Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Wake-up Time Estimation for a Wireless MAC Protocol

Roman Amstutz

TERM THESIS

Summer term 2007

Supervisors: Andreas Meier
Matthias Wöhrle
Professor: Dr. Lothar Thiele

Start Date: 26th March 2007
Issue Date: 8th August 2007

Abstract

Low power listening combined with wake-up time estimation, as it is done by WiseMAC [3], is a energy-saving concept for medium access control in wireless sensor networks. WiseMAC bases on the transmission of a bit-stream preamble. If the concept of wake-up time estimation wants to be taken for a sensor node that uses a packet-based radio, the original protocol has to be adapted.
This thesis presents a protocol that provides low power listening with wake-up time estimation (*synchronous low power listening*) for packet-based transmission. It bases on the chipcon CC2420 radio stack of TinyOS, which already implements packet-based low power listening but without wake-up time estimation.
The protocol and its implementation is described and several evaluations are discussed. The synchronous low power listening protocol turned out to be able to reduce the number of packets used per transmission and to decrease the interval of idle listening.

Contents

Tables

Figures

Introduction

Wireless Sensor Networks make high demands on saving as much energy as possible. Distributed nodes should be able to be in service self-sustaining for years. Usually, wireless sensor nodes spend most of the energy in communication. Transmitting, receiving and listening costs a lot of energy.
There are various concepts of wireless sensor MAC protocols that save energy by turning on and off the radio (*duty cycling*). *Low power listening* [2] is one of these duty cycling MAC protocol. It turned out to be very adapted for low-energy applications. The concept of low power listening can be improved by introducing wake-up time estimation [3].
Low power listening bases on the transmission of a bit-stream preamble. But radio modules that are packet-based are not able to send a bit-stream. Thus, protocols implementing low power listening had to be developed.
The current radio stack for the chipcon CC2420 radio module of TinyOS [6] implements packet-based low power listening but without wake-up time estimation.

The goal of this thesis is to implement an energy-saving wireless MAC protocol that provides packet-based low power listening with wake-up time estimation (synchronous low power listening). The used hardware platform is the TmoteSky sensor node on which TinyOS runs.

In Chapter 1 to 3 the existing work that builds the basis for our implementation is discussed and analyzed.
Chapter 4 describes the concept and the implementation of the synchronous low power listening protocol.
Chapter 5 presents the evaluation of the synchronous low power listening protocol. The performance is analyzed and the synchronous protocol is compared to the asynchronous protocol.

1

Related Work

1.1 Low Power Listening

Radio modules in sensor nodes are one of the main sources of energy consumption. There are four sources of energy waste [1]: idle listening, overhearing, collisions and protocol overhead. An energy efficient MAC protocol must minimize these sources of energy waste. One concept to reduce energy consumption bases on the idea of minimizing the listening interval.

B-MAC [2] is a CSMA protocol that allows low power communication in sensor networks. It duty cycles the radio through periodic channel sampling that is called *Low Power Listening* (LPL). A sensor node periodically turns on its radio to sample the channel to check whether another node is transmitting. If activity on the channel is detected, the node stays listening and is therefore able to receive an incoming packet. If the receiving is terminated, the node returns to sleep (radio is turned off). If no activity is detected in a predefined interval, the node is forced back to sleep by a timeout.
A node that wants to transmit a packet has to make sure that it occupies the channel at the moment when the destination checks the channel. Therefore, a long preamble is sent in front of the data of a packet. The preamble length is matched to the interval that the channel is checked for activity (*wake-up period*) and has to be at least as long as the wake-up period. An example of transmitting is shown in Figure 1-1.

1.2 WiseMAC

For each packet sent with B-MAC, the preamble is of the same length. This leads to an overhead since a sending node consumes more energy for sending the preamble than for the data. WiseMAC [3] reduces the length of the preamble in certain cases when some information about the receiver is given. WiseMAC bases on low power listening, described in Section 1.1. It assumes all nodes in a network to sample the medium with the same constant wake-up period T_W.

