. Furthermore, as the mote is likely to be the most efficient at low voltages it makes sense to place the low threshold close to the low end of the operational range. The high threshold should span enough capacity to power the mote through the longest possible single cycle. It also makes sense to match the high threshold to the voltage scaling, as this will increase the efficiency of the first part of the energy path. Here, efficiency on one side of the storage is traded for efficiency on the other.

Work was done at DELTA by another PhD student to implement dynamic thresholds[66]. This work replaces the comparators with AD conversions by the main microcontroller. The microcontroller wakes up periodically, samples the voltage level and then decides what to do. Two scenarios are evaluated: One where the wakeup is from sleep and one where it is from off. After 6.2s it is preferable to turn off the microcontroller. With the microcontroller controlling the thresholds, it opens up for complex window setups.

Chapter 3

Mote Hardware

Wireless sensor networks are built using multiple mote class computers. This class is not officially defined, but the following properties apply to a wide range of existing motes[28, 7, 6, 16, 53, 38].

- ***Power Source*** Around 20kJ for a pair of AA batteries or typically < 5mJ for motes on harvested energy.
- ***Processor*** 16-32 bit microcontroller (MCU) running < 80MHz.
- ***Memory*** RAM+ROM of ≤ 512kB.
- ***Storage*** < 32Mb of storage on a serial connection.
- ***Communication*** A radio 1kb/s – 2Mb/s ranging up to several hundred meters, depending on conditions.
- ***Sensors Interfaces*** Support for analog and digital (SPI, I^{2}C, RS232) sensors.

The MCU has on-chip memory, timer, communication and analog-to-digital conversion (ADC) modules. Everything else is connected through pins. The following subsectioning is an attempt at describing mote class platforms in a general way. A few details may be specific to the TI MSP430 series of microcontrollers. Other implementations will be different, but comparable.

3.1 Power Source

Active silicon converts power to heat. Antennas converts power to radio waves. Leds convert power to light. The consumption of each hardware component is defined by its state. The radio can be sleeping, in receive mode or sending. Each have different power profiles. The send operation have a profile depending on frame size. Functional units consume power when active. Among others, this includes the USARTs and the ADC.

When operating on a wall socket most deployments dont care about consumption; there are plenty of other problems to spend resources on. Batteries have a fixed amount of energy. Conserving the availabe energy increases the time a mote can stay operational. Techniques to lower the consumption includes

- Keeping functional units off when not in use.
- Keeping external components off or in sleep mode when not in use.
- Keeping track of the lowest possible sleep mode and have the microcontroller enter it when it would otherwise spin.
- Duty-cycle the radio, typically by employing low-power listening.

When sub-sleep energy harvesting is employed the regime changes again. Energy harvesting promises limitless energy[1], but only delivers it at little power. A window comparator and some energy storage system is used to provide the energy at a useful level of power, but only for a short period at a time. This makes things more complicated. Each of these chunks are seen by the mote as a power cycle. The energy storage employed by the harvester needs to be scaled to match the maximal amount of energy needed within a single power cycle. An application could sample a sensor and store the result to flash and transmit it over a radio. A strategy of matching a sample-store-transmit sequence to a power cycle would consume some amount of energy. Having a strategy which performs a sample-store sequence for 9 cycles followed by a single sample-store-transmit cycle is not going to save anything: The storage still have to match the sample-store-transmit sequence. Instead, one could perform the sample-store sequence for 9 cycles and then have a load-transmit sequence. This would lower the maximum required energy to either that of sample-store or that of load-transmit, whichever is largest.

3.2 The Analog in Digital

Digital logic is implemented through abstractions on top of analog circuitry. It operates at a logic level (often 3.3V or 5V), meaning that digital high is coded

[1]Practicalities dictate otherwise.

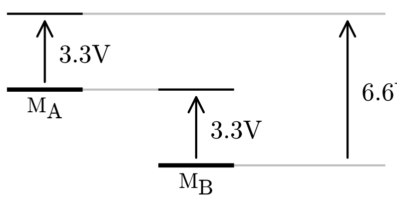


Figure 3.1: Anchoring of ground planes.

as this value and digital low is 0V[2]. In reality everything above some threshold is interpreted as high and everything below as low. The interface of an integrated circuit (IC) expects clear highs and lows on the inputs, and in return guarantees clear highs and lows on the outputs.

Those voltages are measured relative to a ground plane. 0V is in the plane. Two motes have different ground planes unless the two planes have been anchored to each other. Typically one connects the two ground planes in order to agree on logic levels. However, if motes M_A and M_B are both on 3.3V logic levels and the ground of M_A is connected to high on M_B, then M_B will see a high from M_A as 6.6V. This scenario is illustrated in figure 3.1.

A pin is said to be floating as long as it is not anchored to something well defined. When a pin floats its voltage is highly susceptible to outside influences. For this reason it is common to observe a 50Hz signal on floating pins (in 50Hz grid areas) when measuring equipment is attached.

A pin can be anchored, typically using a *pull-up resistor* (to logic high) or a *pull-down resistor* (to logic low). A resistor to anything anchored will do though. The idea is to weakly pull the line towards something. Logic can then easily pull the line in the other direction if needed.

3.3 Mote Channels

Each I/O pin is associated with a configuration that defines which and how functionality is exposed. The configuration has four properties, each mapping to a binary universe of values.

1. ***Route*** $Pin.Route \mapsto \{GPIO, FUNC\}$
   The universe of values contains $GPIO$ (for low-level usage) and $FUNC$ (for routing to a hardwired functional unit).

[2]This is the case for *active high* lines. The situation is reversed for lines which are *active low*. This property is called the *active state* of the line. Active low lines are usually marked with an overline (e.g. $\overline{\texttt{CE}}$ for 'chip enable').

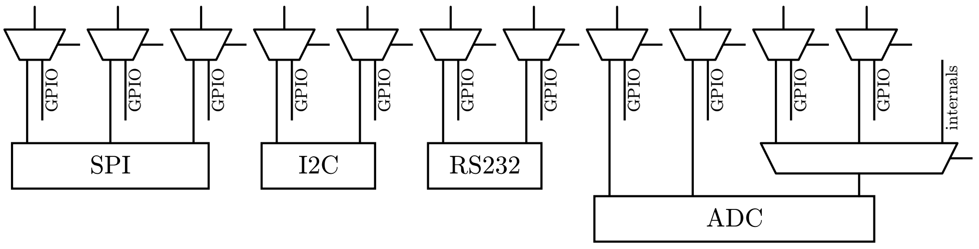


Figure 3.2: Examples of pin to functional unit routing.

2. ***Direction*** $Pin.Direction \mapsto \{Input, Output\}$
The universe of values contains *Input* (for reading) and *Output* (for writing). This is relevant only when $Pin.Route \mapsto GPIO$.

3. ***Interrupt*** $Pin.Interrupt \mapsto \{Enabled, Disabled\}$
The universe of values contains *Enabled* and *Disabled.* This is relevant only when $Pin.Direction \mapsto Input$.

4. ***Edge*** $Pin.Edge \mapsto \{Rise, Fall\}$
The universe of values contains *Rise* and $Fall$. It specifies the type of transition that should generate an interrupt. This is relevant only when $Pin.Interrupt \mapsto Enabled$.

Some pins are associated with a peripheral unit such as a serial block or an AD converter. This peripheral unit is controlling the pin when $Pin.Route \mapsto FUNC$. If instead $Pin.Route \mapsto GPIO$ then the pin is in GPIO mode and has a direction. When it is an *Input* pin the logic level can be sampled, when it is an *Output* pin it is set (pulled to logical high or low). While in *Input* mode it can be configured to generate interrupts on a rising or falling edge.

Figure 3.2 illustrates how pins can be routed to functional units. The top row of demultiplexers can be selected using Pin.Route. This allows each pin to be routed to a single (hard-wired) functional unit. Some pins are associated with serial communication, others with AD conversion.

A channel is a collection of wires connecting two or more hardware components. Peripherals such as radio, flash, sensors and actuators are connected to microcontrollers through channels. We use the channel layout as a way of characterizing the architecture of a mote. Section 3.7 will bring an example.

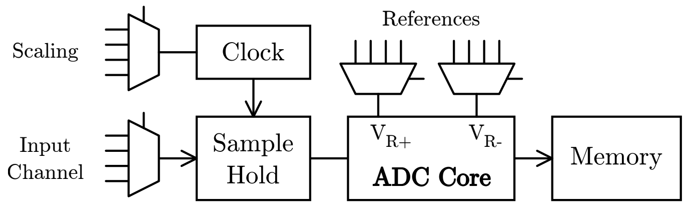


Figure 3.3: Conceptual model of an ADC unit.

3.4 AD Conversion

Digital is an abstraction on top of analog, where specific logic levels are used to separate the 1 from the 0. Sensors work by reacting to some analog property, usually by generating an analog signal which then have to be digitized to be processable to a digital computer. The process of converting an analog input to a digital output is called analog-to-digital conversion or AD conversion and is performed by a module called an AD converter, ADC in short. The reverse process is typically used for actuators and is performed by a digital-to-analog converter, a DAC. A cheaper variant of the DAC is a pulse-width modulation (PWM) engine. These generate a pulsed signal where the high-to-low ratio represents the analog value. The sharp transitions may cause problems with components or humans expecting a stable value.

AD conversion is essentially a two-step process. First a signal is *sampled* by connecting an internal capacitor to an input. Over time this capacitor will charge to the same level as the input. The minimum time required to reach this level is a function of the size of the capacitor and the output resistance of the source we are measuring. The sample time is configured by choosing a clock and scaling it. Disconnecting the signal will end the sampling process. The last step is to convert the sampled value to a digital representation and store this in memory. This conversion is done relative to a *reference*. In other words, a signal outside the reference frame will not be distinguishable from a signal at the edge of the reference frame. Most microcontrollers provide one or more internal references as well as an external one. Figure 3.3 illustrates a basic sample-hold-convert system.

There are different ways of performing the conversion step. The entire conversion could be performed in a single cycle by using a flash ADC, but that would require $2^n - 1$ comparators for n bit resolution. This represents a lot of space on the silicon, depending on n. A successive-approximation ADC perform what is essentially a binary search. A window representing the sample value is iteratively narrowed in by feeding the median value (representing the next bit) through a DAC and comparing to the sample. This solution takes up little silicon but is slow and – because of the iterative approach – fragile to a fluctuating reference.

Other methods exists providing different trade-offs between conversion resolution, sampling frequency, tolerance of reference fluctuations and required die area. A result of these constraints is that microcontrollers tend to have few (e.g. a single) ADC units.

Because of the limited number of ADC units present in microcontrollers it is a common practice to allow multiple pins to be routed to these units. That way many analog sensors can be connected and multiplexed at will. The multiplexing logic is relatively cheap.

3.5 Serial Communication

Serial communication can be accomplished by doing GPIO operations. A stream of bits have to be transmitted by one end and received by another. The transmission is done by setting a GPIO pin high or low. The reception is done by sampling a GPIO or detecting transitions on it. But how do we know when the line is active (that something is being transmitted)? Sampling the line will always give some value as it has a physical state. Either we add a protocol or we involve a second line. The protocol could escape the resting value or encapsulate transmissions in predefined patterns.

How do we know when to sample? Some mechanism for synchronization is clearly needed. A common agreement of bitrate solves the problem of separating individual bits, but the two ends need to have (and keep) synchronized clocks for this to work. Clock drift is influenced by temperature and even clocks from the same series will exhibit different drift under matching external conditions. Due to drift a synchronization mechanism relying only on bitrate will have a *bit error rate*. To minimize this one can add and sync on regularly occurring special bit patterns. This will also result in the two ends being initially synchronized. Another option is to involve a dedicated clock line driven by the master end. In this case an agreement of when to set the value (e.g. on rising edges) and when to sample it (e.g. on falling edges) is needed. Accordingly an active clock means that there is a transmission. The maximum clock speed is limited by the time the receiver needs to process the incoming data.

So far we have looked only at links between two ends, but the concept can be extended in a number of beneficial ways. A *bus* connects multiple endpoints. By extending the protocol of the transmitted bitstream one can target a named endpoint, thus eliminating the need for a bus master. That of course increases the complexity of the protocol-handling logic. This any-to-any model is rarely necessary and thus a simpler solution is often used where a single master is connected to multiple slaves. All slaves receives all transmissions but each slave

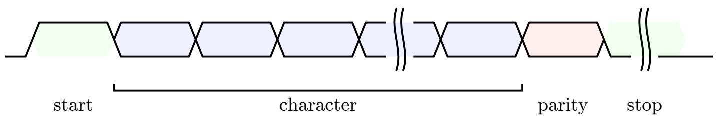


Figure 3.4: UART frame format.

is connected to the master with an additional *slave select* line. The state of this line determines whether the slave reacts to the data line(s). For this to work exactly one slave select line has to be active when requesting a response.

Most microcontrollers provide serial units to automate these tasks and hide some of the complexity. The smallest unit of transmission is a single byte, but sometimes bulk transfers (of larger amounts of data) are available. The following subsections will give examples of different types of serial communication. On some microcontrollers each functional block implements exactly one of these, on others they can be configured to implement any of a subset of them. The common property is that the number of these units is limited.

3.5.1 UART

The term UART is ambiguous as it is used as a general name for a serial functional unit as well as a specific protocol. In this section we refer to the latter meaning. The UART protocol has been widely used over RS232 links for bidirectional communication, and is the most common link between a mote and a general-purpose computer. Here it is adjusted to follow CMOS/TTL logic levels. The two ends have to agree on bitrate. In the most simple version two lines are used for full-duplex:

- ***master-to-slave*** Data flowing from the master endpoint to the slave endpoint.
- ***slave-to-master*** Data flowing from the slave endpoint to the master endpoint.

The unit of transmission is a *character* typically spanning 7 or 8 bits. The format has an optional parity bit to provide robustness towards bit errors. The two ends needs to agree on the semantic of the parity bit. To provide a means of synchronization the data in wrapped by a single start bit and a few stop bits, typically 1-3. The frame layout is shown in figure 3.4.

3.5.2 SPI

The Serial Peripheral Interface (SPI) bus is used to connect a single master endpoint to one or more slave endpoints. The bus employs four lines:

- ***SS*** The slave select line. Slave endpoints reacts to communication if and only if their SS line is active. One line per slave is required.
- ***MOSI*** The master out slave in line. This carries the bit stream from master to slave.
- ***MISO*** The master in slave out line. This carries the bit stream from slave to master.
- ***CLK*** The clock line. During each period a single bit is transferred in each direction. The master provides the clock. While slaves endpoints have a maximum frequency there is no strict minimum and there is no need for it to be static.

To support the mode of operation each endpoint has two registers; one for receiving and one for transmitting. At the beginning of the clock period one bit is shifted out of the transmit register and the outgoing line is set accordingly. In the middle of the period the ingoing line is sampled and the resulting bit is shifted in to the receive register. The unit of communication is a byte. As a result any response on the MISO line is delayed at least 1 byte. To deal with this it is common for higher-layer protocols to pad commands with blank bytes.

As with RS232 there are variations of the standard, but only two properties vary. The first is the clock polarity which defines the resting state of the clock line. A polarity of 1 means that the clock line rests in the high state and is thus active low. The second is the sampling phase which which defines which edge to sample on. A phase of 0 means that sampling should be done on the first edge of the clock signal. Figure 3.5 summarize these settings and highlight a single configuration.

3.5.3 I^{2}C

The inter integrated circuit bus (I^{2}C) is an any-to-any bus. Mote-class hardware usually use it for the one-to-any part of this quality as well as its low requirements regarding linecount. It only occupies two lines and adds some extra complexity to make it work. The lines are:

- ***Data*** This is a half-duplex line. For this to work all endpoints needs to agree on who is transmitting. A protocol is employed to target named

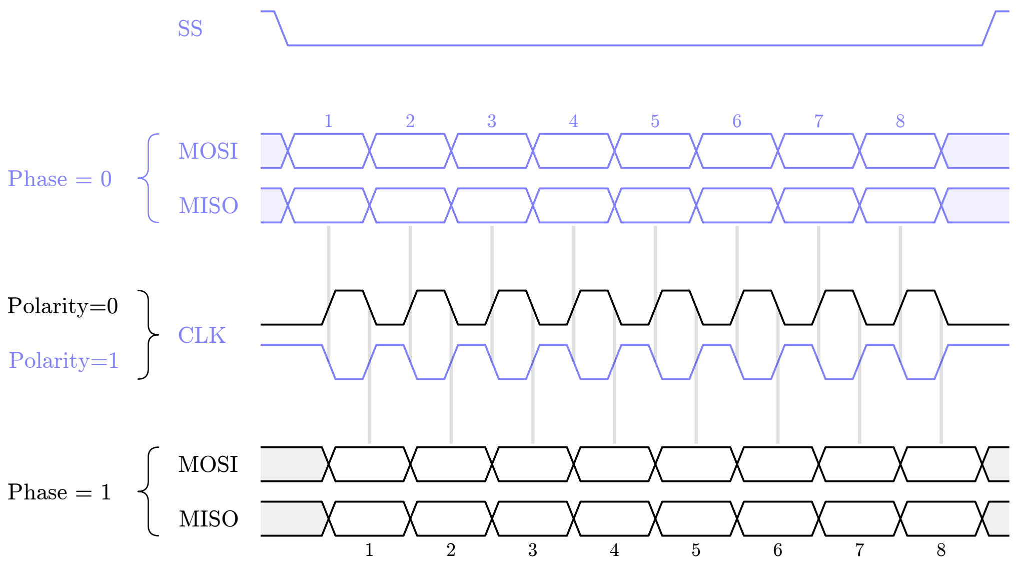


Figure 3.5: SPI line transitions. The $\{Phase \mapsto 0, Polarity \mapsto 1\}$ configuration is highlighted in blue.

endpoints. This provides the same functionality as the slave select lines from SPI.

- ***Clock*** One bit is transmitted on the ***Data*** line per period. Controlled by the active master.

In a single-master setup arbitration is easy. At the end of a request from a master the named slave is supposed to answer. In a multi-master setup all master endpoints employs two techniques to keep the line free of corruption.

- ***Carrier Sense*** Every master monitors the bus for start and stop bits. This way no master will begin transmitting while another is in session. For this technique to work there needs to be some time in between the two masters trying to start communication. When that is not the case another technique is employed:

- ***Feedback*** Two masters grab the bus at the same time and begins transmitting. If they don't transmit the same bit sequence – and this is the only case of interest – then at one point they are trying to pull the line in each direction. This will result in the line going to ground. The reason for this is that the line operates in *open drain* mode where a static pull-up resistor is used to pull the line high. The line needs to actively be set low to go down. When a master switches the pin to input (to weakly pull it high), it also samples the line. If the line is low then it knows that the line is occupied and backs off.

3.6 Peripherals

Many applications of motes come with severe restrictions on power. Sometimes it is even necessary to power cycle individual components by use of digital switches. They often toggle multiple lines so that parasitic currents – where part of the circuitry is fed through lines designed for signals – can be avoided. Functional units within the microcontroller can be turned off and sometimes external components (radio and flash) can be as well.

Motes should be small (to limit form factor and energy consumption) and cheap. For the MCU this translates to a small die and few pins. Both are reused where possible. Pins have multiplexed functionality and may need to be arbitrated. Many can be routed to a single AD converter. Serial units may support different types of communication but only be capable of driving one at a time. Multiplexers are digital switches which selects one among multiple lines.

Due to the way physical components are arbitrated to provide functionality it can be hard to reason about differences in mote architecture. In an attempt to alleviate this we introduce the concept of a *channel* representing the dependency of a bundle of lines. We furthermore describe mote architectures as graphs of links.

3.7 Case Study: TelosB

To highlight the consequences of the architectural choices regarding channel layout we analyze the TelosB[53], one of the most popular mote hardware platforms. In this example we focus one three functional blocks within the MCU as well as all major components on the PCB. The MSP430f1611 of the TelosB has a single ADC and two USARTs (each providing hardware support for a selection of serial buses). It also have limited clock and pin interrupt capabilities, but we refrain from involving these to keep the model simple. Figure 3.6 contains a channel dependency graph of the involved mappings.

In addition the the MCU the TelosB contains:

- ***JTAG*** A programming and debugging interface connected through GPIO pins.
- ***Humidity*** A humidity sensor connected through ordinary GPIO pins. The MCU has to implement a soft-I^{2}C protocol to talk to the sensor.
- ***Photo Sensors*** Two photo sensors connected through pins routable to the ADC.

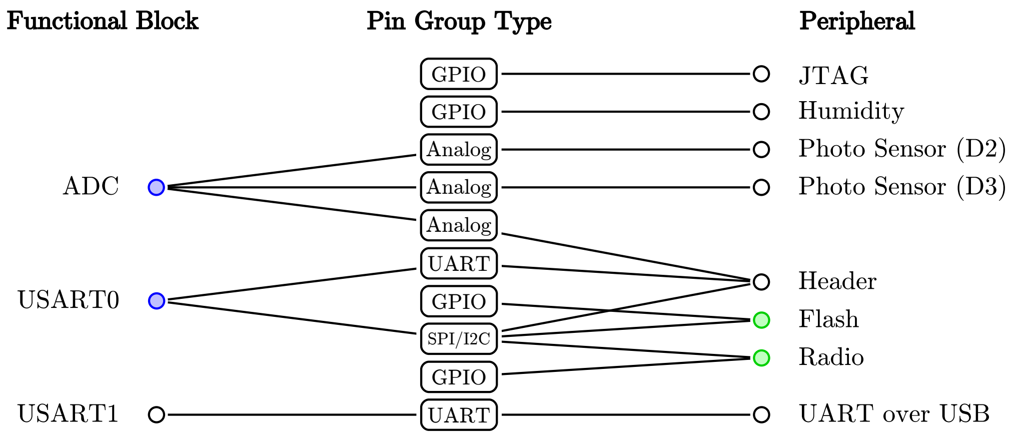


Figure 3.6: The TelosB channel dependency graph. Blue nodes can only operate a single edge at a time, green nodes needs all directly connected edges to operate.

- ***Header*** An expansion header with pins for serial communication (UART, SPI and I^{2}C) connected to USART0 as well as some GPIO pins routable to the ADC.
- ***Flash*** A flash chip is connected to USART0 through SPI and has a few GPIO lines.
- ***Radio*** A radio chip is connected to USART0 through SPI and has a few GPIO lines.
- ***UART over USB*** USART1 is dedicated to an FTDI chip providing a UART connection over USB.

The USART1 is only connected by a single channel and that channel is routed to a USB connector. The USART0 is used for all other serial peripherals. Sampling a serial sensor by means of an USART cannot be done independently of flash or radio operation. This suggests that the main use case for the TelosB is to be connected directly to a general purpose computer for streaming purposes, either as a gateway or an interface to sensors. In order to use it in sample-store or sample-send scenarios this channel would have to be arbitrated. Channel arbitration does not necessarily result in significant overhead.

The added complexity does mean that the programmer needs to keep track of which channels are active and which slices of code is waiting for it. This is essentially a dependency graph. In the general case care should be taken to avoid deadlocks.

Chapter 4

Mote Programming

A given mote must be programmed to provide the required behaviour in the context of a testbed, or more generally, in the context of a wireless sensor network application. In this section, I review how the traditional computer systems abstractions (communication, processing/interpreter and memory [57]) are handled on mote class devices. An additional dimension which is critical for mote programming is power consumption. I cover that in a separate section. I also dedicate a section to mote debugging, which is a key aspect of productive mote programming. I conclude this chapter with two example mote programming frameworks: TinyOS and Contiki.

4.1 Abstractions

4.1.1 Communication

On a mote, the communication abstraction relates to sensors and actuators (including radio and flash). There is a large selection of analog sensors. As we noted in the previous section, most motes have a single ADC which needs to be multiplexed. Some actuators are analog. Analog output is less common than analog input. Some microcontrollers have built-in DACs (Digital to Analog Converters), others only have PWM (Pulse Width Modulation) engines. The software interface could be the same.

Some sensors communicate results back to the mote almost immediately. Thermistors and light sensors belong to this category. Others require significant time to stabilize. The PPD42NS particle sensor uses the heat generated by a resistor to create a stable draft inside the sensor[64]. A beam of light will be reflected by particles in this draft. A light receptor is used to measure the reflected light and based on this give a value. It takes a static amount of time to warm up the resistor and it takes a dynamic amount of time (depending on magnitude of the observed phenomena) to accumulate the output of the light receptor. If a signal is small then the sampling has to cover a long period of time. Actuation may also take time, e.g. if something has to be physically moved.

We review the main mote communication abstractions in the rest of this section.

4.1.1.1 Serial communication

Serial communication is normally done one byte at a time. At high speeds it makes sense to poll for completion of each byte, at low speeds one has to trade off the overhead of polling against the overhead of interrupts. The overhead of interrupts are twofold, first it takes a few cycles to set up the interrupt and second – when the interrupt is produced – two context switches will be performed: To and from the interrupt handler.

Large transfers can be performed by a DMA engine, if the microcontroller has one. Here the situation is the same as the interrupt approach except that (i) only a single interrupt in involved in the entire transfer, and (ii) the overhead of setting up the transfer is significant. In practice this is only relevant for monologues.

4.1.1.2 Radio

From the perspective of the mote a radio is a combination of sensing and actuating logic. There is an interface through which commands can be sent. At the high level frames are sent to the radio, which then transmit them. Upon completion of the transmission the microcontroller is signaled through a GPIO. Upon reception the microcontroller is signaled and the frame is then transferred.

The radio can operate on different abstractions. Some radios are interfaced at a very low level. The frames have to be transferred one bit at a time at the right frequency and jitter on the lines can propagate to the antenna[45]. Other radios expose a command-based interface over a serial bus and buffer the frames. These interfaces are tolerant to imprecise timing in the code running on the microcontroller.

In practice, operating the radio is a bit more complicated as loops are involved. Before sending a frame it makes sense to do CSMA. This is a technique for significantly lowering the risk of frame collision by sampling the channel and waiting for a clear channel before transmitting. This involves polling the radio. If reliable communication is required then logic needs to be implemented to retransmit frames when no acknowledgements have been received.

4.1.1.3 Multiple Streams

Motes are often directed to perform several concurrent duties. A mote could be tasked with routing data over radio while sampling a sensor to flash while sending data from flash over radio.

The problem is that once an operation takes time there is – generally speaking – better things to do than polling. If other operations are active at the same time work could be performed on these to increase overall responsiveness. Otherwise the microcontroller could be placed in sleep mode to save energy. Strictly serial operation is wasteful. It is easy to debug though.

There is a need for an efficient mechanism that multiplexes these streams. Hardware resources tend to have lower bandwidth than what can be processed by the microcontroller. This makes them scarce resources which should be kept under load.

4.1.2 Interpreter

4.1.2.1 Interrupts

Interrupts are used as a means to avoid polling. There are two problems associated with polling; (i) one has to decide on a frequency or pattern thereby trading overhead against reaction time, and (ii) while polling the program counter needs to be active thus limiting the opportunities for the mcu to enter sleep states. For many applications, the event of receiving a packet via the radio is so significant that the system designer will try to minimize the reaction time. This would yield a very high polling frequency and likely make it infeasible to enter sleep states.

By configuring the pin in question to generate an interrupt on the condition change of choice we can avoid the polling. Instead, when the line changes state – signaling the arrival of a packet – the ordinary thread of execution is temporarily *interrupted* in favor of an *interrupt handler*. This is essentially a function whose address is stored at the right location within the interrupt vector.

A change in the line – or some other external event – triggers an interrupt leading to the execution of a predefined interrupt handler. The interrupt handler should implement the logic needed to handle that event. For the radio that would involve moving the received packet from the radio to the MCU, either by itself or by setting a flag, thus making sure that some other component services it.

Interrupt handlers tend to be time critical in the sense that they are hard to make reentrant and the interrupt may fire again after a short period of time. A UART link will send data at a fairly consistent rate. At 9600 baud with 1 startbit, 8-bit character, 2 stopbits and no parity the transmission rate is 872B/s which means that the mcu has less than 1.1ms to deal with the associated interrupt. If this timeframe is exceeded we will start to lose bytes. To avoid reentering interrupt handlers it sometimes makes sense to temporarely turn off interrupts. This option offers consistency at the price of a potential loss of data. Data loss only happens if two interrupts of the same type happens while interrupts are off.

4.1.2.2 Multiple Flows

The mote may have to perform multiple roles. In a networked scenario it may have to take part in multihop routing while performing some actions. Three flows of data can easily be imagined; (i) radio-to-radio routing, (ii) sensor-to-flash sampling, and (iii) periodic flash-to-radio offloading. The resources become potential bottlenecks and it is thus important for the resource to be ready to service the next flow by being able to react quickly to an interrupt. This would also result in a low latency which – in the case of duty-cycled resources – translates to power savings.

Some operations take a significant amount of time to complete but have no dependency on the program counter. The most simple implementations will poll for completion thus blocking the main thread. This essentially serializes all communication with resources. Non-blocking implementations will split up the initiation and completion phases of the operation. In the initiation phase the lower level command is issued to the resource (e.g. configure ADC, start conversion). In the final phase at completion of the operation control is returned to the caller which can then continue work, potentially initiating another long-running operation. The completion event is either polled or signaled. Such split-phase implementations trades simplicity for performance.

Some systems uses events to model flow. One essentially builds an event graph with nodes representing fragmented flows (potentially with associated state) and edges representing events or calls. Figure 4.1 exemplifies this for both polling and signaling. The graph can be either static or dynamic. If the graph is static the same function will handle all callbacks of a given type. If this type is associated with multiple flows then the handler will have to implement a state machine. If the graph is dynamic then it will be updated in both the initiation and the completion phases. When an interrupt is generated the current version of the graph will define which handler is called. To ensure consistency it may in some situations be necessary to disable interrupts while operating on this graph.

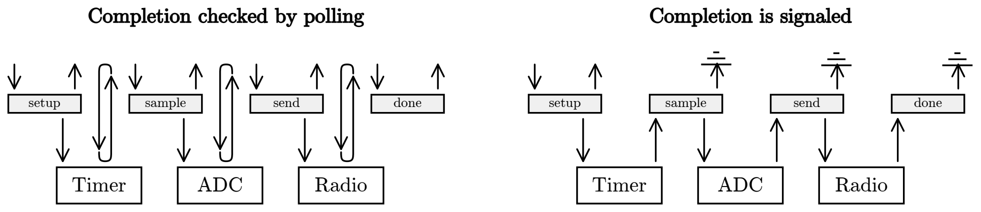


Figure 4.1: Two-phase fragmented flow example.

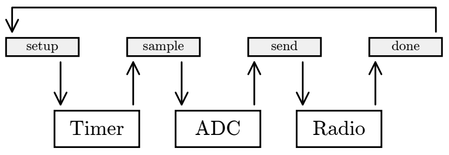


Figure 4.2: Threaded flow example.