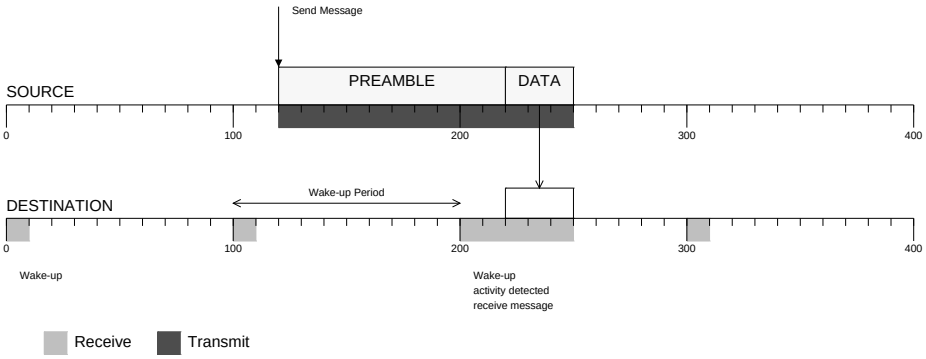


Figure 1-1
B-MAC [2]
The radio of duty cycling node is switched on only for a short interval (the sampling interval) in order to minimize energy consumption. Thus, a transmitter has to send a long preample to hit the sampling interval of the destination.

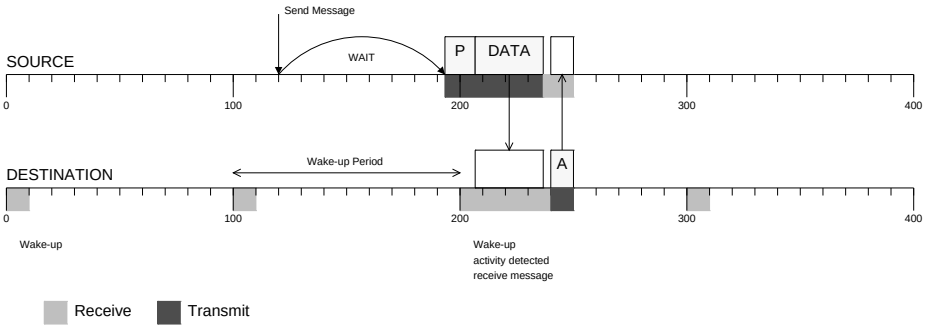


Figure 1-2
WiseMAC [3]
Wake-up time estimation is used to reduce the length of the preamble. The synchronization information is exchanged within the acknowledgement packet.

WiseMAC minimizes the length of the preamble by estimating the next wake-up point of the destination node. For that a node has to learn the sampling schedule of the destination node. WiseMAC ACK packets do not only carry the acknowledgment but also information about the remaining time to the next wake-up. The receiver of an acknowledgement can use this information to calculate the offset to its own wake-up period. Figure 1-2 illustrates how the knowledge about the sampling schedule of the destination node can be used to minimize the length of the preamble.

The duration of the preamble must cover the potential clock-drift between the clock at the source and at the destination. Let L be the time since the last re-synchronization and θ the frequency tolerance of the time-base quartz. Then the necessary duration of the preamble T_P can be calculated as follows:

$$T_P = \min(4L\theta, T_W) \tag{1.1}$$

To center the preamble on the estimated wake-up time, the transmission has to be

Frame type value $b_2b_1b_0$	Description
000	Beacon
001	Data
010	Acknowledgement
011	MAC command
100 - 111	Reserved

Table 1-1: Values of the Frame Type subfield

started at time $L - T_P/2$.
A node that wants to send a packet to a certain destination for the first time has not yet any knowledge about the sampling schedule of the destination. Thus in the first transmission the long preamble (at least T_W) has to be used.

1.3 IEEE 802.15.4

The IEEE standard 802.15.4 [4] is a wireless medium access control (MAC) and physical layer (PHY) specification for low rate wireless personal area networks (LR-WPANs). The Chipcon CC2420 radio module, which is used on the TmoteSky sensor node, provides hardware support for this standard. In this section some specific attributes that are relevant for our implementation of a MAC protocol are depicted.

1.3.1 Frame Format

The frame format defined by the IEEE 802.15.4 standard is shown in Figure 1-3. The first part of a packet consists of preamble and the *Start of Frame Delimiter* (SFD). The SFD allows the receiver of a packet to determine where a frame starts. This first part is called *Synchronization Header*.