Making operations non-blocking is a way of dealing with the limitations of having a single thread of execution. As this suggests there is another solution: To introduce threads. Most flows can be modeled as a single thread on top of blocking operations. Figure 4.2 exemplifies. Complex flows – like ones sampling multiple sensors simultaneously – will need multiple threads and thread synchronization mechanisms though. The mental model required for programming a threaded framework will often be far simpler than the comparable model for a two-phase framework. However, this comes at the price of complexity in the framework scheduler (e.g. thread context has to be shifted). Furthermore, typical threads needs one stack each, thus significantly increasing the memory consumption. Protothreads – as explained in section 4.5 – try to find a middle ground.

4.1.2.3 Arbitration

When multiple – independently engineered – software components are mapped to the same resource the need for exclusive access arises. Exclusive access gives us the ability to design a component without knowing about the internals of others.

If exclusive access is to be ensured resources need to be arbitrated among components. This is the job of an *arbiter*. Components requests access from the arbiter, wait for the resource to become available, perform the intended operation and release the resource.

This approach depends on the components releasing the resources immediately. It also has a more complex dependency. Since we assume that the components are unaware of the internals of each other there is no reason to assume that no circular dependencies exists. Component A may have code acquiring resource R_A followed by R_B while component B concurrently is acquiring resource R_B

followed by R_A. At certain serializations this will result in a deadlock. Even worse; this might happen rarely. If so, this bug will be very hard to reason about (one does not immediately consider the internal workings of seemingly unrelated components) and it won't be reproducible. Two things that hinder the debugging.

There are two immediate solutions to the circular dependency problem. The first is to simply only allow a single resource to be held by each component at any point in time. The second is to introduce two-phase resource allocation. The resources are enumerated. At no point in time is a component allowed to request a resource with a lower number than the maximum number within the set of already acquired resources.

The job of arbitration can be delegated to a virtual resource, wrapping the real one and exposing one interface for each relevant component. The components then have exclusive access to their interface and the virtual resource multiplexes the access.

4.1.3 Memory

Memory can be allocated and released dynamically at runtime. This approach is popular on general purpose computers as it doesn't require the programmer to concretize the dynamics of their program: On a mote, memory would allocated for each object produced and released when it has been sent to another computer. If the bandwidth of the link temporarily becomes slower than the data generation, then the number of allocated slices increases until the bandwidth once again exceeds the production. If the bandwidth stays down for too long then the memory will fill up until the point where there is no more left to allocate. Every site of allocation not handling this situation gracefully represents a bug. Programs are simple to write using dynamic allocation, but hard to make robust.

The alternative is to statically allocate memory at compile time. This requires the programmer to bound all dynamic properties. One could allocate space for 512B of data and use this as a buffer in producer consumer fashion. If the bandwidth drops for too long the buffer fills up, and the producer is forced to stop inserting. The programmer is forced to resolve the allocation trade-off (little overhead vs flexibility). Dealing with lack of space then becomes an obvious problem that the programmer is forced to deal with on an everyday basis. Space on the other hand is mapped to a single use.

4.2 Power

In terms of performance what we care most about is energy, which means that time is of relevance. A LED is around 2mA or 6mW on 3V. That may not seem too significant, but if used carelessly it could become so. If the LED is on half of the time it is going to consume 3mW on average. The AT45DB flash consumes around 15mW while reading. If it is doing so for 1/100 of the time its average consumption becomes 0.15mW or about 20 times less than the LED.

Every component on the the node consumes power. The level of consumption depends on the mode of operation and the current activity. Writing to ROM consumes significantly more than AD conversion or additions. Microcontrollers typically have several low-power modes where the program counter is stopped and different triggers for activating it are available. This includes timers and interrupts. The deepest low-power modes does not retain memory. Many WSN applications are active for a fraction of their deployed time, meaning that the consumption during the inactive period is defining for the energy consumed. In these situations using the low-power modes and choosing the right one will be key. David Culler talks about the importance of *"doing nothing well"* [12].

The MCU is only one component though and it is not even one of the highest consumers. Both the radio and the flash are more expensive in this respect, at least when active. Radio's have different profiles for sending, receiving and sleep. Flash likewise for its operations and states. Sometimes it may make sense to have whole components behind digital switches to avoid even the sleep drain. That off course means that the duty cycling will need an extra step. Marcus Chang found that a secondary low-power radio can be used to improve the latency and duty-cycling of the primary radio[9]. This comes at a price of additional components and complexity.

4.3 Debugging

The goal of mote programming is to end up with an application that solves a specific problem. The application falls short of reaching this goal when its actual behavior differs from the expected. This is known as an anomaly and its cause can be rooted in any of the following reasons:

1. ***Bug*** Code which does not implement the desired logic. Bugs occur largely when the programmer is under pressure. One aspect of this pressure is the mental overhead of working with the programming framework. The programmer is assigning mental resources to grasp the structure and semantics of the framework and use the remaining to write the code. Mistakes happen

when the programmer has too few mental resources left for programming (because she has to juggle with the complexity of the underlying framework), or when the programmer makes simplifying assumptions about the underlying framework (in order to limit the mental resources she spends harnessing it). The time and resources required to track down an existing bug depends strongly on the correlation that exists between the observed anomaly and the bug.

2. ***Mental Model*** A flawed mental model results in code implementing logic with wrong semantics. Flaws in the mental model result from a lack of understanding of the underlying models of environment and platform. Based on prior experiences, study and argumentation the mental model is gradually build and refined. It has weak areas where the system designer relies on expectations and it may inherit flaws if it was built on flawed theories or uncharacteristic experiences. Fixing a flawed mental model means coming to terms with a conflict between the model and the reality. In this process other parts of the model – in particular the surroundings of the flaw in question – will be scrutinized to determine the extent of the conflict. Here, the documentation problems become critical.

3. ***Environment*** Environment differs from expectation, including

    (a) ***Hardware*** Hardware faults. Hardware faults happen due to a combination of construction, external influences and operation. Cheaply made components are more likely to develop faults. Badly packaged components – or entire motes for that matter – are more likely to develop faults. The fault rate of flash chips increase over time due to usage. In less severe cases faulty hardware can be wrapped in redundancy. Health monitoring can be employed to keep such hardware under observation. In more severe cases the hardware needs replacement. People with a background in computer science often assume that the hardware works correctly and base their mental model on this. Among the things they fail to consider is that the printed circuit board (PCB) may not reflect the schematics and that – due to bad soldering – the physical mote may not reflect their perception of the PCB.

    (b) ***Operating Conditions*** Operating conditions outside expected range. Operating conditions fall outside the expected ranges due to wrong expectations or loose assumptions. The results can be temporary or permanent. Components can suffer various degrees of damage which may result in sporadic anomalies. These conditions typically trigger hardware faults or reveal flaws in a mental model. Extreme conditions are often noticed by humans and thus tend to be easy to diagnose.

The ease of debugging comes down to the effort required – when presented with an anomaly – to track down the cause of this anomaly, come up with a solution for removing it and implement that solution. The main component of this is the process of gaining enough perspective to reason about the cause.

When a (critical) anomaly occurs the programmer has to classify and fix it. We call this *debugging*. Often the programmer starts out by assuming that the cause is a bug, and may revise this assumption of category depending on the resulting analysis.

Active ease of debugging is the ability to write anomaly-free code, thus avoiding the *need* to debug. Passive ease of debugging is the ability to make the act of debugging simpler once an anomaly has manifested.

4.4 TinyOS

TinyOS was originally developed at UC Berkeley by David Culler and his group in 1999. TinyOS matured in the context of the DARPA NEST project, and evolved in an open source effort, even though the efforts to spawn a TinyOS alliance to organize an open-community effort never fully materialized.

4.4.1 Programming Model

The TinyOS framework and applications are written in the NesC[22] language. NesC is an abstraction build on top of C. In fact, the compilation is performed by invoking a script which cuts up all named NesC code into small pieces, performs full renaming to avoid naming collisions, pieces the result together in the form of a C file and then compiles it.

In NesC, programs are formulated as component graphs. Some components are themselves defined as graphs. Each component encapsulates state and is associated with a set of ingoing as well as outgoing interfaces for executing logic. For a program to be valid each incoming interface must be connected to an outgoing interface of the same type on another component. This programming model is alien to most programmers with a background in computer science. The presentation of a set of design patterns[21] helped ease the transition to this model.

Memory is allocated statically. This choice trades a consumption overhead for stability. It also trades immediate simplicity for long-term simplicity as few will be accustomed to static allocation while all will be forced to think about limitations.

4.4.2 Hardware Abstraction

A form of Active Messages[61] is used as the main network abstraction. The message itself is associated with an identifier for a handler on the receiving side. Upon reception the handler is looked up and and the message is passed to it as an argument. TinyOS provides mechanisms for transmitting active messages over both radio and UART links. This lowers the complexity involved in gateway programming.

An abstraction allows code to be written once for any of the supported radios[45]. The CC2420 stack is essentially a chain of layers, each consuming an active message interface and providing another. These layers implements features like low power listening, retransmissions and duplicate package detection. By overloading the main configuration the application programmer can enable or disable existing features and add new ones. This approach trades speed for flexibility and portability.

In 2004 a three-layer hardware abstraction model was described[26]. The lower layer should expose the basic means of communication provided by the chip and be completely stateless. This includes register operations. The middle layer builds useful platform-specific abstractions on top of the lower layer. The top layer adapts these to less efficient platform-independent abstractions. Code reuse is possible at every layer and conceptually the layers are well defined. This does result in a lot of complexity though. To trace execution of a write to a digital pin one has to traverse through all these layers.

No common interface exists for analog output. As such, every platform supporting it has their own interface definitions for this. Those that have both DAC and PWM capabilities do not have a common interface. This is likely a result of TinyOSs focus on sensing.

Most components representing hardware devices are associated with an interface for starting and stopping the component. This is used for initialization and control of the device sleep status. It could also be used for turning on and off the device entirely. I am not aware of any TinyOS hardware which supports this though. However, this decision lies in the hardware designs, where – in this case – power is being traded for less hardware components.

The MSP430 code expose the DMA engine to the upper layers, but does not make use of it for normal operations. Operations where it would make sense to pay the setup cost to win in the long run are rare. Incorporating logic to dynamically choose strategy would add a constant overhead and summed up this would likely outweigh the benefits of being able to use DMA through the standard interface.

4.4.3 Execution Model

Execution in TinyOS is made up of event-based code and mutually atomic tasks. Tasks are intended for long-running operations. Hardware interrupts trigger event-based code and interrupts whichever code was being executed at the time of the interrupt. Tasks are enumerated and posted to a queue by setting a flag in an array matching the number of tasks. When no event-based code is active the sheduler will start processing tasks from the queue. When the queue is empty the scheduler will let the microcontroller enter sleep mode.

NesC provides the `atomic` keyword to disable interrupts on a block level. It is recommended to keep these atomic sections as short as possible to (i) increase concurrency, and (ii) ensure that few interrupts are lost. Keeping the atomic sections short often leads to significantly more complex code.

TinyOS is designed around two-phased semantics and a simple event system for serializing multiple concurrent operations. These are two independent design choices, although the use of two-phased operations does suggest event-based multiplexing. The event-system is simpler to implement than threads, but more complex to build applications on. The two-phased semantics is likely a result of choosing the event-based approach. With it comes complexities such as a broken flow and the potential need to implement state machines in the last phase.

Most interfaces are two-phased and – at least for the MSP430 – implemented through interrupts. This approach allows for concurrent operations.

4.4.3.1 Virtual Machines

As a result of the whole-program optimization done in the compile process it makes little sense to do less-than-whole program reprogramming. A full program image has to be transmitted, and this takes time and energy.

Maté[41] addresses this by implementing a small virtual machine on top of TinyOS. The program size is significantly shorter and is independent of any specific location generated by the TinyOS build system. Execution is obviously significantly slower when implementing logic, but for longer two-phased operations it is almost the same. For a small number of executions the total energy budget comes out in favor of the virtual machine. It is also suggested that the safe execution environment of a virtual machine can be used to implement boundary between user- and kernel layers on motes without hardware protection mechanisms.

One of the drawbacks of Maté was the fixed instruction repertoire. Generally, an instruction repertoire on a high level will achieve higher code density and reduce overhead at the cost of flexibility. Application specific virtual machines were proposed to target this by delegating the choice to the application developer[42].

4.4.4 Resource Allocation

Before accessing a shared resource access must be requested and this request must be granted. After use the resource should be released to allow other components to access it. TinyOS even provides the option to request the immediate release of a resource from the component holding it. However, it is only implemented by the SPI code of the MSP430 chip and never used.

Timers, DMA and pin interrupts all require a minimum sleep level to stay active. TinyOS uses the knowledge of resource use to track the minimum required power level so that the sleep mode can be chosen optimally. The execution overhead involved in this is likely to often be significantly less than what can be gained in power consumption. The solution is simple and portable.

4.5 Contiki

Contiki was developed at SICS by Adam Dunkels in 2003[1]. It quickly evolved into an open source effort.

The two defining features of Contiki are (i) the ability to perform dynamic loading and unloading of code at run-time, and (ii) the possibility of running multi-threaded code atop of an event-driven kernel. The framework seem to be designed as lightweight as possible without sacrificing these features.

Contiki is build around protothreads[15] or more likely, the two were designed with each other in mind. Protothreads are described as a response to the two-phased callback model used in TinyOS. In TinyOS the two-phase model was chosen due to the apparent requirement of threads for per-thread stacks. The price was a complex programming model that often required one to implement state machines not obviously inherent to the application. Protothreads uses ordinary C macros to hide the state transitions behind – what appears to be – blocking calls. In reality this call registers the protothread and a state id within this as a listener for some event. Context switching is done by stack rewinding (the function returns) and and thus the stack frame is lost. This is not obvious from the programming model abstraction. Note that a comparison of Contiki to TinyOS can be found in [56].

[1]See `http://www.contiki-os.org/community.html` for the history of Contiki.

4.5.1 Execution Model

Contiki has an event-based kernel. It contains an event scheduler which dispatches events to processes. It also periodically calls all processes' polling handlers. This ensures a level of fairness amongst polling – and non-polling – processes.

Some events are asynchronous, meaning that the execution is deferred. Others are synchronous and thus processed immediately. Having this choice makes it possible to resolve the latency/overhead trade-off in multiple ways, depending on use.

A Contiki process is either an application or a service. It is implemented as a protothread. Inter-process communication is done through a stub residing within the kernel. This stub acts as an abstraction allowing concrete implementations to be loaded and unloaded.

4.5.2 Reprogramming

Services can be loaded by registering function pointers in the stub. Before doing this a version check is performed to ensure compatibility between stub and service. The stub is initialized in a way that triggers a service lookup upon first access. For later accesses this is cached. Unloading a service will reinitialize the stub. All of this happens transparently to the consuming processes. Besides the extra overhead of the first call every call has to go through the kernel.

4.5.3 Preemptive Multitasking as a Library

With Contiki comes a library implementing preemptive multi-threading. Each thread is allocated a stack and a process implements the preemption using the timer interrupt. Threads can be loaded and unloaded dynamically and runs concurrently with processes.

4.5.4 Portability and Abstractions

There is a separation between platform dependent and platform independent code. The platform independent code covers boot code, device drivers, task switching and program loader.

The only abstraction provided by the base system is the CPU multiplexing. It is believed that abstractions are better implemented as libraries or services.

Drivers and applications communicate directly with the hardware as no hardware abstraction layer is provided. This solution delegates complexity away from the base system. That makes the base system simpler at the expense of the receiving code.

Chapter 5

Sensor Network Testbeds

A testbed is a testing framework in which a subject can be suspended for analysis. It shields the subjects from some external influences while subjecting it to others. The level of shielding is chosen mostly as a trade-off between reproducibility and realism.

Mote testbeds usually spans multiple motes to support the networked aspect. Experiments are described as mappings from mote ids to program images.

DELTA wished to deploy their testbed outdoors in order to confront the problems that one might face in an actual deployment (e.g., variations in the environment affecting energy harvesting and communications, geographical dispersion of the motes impacting overall performance as well as maintenance). Most existing testbeds are located indoors. We also wish to experiment with sub-sleep energy harvesting, so we want to minimize the impact of the testbed infrastructure on mote operations at experimentation time.

By its very nature, a testbed must be instrumented to manage the network of motes and to manage experiments. Normally, a backchannel is used to connect the testbed management system to each mote. This backchannel is used to perform administrative duties on the motes. Common operations supported by the backchannel include starting, stopping and reprogramming of the motes as well as different kinds of instrumentation.

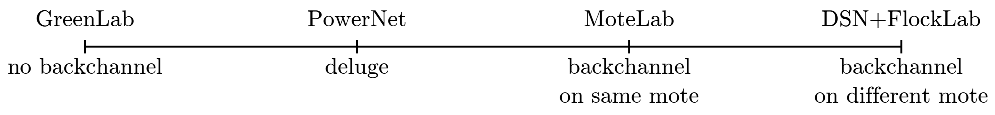


Figure 5.1: Levels of instrumentation among mote testbeds.

Figure 5.1 illustrate the trade-off. At the extreme right we see testbeds where each mote is instrumented by wiring it to another mote which is part of a secondary network supporting the backchannel[18, 46]. This level of instrumentation allows for measuring power consumption and injecting sensor readings. Motelab represents a group of testbeds which have a direct link (JTAG or similar) to each mote. This allows for reprogramming and resetting each mote independantly. PowerNet represents the software-instrumented testbeds where an always-running service provides a bridge to each mote over the wireless network. GreenLab does not require a backchannel and thus minimizes the impact of instrumentation[35].

5.1 Existing Testbeds

In Re-Mote[73] experiments are started when possible: Any user can grab any unused mote at any time and has to release grabbed motes manually to allow recycling. Each mote is programmed separately to allow for arbitrary *mote* $\mapsto$ *image* mappings. Motes which are under control of the user can be started, stopped and reprogrammed. A bidirectional serial connection is also provided. The user needs to keep the connection for the experiment to persist. The power consumption is not monitored.

In MoteLab[68] experiments are scheduled. Jobs are activated by all motes being reprogrammed using a single TinyOS image. While running the user who added the job can interact with it. Job data is transmitted using `SerialForwarder` introspected and stored in a database with tables whose schemas match the transmitted structs. A job takes up the entire testbed or partition in case an administrator has partitioned the testbed. Motes are powered and current consumption is logged.

Kansei[19] takes a hybrid approach to the concept of testbeds: It combines elements from simulation with elements for a physical deployment. The concept is to use TOSSIM[43] as the basic engine, but instead of simulating radio transmissions they are performed on a real grid-topology testbed. This is done automatically and has the benefit of only requiring a small number of real motes when simulating a large-scale network while maintaining a high level of realism. Sensor-readings can be injected at will through a similar mechanism and infrastructure is provided for recording real-world events for this purpose.

SensLAB[8] is a project funded by the French National Research Agency to build a large scale open wireless sensor network platform. The testbed currently consists of 1024 motes split across 4 sites. Each site has a three motes high 3d grid topology and a single robotic train. Each mote is instrumented using a (similar) mote connected to a control network. This allows for real-time monitoring of energy consumption and radio activity, and to some degree the injection of noise and faults.

PlanetLab[51] is an attempt at a planetary scale federated testbed design. It is currently comprised of 1000+ nodes at 500+ sites across the globe. Research projects are given a slice of the total testbed in the form of virtual machines on a subset of the nodes. The virtual machines are connected by an overlay network which is subject to the underlying networks conditions. Code running in a slice will experience realistic network congestion and failure. Contributers can limit the amount of time-slices offered, thereby lowering the price of providing resources. PlanetLab support distributed software services, but it does not incorporate mote class devices in its testbed.

PowerNet[36] is a deployment with testbed-like characteristics. The aim is to characterize the energy consumption of enterprise-style computing infrastructures. Ordinary workspace equipment is powered through motes with the ability to measure consumption and control on/off state. The deployment uses Deluge to push updates to the network.

In the Hogthrob project a platform had to be chosen for sow monitoring. A testbed vas constructed to experiment with different aspects of soft- and hardware codesign[65]. The design included a microcontroller, an FPGA and interfaces for a sensor board and a communications board. Different sensors and radios could be connected. TinyOS was running on the microcontroller. The idea was to flip the role of the MCU, to program the FPGA at boot to implement a microcontroller from a selection. This was to facilitate experiments moving functionality across the traditional soft/hard border.

5.2 Reprogramming

An experiment on a mote testbed involves the execution of experiment-specific code on each of the subject motes. Depending on the experiments one intends to perform, the code to mote mapping may be one-to-all, many-to-many or many-to-one. For most purposes practicality dictates that the reprogramming needs to happen automatically.

5.2.1 Unit of reprogramming

Depending on the type of experiments if may not be necessary to reprogram entire mote images. If the experiments are intended to compare different radio stacks, then it may me enough to simply reprogram this component, saving time and thus energy in transmission. In extreme cases we might only need to reprogram a specific function or even a global variable. We call this the unit of reprogramming. Whole-image reprogramming covers the general case and all smaller granularities involve complexities. How do you replace a component or function with another that is twice the size? Typically you would bend your framework design to support it and thus increase the complexity. Erlang has a mechanism for this[4].

A small unit of reprogramming is practical for *patching* deployments. It doesn't come free though. Indirections has to be inserted to channel access in predefined ways. Also, updating multiple units is complicated as they need to make up a valid program image at any point in (execution) time. This would likely be implemented by disabling all services, applying all new units and reactivating the services. The alternative would often require some units to be compiled for multiple of the intermediary shapes of the total image. This is not feasible.

Transmitting deltas of whole-images using Rsync has been shown to increase transmission cost significantly[33]. The process is to first have the target mote copy its program image to flash. Then – based on prior knowledge of this image – a delta is constructed and transferred to the target mote. Here it is applied to the flash image and then the flash image is loaded.

5.2.2 Time of reprogramming

In TinyOS the Deluge installs a small bootloader which does the reprogramming at boot time. At each boot it updates some state counting the number of failed boots. If this number exceeds 3 then a failsafe image is loaded. The main benefit of this approach is that at boot the mote starts execution at a static component. This component decides which image to load. If the loaded image does not have a minimal amount of functionality spinning it will increase the counter. As long as (i) reboots happen, and (ii) neither the bootloader nor the failsafe image has been compromised, the system will transition towards a safe state. NWProg from Blip uses the same mechanism.

The alternative is to perform the reprogramming at runtime when needed. This approach saves time (and thus energy) during boot at the expense of fault resilience. The bootloader used by deluge is quite isolated and thus does not add significantly to the complexity.

5.2.3 Resident Component

Deluge has a static component in ROM; the bootloader. It needs to be placed here to be loaded at boot time. This approach trades simplicity in the reprogramming code for complexity in the build system.

Without the resident component the reprogramming code would have to be moved into RAM to allow for the space it resides in to be reprogrammed. This requires location independent code, and not all compilers are capable of producing it. In particular, the MSP430 port of gcc is not.

Chapter 6

Debugging a Mote Program

Motes often operate in environments which are alien to the developers. They even take part of their environment. Noisy environments leads to races between potential execution paths and corner cases are reached which were never taken into consideration. This leads to bugs; often the so-called *heisenbugs*, which changes or disappears entirely when one tries to study them. This makes the lack of visibility and control during the execution of a (distributed) wireless sensor application particular problematic.

Deployments starts life as an idea, which then goes into the development phase. In this phase frequent bugs are expected and the developers are willing to trade realism for any help they can get. As the code becomes more stable, the balance shifts towards realism to uncover the remaining bugs (which likely depends on corner cases). At one point a deployment is made. Experience has shown that health monitoring is needed to avoid the risk of significant failure rates[60, 63]. Bugs can manifest as faults anywhere along this process and the tools change along the process.

6.1 Simulation

TOSSIM[43] is a discrete event simulator framework for TinyOS. A single application is build for a special simulation platform. Using python, one sets up the simulator by choosing the number of nodes to simulate using a (single) image and a radio model for each link. Special printf-style commands can be used in the NesC code to log key information about the execution during simulation.

COOJA[49] is a highly extensible simulator for Contiki. It can simulate multiple nodes at different levels. At the network level pure Java nodes allows the user to remove distractions. Libraries can be tested at the operating system level and operating systems can be tested on the machine code instruction set level. KleeNet[58] integrates Contiki with the KLEE symbolic execution framework and automatically injects non-deterministic failures. The two has been integrated in COOJA/KleeNet[50]. A COOJA test scenario is exported to KleeNet which then performs high-coverage testing. Any failed assertions are presented at runtime. The offending execution paths can then be analyzed in COOJA.

Emstar[23] is a software environment for developing wireless sensor networks on a mixture of microservers and motes. A part of the environment is EmSim; a real-time simulator which models radio and sensor channels. Another is EmCee which modifies EmSim to use a real radio setup instead of a radio model. This helps reviling bugs which are triggered by unique dynamics of the environment.

6.2 Source Debugging Systems

Marionette[69] tries to ease the development of algorithms by creating a framework for TinyOS that allows logic to be shifted between sensor nodes and a PC. The idea is to prototype a solution using python (and other available tools) on the PC, and – once verified to be sound – gradually move it to the sensor platform. This is done by adding a step to the TinyOS build system which implements a service exposing variables and allows for remote function calls.

Clairvoyant[71] provides GDB-like functionality to deployed TinyOS code. A service on each deployed mote listens for commands and maintains a minimal overhead by dynamically rewriting parts of the binary code. This allows Clairvoyant to set breakpoints, inspect the stack, access variables and singlestep through the code. This is all done remotely and integrated into a modified version of GDB.

Minerva[59] builds on the idea of Clairvoyant, but uses a different approach. Each mote is attached to a deployment support mote using a JTAG bus. This allows the deployment support network access to the hardware debugging interface of each deployed microcontroller and – since they are networked – checking of global assertions. When a global assertion fails the whole network can be stopped and the combined state inspected. Due to latency and processing time the state will have changed the moment the distributed break has been completed.

6.3 Deployment Debugging

After the Great Duck Island deployment, David Cullers group reported on the lessons learned[60]. One of the lessons was that sensor readings can – given experience – be used as failure indicators. Another Berkeley deployment instrumented a redwood tree and had significant node failures. This lead to SNMS[62]; an interactive management system for wireless sensor networks. It provides the means to enforce two principles: (i) every sensor network should be capable of providing a human manager with the means to determine whether the deployment is functioning, and (ii) that every sensor network should record failure indicators for both post-mortem analysis and real-time requests. These principles should lower deployment failure rate significantly.

EnviroLog[47] is a logging and replay abstraction for TinyOS. It has the core EnviroLog component which performs operations on the flash and one component injected before each target module. The last components receive events from underlying modules. In record mode each event is forwarded to the associated target module as well as the EnviroLog module which then stores it. In replay mode is ignores those events, and instead acts on events from the EnviroLog module, which then performs timed playback of all recorded events. This increases the chances of repeatability significantly. Only annotated events are logged, but by choosing the right level of capture the location of the bug can be narrowed down.

Sympathy[55] provides another way of localizing the root cause of a failure in a collection tree. It constantly gathers general routing metrics and expects data to arrive to the gateway at a relatively consistent pace. On a regular basis is lists the nodes from which less than expected data have arrived, and – based on an analysis of the routing metrics and the topology of the affected nodes – arrive at a diagnose. This is an automatic process.

Chapter 7

Summary

In this state-of-the-art part, I have described the key characteristics of mote hardware, of mote programming and of wireless sensor network testbeds. This work was necessary to inform some of the decisions that DELTA had to made in order to prepare a wireless sensor network testbed, enabling Danish SMEs to experiment with long term monitoring applications based on sub-sleep energy harvesters. As a conclusion to this state of the art, I will now review these decisions as I believe that they correspond to decision points that shape the design space of any mote-based application:

- **New Mote vs. Existing Mote Hardware**: Given the requirements of a given project, the first question is whether an existing mote architecture should be reused or whether a new mote should be designed. Obviously, the way to answer this question is to review existing mote hardware to find whether an existing mote fulfills the requirements of a given application or testbed. Here, the following questions help determine whether an existing mote is appropriate or not (note that these questions are technical and that in any actual projects, non-technical considerations such as money and human factors such as the availability of trained staff play a major role in any decision):
  - What are acceptable limitations in terms of energy budget? What energy sources are available for the mote? How do they impact mote operations (e.g., sample-store, sample-transmit)?
  - What are the limitations in terms of RAM or ROM? In terms of flash storage?
  - Which constraints does mote channel layout introduce in terms of (i) resource arbitration, (ii) duty cycling and (ii) sensing capabilities?

DELTA chose to design a new mote (see Section 7.1) in order to provide a maximum of control and flexibility to their customers so that they could experiment with a range of different energy harvesters, and to base its design on an existing mote (Epic), introducing few modifications in successive design iterations (e.g., the choice of a new MCU to alleviate RAM limitations). The crucial importance of the channel layout and the interdependencies between hardware and programming framework became evident in the new design.

- **New vs. Existing Programming Framework**: The second question is whether an existing programming framework, or which programming framework, can be reused for the given project. Here, the skills already available in the project play a crucial role. However, the following questions should be asked:

  - Is the programming framework adapted to the chosen mote? How easy is it to port the framework? Conversely, how does the choice of the programming framework restrict the space of suitable motes?
  - Should a thread model or an event model be adapted to deal with concurrent data flows?
  - How can resource arbitration be controlled?
  - How is code reuse supported?
  - Which high-level abstractions are already available? Which networking functionalities are available? How is storage abstracted? Are virtual machines supported?

  DELTA chose at first to rely on TinyOS, but changes in the mote design (i.e., the choice of a new MCU) forced me to port TinyOS on the new mote which turned out more complex than originally envisaged. In retrospect, we did not understand the implications of the choice of TinyOS as a programming framework. I will get back to the insights gained in the next part. We then decided to develop a new programming framework tailored for our needs, specially in terms of portability and ease of debugging. Again, I will get back to the design of Njulla, the new programming framework, in the next part.

- **New vs. Existing Programming Testbed**: In the case of DELTA, the last question was whether to reuse an existing testbed infrastructure or to develop a new one. Here, the key questions were:

  - How should the testbed be controlled? What kind of backchannel is available to control the motes and gather data about program executions?

- How should motes be programmed and re-programmed?

DELTA chose to design a new testbed that minimize interferences with the motes. Specially the goal was to operate motes with sub-sleep energy harvesters, in the context of the testbed. As a result, the efficient architecture of FlockLab (based on dual motes) was ruled out, and I had to design a new testbed framework, GreenLab. It is described in the next part.

Part II

Contribution

Our contribution is composed of the insights we gained designing, implementing and testing a new mote hardware (GreenMote4), a new programming framework (Njulla) and a new wireless sensor network testbed (GreenLab) for the sub-sleep energy harvesting regime. First, I review the critique of TinyOS that I have gathered in the process of porting TinyOS on the new mote. This has led to DELTA to give up using TinyOS. Then, I describe the design of Njulla and GreenLab. Finally, I discuss the issues related to debugging mote programs, that drove the design of Njulla and were of paramount importance for productivity at DELTA.

Chapter 8

TinyOS Critique

It is now almost 15 years since David Culler and his team came up with the original design of TinyOS [27]. Since, as we noted in Part I, TinyOS has evolved into an open source effort. Arguably, the most important legacy of TinyOS will be the networking abstractions that have emerged through systematic academic studies (e.g., [31], [45] or [70]), which resulted in new Internet standards such as 6LowPAN[1] or ROLL[2]. Another important legacy from TinyOS is that it enabled the emergence of a wide community. Phil Levis, who has been in charge of TinyOS since 2005, reflects on this latter aspect in his OSDI'12 paper [40]. In this paper, Phil Levis points out both the positive and the negative lessons learned from his TinyOS experience. Phil notes the following critiques:

1. The evolution of TinyOS and NesC made it easier to solve hard problems, but harder to solve easy problems, because the TinyOS developers focused on the needs of a few experts.
2. The Hardware Abstraction Architecture was in retrospect generalization and abstraction for the academic sake of abstraction.
3. Involving companies early on in the design of TinyOS 2 was a mistake because of the tension between their industrial requirements and the academic focus of the core developers.
4. Documentation only aimed at developers led to obtuse tutorials.

In the rest of this section, I reinforce or illustrate some of the critics from Phil Levis (the problems with components and documentation) and I bring a couple of critiques that provide new insights on the design and use of TinyOS.

[1] `http://datatracker.ietf.org/wg/6lowpan/charter/`

[2] `https://datatracker.ietf.org/wg/roll/charter/`

8.1 The Illusion of Hardware Independence

The introduction of the hardware abstraction architecture was a key feature of TinyOS 2. The design goals are discussed in TEP 2 [25] (TEP stands for TinyOS Enhancement Proposals). From the introduction:

```
The introduction of hardware abstraction in operating systems has
proved valuable for increasing portability and simplifying
application development by hiding the hardware intricacies from
the rest of the system. However, hardware abstractions come into
conflict with the performance and energy-efficiency requirements
of sensor network applications.
This drives the need for a well-defined architecture of hardware
abstractions that can strike a balance between these conflicting
goals. The main challenge is to select appropriate levels of
abstraction and to organize them in form of TinyOS components
to support reusability while maintaining energy-efficiency
through access to the full hardware capabilities when it is
needed.
This TEP proposes a three-tier Hardware Abstraction Architecture
(HAA) for TinyOS 2.0 that combines the strengths of the component
model with an effective organization in form of three different
levels of abstraction. The top level of abstraction fosters
portability by providing a platform-independent hardware
interface, the middle layer promotes efficiency through rich
hardware-specific interfaces and the lowest layer
structures access to hardware registers and interrupts.
```

The idea of providing a high level abstraction for each individual driver to promote portability, while exposing a more complete interface to guarantee performance is seducing. However, the risk is that the hardware abstraction architecture gives programmers a false sense of hardware independence. Indeed, there remain fundamental interdependencies between hardware and programming framework that TinyOS 2 hardware abstraction architecture does not address. As a result, most TinyOS programs, and worse, most TinyOS components depend from the underlying hardware platforms (mainly EPIC or TelosB) in a hidden manner, which is obviously not a problem on those platforms but becomes a hassle on other hardware platforms.

I discuss below the interdependencies between hardware and programming framework, and illustrate how widespread they are in the TinyOS code base (both in the original TinyOS and in TinyOS 2). I return to the dark side of TinyOS components in section 8.2.

8.1.1 Interdependencies between Hardware and Programming Framework

I have identified three main forms of interdependencies between hardware and programming framework that are not exposed through the hardware abstraction layers from TinyOS 2. Note that the first item was discussed by Phil Levis in [40], while the other two items are not:

1. ***RAM vs. ROM constraints***: There are fundamental trade-offs between memory and secondary storage, i.e., RAM and ROM when designing the programming framework. Memory is needed for run-time data structures representing the mote state, secondary storage is needed for storing code and data; both persistent data that should remain accessible across program executions and transient data that do not fit in memory. According to Phil Levis, the ratio between RAM and ROM footprint for most TinyOS programs (where the sensed data does not occupy a lot of RAM), is 1:10. So, on a given mote, if ROM is not 10 times larger than RAM then ROM is the limiting factor, while RAM is the limiting factor if it is more than 10 times smaller than ROM. The platforms used to design TinyOS were RAM limited, while the platforms used for TinyOS 2 were ROM limited. As a result, TinyOS 2 components tend to favor RAM usage (e.g., data structures allocated in memory for ADC configuration) rather than ROM usage (i.e., code which is called to generate ADC configuration on the stack when needed). Whether a given TinyOS component API is ROM-optimized or RAM-optimized is not obvious for a programmer. It is not obvious when porting TinyOS on a new mote platform whether code can be reused, or should be reprogrammed to adapt to the RAM or ROM limitation of the target mote.

2. ***Resource Arbitration*** TinyOS and TinyOS 2 were designed for motes (Rene, Telos, Epic) where the microcontroller is connected to radio, flash storage and serial port via a shared bus (as opposed to the Arduino MEGA board for example, where there can be a dedicated bus per resource[3]). In TinyOS resource allocation is covered by TEP 108 [37]. Three cases are defined: (1) the dedicated resource is associated with no sharing policy, (2) the virtual resource is shared as abstract instances provided on top of a dedicated resource and (3) shared resources are multiplexed. I will focus on the shared resources. Control to a shared resource is handled by the Resource interface. It provides commands for requesting and releasing the resource. When a component needs to perform a task on a resource, it will first request it, then perform the task and finally release it. If the task requires access to two resources it will either switch between the two resources

[3]This is the only AVR-based Arduino with multiple serial busses.

as often as needed or simply request both of them throughout the execution of the task. In the first case resources are likely to be wasted by resource allocation tasks and the code is likely to become complicated. Thus the latter method is likely to be used. With the use of this method comes the potential of a deadlock. TinyOS does provide the ResourceRequested interface through which the owner of a resource can be notified when another component is requesting the resource. If implemented correctly and in a nongreedy fashion by either of the components of the earlier example the deadlock would have been avoided. While TinyOS does have a this construction to avoid the deadlocks it is rarely used. Grepping through the source code reveals that for the epic platform the interface is provided only by the USART components and is never used by the rest of the stack.

Such deadlocks are very hard to track down, because (i) access to resources is hidden in the components implementation and completely obscure to the application programmer, and (ii) because it might not be obvious which hardware resources are being requested by the components in the application program. More to the point, all TinyOS components are designed for mote platforms where resource arbitration is a necessity. This is a deep interdependency between the TinyOS programming framework and the underlying mote platforms, whose consequences are far reaching. Indeed, as a consequence of resource arbitration, access to resources such as flash, radio and ADC is not deterministic in TinyOS. Access to the ADC might have to wait until a sequence of atomic (i.e., non interruptible) accesses to flash complete. This is a problem whenever a digital sensor device requires that events are sampled from the ADC at well defined baud rate, or whenever a radio MAC requires that events are serviced within a well defined time frame.

We can thus state that, by design, TinyOS is not well suited for any form of deterministic interaction with mote peripherals. This turned out to be a very strong limitation for the MANA deployment [11]. In his MSc thesis, Javier González showed that TinyOS could not keep up with the baud rate of the water quality monitor (WQM) that an Epic mote was connected to [24]. Note that the initial bug was found on the Arch Rock implementation of TinyOS, where the baud rate was limited to 9600 (compared to the 19200 required by the WQM sensor); a conservative resource arbitration measure to avoid generating too many requests targeted at the serial port [30]. Marcus Chang and Javier González ran the MANA application on top of TinyOS 2 in order to adopt a more aggressive policy in terms of resource arbitration. Javier succeeded in improving the baud rate by the factor of 10 required by the WQM sensor but at the cost of losing events once in a while, in a non-deterministic way [24].

3. ***Hardware Diversity*** The hardware abstraction architecture should make

it easy to incorporate diverse hardware components. Ideally, the Hardware Abstraction Layer (HAL, i.e., the intermediate layer) can encompass a range of different hardware components of a same type. Also, in case of hardware acceleration, the HAL should become a thin layer on top of the accelerated hardware. As long as a hardware component respects the assumptions from the Hardware Interface Layer (HIL, i.e., the top layer), there is no problem. Problems appear when mote hardware breaks an assumption that has been made at the HIL level and throughout an existing TinyOS component. For example, TinyOS assumes three colored LEDs. This requires modifying the HIL abstracting LEDs when porting TinyOS to a mote equipped with four red LEDs [17]. More serious is the problem of diverse timers. While TinyOS 2 introduces virtual timers, it does not expose the MCU timer channels and thus schedules the instantiation of virtual timers in a predefined, yet undocumented way. Here again, TinyOS hides a form of resource arbitration decision from the application program. This is a problem for programs that use many timers, some of them at a high resolution or a high priority that might violate the predefined timer scheduling from the TinyOS timer component. Jan Flora was facing this problem while working on the radio driver for the Freescale EVB13192 evaluation board [20]. A workaround is to define a new Timer hardware abstraction that incorporates both virtual timers and virtual timer channels, but that defeats the purpose of the three layer hardware abstraction architecture.

8.1.2 Analysis of the TinyOS code base

In this section I investigate the interdependencies that exist between TinyOS subsystems (those high level abstractions that have emerged on top of the Hardware Abstraction Architecture, e.g., BLIP or Zigbee). In particular I wish to answer questions such as:

1. *Which subsystems have large amounts of platform specific code?*
2. *Which platforms have large amounts of platform specific code?*
3. *What is the ratio of subsystems without platform specific code to subsystems with platform specific code?*

I accomplish this by analyzing the TinyOS source tree[4]. Figure 8.1illustrate the processing flow which is split into four separate steps. These are:

[4]The code is available here: `https://github.com/aslakjohansen/tinyos-history`

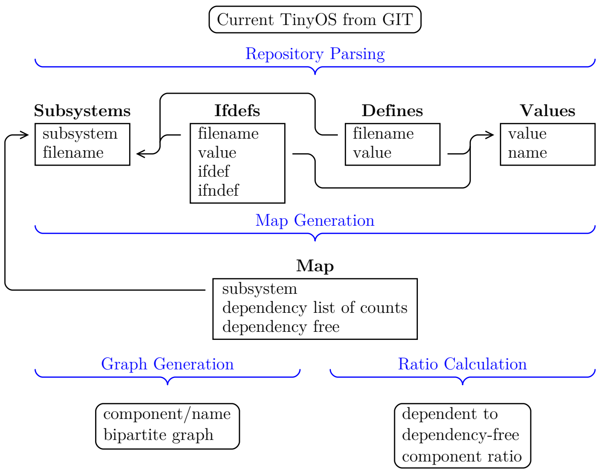


Figure 8.1: Model for analyzing the TinyOS code base. Black indicates data and blue indicates processing.

- **Repository Parsing**
  Input: TinyOS repository checkout
  Outputs Files `spd_subsystems.csv`, `spd_ifdefs.csv`, `spd_defines.csv` and `spd_values.csv`
  All `*.c`, `*.h` and `*.nc` files in the repository are scanned[5] using primitive parsing[6]. The following datasets are generated:

  – Subsystems (file `spd_subsystems.csv`)
    Each scanned file is associated with a subsystem[7].

  – Ifdefs (file `spd_ifdefs.csv`)
    Each preprocessor directive is associated with a type (`ifdef` or `ifndef`) and a filename.

  – Defines (file `spd_defines.csv`)
    Each platform define is associated with a file. This dataset is not currently being used.

  – Values (file `spd_values.csv`)
    Each platform define is associated with a platform name (`PLATFORM_EPIC` $\mapsto$ Epic).

  How: performed by the `spd-extract-data` script.

- **Map Generation**
  Input: Files `spd_subsystems.csv`, `spd_ifdefs.csv`, `spd_defines.csv`

[5]Note that only single-line precompiler directives are parsed at this point.
[6]Comments are respected but includes are not followed.
[7]51 subsystems are defined in the `spd-extract-data` script.

and `spd_values.csv`
Outputs File `spd_map.csv`
All counts (`ifdef` and `ifndef`) for all files associated with a particular subsystem are summed up on a (platform) name basis. If a subsystem has zero references to all platforms then it is marked as being dependency free.
How: performed by the `spd-generate-map` script.

- **Graph Generation**
  Input: File `spd_map.csv`
  Outputs File `spd_graph.pdf` (graph in PDF)
  The graph is bipartite between subsystems and platforms.
  How: performed by the `spd-generate-graph` script.

- **Ratio Calculation**
  Input: File `spd_map.csv`
  Outputs Ratio (as float via STDOUT)
  Each subsystem is either in $\mathbb{S}_d$ (the set of platform dependent subsystems) or in $\mathbb{S}_i$ (the set of platform independent subsystems). The ratio is calculated as $\frac{|\mathbb{S}_d|}{|\mathbb{S}_d|+|\mathbb{S}_i|}$.
  How: performed by the `spd-calc-ratio` script.

For this we mapped every relevant source file in the repository to a subsystem.

Figure 8.2 shows the number of platform references found in subsystems. The largest number of platform references found for a single platform/subsystem combo is 14 for TelosB in Zigbee.

By manual inspection I were able to place the use cases in six categories:

1. Information which could have been part of the HIL Header and footer sizes for different radios.
2. Raised hardware abstraction An abstraction is build on top of the HIL (e.g. when the Active Message component refers to platform specific components like radio).
3. Interface Extensions Extending an interface with relevant platform specific functionality.
4. Warning Trigger A method for giving compile-time warnings (e.g. when the platform has no radio).
5. Debug Debug printputs[8].

[8]likely from a PC build target.

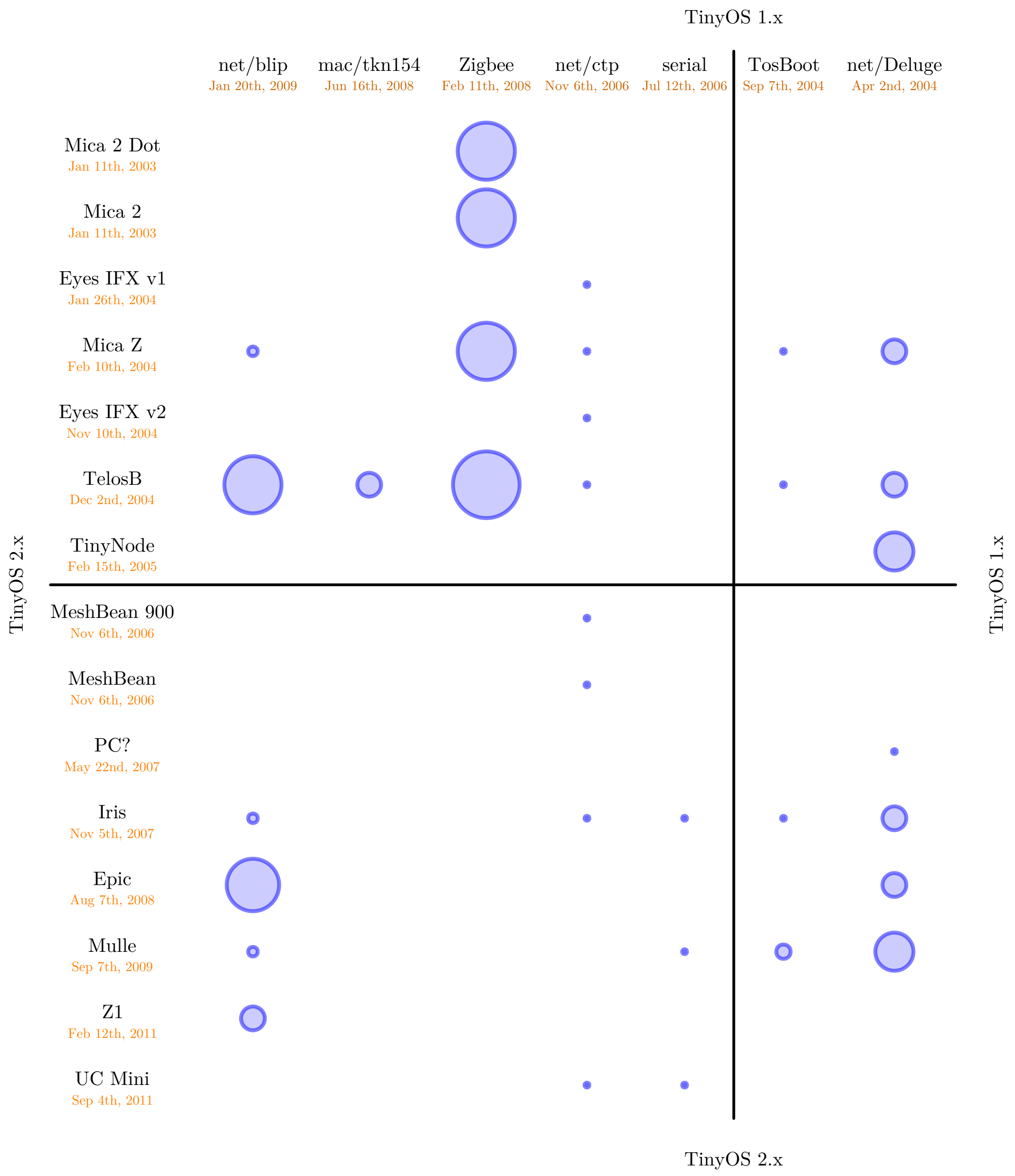


Figure 8.2: Number of platform define uses in subsystems. The circle radius indicates number of encountered references. The circle is omitted if zero references were encountered. Date of subsystem and platform introduction is noted.

Subsystem	HIL Deficiency	Raised HA	Interface Extensions	Warning Trigger	Debug	HPL
mac/tkn154	0	0	5	0	0	0
net/blip	0	9	24	1	0	0
net/ctp	0	8	0	0	0	0
net/Deluge	0	36	0	0	1	0
serial	3	0	0	0	0	0
TosBoot	6	0	0	0	0	0
Zigbee	0	15	5	0	0	21

Figure 8.3: Mapping of platform define uses to causes.

6. HPL Lines belong in the HPL layer.

The distribution is outlined in figure 8.3. Of these categories 1, 2 and especially 6 are problematic. In `/tos/lib/net/zigbee/ieee802154/includes/printfUART.h`[9] register and mask names are exposed even though it belongs in the HPL. This code was introduced[10] on August 25th, 2009.

The ratio of platform dependent to platform independent subsystems turned out to be 0.137 or almost 14%. Declaring "mac/tkn154" a platform independent subsystem (it only uses interface extensions) will lower the value to 0.118 or almost 12%.

The results of this study are associated with a significant amount of uncertainty due to the method of extracting platform define uses. This means that I might have counted uses which are commented out and I will have missed uses of certain platforms. My guess is that we would see a wider spectrum of platforms requirements for the listed subsystems, but no more subsystems. Furthermore the current distribution among platform uses is likely to shift. Overall, I have found that 12% of the TinyOS subsystems contain platform specific code of a problematic type. This is a low estimate as I don't extract the data recursively. There seems to be a sprouting tradition of implementing a raised HAA on top of the HPL, likely due to deficiencies in the HIL. The Zigbee subsystem contains HPL code.

[9]The same goes for `./tos/lib/net/zigbee/wrapper/includes/printfUART.h` (which is an identical copy) and `/tos/lib/zigbee/clusterTree/includes/printfUART.h`.

[10]At changeset hash `1ec320d991e48ec7eef5321797fc757c713e7427`.

8.2 Components Considered Harmful

8.2.1 Understanding TinyOS Components

All interfaces in TinyOS are documented. The reasoning behind and relationships between the interfaces are described through the TinyOS enhancement proposals (TEPs). Tutorials help newcomers get started while design patterns give further insights. These areas represent the documentation for TinyOS and it is clear that thought has gone into the structure of each area.

Today, the promise of the documentation has faltered. Some interfaces do not match their respective TEPs. Their relationships are easy to spot, but commands are left out or have different prototypes. In some components one has to look for comments in the source code in order to gain knowledge of critical information on how to interact with it, in others a study of the code itself is needed.

Combined these aspects shake general confidence in the platform. In particular, it represents unclear requirements when porting TinyOS as it makes little sense to adhere to the documentation if it isn't de facto. How do we know which parts of the documentation are valid?

8.2.2 Porting TinyOS Components

The offerings from TinyOS to the application developer are in a high-level format, in part because of the choice of NesC as the base language. Components are used as building blocks and specific implementations are automatically selected based on target when building the application. The build process converts the relevant NesC code to one big C file and then compiles. The C file is clearly machine generated but comments indicate the NesC source of each line. Working on a high level while getting low-level optimizations is a nice feature to have.

When debugging a port in progress it becomes more important to have a good feeling with this high/low relationship. What you want most of all is a dead-simple starting point, from which you can expand in small steps. And thus it becomes important to have a clear understanding of how the high level abstractions translate to the low level abstractions and the other way around. This could be considered a platform feature.

The same is valid for the build process, where the core of it is made up of a handful of scripts (shell and perl). A long sequence of arguments is passed through these scripts and is modified on each level. Some arguments are acted upon and removed, others are modified and some are added. When debugging a port, knowing each step of the process and being able to control it and look at the intermediate assembly and map files becomes a necessary tool. With TinyOS this requires going through the build scripts to find the final call to the compiler and based on this write our own build sequence.

8.2.3 Debugging TinyOS Components

TinyOS components trade passive ease of debugging for active ease of debugging: Complexity is hidden by isolation with TinyOS components. In addition, TinyOS relies heavily on static code analysis to capture bugs early in the development process. This makes the coding simpler and as a result fewer anomalies should be expected at run time. However, the anomalies which do manifest themselves are obscured by levels of abstraction within each component. Appendix A illustrates some of my frustrations with this problem.

8.3 Summary

So, when is TinyOS a good fit? The short answer is that TinyOS is a good fit for the regimes for which it was designed originally, i.e., (i) for applications that require low power utilization, can tolerate the failure or the data loss from a few motes and have no specific requirements in terms of latency or bandwidth, (ii) on the mote platforms for which TinyOS was designed (TelosB or Epic). TinyOS is not adequate for applications that suffer from data loss on a given mote, as the way TinyOS handles resource arbitration cannot give any guarantee against data loss. TinyOS is not adequate for other platforms on which it has to be ported, as the complexity of the framework and the mismatches between implementation and documentation make any porting effort cumbersome and unproductive. DELTAs use case fell in the latter category.

The problems I faced understanding, controlling and porting TinyOS on the new DELTA mote led us to design a new programming framework that struck a different balance between the portability, efficiency and ease of debugging goals. Indeed, our design stressed simplicity to favour portability and ease of debugging.

Chapter 9

Njulla

Before I describe the Njulla framework, let me briefly present DELTA's new mote, the GreenMote4.

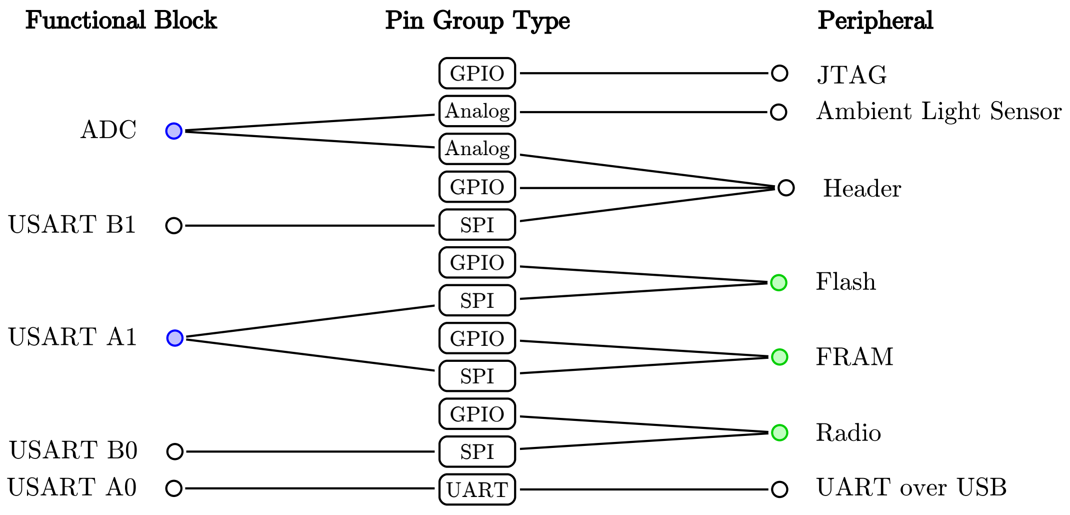


Figure 9.1: The GreenMote4 channel dependency graph. Blue nodes can only operate a single edge at a time, green nodes needs all directly connected edges to operate.

9.1 GreenMote4

The GreenMote4 is the latest mote of a series from DELTA originally derived from the Epic[16]. The current incarnation has a bigger microcontroller, i.e., MSP430f5437, with four USARTs instead of two. The idea was to map each of these to a peripheral and thus completely circumvent the problem of arbitration. The idea was good, but we ended up with five peripherals. Figure 9.1 shows the resulting mapping.

The flash takes considerable time to operate and an erase operation is very expensive in terms of energy. The FRAM was included as a fast and low-energy non-volatile buffer. It can be used to store a blocks worth of data in addition to some configuration metadata. I can also be used as a circular log. The FRAM is not is expected to be involved in time-critical operations and is thus sharing a USART.

9.2 Design Space

I distinguish three main dimensions in the design space for mote programming frameworks:

1. ***Portability*** Portability is important to (i) ensure a smooth transition from development boards to optimized deployment platforms and (ii) to guarantee the perennity of an application in time with different firmware versions and possibly different hardware devices. By designing for portability the dependency of the framework on a single hardware platform is loosened through abstractions so that a single common interface hides multiple implementations.

2. ***Performance*** Performance in the context of sensor networks can be associated to the optimized use of constrained resources, from energy to RAM. Performance also refers to the more classical metrics of throughput and latency.

3. ***Ease of debugging*** For a system to be easy to debug it needs to appear (and be) simple to the developer. This simplicity depends on system complexity and observability. For the end programmer this can be limited to the application layer if the underlying layers are easily understood and can be clearly separated from the design process. Otherwise the application developer will have to extend her mental model of the system to incorporate details of this framework to an extend that restricts the resources she can dedicate to the application layer itself.

It is clear that – while having weak ties to each other – these dimensions are independent. A portable system has additional abstractions which cannot have a positive impact on neither performance nor complexity. In an optimized system structural choices have been made which cannot make it easier to port or debug. In an easily debugable system visibility and simplicity are key properties. Increasing visibility cannot make the system more portable, although it may reveal potential ways of doing so. Making a system simpler cannot increase performance, unless it has side-effects revealing new ways of optimizing.

Focusing on optimizing a single of these dimensions leaves space for balancing trade-offs between the other two. Focusing on optimizing any two leaves no leverage for protecting areas of the third. We would have to balance the goal of optimizing two dimensions against the repercussions in the third. Focusing on all three leaves no room to maneuver when faced with any of the obstacles which are bound to appear. One has to balance the three dimensions according to some set of guiding principles. In a way system design is a puzzle; you must find a configuration of the pieces which has no inherent conflict and fills out the the required space.

As we have seen, TinyOS 2 introduced a three layer abstraction for all its drivers in order to promote portability. Performance trade-offs were dictated by ROM constraints (as opposed to RAM constraints for TinyOS). In terms of ease of debugging, TinyOS relies on components embodying high level abstractions to simplify application code and on static analysis to capture as many anomalies as possible at compile time. Let us now review the design decisions I took with Njulla.

9.3 Programming Framework

Plain C was chosen as the programming language due to its portability, relatively low level and simplicity. As a language C offers constructs for implementation of sequence, choice and indirection – both for data and logic – and little more. As such it takes little effort to consider how the language can help you solve a problem or figure out precisely how a solution functions. For reasons related to portability I decided to go with the `gcc` compiler.

9.3.1 Build System

The build system is a python script. Figure 9.2 lists the parameters. It allows one to choose the target platform (thus modifying the include path), the mote id, the programmers device, the framework path as well as the mode of operation. The framework path defaults to the current directory and a search is performed from this path towards the root of the filesystem to determine the framework

```
/home/aslak/vcs/git/njulla/samples/uart/blink$ build --help
Usage: build [options]

Options:
  -h, --help            Show this help message and exit
  -t TARGET, --target=TARGET
                        Build for TARGET [default: greenmote4]
  -i ID, --nodeid=ID    Specify node ID [default: 42]
  -d DEVICE, --device=DEVICE
                        Specify DEVICE for programming [default: /dev/ttyUSB0]
  -m MODE, --mode=MODE  Specify MODE of operation (full, compile, program)
                        [default: full]
  -r ROOT, --root=ROOT  Specify ROOT of framework installation [default:
                        /home/aslak/vcs/git/njulla/samples/uart/blink/]
```

Figure 9.2: Help message from build script

root. The mode of operation is compilation, programming of an existing image or the combination.

A significant number of programs goes into the build process. These come from four different packages and can – depending on distribution – be installed at a few different locations. The build script scans each of these locations for the required programs and produces an error message if one is missing. This message explains which package is missing and how to get a hold of it.

Each build is injected with metadata. They cover time, path, current git checkout, whether there are uncommitted changes, the system uname, the username, the hostname and the build parameters. This allows one to recreate the build conditions that results in an anomaly after extracting the image from a mote. The data is also addressable from logic on the mote.

All intermediate steps of the build process are left on disk for inspection. This includes individual and combined assembly files, object files, the memory map file, the hex file and a disassembled version of the image file. The hex file is essentially a sequential program, listing commands for modifying the contents of a memory space. The raw hex file is additionally pretty printed to HTML with highlighted command types and fields for easy inspection. The disassembled file is convenient (compared to the machine code) for verifying that the compiler does as expected.

9.3.2 Memory Management

Static memory allocation allows one to know at compile-time whether some program will fit in the RAM available on a given mote. Once we lose this property a new class of potential problems is introduced. We would then have to start coding for run-time memory shortages.

One could argue that this isn't always necessary as it is obvious for many programs that they fit in memory when implemented bug-free. But what happens when we observe an anomaly? Then it is likely to be caused by a bug. The bug could cause a memory leak, or it could be something completely different. The symptoms may at times provide clues but we cannot rely on these. What we need is visibility at post-mortem and the only way to get that is to insert code for it. This code serves the purpose of debugging and is thus dead-weight in terms of simplicity of the code implementing the desired target functionality.