The first two bytes of the *MAC Header* are reserved for the *Frame Control Field* (FCF)(Figure 1-4). The FCF contains information about the frame itself. Especially, it defines the type of the frame in a 3 bit Frame Type subfield. The different frame types are listed in table 1-1. The frame types 4, 5, 6 and 7 are reserved for future expansions of the IEEE 802.15.4 standard.

The IEEE 802.15.4 allocates a frame format specifically for acknowledgement packets. Acknowledgement packets are of frame type 2 and always of a length of 5 bytes. They do not carry any address information. Whether a received acknowledgement refers to the last sent packet of a transmitting node can be evaluated by checking the *Data Sequence Number*.

1.3.2 Address Recognition

The CC2420 provides hardware support for the address recognition defined by the IEEE standard 802.15.4. The address recognition bases on requirements listed in Section 7.5.6.2 in [4]. Received frames with frame type are only accepted if the `RESERVED_FRAME_MODE` control bit in `MDMCTRL0` is set.

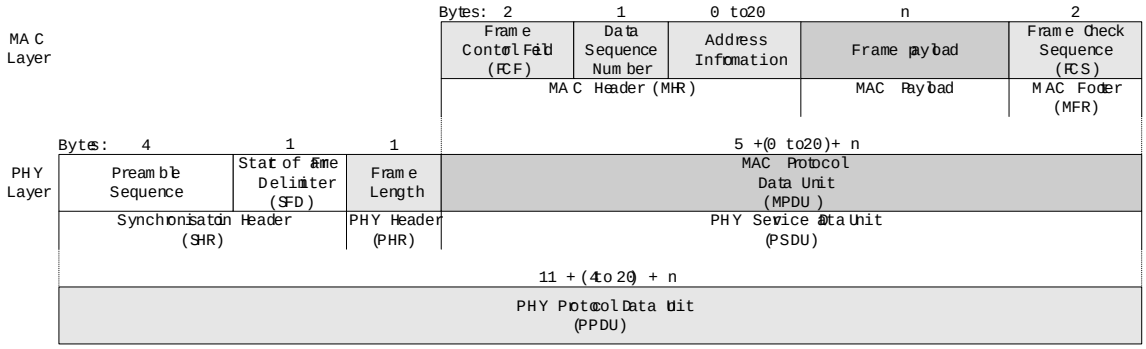


Figure 1-3
IEEE 802.15.4 packet frame

Bits: 0	3	4	5	6	7-9	10-11	12-13	14-15
Frame Type	Security Enabled	Frame Pending	Acknowledge request	Intra PAN	Reserved	Destination addressing mode	Reserved	Source addressing mode

Figure 1-4
IEEE 802.15.4 frame control field

2

The Hardware: TMoteSky

2.1 The Microcontroller: TI MSP 430

2.1.1 The Clock System

The TMoteSky sensor node includes a low-frequency oscillator that is used with a 32768-Hz watch crystal. 32768 Hz correspond to 32 binary kHz. In the further discussion we use $clocks_{32kHz}$ as unit to describe 1 clock of the 32768-Hz watch crystal. 1 $\mathrm{clock}_{32\mathrm{kHz}}$ is the same as $3.05 * 10^{-5}$s.

2.1.2 The Timer System

The *TI MSP 430* of the *TmoteSky* provides two independent counters. The clock which drives the counter can be chosen for each of them separately. In the current implementation of the CC2420 radio stack *TimerA* used an internal digital clock. This clock source allows the usage of high frequencies but offers only limited clock stability. On the other Hand *TimerB* uses an external 32 kHz Quartz with appropriate clock stability. The according clock drives a counter that is increased by one with each clock. The counter can be read out by software. Hence it can be used in order to measure time constraints in the program flow. TimerA is a 16-bit counter with three compare registers while TimerB is also 16-bit, but features seven compare registers. If a value is written into a compare register, the MSP 430 generates an interrupt if the counter matches the value of the compare register. Thus the number of compare registers determines the actual number of timers which can run in parallel.

2.2 The Radio Module: Chipcon CC2420

The Chipcon CC2420 is a packet based radio that implements the IEEE 802.15.4 standard in hardware. It provides an effective data rate of 250 kbps. One 128 byte FIFO buffer (RX) stores the received data and another 128 byte FIFO buffer (TX) is responsible for transmitting data.