Static allocation is not without faults. Many programmers are used to think in terms of dynamic memory allocation. This means that the subjective (perceived) simplicity often suffers.

A more abstract programming language – like Java – could have provided a set of guarantees and support functionality to lessen the problems associated with dynamic allocation. You would then need to be able to trust this language. With the choice of a language of a lower level of abstraction I chose the static model. So far this has not given us any gripes.

9.3.3 Object (state) System

Some functions of logic operate on parameterized state. A function to set a pin high needs a parameter representing the pin. In Java it would be an object hidden behind the abstraction of a method. In the Arduino framework it would be an enumerated value.

Many straight C libraries for general purpose operating systems are made up of functions operating on pointers to generic *contexts*. Several plain C object systems exists that are comparable to native object systems (both in features and complexity), at the cost of simplicity.

Few of these features are strictly necessary for us and I thus stick to the simplest form which can be type checked: a pointer to a struct. An example of how such structs are defined is shown in figure 9.3.

9.3.4 Execution Model

To poll, or not to poll, that is the Question: Whether it is better to design for strictly sequential operations, or to accept the bulk associated with the infrastructure to support concurrent operations. For sequential operations polling is

```
typedef struct _spi_context_t {
  volatile unsigned char* ctl0; // register: control 0
  volatile unsigned char* ctl1; // register: control 1
  volatile unsigned char* mctl; // register: modulation control
  volatile unsigned char* br0;  // register: baud rate 0
  volatile unsigned char* br1;  // register: baud rate 1
  volatile unsigned char* ie;   // register: interrupt enable
  volatile unsigned char* ifg;  // register: interrupt
  volatile unsigned char* tx;   // register: transmit buffer
  volatile const unsigned char* rx;   // register: receive buffer
  pin_t*    miso; // pin: Master In, Slave Out
  pin_t*    mosi; // pin: Master Out, Slave In
  pin_t*    clk;  // pin: Clock
  pin_t*    ss;   // pin: Slave Select
} spi_context_t;
```

Figure 9.3: Njulla object definition example.

the simple choice, and even the low-latency choice. For concurrent operations polling grows in complexity and latency, and trade-offs emerge weighting the different operations against each other.

Threads provides a relatively simple concurrency model to application programmers. This works well for everyday coding, but it breaks knowledge of causality. When an anomaly manifests it will be unknown how the active threads were serialized to result in this anomaly. It may even be unknown which threads were active. It also requires additional complexity in the core of the framework for switching between the threads. The alternative also involves potential race conditions, but we can make sure that these events have limited propagation while still providing concurrency. I thus choose to not offer threads.

Events provides an elegant solution at a constant cost in complexity. We felt that the option to have concurrent operations was necessary and chose to go for events.

9.3.4.1 Callback System

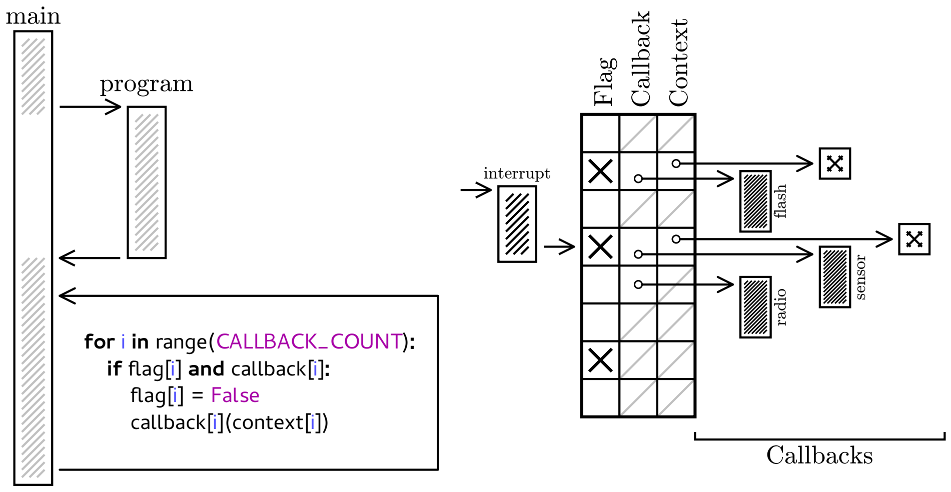
Events are used to signal the completion of the first phase of a two phase operation. During the first phase a handler is configured to allow for the control to be returned. Because of this the handler is often known as a *callback function* and the operation of calling it is a *callback*. The lowest layer of callbacks are incarnated by interrupts.

Callback functions without parameters are registered as `void*` $\rightarrow$ `void` function pointers. This allows us to pass an arbitrary context to any callback handler, while treating different types of handlers in the same way. If extra parameters needs to be passed to the callback function then the interface reflects this, as

```
typedef void (*adc_callback_t)(void* context, uint16_t value);

void adc_sample (adc_context_t* context,
                 adc_callback_t callback, void* callback_context);
```

Figure 9.4: Njulla callback example.



```
for i in range(CALLBACK_COUNT):
    if flag[i] and callback[i]:
        flag[i] = False
        callback[i](context[i])
```

Figure 9.5: Overview of the Njulla execution model.

exemplified in figure 9.4.

9.3.4.2 Interrupt Processing

Figure 9.5 illustrates the operation of the main loop of Njulla. The main function is located within the framework. It first initializes some data structures and peripherals and then calls the main application function, `program`. After the return of this function the main loop is entered.

The main loop services the flag-callback-context structure. This structure has an entry for every interrupt implemented. For each interrupt is has a flag field as well as fields for storing a callback handler and a context in which to execute it. Interrupt processing is split into two steps. During the first any time critical operations is performed, the corresponding flag is set and the thread of execution is returned.

When the main loop comes across a flagged interrupt with a valid (non-null) handler it will clear the flag and call the handler, thereby completing the second step. This approach ensures that the callback handlers are executed atomically with respect to each other.

```
#include <stdint.h>
#include <uart.h>

#define BUFSIZE 256

uint8_t const_rxbuffer[BUFSIZE];
uint8_t const_txbuffer[BUFSIZE];
uint8_t* rxbuffer = const_rxbuffer;
uint8_t* txbuffer = const_txbuffer;
uint8_t i = 0;

void transmit_callback () {}

void receive_callback (uint8_t byte)
{
  uint8_t *tmp;

  rxbuffer[i++] = byte;
  if (byte == '\n') {
    tmp = txbuffer; txbuffer = rxbuffer; rxbuffer = tmp; // flip buffers
    uart_transmit(txbuffer, i, transmit_callback);
    i = 0;
  }
}

void program (void)
{
  uart_init();
  uart_open(UART_BAUD_9600, UART_DATABITS_8, UART_PARITY_EVEN, UART_STOPBITS_1);
  uart_receive(receive_callback);
}
```

Figure 9.6: Njulla example of a double-buffered UART echo client.

9.3.4.3 Programming Model

This leads us to a programming model where the application code is called during startup. It then registers callback handlers for certain events in a callback bank and returns so that the main loop gets activated. Eventually the events happen, the callbacks are made and control is thus returned to the application code. While the control is here, the callback bank can be reconfigured. Figure 9.6 gives an example of how callbacks are registered.

The reconfiguring is done by library code to avoid unnecessary clutter in the application code. This approach also has the benefit of ensuring a canonical position for the callback logic with some degree of isolation.

9.3.5 Timers

The MSP430 has a small number of hardware timers. A framework needs to be able to support significantly more virtual timers. I thus define an abstract timer type, for which all instances are based on one of the hardware timers. Each time an active abstract timer is reconfigured (added, expired, disabled or removed) the shortest active timeout is found and the hardware timer is configured to match this. When the hardware timer expires all active abstract timers have their timeouts decremented, the hardware timer is setup again and the expired abstract timers are signaled.

Each abstract timer is associated with a context and the initialized timers are chained together by a linked list. This implementation is simple but inefficient: All initialized abstract timers (including the disabled ones) have to be scanned each time the hardware timer expires. It would be easy to replace the single list with two; one for active abstract timers and one for disabled ones. Figure 9.7 shows how this is interfaced from the application layer.

9.3.6 ADC

The MSP microcontroller is designed for low power operations. This is evident in the design of the ADC subsystem. 16 inputs can be routed to a single ADC. There are 16 sets of configurations specifying parameters for which input is routed and how the conversion is done. The ADC subsystem can be configured to iterate through these as a chain and do multiple cycles, potentially filling up the entire memory. All of this can be done without a running program counter.

This power comes at an inherent complexity. Exposing all the dependency involved through a framework is problematic; overhead would be significant. Doing so in a portable manner would be impossible.

As we are trying to simplify things and we rarely have many sensors the solution is simple: Perform a single conversion at a time. Each input can be configured independently, so the overhead of sequencing conversions has been minimized. Figure 9.8 shows the interface.

9.3.7 Reprogramming

Reprogramming a mote is the act of programming it by means of logic implemented on the mote itself. This is in contrast to the situation where a JTAG or similar programmer is used to stream an image to ROM. A sample application if shown in figure 9.9.

```
#include <stdint.h>
#include <led.h>
#include <abstracttimer.h>

#define PERIOD (512)

abstracttimer_context_t timer_contexts[3];

void callback1 (uint16_t time)
{
  led_toggle(1);
  abstracttimer_timeout(timer_contexts+0, 1*PERIOD, callback1);
}

void callback2 (uint16_t time)
{
  led_toggle(2);
  abstracttimer_timeout(timer_contexts+1, 2*PERIOD, callback2);
}

void callback3 (uint16_t time)
{
  led_toggle(3);
  abstracttimer_timeout(timer_contexts+2, 4*PERIOD, callback3);
}

void program (void)
{
  for (int i=1 ; i<4 ; i++) {
    led_off(i);
    led_enable(i);
  }

  for (int i=0 ; i<3 ; i++) {
    abstracttimer_init(&(timer_contexts[i]));
  }

  abstracttimer_timeout(timer_contexts+0, 1*PERIOD, callback1);
  abstracttimer_timeout(timer_contexts+1, 2*PERIOD, callback2);
  abstracttimer_timeout(timer_contexts+2, 4*PERIOD, callback3);
}
```

Figure 9.7: Njulla example of a 3 bit led clock based on abstract timers.

```c
#include <stdint.h>
#include <pin.h>
#include <adc.h>

void sample_callback (void* context, uint16_t value)
{
  led_toggle((value%3)+1);
  adc_sample(&adc0, sample_callback, NULL);
}

void program (void)
{
  for (uint8_t i=1 ; i<4 ; i++) {
    led_off(i);
    led_enable(i);
  }

  adc_open(&adc0);
  adc_set_pin(&adc0, &pin60);
  adc_sample(&adc0, sample_callback, NULL);
}
```

Figure 9.8: Njulla example of using the ADC to implement a random generator.

The ability to perform reprogramming allows us to perform remote updates of the software running on motes. In particular, I am using this for switching between multiple images in the testbed of section 10.

9.3.7.1 Compiler limitations

Writing position independent code (PIC) means using relative branching. The MSP430 lacks the instructions to perform relative jumps. This means code performing relative jumps would have to calculate the difference at runtime and that takes time. In the MSP port of `gcc` this overhead combined with the rare need for PIC means that no-one ever implemented support for PIC.

The consequence of this is that the logic performing the reprogramming itself cannot be moved during reprogramming. This leaves us with a static section that we can't update by reprogramming. This situation could be bypassed by writing the reprogramming logic in assembly. That however, bypasses the point of having a framework and would require debugging at the assembly level.

Texas Instruments are currently in the process of taking over the control of `mspgcc`. They are hiring Redhat to do the work. In time this may give us the option of PIC, but until that happens it is simply not available through `mspgcc`.

Program 1: Flicker LED 1

```
#include <stdint.h>
#include <led.h>
#include <hil_at45db.h>
#include <reprog.h>

#define LED (1)
#define OTHER_IMAGE_BLOCK (3)
#define ROM_BLOCK_SIZE (512L)
#define OTHER_IMAGE_SIZE ((82)*ROM_BLOCK_SIZE)

void program (void)
{
  // led init
  led_enable(LED);

  for (uint8_t i=0 ; i<100 ; i++) {
    led_toggle(LED);
  }

  // flash init
  at45db_init(&at45db_context, at45db_context.type);

  // switch image
  reprog_load_image(OTHER_IMAGE_BLOCK, OTHER_IMAGE_SIZE, FALSE);
}
```

Program 2: Flicker LED 2

```
#include <stdint.h>
#include <led.h>
#include <hil_at45db.h>
#include <reprog.h>

#define LED (2)
#define OTHER_IMAGE_BLOCK (3+82)
#define ROM_BLOCK_SIZE (512L)
#define OTHER_IMAGE_SIZE ((82)*ROM_BLOCK_SIZE)

void program (void)
{
  // led init
  led_enable(LED);

  for (uint8_t i=0 ; i<100 ; i++) {
    led_toggle(LED);
  }

  // flash init
  at45db_init(&at45db_context, at45db_context.type);

  // switch image
  reprog_load_image(OTHER_IMAGE_BLOCK, OTHER_IMAGE_SIZE, FALSE);
}
```

Figure 9.9: Njulla example jumping back and forth between two program images.

9.3.7.2 Time of Reprogramming

Without the PIC option we had to stick with a static section. This leaves us with the problem of *when* to perform the reprogramming. It can either be done at boot time or at run time.

Boot time reprogramming would require additional logic to be executed at every boot. This takes time and power, but more importantly it adds to the complexity of the core functionality. Run-time reprogramming does not affect anything but the library implementing it.

9.3.7.3 Approach

A program image is stored continuously on external flash. It has a location and a size. The reprogramming is initiated by turning off interrupts to make sure that no partially reprogrammed code is executed. Then, for each involved memory segment not a part of the static section, the following operations are performed:

1. ***Read ROM*** The segment is read into a buffer in RAM.
2. ***Modify*** The largest relevant subset of the buffer is overridden by the matching segment from flash.
3. ***Erase*** The ROM segment is erased.
4. ***Write*** The buffer is written to the ROM segment.

After completion the stack is no longer guaranteed to be valid (it may contain pointers to the old contents). This is fixed by restarting the mote, which activates the new image.

9.3.8 Implementation

The actual implementation mirrors the above mentioned intentions quite closely. The only real differences are (i) the lack of a strict adherence to the passing of contexts, and (ii) the lack of a strict adherence to a generic function pointer type. Deviations are limited to the library internals and have no technical foundation.

9.4 Evaluation

The evaluation of the Njulla framework is directed at its ability to function as a prototyping framework for SMEs working with sub-sleep energy harvesting sensor networks. More specifically, I postulate three hypotheses:

1. To make a mote programming framework attractive for SMEs the learning curve needs to be shallow.
2. TinyOS trade-off between high-level abstractions and performance does not allow for prototyping of sub-sleep energy harvesting sensor networks.
3. Njullas trade-off between simple abstractions and performance is well suited for sub-sleep energy harvesting applications.

9.4.1 Complexity

The framework complexity relates to both the portability (covered in the next section) and the ease of debugging (covered in section 9.4.3). As described in section 1.4, the complexity of a framework depends on the components involved and their connections. In this section I compare the code bases of Njulla and TinyOS based on the number of files, lines and ifdefs involved in key areas of the source code. These metrics cross the boundaries of components and should be considered in the context of the framework design philosophy.

A quick look at the TinyOS code base relating to the TelosB (the primary platform) hints at a fairly complex code base. Figure 9.10 tabulates some key properties of the code base. The data was generated by a script[1] scanning the source tree. Files we categorized based on name, and the TelosB `.platform` file was used to pick out platform dependent code. Figure 9.11 shows the similar numbers for Njulla.

Three platforms are defined in Njulla; GreenMote 1, 2 and 4. The GreenMote 2 is left out for brevity. The GreenMote 1 was replaced early as the main platform. It is kept in the table because it can be used to program the TelosB, although this wasn't verified until late in the process. The GreenMote 4 is the main platform. The large number of ifdefs in the Njulla interface line is due to import guards and a single automatically generated header file containing symbol sizes.

The comparison is not trivial. TinyOS has more features – both in base code and libraries – and it is not immediately clear how much can be easily removed. Where TinyOS has the distinction of system, libraries and (various categories of)

[1]The code is available here: `https://github.com/aslakjohansen/tinyos-history`

Type:	Files:	Lines:	Ifdefs:	Comments:
Platform Dependent	451	53490	669	
System	79	6213	134	
Interfaces	83	6576	0	Not all relevant for TelosB
Libraries	909	112183	1751	Most are outside the scope of Njulla
Apps	589	42990	450	Sample code, not all applicable for TelosB

Figure 9.10: Distribution of TinyOS code relating to the TelosB. Platform dependent code means code that is not shared with *all* platforms.

Type:	Files:	Lines:	Ifdefs:
Platform Dependent (GreenMote1)	20	1610	20
Platform Dependent (GreenMote4)	28	3203	40
Platform Independent	16	3239	72
Interfaces	26	2270	327
Samples	169	16673	72

Figure 9.11: Distribution of Njulla code relating to the TelosB.

platform dependent code, in Njulla the system is a part of the platform dependent code and the libraries are platform independent. The platform dependent code from the Njulla table thus maps to both the platform dependent code and the system code in the TinyOS table, and the platform independent code from the Njulla table maps to the libraries in the TinyOS table. In addition, the sample code from the Njulla table maps to the apps code in the TinyOS table.

TinyOS is – due to features – naturally larger. The key difference is that while TinyOS employs multiple layers of abstractions between the interfaces normally used for application development and the hardware, njulla has the platform dependent code and then libraries with a short longest dependency chain (routing→radio→spi→pin). As a result – and as long as the abstraction can bear it – the complexity of Njulla grows slower with features than TinyOS.

9.4.2 Portability

One of the main dimensions of the Njulla design space was portability. Njulla currently supports 4 versions of the GreenMote line of motes. However, they all use MSP430 microcontrollers, AT45DB flashes and the CC2420 radio.

Porting Njulla to a new microcontroller is a matter of porting the platform dependent code. This is 3203 lines of code split across 28 files (of which half are headers). Each of these 14 modules have multiple corresponding test cases in the sample code, typically highlighting corner cases. Due to the low number of connections between these core modules, most have few dependencies and can thus be tested almost independently. This means that the porting process only requires you to port one module at a time, and the order is fairly obvious.

Similarly, porting TinyOS to a new microcontroller is a matter of porting the platform dependent code. If the TelosB is taken as a base, then this process involves 53490 lines of code in 451 files. Moreover, the dependencies of these files are not immediately clear. The process of finding a path for gradually adding platform support is complex. The first step is to identify a minimal core set of components needed to do something and make sure that these are free of external dependencies. Given the complexity and the lack of up-to-date tools for dealing with it, this is not a simple task.

Porting Njulla to a new radio requires a radio module to be produced. Integrating into the routing library requires overloading of the radio context. Operating multiple radios of different types requires writing an additional abstraction. The same is the case for flash.

Porting TinyOS to a new radio requires interfacing with a large existing stack. The stack provides a significant amount of functionality, but limits messages to a maximum of 256 bytes. At the other end one has to build on components whose internals are not clearly specified.

9.4.3 Ease of Debugging

The ease of debugging is the main design point of Njulla. In the evaluation I compare it to TinyOS.

Njulla has a shallow stack where dependencies can be seen as includes among a very limited set of files at clearly defined locations. The state of individual modules is observable through the contexts at runtime. At compiletime the build system constructs a series of files relevant for debugging: Per-module assemblies, full-program disassemblies, map and lst files. These provide observability to different phases of the build process.

TinyOS has a complex stack often with multiple connections between any two components. The components are designed to hide the complexity. Which components are pulled in to construct the application is unclear until you manually trace the calls in the automatically generated single-file source code. Similarly, the build system provides you with a processed c file and a binary program image. How it is produced is unclear until you manually trace the execution of a series of Bash and Perl scripts. Overall, care has been taken to hide the details. This is often something to appreciate when writing code, but now and then the details are needed. And then they are far away. In this way TinyOS makes easy problems easy and hard problems harder.

9.4.4 Bar of Entry

When a SME is deciding on a programming framework one of the more important properties of a framework is the size of the body of potential labor; if significant resources need to be applied to educate workers then both cost and risk goes up. The universities provide a glimpse into the future of the workforce. At ITU the knowledge of sensing platforms fall roughly into three categories:

- ***TinyOS*** Experience is limited to four members of staff, and that number is more likely to fall than to rise.
- ***Arduino*** Used extensively in three labs and is taught in multiple courses.
- ***Raspberry Pi*** Full student body from the two CS lines, by virtue of a LAMP stack.

This indicates that it is virtually impossible to find people with experience in TinyOS. There exists a community around Arduino and successful workshops are held regularly for people with little or no technical background. The learning curve is simply level enough to allow for easy adoption. The Raspberry Pi covers a different area of the solution space where computational power and and the convenience of a general purpose operating is favored over low-level IO[2] and low power consumption. However, it could be applied to a significant portion of the problem space normally dominated by mote class hardware. It is cheap, physically small and can perform both serial and digital pin operations. It does, however, consume significantly more power than mote class hardware and does thus require a wall socket or other significant power source to be available in the problem space.

Njulla comes with a large selection of sample code. Figure 9.11 indicates that there are 16673 lines of sample code per $(3203 + 3239 = 6442)$ lines of platform code (a ratio of 2.59). Similar numbers for TinyOS are at most 42990 lines of sample code per at least $(53490 + 6213 = 59703)$ lines of platform code (a ratio of 0.72). It could be argued that TinyOS – due to its component encapsulation – only need sample code for the components normally used by developers, but – as one is expected to overload individual components – it is unclear how this subset is defined. Njulla is conceptually much simpler than TinyOS and the source code will be familiar to anyone with experience in callback functions. In order to create

[2]Although RISC OS is available (and capable of single-tasking), one of the attractions is to have a general purpose operating system with code running in a rich user space. Having that implies the code is subject to time slices and thus cannot have time critical components.

a TinyOS application one has to create a configuration (subgraphs of modules) linking interfaces. Finding the component which exposes a needed functionality involves some guesswork. Experience will increase the quality of guesses, but in the beginning this seems far off. In Njulla, the stack is sparse, and there is little guesswork involved in finding the provider of some functionality. Even if one guesses wrongly it is highly likely that the guessed module will include a reference to the functionality. Because all functions are named according to providing module, it is a simple job to link such a reference to a specific interface.

Why choose Njulla over Arduino then? It doesn't have a significant community, but it does have one of the main properties that initially drew people to Arduino: Simplicity. The other property is readily available hardware at low prices. This property is not as crucial for SMEs as it was for the *maker community*.

9.4.5 Performance

The third dimension of Njulla's design space was the performance in the context of sub-sleep energy harvesting. To evaluate how the focus on the ease of debugging impacted the performance of the framework, I evaluate the boot sequence and the communication. I use the TelosB mote for comparing Njulla to TinyOS 2 (which was designed around the TelosB platform). It is schematically close to the Epic on which the first GreenMote was based. Since the GreenMote 1 is still supported by Njulla little trickery is required to program the TelosB.

For the boot sequence I cover boot time, power profile and finally energy requirements. For communication I analyze pin toggling and serial communication.

9.4.5.1 Boot Time

The boot sequence is one of the major contributers to the power consumption of sub-sleep energy harvesting sensor nodes. The boot time gives us a hint of the energy consumption and the variation of the boot time tells us how stable it is. The experimental setup for collecting the needed data is illustrated in figure 9.12. An Arduino Uno is used to toggle the power of a TelosB mote at 5V at a cycle consisting of 500ms high and 30s low levels. The pins of the Arduino's AVR microcontroller are rated at 40mA each (max 200mA combined). The TelosB can be expected to pull more than 40mA while charging capacitances during boot, so more than a single pin is needed. The only port of the AVR that has all 8 pins exposed is port D. All these pins can be set in parallel. I thus connect all pins of this port to VCC of the TelosB. The TelosB is programmed – at the user level – to raise a pin and then enter a while true loop. A logic analyzer is used to measure the time it takes for the pin to go high after VCC has gone high. Three

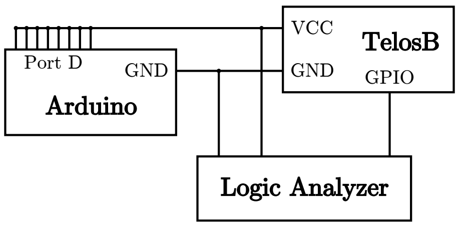


Figure 9.12: Experimental setup for measuring boot time.

software configurations are compared:

1. TinyOS performing the service of booting.
2. Njulla performing the service of booting.
3. Njulla performing the service of boothing, sampling an analog input and transmitting a frame. Here the TelosB pin is toggled to indicate transitions between step. The time used for comparison indicates the completion of the transmit operation.

Figure 9.13 illustrates the results of more than 1200 repetitions of each software setup as histograms. Figure 9.14 tabulates the key values and select derivatives. Although the boot time is rarely important by itself it is one of the key components of the energy spent when operating on harvested energy. This gives us reason to believe that Njulla may be more efficient for boot-sample-transmit cycles than TinyOS. The next experiment will dig deeper into this.

For homogeneous processes we can expect the standard deviation of the processing time to be constant relative to the mean. For Njulla this ratio drops when adding sampling and transition to the boot sequence. This indicates that the boot phase is less deterministic that the combination of sampling and transmission. With regards to the boot service alone TinyOS has a larger ratio than Njulla. This suggests that the boot procedure of TinyOS has components exhibiting nondeterministic temporal behavior. A likely cause is initialization routines waiting for some hardware operation(s) to complete and/or stabilize.

9.4.5.2 Boot Sequence

In the last test we saw the distribution of boot times. In this test I add the dimension of power draw and focus on a single sequence from each category. The experimental setup is illustrated in figure 9.15. It refers to PhonePower, an Arduino shield for measuring power consumption described in appendix B. The TelosB is again programmed – at the user level – to raise a pin and then

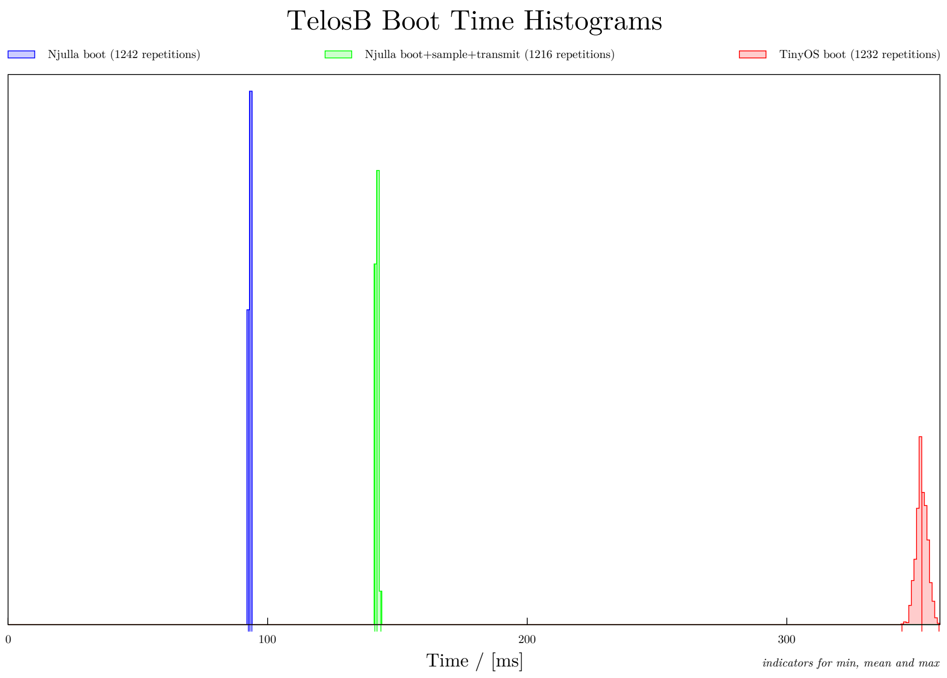


Figure 9.13: Histograms of boot times.

Software	Service	Mean	Normalized	Std. Deviation	Deviation/Mean
Njulla	Boot	93.17ms	0.26	0.32ms	3.42×10^{-3}
Njulla	B+S+T	142.12ms	0.40	0.37ms	2.61×10^{-3}
TinyOS	Boot	352.04ms	1.00	2.13ms	6.06×10^{-3}

Figure 9.14: Values for boot times.

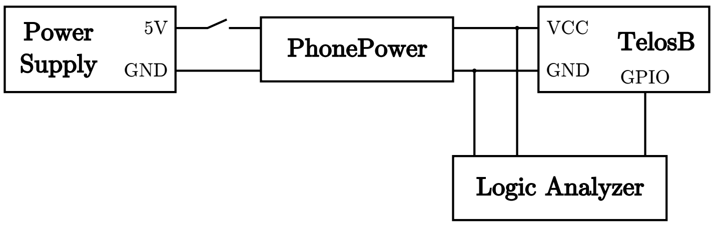


Figure 9.15: Experimental setup for measuring the boot sequence.

enter a while true loop. A logic analyzer is used to measure the time it takes for this pin to go high after power has been applied to VCC. The PhonePower board is mounted on an Arduino Uno and used to stream current×voltage pairs to a laptop at a known rate. The boot process is initiated manually by use of a switch.

By combining the datasets from the two instruments we can see how the power consumption evolves throughout the boot sequence. Figure 9.16 plots the dataset from two implementations of the boot service, one in TinyOS and one in Njulla. To place this in context I have added a plot of Njulla performing the boot+sample+ transmit service.

In the beginning the power consumption is hardware defined; capacitances throughout the board has to be filled. Then all boards enter a period of a relatively stable low consumption. The length of this period is dependent on the amount of global variables that have to be initialized (according to C specification). For Njulla, the increase in codebase caused by adding the functionality of sampling a pin and transmittting a packet increases the boot time from 91.6ms to 100.50ms. The increased number of global variables are likely contributors.

TinyOS boots in 348.84ms and consumes 4.36mJ in the process. Njulla boots in 91.6ms during which 1.66mJ is consumed. If the Njulla program is extended to sample and analog input and transmit the resulting value then the time increases to 139.30ms and the consumption to 3.37mJ. In other words, Njulla can boot, sample and transmit a packet in less energy than TinyOS needs to boot.