2.2.1 Configuration and Interfaces

The CC2420 possesses 33 16-bit configuration and status registers, 15 command strobe registers and two 8-bit registers to access the separated transmit and receive FIFOS. These registers are accessed by a 6-bit address.

The CC2420 can be configured by programming the different configuration registers. A detailed description of the registers can be found in the data sheet of the chipcon CC2420 [5].

To allow the microcontroller to affect the operation of the radio module, the CC2420 supports so called *command strobes*. A command strobe can be viewed as a single byte instruction to the chipcon CC2420 that starts an internal sequence by addressing a command strobe register.
There are specific registers reserved for command strobes. Addressing a strobe register makes the radio module to run an internal sequence. In doing so, no data has to be transferred. The internal sequence is started only by addressing the command strobe register.

The CC2420 has several pins to indicate certain states or events. In the receive mode, the SFD pin goes high if a full start of frame delimiter has been received and goes low again, if the last byte of the message has been received. It also goes low if the hardware-implemented address recognition fails. In the transmit mode, the SFD pin is active during the transmission of a data frame. It goes high if the start of frame delimiter has been transmitted completely and goes low again after the transmission of the last byte.

The CCA pin is used for clear channel assessment. It indicates whether a signal is detected on the channel.

2.2.2 Acknowledgments

The IEEE standard 802.15.4 (Section 1.3) bases on packet based transmission. After each packet received, an acknowledgment packet of a certain frame format is sent back. There are various possibilities how this acknowledgment packets can be generated.

The chipcon CC2420's hardware allows two kinds of acknowledgments. On one hand, the hardware can be set to automatically send acknowledgment packets with no influence of the software running on the TI MSP 430. The acknowledgment packet is sent by hardware as soon as the message is accepted by the address recognition.
On the other hand, the hardware can be set only to send back an acknowledgement when the software initiates the transmission. This is done by using a command strobe. The packet itself is generated automatically by hardware, but it is only transmitted if the command strobe arrives.

2.2.3 The FIFO

128 byte RXFIFO and the 128 byte TXFIFO buffers are part of the internal 368 byte RAM. If a data overflow occurs, the stored data remains in the buffer while new data

arriving is discarded.

3

TinyOS 2.0

TinyOS [6] is an open-source operating system for wireless embedded sensor networks. It has component-based a structure and is programmed in nesC [7]. nesC(network embedded system C) is a component-based C dialect.

3.1 Operating System

3.1.1 Components and Interfaces

A nesC application consists of components. Each component can be seen as a black-box that provides certain functionality. Interfaces represent this functionality to other components. A component A that wants to use the functionality provided by another component B can do this by linking to the corresponding interface of component B.
Interfaces are bidirectional. Each interface can specify a set of commands, which are functions that have to be implemented by the interface's provider. On the other hand, the interface specifies a set of events, which are functions that have to be implemented by an interface's user. A component that uses an interface by calling its command has to implement an event handler for each event of the corresponding interface. Both, commands and events are actually function calls. Commands are called by the user of an interface while events are signaled by an interface's provider.

The interfaces of application components are directly wired to the interfaces of components of the operating system. Thus there is no strong separation between application and operating system. The application and the operating system are integrated in one executable program. At compile time, it is already defined whether a component is included in an application and thus only the components that another component wires to have to be used to build the binary image. Hence the code size of an application can be minimized by only compiling the actually used components.

Components can be divided into two types: *configurations* and *modules*. The difference of the two types lies in their implementation.

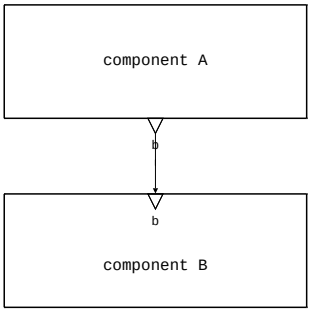


Figure 3-1
Example of two components wired

Modules are components that actually implement the functionality of its provided interfaces. They define functions and allocate state. A module must implement every command of its provided interfaces and every event of its used interfaces.

Configurations do not implement any interface on their own. Indeed, configurations can also provide and use interfaces, but the implementation is done somewhere else. Configurations wire used interfaces of itself or other components with provided interfaces. They must guarantee that each of the used interfaces is wired with an implementation of this interface.

3.1.2 Execution Model

An example of the execution model in TinyOS shall encourage a better understanding. First, we assume a component A, which uses interface b. Interface b is provided by component B (Figure 3-1). Somewhere in the code of A a command $b.1$ from the set of interface b is called. This leads to a function call and the code implementing $b.1$ in component B is executed. The code just described is *synchronous*. It runs in a single execution context and does not have any kind of preemption. This means, if synchronous code is once started, it runs to the end and can not be preempted by other synchronous code. Thus, synchronous code is able to prevent other synchronous code from running and should therefore be hold as short as possible.

To execute large computations, *tasks* can be used. Tasks allow the operating system to defer large computations to a point of time where the CPU is not needed. A task is a function that is not run just when it is called but some time later.
If a task is posted, it is added to an internal task queue which is processed in FIFO order. A task which is once started runs to the end and can not be preempted by another task or synchronous code in general.

3.1.3 Concurrency

As far, we have looked at synchronous code, which is run to the end if once started. The execution is sequential. If a command is called, the execution is proceeded at the point where the corresponding function is implemented. To call back the component above which has called the corresponding command, an event can be signaled. The mentioned program flow is deterministic and known at compile time.

But synchronous code can not be used when it comes to time critical functionality. A hardware interrupt for example can not be handled in a task. If an interrupt occurs, the processor immediately jumps to the handling code. The interrupt handler has to be executed as soon as possible after the interrupt occurs.
Therefore *asynchronous* function calls are introduced. These function calls are assigned with the *async* keyword. An asynchronous function call interrupts synchronous code. This means, an event with the async keyword is a function call that preempts already running synchronous code. Asynchronous code can only call asynchronous functions. In the implementation of an asynchronous command it is allowed to call another asynchronous command or to signal an asynchronous event but calling synchronous functions has to be sourced out to a task. On the other hand, asynchronous functions can be called from within synchronous code.

3.1.4 Generic Components

Normal components in TinyOS are *singleton*. This means that a component exists only once. That is, a component's name is a single entity in a global namespace. If there are more than one component wired to it, their calls are handled always by the same piece of code. A generic component is not singleton. A new instance of it can be allocated within a configuration. For example the timer component *TimerMilliC*, is generic. Hence, applications can instantiate a timer component for each different timer used instead of accessing one single timer component. Using one timer component on an application near layer would complicate the using of timers.

3.2 Hardware Abstraction Layers

The hardware abstraction in TinyOS can be divided into three layers. The lowest and hardware nearest level is the hardware presentation layer *HPL*. The next higher abstraction is called hardware adaption layer *HAL* and the hardware-independent layer is called hardware interface layer *HIL*.

3.3 The Timer-System

Wake-up time estimation in a duty cycle based low power listening protocol requires a timer system which provides timing with a constant period. To determine the duty cycle behavior it is necessary to get a deeper view into the timer system of TinyOS used on a TmoteSky.

3.3.1 Hardware Presentation Layer

The configuration *Msp430TimerC* implements the HPL. Both TimerA and TimerB can each be controlled by a separate interface provided by the configuration. Each of three and seven respectively registers can be set and read by a separate interface. All the Msp430-specific interfaces are implemented with asynchronous commands and events which means that corresponding commands and events are able to preempt synchronous code. Thus a timer-interrupt on the HPL is not delayed by a long-running task.