9.4.5.3 Waking from Sleep

So far the focus has been on the aspect of performance that matters for energy harvesting. TinyOS was designed to boot up once, and spend as much time in sleep mode as possible. The effort has thus been directed at minimizing the time and energy needed to wake up from sleep. This caused David Culler to suggest highlighting the differences in use cases by comparing the wake from sleep time. The MSP430f1611 employed by the TelosB has four levels of low

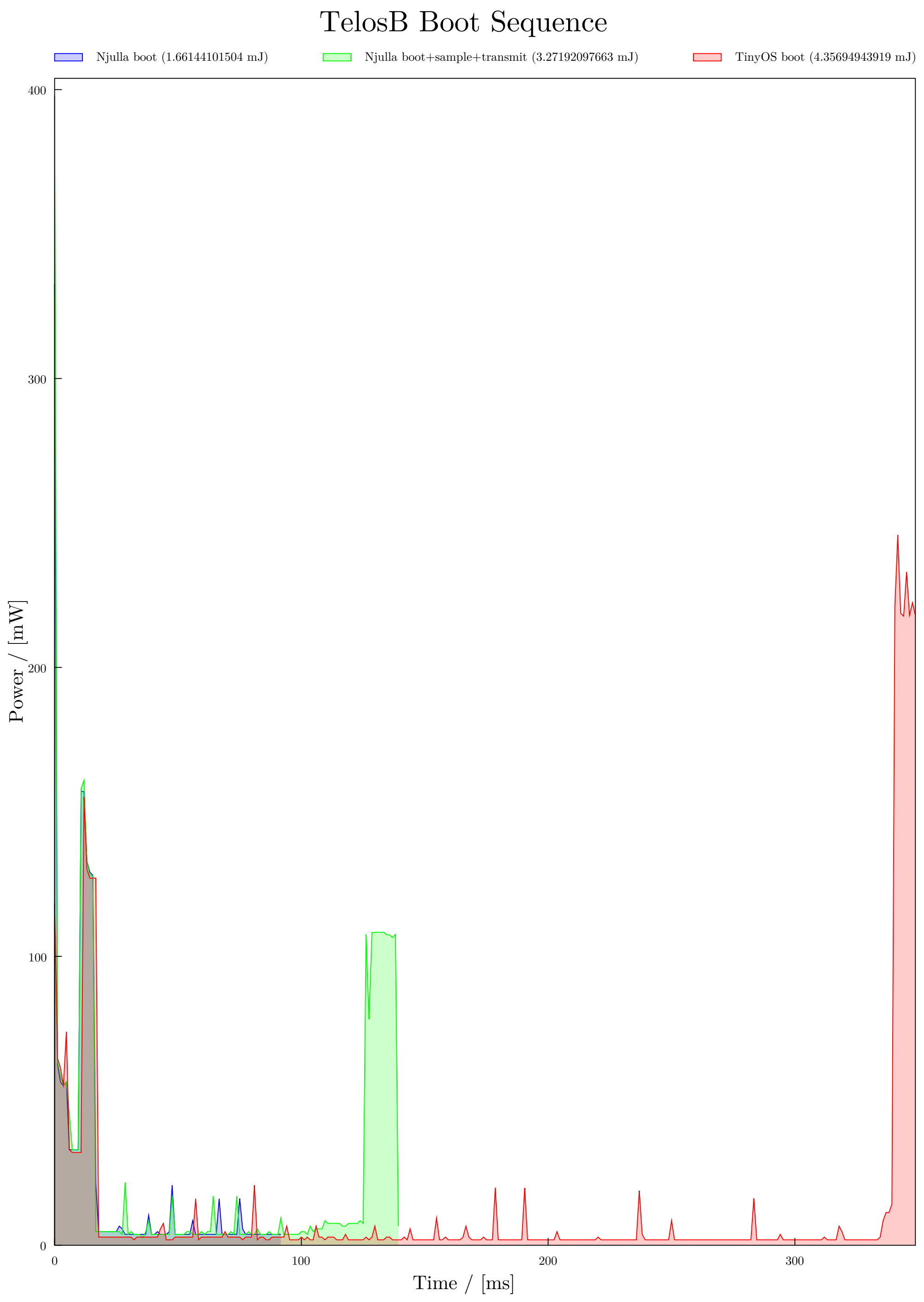


Figure 9.16: Power draw during boot for Njulla and TinyOS on the TelosB. Boot+sample+transmit for Njulla added for comparison.

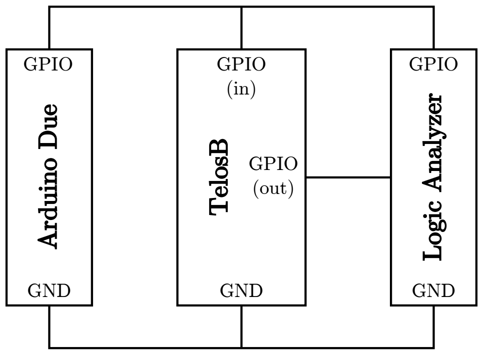


Figure 9.17: Experimental setup for measuring the time it takes to wake up from sleep initiated by a pin transition.

power modes (also known as sleep modes). These turn of the main clock thus keeping the program counter static, and – depending on the level – some derived and auxiliary clocks as well. I focus on pin interrupts which are supported by all low power modes. TinyOS should thus select the lowest power mode, LPM4.

The experimental setup is illustrated in figure 9.17. An Arduino Due board was chosen – due to its 3.3V logic level – to generate a clock signal. This signal was routed to an interrupt pin on the TelosB. The TelosB was then programmed to toggle an output pin twice whenever it received the interrupt and this had been propagated to user code. Both input and output of the TelosB was monitored by a logic analyzer.

The results of the test is tabulated in figure 9.18. TinyOS is two orders of magnitude quicker than the original variation of Njulla. After increasing the clock rate to match TinyOS, Njulla is still 22 times slower. After also matching the optimization level of TinyOS, Njulla is still 15 times slower. The handler itself is called after 5.98μs, meaning that the time spent propagating the signal by TinyOS is roughly equal to the time spent by the hardware. Njulla's main loop serializes userspace events as described in section 9.3.4.2. This was done to offer a guarantee of atomicity between user-code event handlers while keeping the framework reactive to new inputs. The cost of this approach represents an overhead of almost 200μs. This comes from (i) posting to and iterating through the flag-callback-context structure, and (ii) not modifying the interrupt vector dynamically to allow for more optimizations.

9.4.5.4 Toggling of a Pin

Pin operations are used for low-level communication and activation. I use the operation of setting the value of a pin as a representative of these operations.

Software	Time
TinyOS	13.09μs
Njulla	1671.60μs
Njulla v1	292.75μs
Njulla v2	202.18μs
Handler	5.98μs

Figure 9.18: Reaction times to a pin transition following LPM4 sleep.

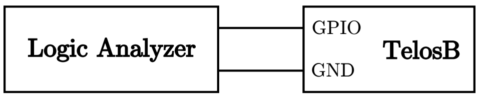


Figure 9.19: Experimental setup for measuring the time it takes to toggle a pin.

The experimental setup is – as illustrated in figure 9.19 – quite simple. A logic analyzer is connected to an output pin on the TelosB. The TelosB is programmed – using a selection of frameworks – to toggle this pin 21 times. This results in 20 measurable transition times. The average time between transitions is then calculated. The selection of frameworks are:

- ***TinyOS*** For comparison.
- ***Njulla v0*** This is the unmodified variant that was used int the boot sequence tests. The configuration and build procedure leaves it at a disadvantage compared to TinyOS. The two remaining variants are included to offset this in a controlled way.
- ***Njulla v1*** TinyOS raises the clock frequency during boot. So does this variant of Njulla. The settings used are: $\{\texttt{DCOCTL} \mapsto 196, \texttt{BCSCTL1} \mapsto 135, \texttt{BCSCTL1} \mapsto 4\}$. This configuration was found by programming TinyOS to dump them over a serial link.
- ***Njulla v2*** TinyOS does full program optimization and optimizes for size. This variant of Njulla gives the compiler the potential for inlining and optimizes for size as well.

Figure 9.20 tabulates the results. The difference in clock frequency has a linear – and thus significant – impact on the toggling time. This brings Njulla v1 close to TinyOS. The Njulla v2 time is less than half of that of TinyOS.

Software	Toggle
TinyOS	$9.77\mu s$
Njulla v0	$67.25\mu s$
Njulla v1	$11.84\mu s$
Njulla v2	$4.53\mu s$

Figure 9.20: Times for toggling of a pin.

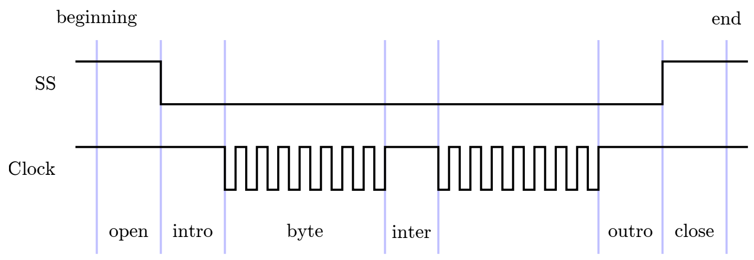


Figure 9.21: Relevant metrics for SPI benchmarking. The beginning and end marks the last and first user statement independent of the SPI communication. The interbyte time is averaged over 10 samples (11 bytes being transmitted).

9.4.5.5 Serial Communication

Serial communication is typically used for inter-chip communication. I have chosen SPI as a representative for these buses. Measuring SPI communication is a complex task with dependencies in both clock configuration and software stack. The software operates a serial unit inside the microcontroller. The rate at which this unit serializes a byte is defined by the clock that is fed into it. The serial unit needs configuration, the slave needs selection and deselection, and logic needs to decide what to happen in between bytes. This is controlled by the software stack. In figure 9.21 each of these phases are named and have an associated temporal metric.

Figure 9.22 illustrates the experimental setup I have used to capture these characteristics. The apparatus under evaluation are TinyOS and a series Njulla variants. These are:

- ***Variant 0*** This is the original Njulla framework used in the boot sequence analysis. The remaining variations are created to put the results into context. These are all small modifications at intuitive locations.
- ***Variant 1*** TinyOS raises the clock frequency at boot. In this variation Njulla matches that frequency by setting $\{\texttt{DCOCTL} \mapsto 196, \texttt{BCSCTL1} \mapsto 135, \texttt{BCSCTL1} \mapsto 4\}$. This configuration was found by programming TinyOS

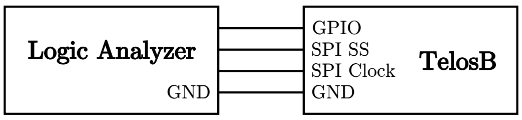


Figure 9.22: Experimental setup for measuring the SPI communication.

to dump them over a serial link. This variation reveals a bug, which is described in the next paragraph.

- ***Variant 2*** TinyOS does full-program optimization and optimizes for size whereas Njulla – in its current state – compiles modules separately, turns off optimizations and then links the modules together. This method was used to have a higher granularity while debugging. However, one does not necessarily preclude the other. In this variation Njulla fixes a bug revealed in the first variation and matches TinyOS regarding inlining potential (cross module compilation and optimization for size).

- ***Variant 3*** The interface of Njullas SPI module leaves something to be desired: The configuration of the a SPI bus and the operation of the slave select line are linked. For this reason configuring the SPI bus also selects a slave, and deconfiguring it deselects the slave. Furthermore, due to the ordering of operations the *intro* and *outro* metrics include some unnecessary configuration operations that really belong to the *open* and *close* metrics. TinyOS has a clearer (and better) differentiation between these. This variation follows the TinyOS split. Njulla does – unlike TinyOS – wait for the transmit buffer to be ready. I disabled this feature, but had to insert a delay to keep control over the resulting race condition.

The results are tabulated in figure 9.23. The original framework stayed on the startup frequency. The frequency/consumption tradeoff was never touched in Njulla, and the comparison is thus not fair. The v1 variant evens this out (as can be seen in the *Byte* column), but reveals a bug caused by a race condition: The slave is deselected before the last byte has been transceived. The increase in clock frequency revealed that the framework does not wait for the transceive operation to finish. Instead the path to deselecting the slave had become comparably shorter, resulting in the negative time. While the race condition is present in the original variant the bug remains dormant due to the low frequency. All later variants include a fix for this bug. The v2 variant illustrates the case where the compiler has been given the means to inline function calls. This removes a significant amount of overhead, and can be easily implemented in Njulla. The v3 variant illustrates the potential gains from a better functionality to interface function mapping. The additions of v2 and v3 are all present in TinyOS.

Software	Open	Intro	Byte	Inter	Outro	Close
TinyOS	85.00μs	9.37μs	14.45μs	10.59μs	10.16μs	38.98μs
Njulla v0	1084.19μs	422.09μs	21.15μs	210.36μs	197.68μs	155.28μs
Njulla v1	183.15μs	74.82μs	14.32μs	30.98μs	$-$ 0.19μs	45.56μs
Njulla v2	79.88μs	25.02μs	14.31μs	21.08μs	15.07μs	10.50μs
Njulla v3	90.02μs	9.01μs	14.31μs	4.75μs	10.06μs	10.45μs

Figure 9.23: Times for the components of SPI transmission.

9.4.6 Conclusions

Based on this evaluation I can revisit the hypotheses from section 9.4.

1. ***To make a mote programming framework attractive for SMEs the learning curve needs to be shallow.*** According to DELTAs research a typical SME budgets for $1-2$ weeks of extracurricular activities per employee per year. Furthermore, only $1-5\%$ of the turnover is used for research and development. This makes it critical for SMEs to have a low stepping stone to evaluate potential and explore business opportunities. With regards to tools in general and programming frameworks in particular, this translates to the need for a shallow learning curve. In this respect Njulla has a significantly better position than TinyOS (section 9.4.4).

2. ***TinyOS trade-off between high-level abstractions and performance does not allow for prototyping of sub-sleep energy harvesting sensor networks.*** I have found no basis for either confirming or falsifying this hypothesis. However, the high-level abstractions of TinyOS makes programs harder to debug (section 9.4.3) and represents a burden for SMEs to overcome in order to get started prototyping (section 9.4.4). Furthermore, while TinyOS performs well in serial and pin operation (sections 9.4.5.5 and 9.4.5.4), it has been optimized for an operational pattern where low power modes is used to save energy and boot done once (thus representing a constant energy expense). This is evident from the the experiment measuring the time it takes to wake from sleep where nearly half of the time is spent in hardware (section 9.4.5.3). It is also evident that trade-offs have been resolved in a manner that causes the boot sequence alone to consume significantly more energy than Njulla spends booting, sampling a sensor and transmitting the resulting value (section 9.4.5.2). As this is the typical scenario, it is clear that TinyOS – in its stock form at least – is ill suited for sub-sleep energy harvesting purposes.

3. ***Njullas trade-off between simple abstractions and performance is well suited for sub-sleep energy harvesting applications.*** Despite

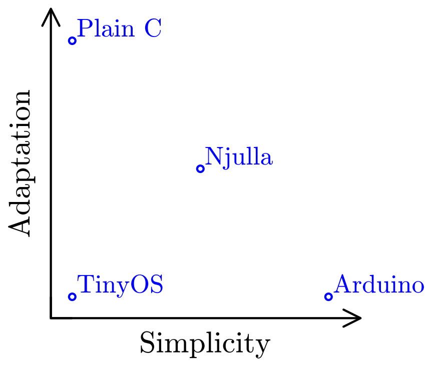


Figure 9.24: The design space of simplicity and adaptation to the sub-sleep energy harvesting regime.

> having the ease of debugging as the main focus Njulla consumes significantly less energy (compared to TinyOS) in the typical sub-sleep energy harvesting usage scenario: To boot, sample and transmit (section 9.4.5.2). This metric is the key when sizing the energy storage and harverster itself.

We are not yet at a point where the abstraction level can be linked directly to a sub-sleep energy harvesting relevant performance penalty. The main contributor is the boot sequence and none of the existing frameworks are particularly well suited for this. To clarify the situation facing SMEs, we can extend a design space along the axes of simplicity and adaptation to the sub-sleep energy harvesting regime. TinyOS is not simple and not adapted. Arduino is simple but not adapted[3]. Going without a framework in plain C would not be simple but adapted. Njulla is another point in this space and it fits somewhere in the middle; it is relatively simple and partially adapted. It is better in both dimensions than TinyOS. Figure 9.24 illustrates this space.

9.5 Discussion and Future Work

In the current incarnation of Njulla there are no abstractions on top of peripheral code. Code is thus not written for a generic radio, but for the specific CC2420. The interfaces of the functions implementing this code are defined in a general way to allow for abstract modeling and future abstractions. At the moment the routing library refers to the CC2420 code. If the framework is to grow it would make sense to introduce abstract classes of objects. This would however decrease visibility by introducing an extra layer.

[3]Applying the setup from section 9.4.5.1 to the Arduino Uno reveals that this platform takes more than 1.5s to boot in stock configuration.

Another approach is to modify the framework instead of building on top of it. Instead of having to go through a function on an abstract class – which only forwards the call to a function on a specific one anyways – the direct call would simply be made. The simplest implementation would be to make all functions on abstract classes macros which are mapped at compiletime. This of course means that the abstract class can only be mapped to a single specific class at any given time.

In many current microcontrollers (e.g. MSP430, AVR, Cortex M3) there is a single ADC to which several pins can be routed. This is often enough, but as a general design it is limited. If one wants to sample two analog sensors at the same time then external ADC's are needed. This is the case when voltage and current are sampled to calculate power. Multiple external SPI chips can be addressed simultaneously by having them share the same bus and selecting both.

Ideally the number of USARTs should match the number of peripherals. Vendors however try to lay out their product lines in such a way that getting an extra USART results in an upsell. As a result the microcontroller gets more expensive and covers more silicon. This results in higher power consumption. USARTs implement one or more serial busses. If one is looking for a small microcontroller it may be possible to get away with one that has a single USART supporting multiple busses.

In TinyOS resource arbitration is a key concept. Resources are requested and released throughout the stack. Some paths may request multiple resources, and the consequence of this is not obvious. If one path requests R_1 followed by R_2 concurrently with another requesting R_2 followed by R_1 then there are potential serializations resulting in a deadlock. Anomalies caused by such deadlocks can be very hard to explain. Introducing a two-phase resource allocation scheme modeled after two-phase locking would remove this problem altogether.

The job of the framework is essentially to process and route data between peripherals over the topology of the communication channels within the mote. This is a problem of concurrency. I have tried to limit the concurrency by dedicating functional units to single peripherals, but this can only remove part of the concurrency. Many of the complexities observed are byproducts of trying to map concurrency to soft logic executed by a single interpreter. Placing the logic in an inherently concurrent medium like an FPGA could potentially remove many of the current challenges. Of course, new ones would be introduced in the process, but it would be interesting to compare the two approaches. A less drastic approach would be to add distributed logic: A small MCU in front of the radio to take care of the send queue, another to perform sampling and yet another to serialize access to flash.

The strategy has been to not do any optimizations, but think enough about efficiency to know what is inefficient and how the known inefficiencies can be solved. The variations generated during the evaluation showed that this approach was flexible enough to allow the programmer to quickly resolve most of the inefficiencies found. Finding a way to improve the reaction time to hardware interrupts without sacrificing the ease of debugging is left as future work.

Chapter 10

Greenlab Testbed

The goal of the testbed is to allow for experimentation on sub-sleep energy harvesting motes. What distinguishes this from other mote testbeds is the impact of instrumentation overhead, which could be significant due to the fragility which stems from having extremely asymmetric components in the charge-discharge cycle. Limiting the overhead of instrumentation is thus a key design parameter. The one kind of instrumentation that is required is a backchannel. The backchannel is needed to upload experiment definitions, begin experiments and download experiment results. The backchannel is influencing the experiments because it is active at the time of experimentation. By separating the backchannel from the experiment in time we break this relationship. This is accomplished by (i) having the motes operate in either a service state or an experiment state, and (ii) having a software backchannel only in the service state.

10.1 Experiment Form

From the testbeds point of view, experiments take the form of a mapping from mote id's to full binary program images. This allows multiple motes to have the same identity during an experiment and two motes to execute significantly different code.

Figure 10.1 illustrate the states at a testbed scale. While in the service state each mote operates as router and endpoint. Experiment images are uploaded to motes according to the experiment mapping and the transition towards the experiment state begins. Because of the experiment images not having backchannel capabilities it is necessary to begin from the leaves of the routing tree and work our way towards the root. The experiment state of the testbed is reached when all motes are in the experiment state. Each mote then has the responsibility to

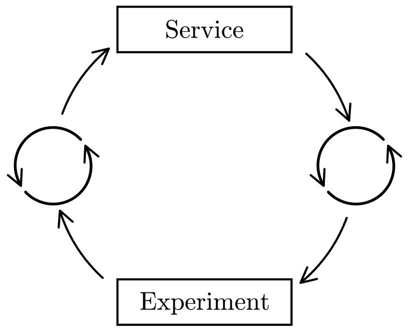


Figure 10.1: Testbed states.

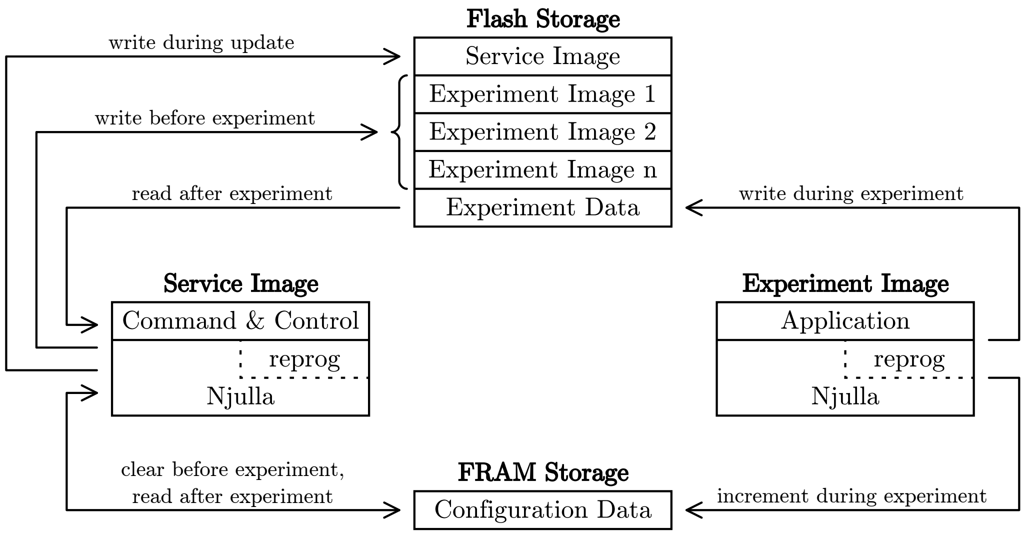


Figure 10.2: Flow of data between service and experiment images.

switch back to the service image at an agreed-upon time. When this happens the motes – and the testbed – switches back to the service state and we can download the results of the experiment. The results are what have been written to flash.

10.2 Mote Lifecycle

The motes run two different types of images. These communicate strictly asynchronously through two storage chips; a flash and a FRAM. Figure 10.2 shows the relevant flows of data.

Both involved images are build using the Njulla framework, and both uses its `reprog` library for switching. The service image implements a command and control service which will be covered in section 10.2.1. The experiment image includes the relevant Njulla libraries to implement the logic required to make the mote perform its role during the experiment.

The flash chip is used as the primary medium for storage. Here, program images and experiment data is stored. The service image is located at a static offset to eliminate the risks involved with forgetting to update a reference. The flash is byte addressable from the Njulla framework. During the experiment state experiment data is written to the flash. During the service state these are offloaded and experiment images can be written.

The FRAM is used to keep track of how much experiment data has been written to flash. For the experiment image the sequence is:

1. Read offset from FRAM.
2. Calculate index on flash from this offset and the address of the flash "partition" holding experiment data.
3. Write data to this index.
4. Write updated offset to FRAM.

This approach ensures that all data up till the offset are valid, even if something goes wrong during steps 1 through 3.

10.2.1 Service State

In the service state we need to be able to trigger the execution of code on any mote within the testbed from a general purpose computer (from hereon a *PC*). This depends on the ability to route commands to specific motes. Due to the energy harvesting nature of the experiments we also need to be able to configure the power subsystem.

The PC and the motes involved take on roles to accomplish this. Figure 10.3 highlights the relationship between these roles. The PC has a single role: To direct the testbed. All motes implement three roles; (i) that of the gateway connecting the motes to the PC, (ii) that of the router forwarding frames, and (iii) that of the target mote receiving commands and acting upon them.

The PC is running Linux and a Python script is used to direct the testbed. It does so by calling programs written in C. These programs perform high-level operations (e.g. upload a file to a mote) by issuing multiple low-level commands (e.g. erase flash segment). Each command has a matching function in the `cmd`

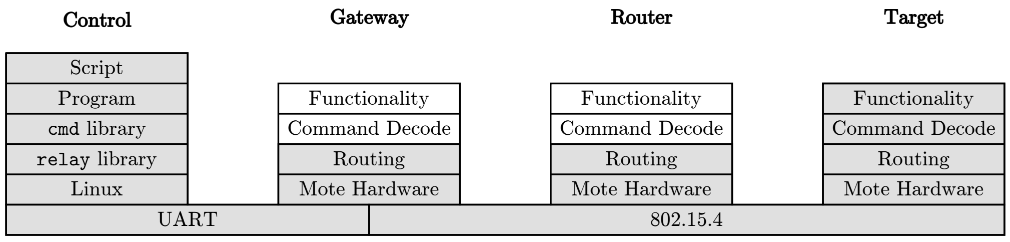


Figure 10.3: Active part of stacks for different roles in testbed.

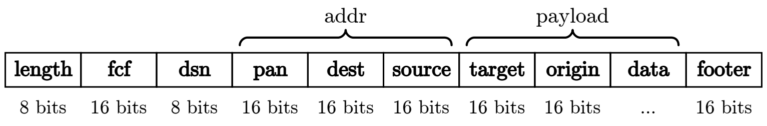


Figure 10.4: Frame format.

library, so that the complexity of retransmissions and waiting for reply is hidden. This library operates on top of another library `relay` which handles the UART connection as well as frame construction and deconstruction. When using these libraries a timestamped log of events is generated for later analysis.

Normally only a single mote is used as gateway, but this is due to reasons of simplicity on the PC. For reasons of simplicity on the motes all motes are potential gateways. This allows us to connect to any mote for debugging purposes. Due to a need to listen for frames signaling the return to the service state the active gateway does not participate in experiments.

When a frame has reached its target the contents is decoded. The relevant functionality is then looked up and called.

10.2.2 Routing

Figure 10.4 shows the frame format. It has fields for both hop-level source + destination and path-level source+destination (called origin+target). Having both results in a little overhead but is very simple to think about. The address part of the frame is used by the radio for filtering purposes. The data field is (8-bit) ASCII encoded except for parts containing inherently binary data. This improves visibility significantly.

A routing table is maintained, mapping path destinations to the id of the next hop destination and the link through which it is reachable. The table thus has 4 fields (this includes a field marking validity) and the lookup is implemented as a linear scan. The size of the table is determined at compile time by a constant. The routing tables can be populated from the PC from the root out, incrementally growing the routing tree.

At the moment the only supported links are the CC2420 radio and the UART. Adding support for other links is a matter of implementing the link logic and adding an entry to a switch construction.

The idea was to start simple, with the whole routing tree controlled manually by the PC. This was designed in a way that allows for later automation. Based on signal strengths – or other relevant metrics – the motes could update their own routing table to adapt to dynamics in the environment.

10.2.3 Command Instruction Set

I use the routing as a conduit for sending commands to motes and in return receive responses. Figure 10.5 lists the most important commands. Each of these commands were chosen for at least one of the following reasons:

- ***Functional*** Some functionality needed to be exposed in order to reach the desired level of system functionality.
- ***Verification*** CRC checks does not ensure that what was received matches what was sent. Adding functionality to do a final check before applying some configuration allows us to catch errors that would otherwise have left the mote in a faulty state.
  Example: When writing to the flash this is done one block at a time. For each block a transfer buffer is first cleared and then filled up using multiple frames (one frame cannot hold a large enough payload). The mote is then asked to calculate a CRC for the entire buffer and this is checked on the PC. If the check is favorable the buffer is flushed it flash, otherwise the process is repeated.
- ***Visibility*** Allowing inspection of internal state. This can be used to track down the cause of an anomaly.

For interactive debugging I implemented a simple shell on top of the `cmd` library. It allows one to construct and send frames by declaring source, destination, origin, target and data fields. Any frames received by the PC are printed to the screen. This provides us with a quick way of sending arbitrary sequences of commands while observing the responses. To aid in locating potential issues I made sure to keep a high granularity of commands: Functionality was implemented through more commands than strictly necessary.

The `command.*` commands allows one to inspect a mote without knowing the exact version of the service image. The `build.*` commands allows one the extract

`led.on ID`	Turns on led ID
`led.off ID`	Turns off led ID
`ping`	Echos frame
`link.add T L R`	Adds an entry to the routing table for target T using router R on link L
`link.remove T`	Removes an entry from the routing table
`sensor.subscribe`	Substribe to periodic readings from an onboard sensor
`sensor.unsubscribe`	Unsubstribe from periodic readings from an onboard sensor
`buffer.clear`	Nulls out buffer
`buffer.crc`	Calculates and returns a CRC from buffer
`buffer.read O L`	Read L bytes from the offset O in the buffer
`buffer.write O L D`	Write L bytes from D to the offset O in the buffer
`buffer.flush B`	Flush the buffer to block B in the flash
`buffer.load B`	Load block B of the flash into the buffer
`buffer.state`	Does the buffer contain data which has not been flushed
`buffer.volatile V`	Set the state of the buffer to V
`experiment.set S_1 S_2 H T D`	Set experiment properties (see section 10.3 for details)
`experiment.get`	Get experiment properties
`experiment.do`	Begin experiment
`build.lenght`	Get number of meta information key×value pairs
`build.index I`	Get the I'th key×value pair
`config.read A L`	Read L bytes of data from address A of the FRAM
`config.write A L D`	Write L bytes of data from D to address A of the FRAM
`command.length`	Get number of commands
`command.index I`	Get the I'th command name

Figure 10.5: Testbed command instruction set.

build information about the service. By linking this to the log of the versioning control system one may be able to explain observed anomalies.

10.3 Power Subsystem

The power subsystem is reconfigured while the mote is in the service state. When the mote has reprogrammed itself it sends a command to the power subsystem to start the experiment. The AVR then applies its configuration and cuts power to the mote. The experiment has started. When the experiment timer runs out the power subsystem indicates the end of experiment and switches the mote on to stable power. The end of experiment indication is registered on the mote as an interrupt and this triggers the reprogramming into the service state.

10.3.1 Configuration

When configuring an energy harvest power supply system the parameter space has several dimensions. The important ones are:

- ***Source*** Which harvester to use. Choices include type (photovoltaic, thermal and vibrational) as well as size.
- ***Storage*** How energy is stored. This includes the configuration of capacitors, including sizes and types.
- ***Window Comparator*** The high threshold. It doesn't make much sense to change the lower level, as there is an optimal one. This is the lowest voltage at which the mote can be expected to operate within specs.

Two extra parameters determines how the switching between states on the mote level is performed:

- ***Time*** How long time before supplying stable power and indicating the end of experiment.
- ***Discharge*** Whether to discharge the storage before the start of the experiment.

All of these parameters can be configured through the `experiment.set` command. Figure 10.6 illustrates the relationship between these parameters.

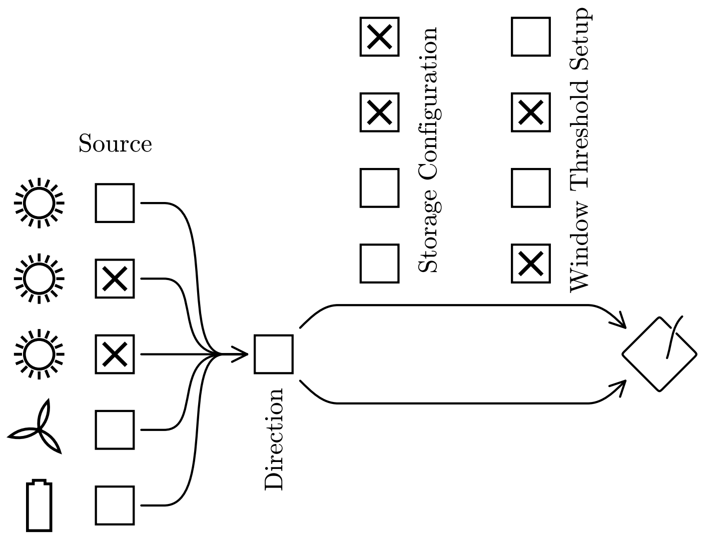


Figure 10.6: Parameters of an energy harvesting power subsystem.

10.3.2 Interface

The power subsystem is implemented as a separate PCB using an AVR microcontroller running Arduino code. To limit the influence of instrumentation the board is electrically isolated from the mote using optocouplers. It operates on stable power (e.g. batteries). Through a series of digital switches it routes power from a subset of sources to a subset of storages. It implements a software window comparator by using a comparator IC to do thresholding against a subset of a resistor bank. This subset represents the high threshold.

The board communicates with the GreenMote4 through the USART_B1 functional block. SPI is used as bus and the chip select line is used to wake the AVR so that it can spend most of its time sleeping. This gives the switchboard an expected battery life of 1+ years without interaction, months with the expected amount of interaction. A GPIO line is used to indicate the end of the experiment.

The protocol used to communicate with the AVR uses the first byte transferred as an upcode. Commands exists for setting the relevant parameters and for starting an experiment. All bytes send are echoed back to allow for verification.

10.4 Evaluation

The evaluation of the GreenLab testbed is directed at the overhead involved in the operation of the testbed. The main hypothesis is *A testbed without* *backchannel is well suited for sub-sleep energy harvesting motes intended for outdoor deployment.* However, I have neither the tools nor the qualification to test the overhead in terms of power profile fidelity. This degrades the main hypothesis to *The temporal overhead for a testbed without backchannel is not significant for experiments on sub-sleep energy harvesting motes intended for outdoor deployment.* This in turn leads to two sub-hypotheses:

1. The temporal overhead of a testbed without a backchannel is not significant for experiments powered by sub-sleep energy harvesting.
2. A testbed without a backchannel can scale.

To evaluate these I identify 5 main components from the GreenLab experiment model:

1. Upload of an experiment program image. This is quantified in section 10.4.3.
2. Switching to the experiment image. This is quantified in section 10.4.4.
3. Performing experiment. The experiment itself is necessary, and does thus not contribute to the overhead.
4. Switching back to the service image. This is quantified in section 10.4.4.
5. Downloading any resulting data. This is quantified in section 10.4.5.

Components 2 and 4 are two applications of the same functionality – reprogramming – and thus evaluated together. The remaining components are evaluated separately.

10.4.1 Testbed Experiments

As mentioned in section 2 sub-sleep energy harvesting changes the scale of things. When single cycles can take up to multiple weeks, a two-week long experiment may very well be required to get enough cycles for a decent signal. A 1% overhead of such an experiment is 3 hours 22 minutes. This is the context in which the evaluation should be performed.

10.4.2 Input-varied Distribution Plots

Given a significant number of repetitions of an experiment that maps a single input value to a single output value, how can we visually present the outcome of all repetitions? Multiple solutions exists for this problem, including

- ***Median Value*** The median value is an easily understandable way to simplify a dataset.
- ***Errorbars*** Errorbars can be used to add the range covered.
- ***Standard Deviation*** The standard deviation hints at the distribution, but we really don't know.
- ***Multipoint*** Plotting every single point will display all results without loss of information. However, the information is not readily available. When the number of repetitions crosses a handful, it becomes hard for humans to extract the distribution.
- ***Histogram*** A histogram solves this problem, but takes up an extra dimension and needs a significant number of repetitions to be reliable.
- ***Heatmap*** A heatmap is essentially a histogram which relies on color or intensity instead of form. It inherits both negative properties.

What happens when the input value becomes a factor in the experiment? Then the problem expands to dealing with multiple values for multiple inputs. For each discrete input we can apply the same methods as before. However, not all methods expand gracefully. The histogram will refer to the secondary axis and take up space on the primary axis. This is not intuitive. Something similar happens to the heatmap, where the expectation is that each value covers a range on both axes. Despite this, I have – due to their superior distribution presentation qualities – chosen to use histograms for representing such datasets. The histograms I use are mirrored across the input value for centering purposes. Figure 10.8 exemplifies such a plot.

10.4.3 Upload of Program Image

The upload of experiment images is the most significant contributer to the overhead of running the testbed. To measure the time it takes to upload an image I configure the testbed to employ a linear routing topology as illustrated in figure 10.7. The current implementation of the upload procedure loads the hex file, parses it, and recreates it on the target node by transmitting the complete (ROM) address space. For this reason, the upload time can be considered independent of the size of the actual program image. Accordingly, we keep the program image size constant during this test. The only factor is the number of hops, which translates to a node id. The test was conducted in a 12m^2 office. 76 repetitions were performed.

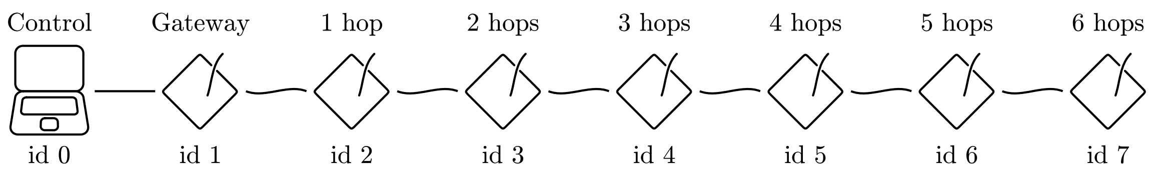


Figure 10.7: Experimental setup for measuring routed transmission time.

For each hop distance I expect to see a distribution which is derived from a normal distribution. Several events may offset parts of the distribution. They include (i) packet loss causing missing ACKs, (ii) transmission errors causing CRC checks to fail at block granularity, and (iii) flash write errors causing validation errors. Each of these events trigger a response after a static amount of time. The distribution should thus be a sequence of normal distributions and the width of the distribution should naturally follow the number of hops. Furthermore, the quickest result from each of the hop distances should fit a line with a static component – representing time used for work performed on the controlling computer, the gateway and the target node – and a dynamic component representing the per-hop routing cost. This line represents the best case.

The result of the test is visualized in figure 10.8 using the method described in section 10.4.2. Retransmissions can be seen as separate trailing distributions at one and two hop distances. These are lacking multiples of 5s behind the ideal. The 5s is the current timeout for retransmissions. It could be optimized according to individual command types. The best-case line is $(41.97h + 127.92)$s, where h is the number of hops.

The offset of 127.92s has an obvious contributor; the link between controlling computer and the gateway is a 9600baud serial link. This speed could be raised. The dynamic component of 41.97s/hop represents the time spent routing. This could be lowered by employing full-program optimization. In sections 9.4.5.4 and 9.4.5.5 I have highlighted the effect of this optimization. Implementing optimizations for only transmitting the relevant parts of the address space, would save time in all components. For now I define the upload penalty as:

$$t_u(h) \quad = \quad (41.97h + 127.92)\text{s}$$

Real deployments have unique radio propagation properties. This makes it hard to derive anything beyond the best case.

10.4.4 Reprogramming

In the configuration phase of each experiment, all motes involved are reprogrammed. I use the setup from figure 10.9 to measure how long this takes. The

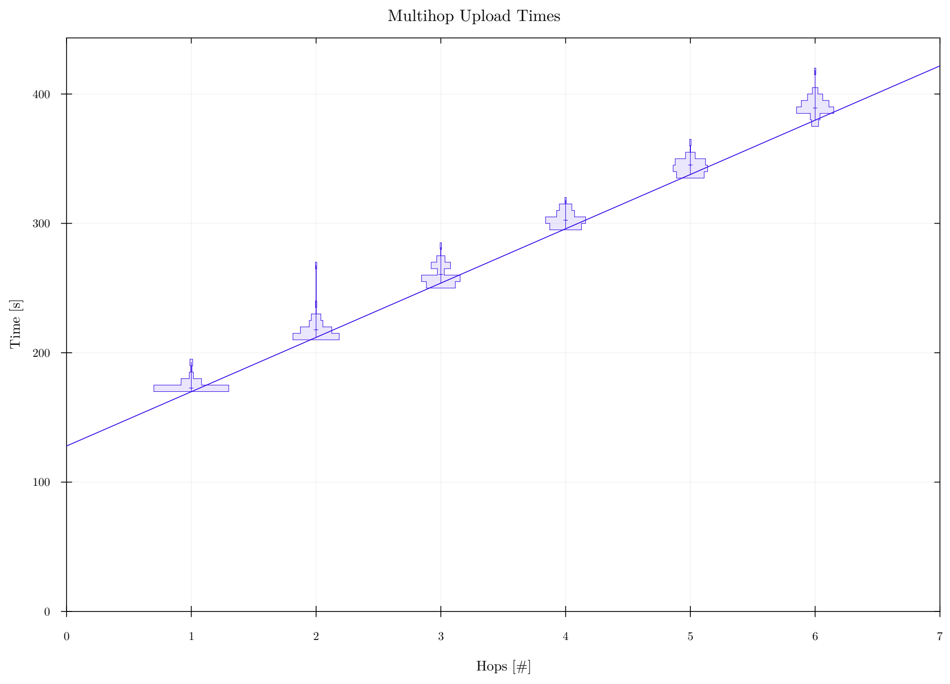


Figure 10.8: Full-program image upload times.

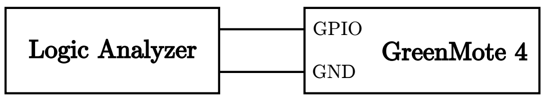


Figure 10.9: Experimental setup for measuring single mote reprogramming time.

mote in question is programmed to be a part of the testbed network. An experiment image is then uploaded and the mote is instructed to reprogram itself using this image. The image pulls a pin high, wait a small amount of time, pull the pin low, and then reprogram itself using the same image. This causes the process to loop. The reprogramming time can then be measured as the time where the pin is low. This test is repeated 670 times to get a good model of the distribution.

The reprogramming phase includes copying the program image from flash to ROM and then rebooting the mote. Copying the program image involves iteratively loading each flash block to a flash buffer, and then fetching it. The loading command has to be polled for completion. Besides this, the only other operation involved that is not operating deterministically based on clock is the microcontrollers hardware boot sequence. This causes me to expect a normal distribution with a small standard deviation.

The resulting distribution of reprogramming times is illustrated as a histogram

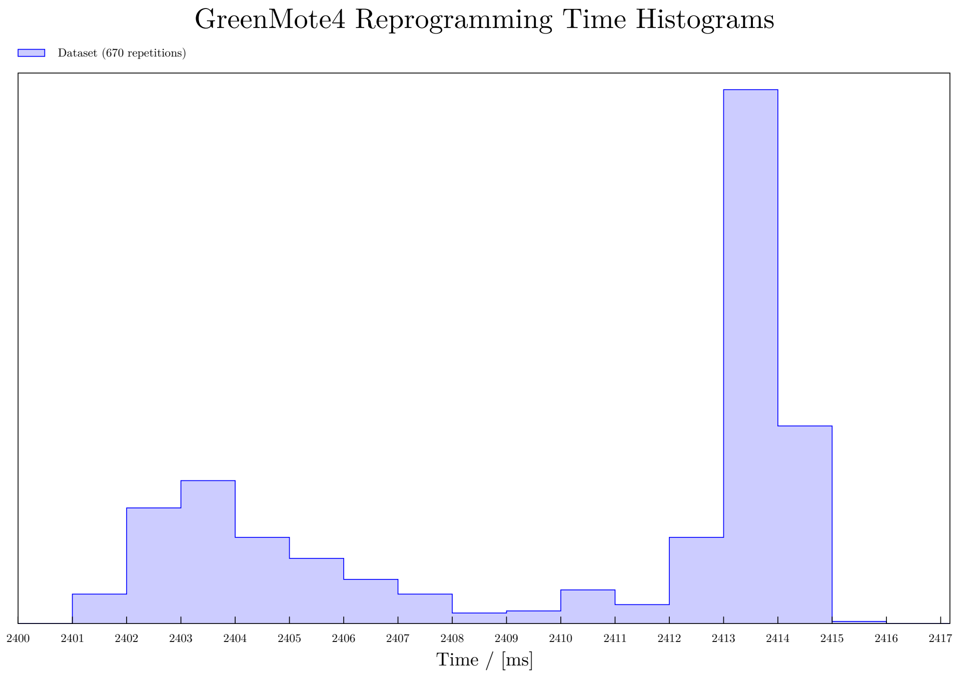


Figure 10.10: Histogram of reprogramming time.

in figure 10.10. The mean is 2409.48ms and the standard deviation is 4.66ms, or 0.19% of the mean. However, it is not normally distributed. It seems to be consisting of at least three distributions, around 2403ms, 2410ms and 2414ms. From section 9.4.5.1 we know that the standard deviation of the boot time for a similar image is 0.37ms. This suggests that distribution of the flash load operation has two peaks, perhaps due to some internal error checking. I define the reprogramming penalty as:

$$t_r = 2.409\text{s}$$

10.4.5 Download of Experiment Results

Some experiments may have stored data on the flash chips of the involved motes. For most experiments this will be a very small amount, if any at all. To measure the time it takes to download result data I used the same experimental setup as in the upload test. This is illustrated in figure 10.7. The test involves two factors: The number of hops and the workload (the amount of data to download). The current implementation of the testbed allows downloads of an integral number of blocks; where each block contains 512B. The test was conducted in the same 12m^2 office which was used for the upload test. 76 repetitions were performed.

This test is closely related to the upload test. This is mirrored in the expectations, which are essentially the same, only with the added dimension of the workload. The expectations are:

- Each hop $\times$ workload combination has a distribution which is derived from a normal distribution. Parts of this distribution are delayed – due to reasons described in section 10.4.3 – so that the result is a sequence of normal distributions where the width of the distribution follows the median.
- For each hop distance, the quickest result from each workload distribution should fit a line. This line has a static component – representing the work performed on the controlling computer, the gateway and the target node – and a dynamic component representing the per-block processing cost. The static components should be shared while the dynamic components should be a linear function of the hop count.

The results of the test are visualized in figure 10.11 using the method described in section 10.4.2. Each hop distance is represented using a unique color; with one hop having the shortest execution time and 6 hops having the longest. For each hop distance a line is fitted to the best results of each distribution. These lines represents the best-case scenarios, and are tabulated in figure 10.12.

From the plot it is unclear how well the datasets fit the multiple normal distributions model. However, the retransmissions are easy to spot. They are trailing multiples of 5s behind the main distribution. As mentioned in the upload test, the 5s is the current timeout for retransmissions. It could be optimized according to individual command types.

Looking at the set of formulas from figure 10.12, there is an obvious pattern to the offsets: They are in order. The reason behind this is that there is a *hidden* 0th hop in the serial line between the controlling computer and the gateway. The offset follows a linear function of the hop count: $0.11h + 0.28$ where h is the hop count. The angle of this function is (0.11 ± 0.0011)s/block/hop and the base is (0.28 ± 0.0044)s/block. This is a pretty good fit. The multiplicand component also follows a linear function of the hop count: $0.45h + 1.37$. The angle of this function is (0.45 ± 0.00029)s/hop and the base is (1.37 ± 0.0011)s. This is also a good fit. Putting it all together, we have that the download time is given as:

$$\begin{aligned} t_d(h, w) &= ((0.45h + 1.37) \cdot w + (0.11h + 0.28))\text{s} \\ &= (0.45hw + 1.37w + 0.11h + 0.28)\text{s} \end{aligned}$$

10.4.6 Conclusions

Based on this evaluation I revisit the hypotheses from section 10.4:

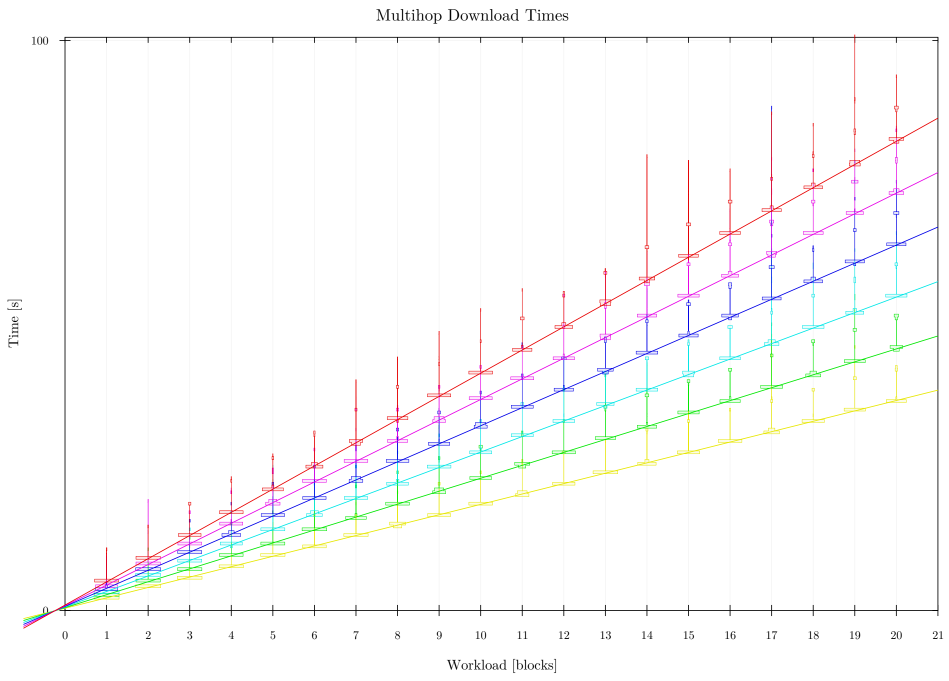


Figure 10.11: Multihop download times.

Distance	Formula
1 hop	$(1.82w + 0.40)$s
2 hops	$(2.27w + 0.50)$s
3 hops	$(2.72w + 0.62)$s
4 hops	$(3.17w + 0.72)$s
5 hops	$(3.62w + 0.83)$s
6 hops	$(4.07w + 0.95)$s

Figure 10.12: Best-case download models. The formulas as functions of the workload, measured in blocks of 512B.

1. ***The overhead of a testbed without a backchannel is not significant for experiments powered by sub-sleep energy harvesting*** Each of the sections 10.4.3, 10.4.4 and 10.4.5 quantifies the best-case overhead of a specific component. Putting these together gives us the scaling function of the overhead:

   $$\begin{aligned} t(h,w) &= t_u(h) + 2 \cdot t_r + t_d(h,w) \\ &= 0.45hw + 1.37w + 42.08h + 133.018 \end{aligned}$$

   If we have 20 motes, each 4 hops away from the gateway and producing 2 blocks worth of data in a single experiment, then the overhead will be $20 \cdot t(4,2) = 102\text{m}33\text{s}$. As argued in section 10.4.1, this number should be considered in relation to the length of the experiment. If we set the length of the experiment to 2 weeks, then the overhead ends up being 0.51%. I do not consider this to be significant.

2. ***A testbed without a backchannel can scale*** There are two variable inputs to the testbed experiment process: the number of hops and the download size. Section 10.4.3 shows that the upload process is linear in the number of hops. Section 10.4.5 shows that the download process is linear in both the number of hops and the download size. No other component depends on these parameters.

The operations GreenLab uses to perform experiments – upload, reprogramming and download – are not optimized. The absolute overhead could be improved significantly. However, given an expected typical experiment duration of days or weeks the current implementations relative overhead becomes insignificant. The overhead scales linearly with the amount of data to be downloaded and – at least in the range tested – the number of hops. With the potential exception of the power profile, this makes the testbed well suited for experimentation on sub-sleep energy harvesting motes intended for outdoor deployment.

10.5 Discussion and Future Work

This testbed was designed mainly to answer high level questions. *How frequently can I get a sample-store-transmit cycle from a solar panel of this size under typical conditions?* The answer is seeked by a single experimental setup. It gives us results which are easy to apply. The GreenLab testbed fulfills this function. However, the lower level insights needed to reason about the design of a system are not covered: *How do I design a system for this many sample-store-transmit cycles under typical conditions?* This knowledge needs to either come from somewhere else – in which case it can be verified using the testbed – or be build on top of the results of multiple testbed experiments.

The mote $\mapsto$ image mapping enables experiments to give a subset of the motes a monitoring role. They can run on stable power and log observed properties of ongoing experiments, as observed by means of eavesdropping.

It is still unclear how data generated by experiments should be stored and exposed. At the moment I simply dump experiment results to the filesystem in its raw form and use C programs to convert the binary data into a CSV file.

The current approach has one fundamental weakness: For a mote to return to its service state it needs to (i) have a valid experiment image loaded, and (ii) have interrupts enabled. If for any reason an invalid image is loaded (e.g. one not listening for the interrupt) or interrupts are turned off, then the mote will become stuck and need physical intervention. This is a direct result of the two-state approach, and it *may* not be necessary. The assumption of instrumentation interference is not quantified and the optocoupler approach used to keep electrical separation between the mote and the power subsystem could just as well be applied to the link to a deployment support network[18].

At the moment each mote is connected to the power subsystem, and this is essentially another mote. Add a radio and some storage, and the only missing piece is a reliable source of power for it to take the role as a support network. As the testbed is designed for outdoors operation this gets a bit tricky. Long wires would not be acceptable, but in some situations oversized solar panels may provide a reasonable choice. There are thus gains to be made from switching to a deployment support network model, but this requires stable energy sources at each target mote.

The approach taken for the GreenLab testbed gives a high level of visibility. This allows us to observe the inner workings and reason about faults. These are positive infrastructure properties which could be leveraged in other areas. The type of infrastructure employed by the testbed could be used in deployments with a stable source of power. This would result in an easily debugable pipeline from the gateway to logic on the individual motes. What is missing to make this work, is a layer of security. Enough security should be added to ensure that only authorized communication is accepted. Ideally one should be able to enable and disable the security so that most debugging sessions can take place without being obscured by it.

Chapter 11

Debugging a Mote Program

In this section I cover a selection of debugging aids, give two examples of their use and present a methodology for debugging. Section 11.2 contains the first example covering mote-level debugging. Section 11.3 contains the second covering PC-level debugging.

11.1 Tools

While coding the tools available relates mostly to active ease of debugging; lowering the risk of the existence of a bug. Static analysis, simulation and decomposition design help us spot bugs before deployment.

Under normal execution a bit can be observed at low frequencies through a led. At high frequencies a logic analyzer is needed to observe a bit. The logic analyzer is also capable of observing multiple bits over time. Byte granularity observation is best done using an FTDI[1] or a logic analyzer, depending on format. Logging to RAM is also an option.

For debugging purposes we often want to synthesize execution; trigger execution of a slice of logic. This is used to verify state transitions. The choices of decomposition define to which degree this is possible.

11.1.1 Static Analysis

Without a dedicated set of tools static analysis of the codebase is a matter of *"being very careful"*. This involves going through the code and considering what

[1]An FTDI is a chip implementing two-way USB $\leftrightarrow$ UART translation.

could go wrong. This is what you do when you don't have any better options and the cost of an anomaly is too severe to ignore.

11.1.2 Serial

At times one can use a serial link for printing out debug information. To observe the printouts an extra device needs to be connected when they happen. This device would typically be a PC, but dataloggers could be used for deployed motes.

Transmitting a human readable string over a serial link takes significant time. Time sensitive operations can be affected. The use of serial debugging should be limited on deployed motes during normal operation.

11.1.3 Logging to RAM

Logging can sometimes be done in memory. Our flash library supports self-checking writes by performing loops of write-read-compare operations. I have a compile-time option to log the writes and reads of a single self-checking write. If the write fails (the maximum number of retries is reached) then the log is dumped to the UART link.

We can do this because (i) we have enough memory, and (ii) it is clear when the data is relevant. In this situation we need the entire log, so we must allocate space enough for that. For other problems it may make sense to employ a circular buffer to limit the memory consumption. This comes with the added problem of when to freeze the buffer. Unless stopped the circular buffer will be overwritten given time. In order to keep important data from being lost it is essential to detect interesting events and act on their occurrences, either by freezing the log or by dumping it over an UART.

11.1.4 Leds

Motes often have three LEDs of different colors attached to GPIOs. The number is arbitrary. Eight would have been preferable, but that requires both physical space and pins. These leds can be used for signaling a state during deployment or debugging information during development. To some degree the leds take the role of `printf` on motes.

The problem with toggling leds is that it can be very hard for humans to decode. Relative to execution time it takes ages for a human to register that the state of an led has changed. If multiple leds are used we additionally have to remember the pattern.

11.1.5 Logic Analyzer

A logic analyzer is a device capable of logging the state of a significant number of digital lines. This includes leds as well as almost any other line on the mote PCB. Tools exists to visualize and process these logs.

This allows one to reason about temporal relationships between events happening at frequencies impossible to observe with the naked eye. It allows for protocol decoding so that a serial communication is shown by the byte-values of the payload instead of a sequence of bits also covering control information. At last we can measure time precisely to locate bottlenecks or benchmark operations.

The Open Bench Logic Sniffer is a 50$ logic analyzer build around an FPGA and a PIC18 microcontroller. All code – including microcontroller, FPGA, schematics and client – is open source. It has 32 digital input channels which can be sampled at up to 100Msps (or 16 channels at 200Msps). The typical mode of operation is to configure a trigger (e.g. some line going high) and have the logic analyzer wait for that event to occur. When it does the logic analyzer will begin sampling to an on-board flash and when that is full transfer the logged data to a computer for analysis. The flash has the capacity to store 128k samples. Sampling 8 channels at 1024Hz will thus allow for 16s of sampling. Note that the sampling theorem still applies.

11.1.6 Decomposition

One implementation strategy is to design for inspection. This means decomposing the logic to make space for inspection sites, and exposing these through easily accessible interfaces.

Conceptually most applications consists of layers. We even refer to a *stack*. We associate each layer with state, some logic that operates on this state and interfaces to neighboring layers. By monitoring the interface we can localize an anomaly to a layer and by observing how the state reacts to interface stimuli we can even reach the granularity of a segment of logic; a group of related functions. Obviously, we can't have this level of instrumentation active all the time, but if access is available and the need arises then we can inject activity through the interfaces and data through the logic.

We can, however, work with the interface granularity. Complex operations can be split up in verifiable parts. Having a function for write n bytes to address a doesn't give us much to work with if the result of the operation turns out to be faulty. If instead we realize that it is a read-modify-erase-write operation and expose each of these components then the visibility has improved. Note that this can be done while keeping the old function, only now as a thin wrapper.

11.1.7 Simulation

Another option would be to simulate, preferably of unmodified machine code. Virtual leds, serial devices and logic analyzers could be used for instrumentation. Snapshots could be used to dump the state and fork the execution around inspection sites. Then the virtual instruments could be used to probe the state and inspect the response.

Even a dumb implementation of peripherals would allow us to verify implementation details. In the most basic form a peripheral could be implemented as a separate inspection site which blocks simulation. Synthetic traces could then be used to first construct the desired state and then prod it to observe behavior. A proper simulation of the peripherals would mostly obsolete the manual construction of such traces.

All of this depends on peripherals exhibiting correct – or at least expected – behavior. An execution is essentially a path of a search tree. Each access to a peripheral holds the potential to cause or expose a fault. By presenting this tree to the developers and letting them interactively chose which subtree to explore the risk of missing a case can be reduced.

To account for faults caused by race conditions a instruction/state dependency graph should be maintained during the simulation. All potential flows of different results should be presented and the differences in state highlighted so that the developer can choose which subset of potential outcomes to proceed with.

11.1.8 JTAG Debugger

JTAG Debuggers attach on the JTAG header of a mote. This header is routed to the microcontroller where it exposes a low-level interface. Through the JTAG code can be uploaded to the microcontroller ROM, the program counter can be controlled and registers read. This includes the ability to set breakpoints at program addresses and step through instructions.

The exact implementation of this differs from implementation to implementation. Two limitations exists though; (i) timing is lost at the first breakpoint so the state

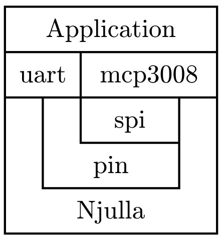


Figure 11.1: Stack for MCP3008 support.

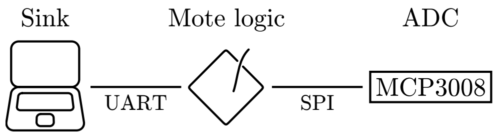


Figure 11.2: Experimental setup for lowlevel debugging example.

of the mote is no longer consistent, and (ii) the power needs to be cycled after attaching the debugger. It is still a powerful tool though.

11.2 Low-level Debugging Example

This example goes through the process of adding support for the Microchip MCP3008 8-channel 10 bit AD converter. The intended software stack is illustrated in figure 11.1.

11.2.1 System

We enter the process at the point where the code has been written and it compiles. Now it needs to be tested and debugged. Figure 11.2 illustrates the experimental setup. The mote is programmed to stream samples from the external ADC over an UART link to a PC.

11.2.2 Process

At first, nothing worked. This was obvious from listening on the UART which was silent. After connecting the logic analyzer it was revealed that the only signal being transferred between the MCU and the ADC was the SPI clock. There was quite a lot of noise though.

Checking the setup revealed that the physical pin used as *chip enable* did not match the one referred to in the program. This was corrected and then this line was visible on the logic analyzer.

At this point the experimental setup was moved to the middle of a nearby hallway to combat noise on the lines. The PC also had to run from batteries. Still noise appeared, but significantly less. The MOSI line appeared to be flat though.

After consulting a colleague I set up the directions of the MISO, MOSI and clock pins before communication. Apparently this wasn't done by the functional unit and the noisy pins had been floating. The setup was moved back into the office.

With visibility that could be trusted, the debugging sped up. First, the principal code for operating the ADC was moved from the library into the application layer for simplicity. Then it was discovered that a dummy command was being transmitted. After encoding the command according to the ADC datasheet the MOSI line became active.

From the logic analyzer visualization it was clear that the bit order was reversed compared to the datasheet. This was quickly fixed, but still didn't result in a reply from the ADC.

At this point, revisiting the datasheet revealed that the chip select pin is called $\overline{\text{CS}}$, and not CS. This means that it operates on reversed logic levels. Accordingly, there was no overlap between the times at which the MCU was sending and the times at which the ADC was receiving.

This gave us a reply on the SPI line, but not the UART. After some frustration I decided to try some old sample code for the UART and it also didn't work. I then consulted the settings of the serial console used on the PC and got a number through. The logic analyzer would have caught this mistake, but I expected it to be working and thus chose not to instrument it.

The number was wrong though; 1531 is simply too big when it comes from a 10 bit converter. The logic analyzer revealed that the correct value was being transferred on the SPI line so the bug was likely to be found in the decoding section. This ADC transfers its 10 bit value across 3 bytes resulting in some shifting operations in the decoding code. There was an off-by-one error resulting in the most significant bit being twice as significant as it was supposed to.

In the end the code for operating the ADC was moved back into its respective library and cleaned. Tests revealed that the code still worked.

11.3 High-level Debugging Example

This example lists the steps taken to locate a bug in the GreenLab testbed.

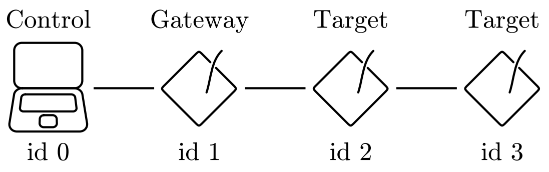


Figure 11.3: Experimental setup for highlevel debugging example. Routing tree highlighted.

Gateway	1st hop	2nd hop	Outcome
Node A	Node C		Download size way too big
Node A	Node C	Node B	Upload never finished
Node A	Node B		Seems to work
Node A	Node B	Node C	Download size way too big

Figure 11.4: Physical id to network id node mapping.

11.3.1 System

The experimental setup is illustrated in figure 11.3. Three motes were involved. One had the static role of gateway and two were available for experimentation.

11.3.2 Diagnostics

After having been away for 2 months I continued the debugging. It wasn't clear which state the code was left in, so figuring that out was the first step. First I set up the whole system and made it run until something was clearly wrong.

Binary search through the logs revealed that `900_reprogram-cmd_2.log` went through just fine but `902_sync-cmd_.log` failed completely. The log in between (`901_reprogram-output_2.log`) holds the STDOUT+STDERR of the reprogram command and shows no abnormalities. Visual inspection of nodes 2 and 3 reveals that they both have their flash, radio and leds turned off, and that a power cycle can't bring them back to a responsive state.

The next day I set up the experiment again and got different results. This included shuffling the motes and reprogramming them. I thus concluded that a hardware fault was a likely candidate and labeled all physical motes alphabetically. I then reran the testbed code with a single target mote and the intension to go through every permutation of the three involved physical motes. The results are listed in figure 11.4.

Targeting the physical mote C appeared to result in huge datasets. Rerunning the test code for the FRAM revealed no faulty operation. This led us back to the main python script, controlling the testbed. It turned out that in the main loop of this script the value stored in the FRAM was cleared after the experiment data had been downloaded. This order requires all target motes to have their FRAM's pre-cleared before the execution of the script. Mote C had had some random value – likely written in a completely different context – stored in the FRAM. During the experiment this value was incremented and after the experiment a dataset of the resulting size was attempted downloaded. Reversing the order of operations fixed the problem.

11.4 Methodology

I see debugging of complex systems mainly as an experimental process. Given what you know about the bug, you come up with hypotheses. You then gain additional knowledge by iteratively evaluating these and posing new hypotheses. At each step you simplify and add trust to your mental model of the system encompassing the bug. At one point, the bug will become obvious.

11.4.1 Design Principles

The following design principles are meant as guidelines for building easily debugable systems. As always, trade-offs are involved making compromises key.

- ***Reduce Arbitration*** Remove potential need for arbitration by design.
- ***Manual First*** When implementing a component first expose manual controls, then – if needed – create automation on top of these.
- ***Expose State*** Expose internal state for reading and writing.
- ***Expose Logic*** Expose operations on internal state for execution.

11.4.2 Approach

Figure 11.5 maps three classes of debugging techniques to the timeline. At the moment when an anomaly is detected knowledge of past state is limited to what had been logged. Current and future state can be inspected, but the past is set. If the anomaly is not repeatable then this is all we are going to know.

When an anomaly is observed the first step is to extract all available state. This is the information available and it is unlikely to be enough to locate a bug. We

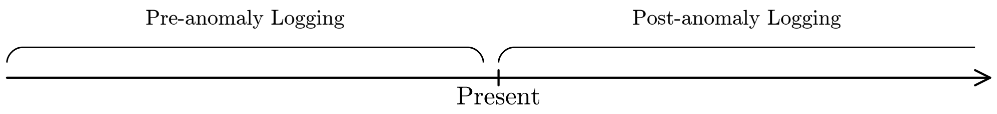


Figure 11.5: Changes in state as a bug manifests as an anomaly.

can only hope that it is enough to reproduce it. If not, then the best option is to add further logging, so that we stand a better chance next time it shows its head.

A helpful method is to employ a cyclic buffer to log events. For this to work the anomaly in question typically needs to be automatically detected so that the logging can be stopped. Otherwise the events leading up to the anomaly will be overwritten. The log allows us to glean into the past. If something is not logged, then it is lost.

When designing for post-anomaly logging the first step is to make a list of access sites; places through which information can be gleaned. On the mote-level these includes GPIOs (including LEDs) and serial ports. On the network-level this includes any PC rooting the network as well as any deployed sniffers.

11.4.2.1 Adding Functionality

Functionality is implemented as state with surrounding logic. Designing functionality is an opportunity to think about the structure of the functionality and consider how that that can be instrumented to allow access.

What state is involved? Having read access will allow us to inspect the state when faced with an anomaly. Having write access will allow us to reproduce states which could potentially lead up to the anomaly.

Which logic is involved? The logic implements transitions between states. Being able to parameterize and execute this logic allows us to artificially construct a state as if a specific pattern of events had happened. This state could then be inspected for verification.

11.4.2.2 When something goes Wrong

First the fault causing the anomaly has to be ***located***. This involves reviewing the logs and the state of the anomaly. GreenLab does extensive logging of all communication to increase the chance of having the needed information. Njullas shallow stack simplifies the task of locating the relevant components.

Further ***exploring*** is needed to reveal what state and which logic is related to the fault. This is likely to involve further instrumentation and repeating variations of the steps which are known to lead to the fault. In Njulla all state relating to a specific type of component is grouped into a context. This makes all state-logic relationships explicit and accordingly the process less error-prone.

What is the most convenient way to ***access*** this state and logic? GPIOs, serial, radio, flash and FRAM are the obvious choices. Njulla has options for enabling logging of flash operations at compile-time. This fills up a buffer with key information regarding the operations and then – after a certain number of operations – dumps everything to serial in a human readable format.

Observe what state and which input on which logic makes the transition to a faulty state. The shallow stack of Njulla makes it manageable to instrument all layers with code for toggling GPIOs. This combined with the use of a logic analyzer is very convenient for tracing execution. The lack of arbitration means that it is easy to insert temporary code for printing out information over a serial line.

What could have caused this? Put forth a ***hypothesis*** of what went wrong. A simple framework – like Njulla – comes with few complexities to account for while formulating a hypothesis.

Verify that the hypothesis is true. This is likely to involve injecting synthetic data through the logic and observe the response of the system. Much verification can be done on the component-level if the relevant components are decomposed in a way that allows instrumentation. In Njulla this was a design principle.

Write and apply a ***fix*** for the bug from the hypothesized cause. Bugs in a simple framework are likely to be simple to reason about. Unless a structural flaw is uncovered they should also have simple fixes.

Verify that the fix actually removed the anomaly. The same means that allow us to detect an anomaly can be used to verify the lack of it.

Chapter 12

Conclusion

The original problem defined by DELTA was to develop a wireless sensor network testbed that would allow Danish SMEs to experiment with long term monitoring solutions based on sub-sleep energy harvesters. This was a very ambitious goal. It brought technical problems forcing me to divert focus. The Njulla framework was a product of such a diversion.

Today Njulla serves as a general purpose mote programming framework with support for sensing, actuation, communication and storage. It has an easily observable code base which lowers the burden of debugging as well as the task of modifying the lower layers when porting it to a new mote platform. Njulla thus makes it easy to reason about tradeoffs in application-specific ways. This is especially beneficial for sub-sleep energy harvesting applications.

I designed Njulla because DELTA chose not to rely on TinyOS for mote programming. Indeed, DELTA favoured portability and TinyOS turned out to be a very complex system to port. While the TinyOS Hardware Abstraction Architecture was meant to ease portability, our experience showed that there remained many hidden interdependencies between hardware and programming framework. The opaque build system and the mismatches between documentation and implementation make it very hard to tackle the anomalies that surface when porting TinyOS to a new platform. In fact, Njulla can be seen as an effort to steer away from the complex component architecture of TinyOS. It puts emphasis on simplicity in the programming framework, leaving applications to deal with complexity as it arises.

In terms of testbed infrastructure, the backchannel-less approach of greenlab works, but increases the pressure on the experiment programmers. Whenever a mote locks up, manual intervention is needed. This actually increases the need for a simple framework. For experimentation on outdoor sub-sleep energy harvesting motes this is a necessary trade.

In the long run Njulla is likely to turn out to be an evolutionary dead end. Another mote programming framework will take the high-impact role. When that happens there are a few lessons to learn from this dissertation:

1. When looking for a solution to a set of application requirements, the design space of both software and hardware should be covered.
2. Designing a system for introspection makes debugging easier. This effectively flattens the learning curve enough to extend the set of potential mote programmers. Perhaps to start including SMEs.
3. Documentation is key for a sustainable software ecosystem. The documentation should be trustworthy, complete and cover all relevant aspects. Keeping such documentation up to date requires a significant amount of work, and for that to be done it must be recognized as a contribution as well as a requirement.
4. Programmers assume that components are isolated. The documentation needs to be vocal about any underlying dependencies. Just as important, the state relating these dependencies needs to be observable to reason about the relationships.

Getting back to the original DELTA problem, part of the goal was to involve SMEs. DELTA did not make significant breakthroughs in that respect. It is still a challenge to involve SMEs in mote programming. There is a steep learning curve and once a programmer has learned enough about mote hardware and programming framework and testbed, she is faced with application problems which are often hard to debug. Njulla attemps to mitigate these debugging problems, but lack the high-level constructs to quickly assemble applications. These are needed for the framework to have any chance of large scale adoption. Today, it has enough functionality to be useful at DELTA, but it is not ready to become a widespread programming framework.

This leads us to the question of who should actually do mote programming. Can we expect SMEs to have people of the right skillset for programming motes? If so, how can we best support them? If not, which tools are needed to aid the communication between domain experts and instrumentation experts? In fact, an interesting area of research is to abandon motes altogether for wireless sensor networks. An idea is to rely on recycling smart phones for this purpose. New smartphones are expensive, but used smartphones are cheap. Soon, there will be many used Android phones ready to be recycled. For monitoring applications where power consumption is no problem, smart phones could easily be deployed.

A question in this respect is what kind of sensing interfaces could be supported (in addition to the many sensors already embedded on the phones)? More interestingly, how far can we go using smart phones for power constrained applications? Can we rely on a stripped down version of Android for this purpose? Can the Java programming framework still be used in such Android configurations? These are interesting open questions. I have started to work on these research questions, together with Geoffrey Challen, during my stay at SUNY Buffallo in the summer 2013.

Bibliography

[1] Wireless Sensor Networks, Second European Workshop, EWSN 2005, Istanbul, Turkey, January 31 - February 2, 2005, Proceedings. IEEE, 2005.

[2] AQ Ansari and DJF Bowling. Measurement of the trans-root electrical potential of plants grown in soil. New Phytologist, 71(1):111–117, 1972.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange. Sensorscope: Out-of-the-box environmental monitoring. In Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on, pages 332–343, 2008.

[4] Gabor Batori, Zoltan Theisz, and Domonkos Asztalos. Robust reconfigurable erlang component system. In Erlang User Conference, Stockholm, Sweden, 2005.

[5] Gordon Bell. Bell's law for the birth and death of computer classes: A theory of the computer's evolution. Technical Report MSRTR2007146, Microsoft Research, 2007.

[6] Jan Beutel. The btnode story - reflections on almost a decade of mote class devices. NCCR MICS Conference, ETH Zurich, 2008.

[7] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping wireless sensor network applications with btnodes. In Holger Karl, Andreas Willig, and Adam Wolisz, editors, EWSN, volume 2920 of Lecture Notes in Computer Science, pages 323–338. Springer, 2004.

[8] Clément Burin Des Rosiers, Guillaume Chelius, Eric Fleury, Antoine Fraboulet, Antoine Gallais, Nathalie Mitton, and Thomas Noël. SensLAB Very Large Scale Open Wireless Sensor Network Testbed. In Proc. 7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCOM), Shanghai, Chine, April 2011.

[9] Marcus Chang. Power efficient duty-cycling with ultra low-power receivers. MSc Thesis, Department of Computer Science, University of Copenhagen, 2006.

[10] Marcus Chang and Philippe Bonnet. Meeting ecologists' requirements with adaptive data acquisition. In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10, pages 141–154, New York, NY, USA, 2010. ACM.

[11] Marcus Chang and Philippe Bonnet. Monitoring in a high-arctic environment: Some lessons from mana. Pervasive Computing, IEEE, 9(4):16–23, 2010.

[12] David Culler. Sustainable energy networks - a sensys grand opportunity. Presented as the keetnote for the 10th ACM Conference on Embedded Networked Sensor Systems, 2012.

[13] Peter J. Denning. Acm president's letter: What is experimental computer science? Commun. ACM, 23(10):543–544, October 1980.

[14] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Proceedings of the 4th international conference on Embedded networked sensor systems, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

[16] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensornet systems. In Proceedings of the 6th ACM conference on Embedded network sensor systems, SenSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[17] Mads Bondo Dydensborg. Connection oriented sensor networks. PhD thesis, Department of Computer Science, University of Copenhagen, 2004.

[18] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network a toolkit for the development of wsns. In Proceedings of the 4th European conference on Wireless sensor networks, EWSN'07, pages 195–211, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Emre Ertin, Anish Arora, Rajiv Ramnath, Mikhail Nesterenko, Vinayak Naik, Ip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, and Hui Cao. Kansei: A testbed for sensing at scale. In in Proceedings of the 4th Symposium on Information Processing in Sensor Networks (IPSN/SPOTS track, pages 399–406. ACM Press, 2006.

[20] Jan Flora and Philippe Bonnet. Tiny15four: A portable, yet efficient 802.15.4 stack. In 2009 IEEE 34th Conference on Local Computer Networks, pages 842–849. IEEE, October 2009.

[21] David Gay, Philip Levis, and David Culler. Software design patterns for tinyos. ACM Trans. Embed. Comput. Syst., 6(4), September 2007.

[22] 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 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[23] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. Emstar: A software environment for developing and deploying wireless sensor networks. In USENIX Annual Technical Conference, General Track, pages 283–296, 2004.

[24] Javier González González. The mana testbed. MSc Thesis, IT University of Copenhagen, 2011.

[25] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, David Culler, and David Gay. Tep 2: Hardware abstraction architecture. 2007-02-22). http://www. tinyos. net/tinyos-2. x/doc/tep2, 2012.

[26] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, and David E. Culler. Flexible hardware abstraction for wireless sensor networks. In EWSN [1], pages 145–157.

[27] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, ASPLOS IX, pages 93–104, New York, NY, USA, 2000. ACM.

[28] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. IEEE Micro, 22(6):12–24, November 2002.

[29] Carlton Himes, Eric Carlson, Ryan J Ricchiuti, Brian P Otis, and Babak A Parviz. Ultralow voltage nanoelectronics powered directly, and solely, from a tree. Nanotechnology, IEEE Transactions on, 9(1):2–5, 2010.

[30] Wei Hong. Arch Rock Private communication in the context of the MANA Project, April 2008.

[31] Jonathan W. Hui and David E. Culler. Ip is dead, long live ip for wireless sensor networks. In Proceedings of the 6th ACM conference on Embedded network sensor systems, SenSys '08, pages 15–28, New York, NY, USA, 2008. ACM.

[32] Ioannis Ieropoulos, Chris Melhuish, John Greenman, and Ian Horsfield. Ecobot-ii: An artificial agent with a natural metabolism. International Journal of Advanced Robotic Systems, 2(4), 2005.

[33] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. In Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on, pages 25–33, 2004.

[34] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual environmentally powered sensor networks. In Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[35] Aslak Johansen, Thomas Sørensen, and Philippe Bonnet. Service and experiment: Towards a perpetual sensor network testbed without backchannel. In The Eighth IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2011), Valencia, Spain, October 2011.

[36] Maria Kazandjieva, Brandon Heller, Philip Levis, and Christos Kozyrakis. Energy dumpster diving. In SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, New York, NY, USA, 2009. ACM.

[37] K Klues, P Levis, D Gay, D Culler, and V Handziski. Tep 108: Resource arbitration. Core Working Group, TinyOS Community, 2007.

[38] JeongGil Ko, Qiang Wang, T. Schmid, W. Hofer, P. Dutta, and A. Terzis. Egs: A Cortex M3-Based Mote Platform. In Proceedings of the 7th Annual IEEE Communications Society Conference on Sensor Mesh and Ad Hoc Communications and Networks, SECON '10, pages 1–3, June 2010.

[39] A. Lal, R. Duggirala, and H. Li. High efficiency radio isotope energy converters using both charge and kinetic energy of emitted particles, November 27 2007. US Patent 7,301,254.

[40] Philip Levis. Experiences from a decade of tinyos development. In Proceedings of the 10th USENIX conference on Operating Systems Design

and Implementation, OSDI'12, pages 207–220, Berkeley, CA, USA, 2012. USENIX Association.

[41] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, ASPLOS X, pages 85–95, New York, NY, USA, 2002. ACM.

[42] Philip Levis, David Gay, and David Culler. Active sensor networks. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI'05, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[43] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In Proceedings of the 1st international conference on Embedded networked sensor systems, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[44] Philip Levis, S Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, and Eric Brewer. TinyOS: an operating system for sensor networks. pages 115–148, 2005.

[45] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1, NSDI'04, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association.

[46] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In Proceedings of the 12th International Conference on Information Processing in Sensor Networks, IPSN '13, pages 153–166, New York, NY, USA, 2013. ACM.

[47] Liqian Luo, Tian He, Gang Zhou, Lin Gu, Tarek F. Abdelzaher, and John A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In In Proc. INFOCOM'06, 2006.

[48] National Semiconductor. Op Amp Circuit Collection: Application Note 31, September 2002.

[49] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. 37th Annual IEEE Conference on Local Computer Networks, 0:641–648, 2006.

[50] Fredrik Österlind, Adam Dunkels, Raimondas Sasnauskas, Oscar Soria Dustmann, and Klaus Wehrle. Integrating symbolic execution with sensornet simulation for efficient bug finding. In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10, pages 383–384, New York, NY, USA, 2010. ACM.

[51] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. SIGCOMM Comput. Commun. Rev., 33:59–64, January 2003.

[52] Kris Pister. Smart dust. Technical Report BAA 97-43, Department of Computer Science and Electrical Engineering, UC Berkeley, 1997.

[53] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on, pages 364–369. IEEE, 2005.

[54] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava. Design considerations for solar energy harvesting wireless embedded systems. In Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[55] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys '05, pages 255–267, New York, NY, USA, 2005. ACM.

[56] Tobias Reusing. Comparison of Operating Systems TinyOS and Contiki. Sensor Nodes–Operation, Network and Application (SN), 7, 2012.

[57] Jerome H. Saltzer and Frans Kaashoek. Principles of Computer System Design: An Introduction. Morgan Kaufmann, 2009.

[58] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN '10, pages 186–196, New York, NY, USA, 2010. ACM.

[59] Philipp Sommer and Branislav Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems, SenSys '13, pages 12:1–12:14, New York, NY, USA, 2013. ACM.

[60] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a sensor network expedition. In Wireless Sensor Networks, pages 307–322. Springer, 2004.

[61] David Tennenhouse. Active networks (abstract). In Proceedings of the second USENIX symposium on Operating systems design and implementation, OSDI '96, pages 89–, New York, NY, USA, 1996. ACM.

[62] Gilman Tolle and David E. Culler. Design of an application-cooperative management system for wireless sensor networks. In EWSN [1], pages 121–132.

[63] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys '05, pages 51–63, New York, NY, USA, 2005. ACM.

[64] Tracy Allen. De-construction of the Shinyei PPD42NS dust sensor. EME Systems LLC, 30 May 2013.

[65] Kashif Virk, J Madsen, Andreas Vad Lorentzen, Martin Leopold, Philippe Bonnet, and M Hansen. Design of a development platform for HW/SW codesign of wireless integrated sensor nodes. pages 254–260, 2005.

[66] D. Vuckovic. Microcontroller-based power management for nanowatt and microwatt energy harvesters. In Sensors, 2013 IEEE, pages 1–4, Nov 2013.

[67] Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on, pages 108–120. IEEE, 2005.

[68] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: a wireless sensor network testbed. In Proceedings of the 4th international symposium on Information processing in sensor networks, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[69] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In Proceedings of the 5th International Conference on Information Processing in Sensor Networks, IPSN '06, pages 416–423, New York, NY, USA, 2006. ACM.

[70] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In Proceedings of the

1st international conference on Embedded networked sensor systems, SenSys '03, pages 14–27, New York, NY, USA, 2003. ACM.

[71] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys '07, pages 189–203, New York, NY, USA, 2007. ACM.

[72] Lohit Yerva, Brad Campbell, Apoorva Bansal, Thomas Schmid, and Prabal Dutta. Grafting energy-harvesting leaves onto the sensornet tree. In Proceedings of the 11th International Conference on Information Processing in Sensor Networks, IPSN '12, pages 197–208, New York, NY, USA, 2012. ACM.

[73] Esben Zeuthen. Re-mote testbed framework. MSc thesis, Department of Computer Science, University of Copenhagen, 2007.

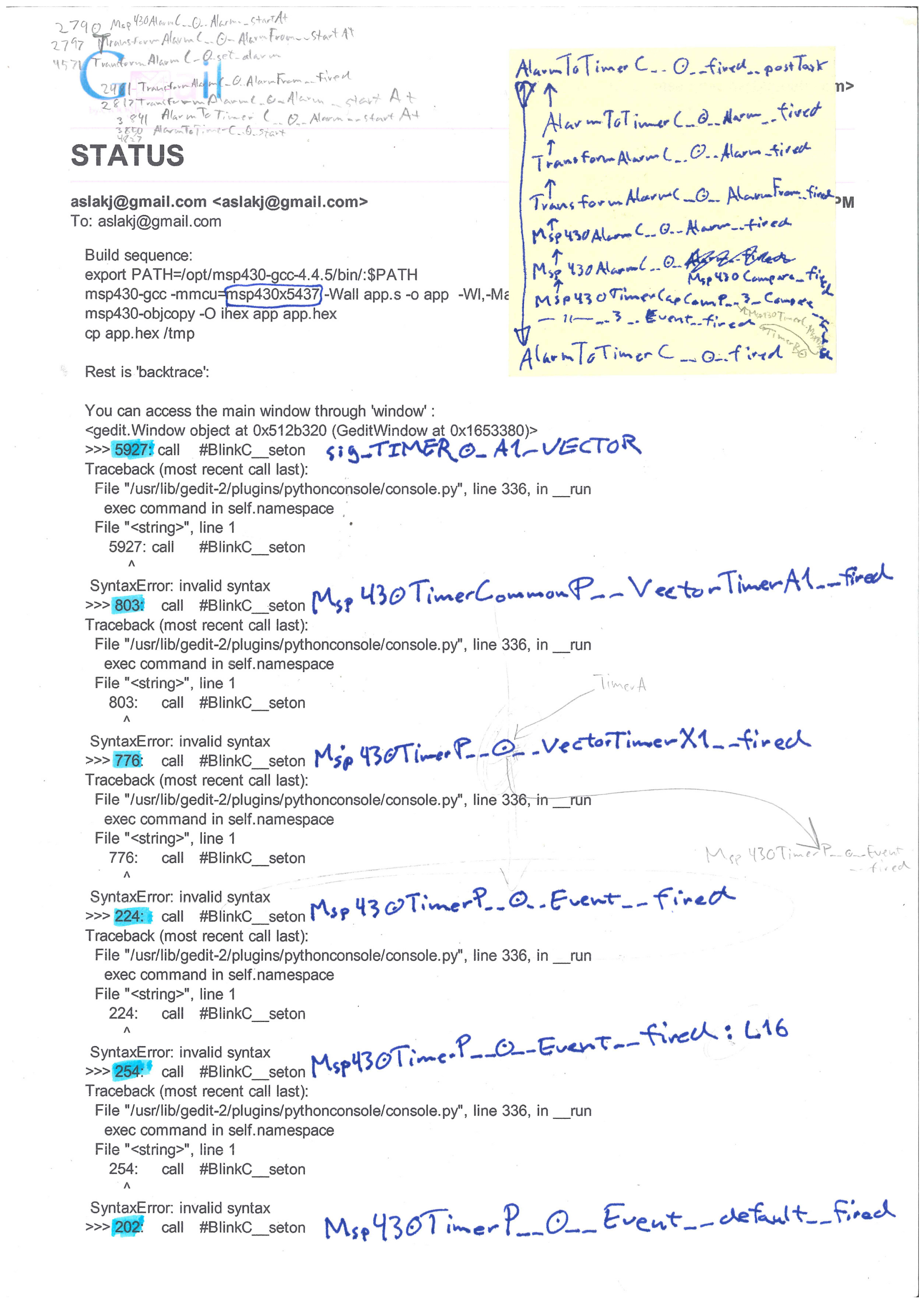
Appendix A

TinyOS Timer Abstraction

The following four pages contains scans of handwritten notes generated during my process of tracing the execution within the component graph of the timer. These notes are an illustration of what Phil Levis refers when he writes that TinyOS makes it harder to solve easy problems. Here, the problem is to find out why a timer is not firing as it should on the micro-controller of the GreenMote4 (i.e, MSP430f5437). This led me to the step which is illustrated below: Which TinyOS events are generated when running on the TelosB microcontroller (MSP430f1611)? Which TinyOS events are so generated on the new microcontroller? Is there any difference? The answer was no. But it took a lot of time and energy to find out. The complexity of the notes below should illustrate this fact.

The notes were created by a combination of the following techniques:

- Manual tracing of includes.
- Instrumentation of code to perform conditional LED operations to figure out what was going on. Repeat for a trace. At this point I did not have a logic analyzer and had to make do with 3 LEDs. Knowing how to use a JTAG debugger would also have helped.
- Looking through the autogenerated `app.c` file to figure out what is calling what. This quickly becomes a hard.
- Compile to assembly (it's easy to parse) and write a script to generate call trees.



2790 Msp430AlarmC__0__Alarm__startAt
2797 TransformAlarmC__0__AlarmFrom__startAt
4571 TransformAlarmC__0__set_alarm
2481 TransformAlarmC__0__AlarmFrom__fired
2817 TransformAlarmC__0__Alarm__startAt
3841 AlarmToTimerC__0__Alarm__startAt
3850 AlarmToTimerC__0__start
4827

STATUS

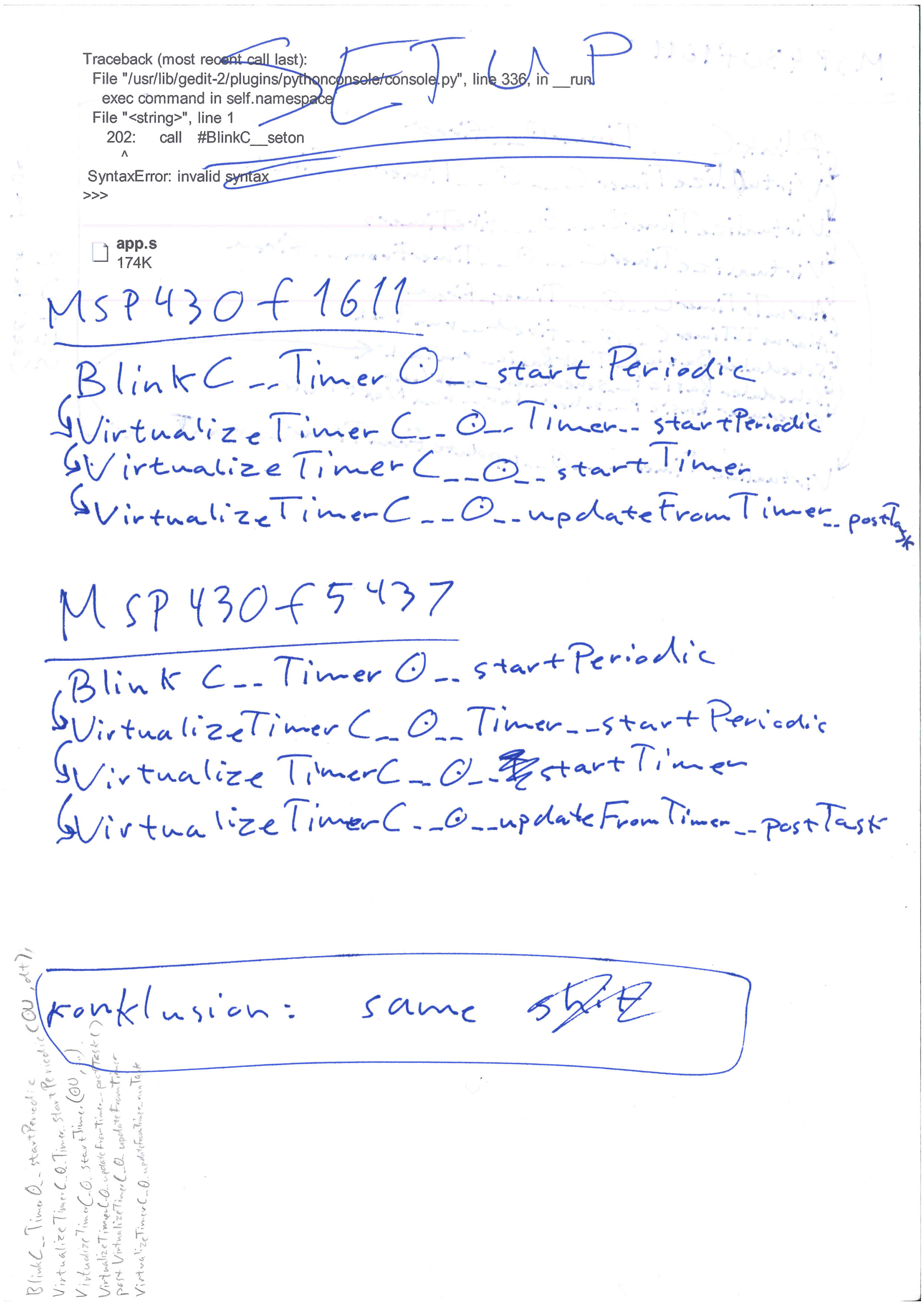
aslakj@gmail.com <aslakj@gmail.com>
To: aslakj@gmail.com

AlarmToTimerC__0__fired__postTask
↑
AlarmToTimerC__0__Alarm__fired
↑
TransformAlarmC__0__Alarm__fired
↑
TransformAlarmC__0__AlarmFrom__fired
↑
Msp430AlarmC__0__Alarm__fired
↑
Msp430AlarmC__0__~~Alarm__fired~~ Msp430Compare__fired
↑
Msp430TimerCapComP__3__Compare
— 〃 — __3__Event__fired
Msp430TimerC__Msp430TimerB0
AlarmToTimerC__0__fired

```
Build sequence:
export PATH=/opt/msp430-gcc-4.4.5/bin/:$PATH
msp430-gcc -mmcu=msp430x5437 -Wall app.s -o app  -Wl,-Ma
msp430-objcopy -O ihex app app.hex
cp app.hex /tmp

Rest is 'backtrace':

You can access the main window through 'window' :
<gedit.Window object at 0x512b320 (GeditWindow at 0x1653380)>
>>> 5927: call    #BlinkC__seton   sig_TIMER0_A1_VECTOR
Traceback (most recent call last):
  File "/usr/lib/gedit-2/plugins/pythonconsole/console.py", line 336, in __run
    exec command in self.namespace
  File "<string>", line 1
     5927: call    #BlinkC__seton
         ^
 SyntaxError: invalid syntax
>>> 803:    call   #BlinkC__seton   Msp430TimerCommonP__VectorTimerA1__fired
Traceback (most recent call last):
  File "/usr/lib/gedit-2/plugins/pythonconsole/console.py", line 336, in __run
    exec command in self.namespace
  File "<string>", line 1
     803:    call   #BlinkC__seton                TimerA
        ^
 SyntaxError: invalid syntax
>>> 776:    call   #BlinkC__seton   Msp430TimerP__0__VectorTimerX1__fired
Traceback (most recent call last):
  File "/usr/lib/gedit-2/plugins/pythonconsole/console.py", line 336, in __run
    exec command in self.namespace
  File "<string>", line 1
     776:    call   #BlinkC__seton                                Msp430TimerP__0__Event
        ^                                                                      __fired
 SyntaxError: invalid syntax
>>> 224:    call   #BlinkC__seton   Msp430TimerP__0__Event__fired
Traceback (most recent call last):
  File "/usr/lib/gedit-2/plugins/pythonconsole/console.py", line 336, in __run
    exec command in self.namespace
  File "<string>", line 1
     224:    call   #BlinkC__seton
        ^
 SyntaxError: invalid syntax
>>> 254:    call   #BlinkC__seton   Msp430TimerP__0__Event__fired: L16
Traceback (most recent call last):
  File "/usr/lib/gedit-2/plugins/pythonconsole/console.py", line 336, in __run
    exec command in self.namespace
  File "<string>", line 1
     254:    call   #BlinkC__seton
        ^
 SyntaxError: invalid syntax
>>> 202:    call   #BlinkC__seton   Msp430TimerP__0__Event__default__fired
```



```
Traceback (most recent call last):
  File "/usr/lib/gedit-2/plugins/pythonconsole/console.py", line 336, in __run
    exec command in self.namespace
  File "<string>", line 1
     202:     call    #BlinkC__seton
        ^
 SyntaxError: invalid syntax
>>>
```

app.s
174K

SETUP

MSP430f1611

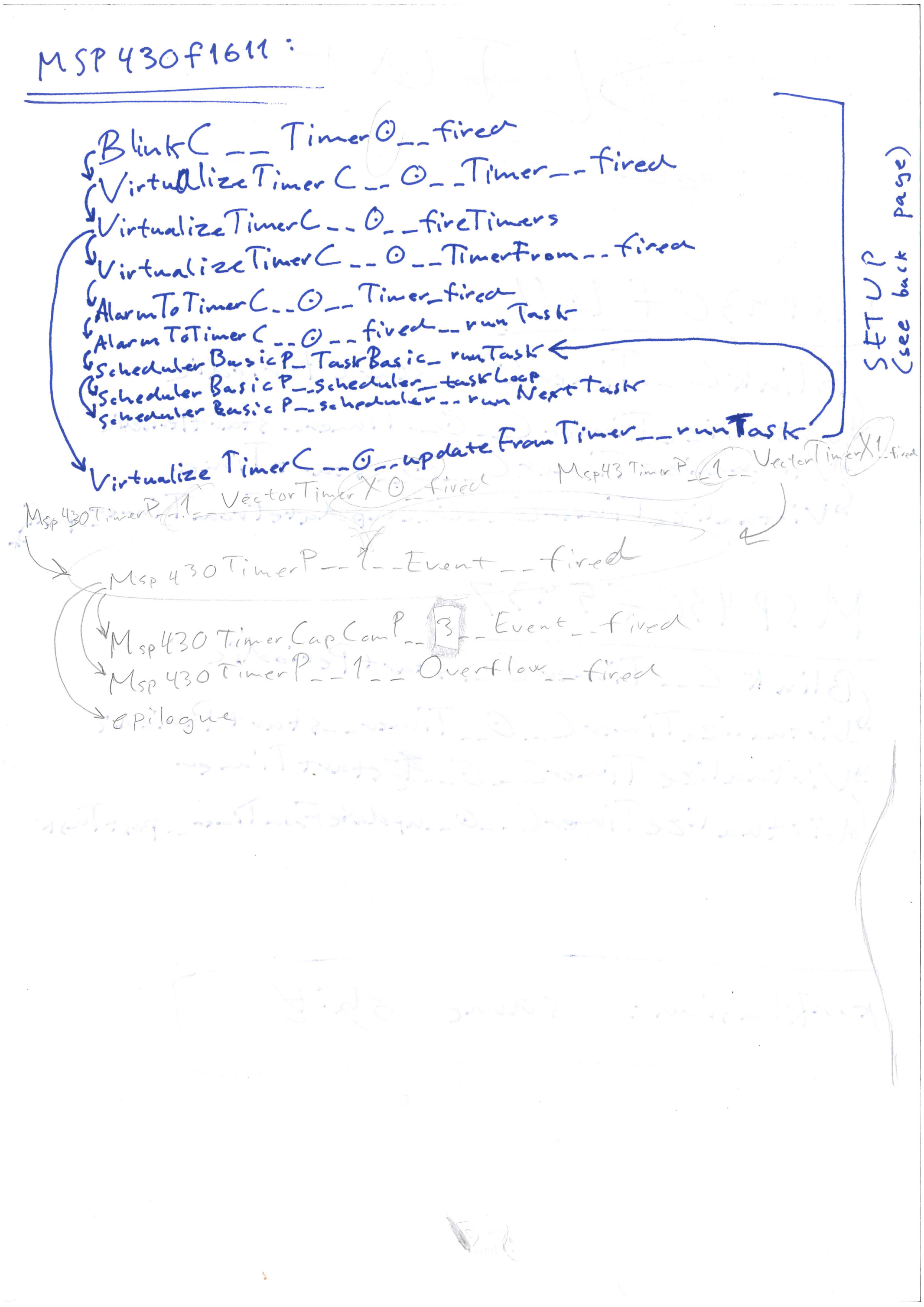
BlinkC__Timer0__startPeriodic
↳VirtualizeTimerC__0__Timer__startPeriodic
↳VirtualizeTimerC__0__startTimer
↳VirtualizeTimerC__0__updateFromTimer__postTask

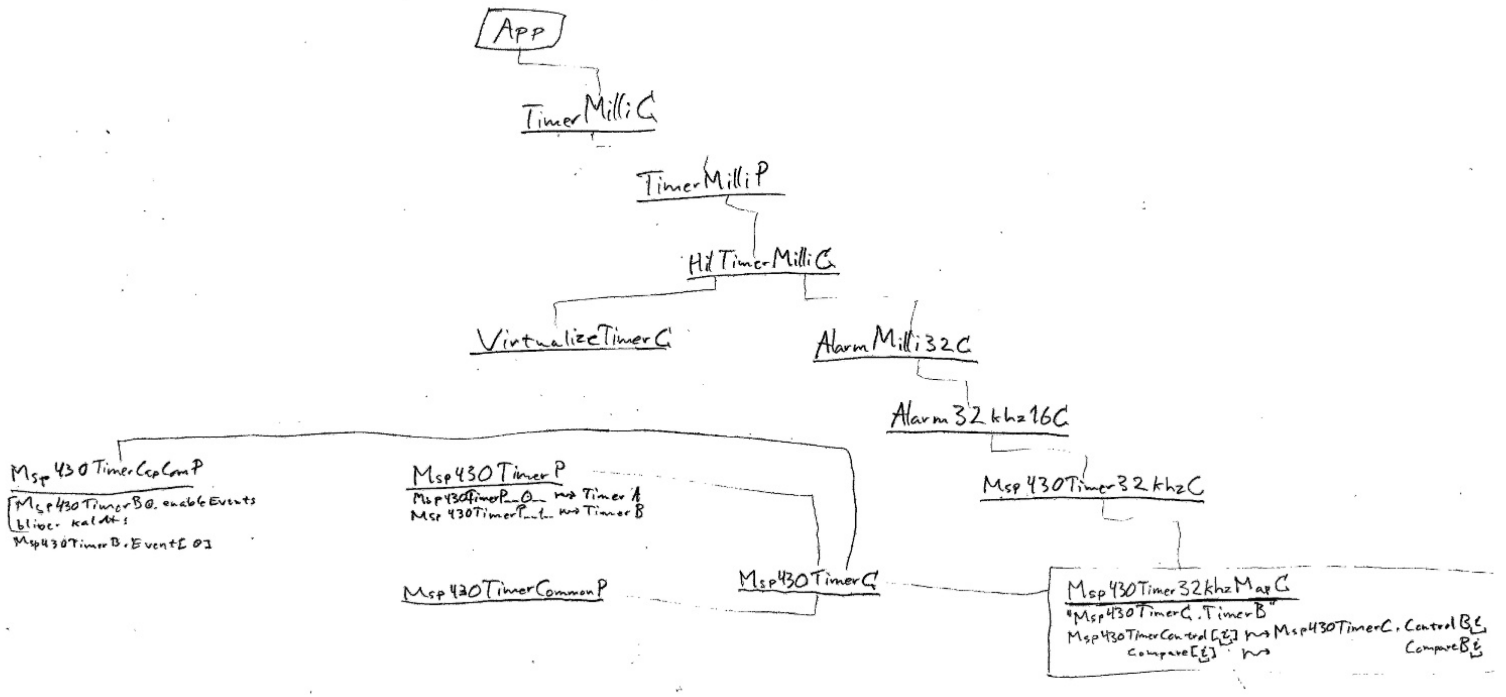
MSP430f5437

BlinkC__Timer0__startPeriodic
↳VirtualizeTimerC__0__Timer__startPeriodic
↳VirtualizeTimerC__0__startTimer
↳VirtualizeTimerC__0__updateFromTimer__postTask

konklusion: same ~~shit~~

BlinkC__Timer0__startPeriodic
VirtualizeTimerC_0_Timer_startPeriodic(0U, dt);
VirtualizeTimerC_0_startTimer(0U, ...);
VirtualizeTimerC_0_updateFromTimer__postTask()
post VirtualizeTimerC_0_updateFromTimer
VirtualizeTimerC_0_updateFromTimer_runTask


MSP430f1611:
BlinkC__Timer0__fired
VirtualizeTimerC__0__Timer__fired
VirtualizeTimerC__0__fireTimers
VirtualizeTimerC__0__TimerFrom__fired
AlarmToTimerC__0__Timer_fired
AlarmToTimerC__0__fired__runTask
SchedulerBasicP_TaskBasic_runTask
SchedulerBasicP__scheduler_taskLoop
SchedulerBasicP__scheduler__runNextTask
VirtualizeTimerC__0__updateFromTimer__runTask
SETUP (see back page)
Msp430TimerP__1__VectorTimerX0__fired
Msp43TimerP__1__VectorTimerX1__fired
Msp430TimerP__1__Event__fired
Msp430TimerCapComP__3__Event__fired
Msp430TimerP__1__Overflow__fired
epilogue


App
TimerMilliC
TimerMilliP
HilTimerMilliC
VirtualizeTimerC
AlarmMilli32C
Alarm32khz16C
Msp430Timer32khzC
Msp430TimerCapComP
Msp430TimerB0.enableEvents
bliver kaldt i
Msp430TimerB.Event[0]
Msp430TimerP
Msp430TimerP_0_ ⇝ Timer A
Msp430TimerP_1_ ⇝ Timer B
Msp430TimerCommonP
Msp430TimerC
Msp430Timer32khzMapC
"Msp430TimerC.TimerB"
Msp430TimerControl[i] ⇝ Msp430TimerC.ControlBi
Compare[i] ⇝ CompareBi

Appendix B

PhonePower Shield

This board was designed in collaboration with Jonathan Fürst, a PhD student at ITU.

B.1 Problem

I noted in section 9.4.4 that TinyOS is a bad fit for students at ITU. As alternatives to TinyOS, we are evaluating Arduinos, the Raspberry Pi and discarded smartphones with regards to building monitoring and automation. One aspect of the evaluation deals with power consumption. The charging profile of a smartphone is complex. To get a deeper understanding of it we have been looking into long time logging of the consumption.

B.2 Approach

We first evaluated existing solutions for measuring DC power consumption. Most used a shunt resistor[1] and seem to fall into three categories:

1. ***USB Oscilloscopes*** These oscilloscopes cover products from PicoScope, Hantek and others. They typically operate in a single-buffer style with a loop of two phases after the initial trigger event, namely (i) fill buffer with samples, and (ii) empty buffer over USB. During phase (ii) no samples are

[1]A shunt resistor is a resistor used to generate a voltage drop proportional to a current for the purpose of measuring that current. This is a common way of measuring current indirectly. A shunt needs to be well defined and typically has a low value to keep the interference minimal.

collected. As a result, the final dataset contains gaps. Some of these oscilloscopes provide a *continuous* mode which essentially bypasses the buffer so that the sampling will be done at a constant frequency. We did, however, not find a simple interface to this mode.

2. ***Cheap Bench Oscilloscopes*** The cheap oscilloscopes have very small buffers (1.5kPoints for the Tektronics TDS2000 series), but (relatively) fast streaming capabilities.

3. ***Expensive Bench Oscilloscopes*** The expensive oscilloscopes have surprisingly small buffers ($5-20$MPoints for the Tektronics MSO4000 series), but (relatively) fast streaming capabilities. They are, however, out of our reach, especially given the intended long term deployment which would incapacitate it for other purposes.

The cheap bench oscilloscope was a real option, but it had two drawbacks: (i) the per-unit price didn't allow for multiple deployments, and (ii) while it could give us the dataset we wanted we didn't have the background for understanding *how* it was obtained. Instead we decided to create a more dedicated solution. Designing this system would help us better understand the process leading to the final datasets. This solution took the shape of an Arduino shield.

B.3 Design

B.3.1 Hardware

The idea behind the design is illustrated in figure B.1. It is essentially an instrumented USB cable where power is supplied from the input side and ground is offset slightly on the output side. This offset is caused by a shunt resistor. The general idea is that a load (e.g., a cellphone) is connected to the output plug while a source is connected to the input plug. The load represents a resistance, which in series with the shunt resistor make up a voltage divider. By measuring the voltage drop across the shunt V_s and applying Ohms Law we can calculate the current I_l being driven by the load. The remaining voltage can be calculated from the total (V_t) as $V_l = V_t - V_s$ and the discrete power is then $P_l = V_l \cdot I_l$. The shunt is replaceable to support a range of different loads.

Naturally we want the voltage as seen by the load to be as close to the input voltage as possible and remain stable as the current draw of the load fluctuates. This wish translates to a small voltage drop over the shunt V_s and thus a small shunt R_s. However, a small voltage is hard to measure. To compensate for this we amplify the signal using an operational amplifier (OpAmp in the figure).

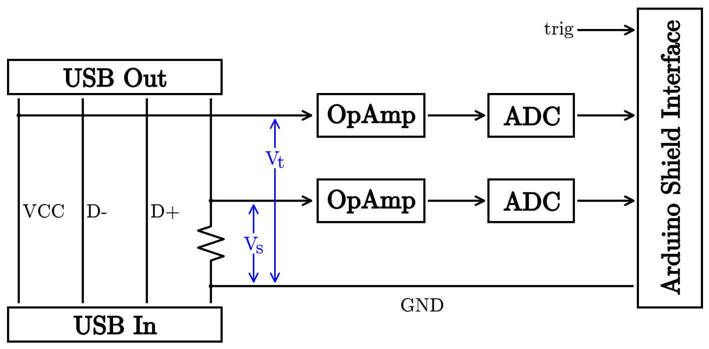


Figure B.1: Highlevel overview of the PhonePower design.

An operational amplifiers is a type of amplifier which had analog computers as its first application. In addition to two supply pins, the type we are interested in has two inputs and one output. The internals of the component amplifies the difference between the two inputs using a high gain factor. This property can be used to implement a feedback mechanism if proper support circuitry is fitted. Pairs of resistors implementing voltage dividers are often used to modify one input. The intension is to make the system reach an equilibrium where the the difference between then two inputs is zero. The amplifier will saturate in the process if the feedback mechanism does not have a strong enough effect. National Semiconductors has as substantial list of operational amplifier based designs[48].

In AC setups certain (non-ohmic) loads shifts the current phase. To get a representative measure of the power, current and voltage needs to be sampled at the same time. In DC setups this is less important as long as the voltage is kept stable. As we had plenty of space on our PCB we decided to implement equitemporal signal paths and thus support AC setups. The two voltages are thus fed through operational amplifiers into two ADCs. We use the MCP3008 from Microchip. It can be sampled in a single SPI command. Accordingly, we can connect the two ADCs as parallel slaves (sharing CLK, MOSI and SS lines) and thus ensure sampling at the same time.

The board has a trigger which can be used to synchronize logging with an external event. This can be used to correlate logs from several instruments.

B.3.2 Software

The main service loop is described as pseudocode in figure B.2. First the code waits for the trigger event. Automatic triggering can be configured in hardware by setting a jumper. Once triggered – and as long as it remains triggered – it will sample the ADCs, encode their values into a string and transmit this string over

```
trigged = False
buffer = ""

while True:
  # wait or trigger
  while (!trigged) trigged = pin_d2_get()

  while (trigged):
    # sample
    pin_d5_low() # select slave
    spi_send_adc_configuration()
    spi_skip_zero_response_part()
    result = []
    for (i in range(10)): result.append(spi_read_port())
    pin_d5_high() # deselect slave

    # encode
    v = encode_from_result_indexed_by_bit(3)
    c = encode_from_result_indexed_by_bit(4)
    sprintf(buffer, "v=%4u c=%4u\n", v, c)

    # transmit
    write_to_serial(buffer)

    trigged = pin_d2_get()
```

Figure B.2: Pseudocode for sampling the PhonePower board.

the serial line. Each of these steps take a static amount of time. By measuring the time between the SS line going high, we can calculate the actual sampling rate. The sampling is done on a port level[2] and stored in 10 variables; one for each bit of the result. During the encoding process we extract the two bits of interest from each variable and apply bitshifting to get the final values.

The Arduino standard libraries for pin and port operations are very slow. To compensate for this, I wrote a script that generates `fastardu.h`. This file contains a list of macros for doing these operations at assembly speed. The header file is constructed in such a way that the macros are defined according to architecture. Currently the Uno is fully supported and the Due is partially supported.

[2]This means that 8 pins are sampled in parallel.

B.4 Calibration

For calibration of the amplification factor for the current channel F_{C} we fit a dataset – matching reference loads to board responses – to the known processing sequence. This depends on the individual reference loads R_{load}, the shunt R_{shunt}), the supply voltage V_{supply}, the reference for the ADC V_{Cref}, the analog resolution $S = 1024$, the analog offset $A_{\mathrm{base}} = 27$ (for the board used) and of course the analog value itself A:

$$\begin{aligned} I_{\mathrm{known}} &= I_{\mathrm{processed}} \Leftrightarrow \\ \frac{V_{\mathrm{supply}} \cdot (R_{\mathrm{load}}/(R_{\mathrm{load}} + R_{\mathrm{shunt}}))}{R_{\mathrm{load}}} &= (A - A_{\mathrm{base}}) \cdot \frac{V_{\mathrm{Cref}}}{S \cdot F_{\mathrm{C}} \cdot R_{\mathrm{shunt}}} \end{aligned}$$

This fit resulted in an amplification factor (F_{C}) of 268.39 ± 0.18. The resulting fits from two boards can be seen in figure B.3

For calibration of the amplification factor for the voltage channel F_{V} we fit a dataset – matching reference loads to board responses – to the known processing sequence. This depends on the individual reference loads R_{load}, the shunt R_{shunt}), the supply voltage V_{supply}, the reference for the ADC V_{Vref}, the analog resolution $S = 1024$, the analog offset A_{base} and of course the analog value itself A:

$$\begin{aligned} V_{\mathrm{known}} &= V_{\mathrm{processed}} \Leftrightarrow \\ V_{\mathrm{supply}} \cdot \frac{R_{\mathrm{load}}}{(R_{\mathrm{load}} + R_{\mathrm{shunt}})} &= (A - A_{\mathrm{base}}) \cdot \frac{V_{\mathrm{Vref}}}{S \cdot F_{\mathrm{V}}} \end{aligned}$$

The fit resulted in an amplification factor (F_{V}) of 0.51 ± 0.0001 and an analog offset (A_{base}) of -1.54 ± 0.17.

The resulting calibrated board has the specifications:

- ***Shunt Resistor*** 0.1Ω
- ***Sampling Frequency*** Current$\times$Voltage pairs at 839.92Hz
- ***Current Range*** $[0 : 181.30]$mA
- ***Current Resolution*** 0.18mA
- ***Voltage Range*** $[0 : 6.65]$V
- ***Voltage Resolution*** 6.50mV

Note that the shunt causes the voltage range to depend on the current. This lowers the upper bound of the range slightly for practical purposes.

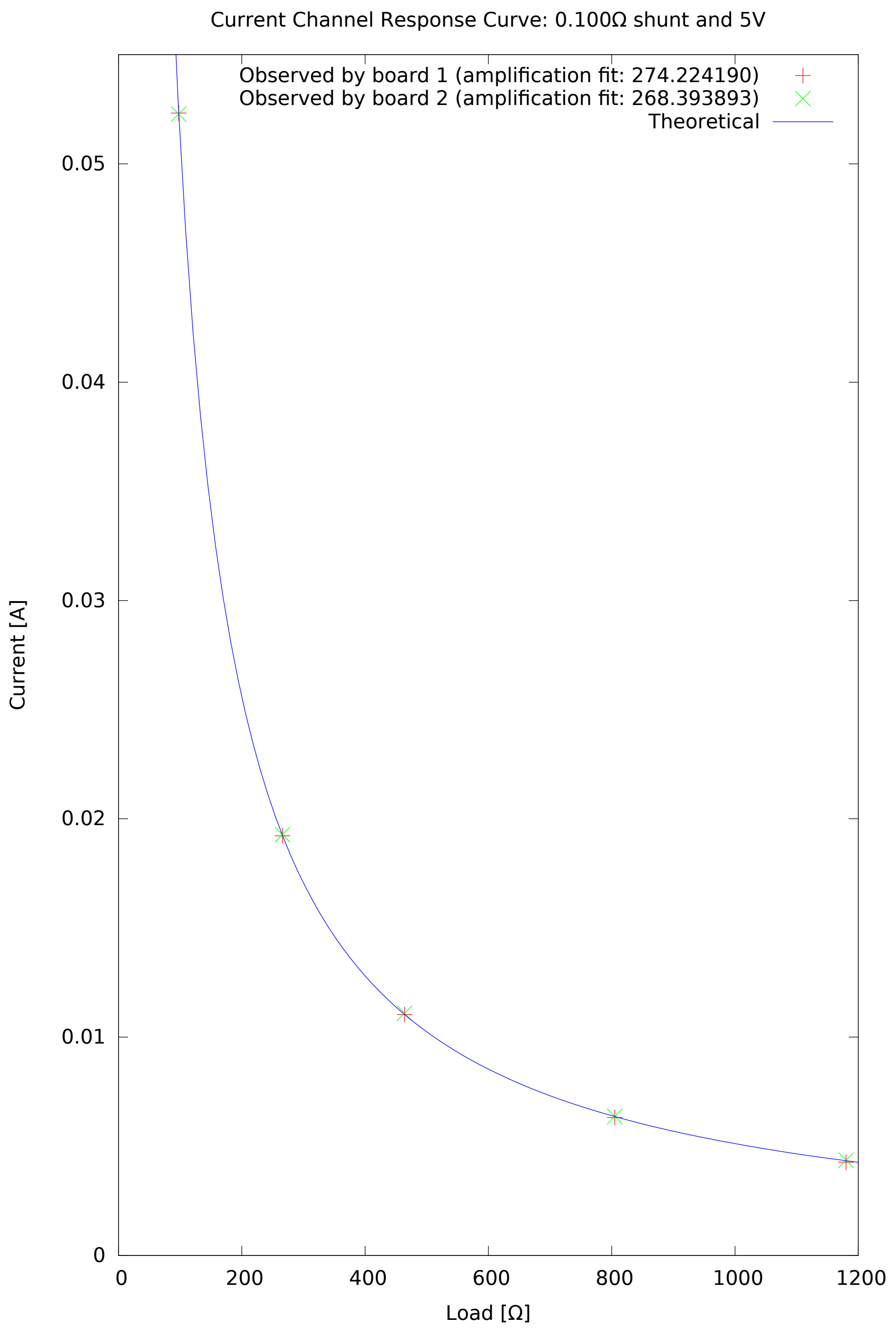


Figure B.3: Calibration result for the current channel of two PhonePower boards.

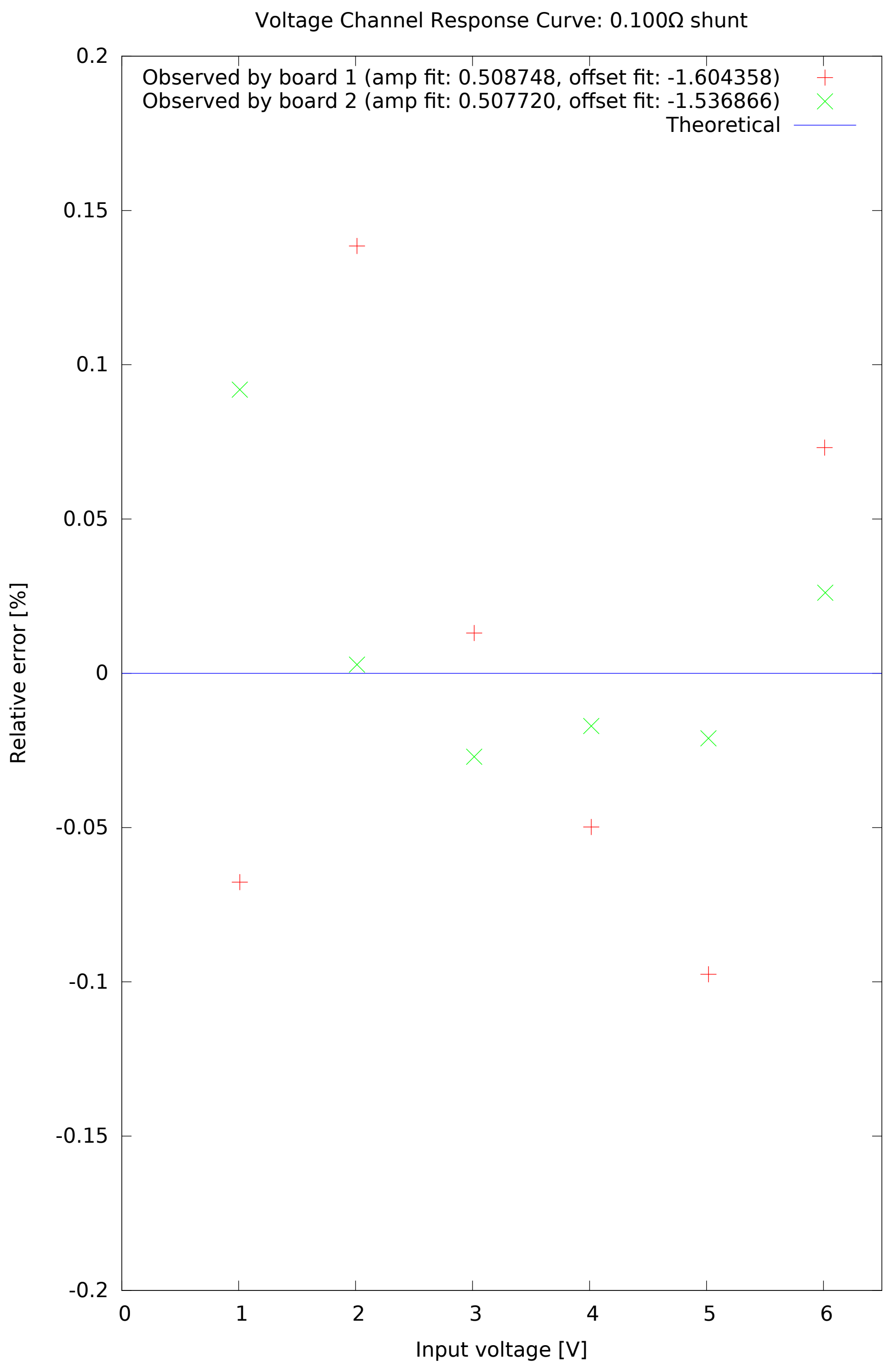


Figure B.4: Calibration result for the voltage channel of two PhonePower boards.