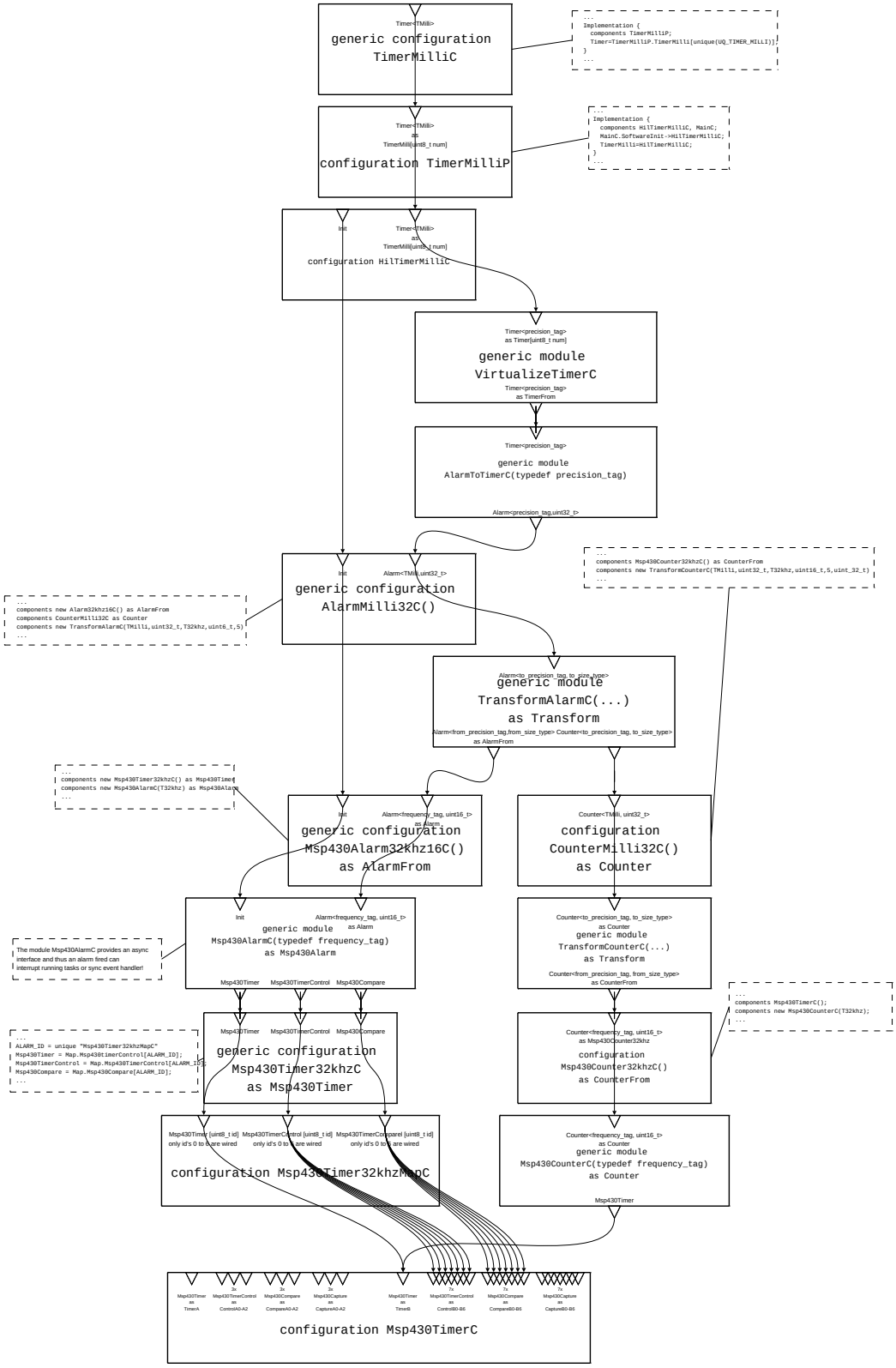


Figure 3-2
Upper level of the wiring of the timer system in TinyOS

The complete wiring of the timer system form the HIL down to the HPL is shown in Figure 3-2.

3.3.2 Hardware Adaption Layer

On the HAL different components use the Msp430TimerC configuration to implement further abstractions. Various components are used to provide timers with different precision and width. In the coming explanations we will focus on the wiring path that finally leads to the component *TimerMilliC*, which is used mostly on the application layer.
The generic configuration *Msp430Timer32khzC* provides an *Alarm* interface that is wired to Component *TimerB*. The Alarm Interface consists of the following commands and events:

```
/**
 * An Alarm is a low-level interface intended for precise
 * timing.
 *
 * <p>An Alarm is parameterised by its "precision"
 * (milliseconds, microseconds, etc), identified by a type.
 * This prevents, e.g., unintentionally mixing components
 * expecting milliseconds with those expecting microseconds
 * as those interfaces have a different type.
 *
 * <p>An Alarm's second parameter is its "width", i.e., the
 * number of bits used to represent time values. Width is
 * indicated by including the appropriate size integer type
 * as an Alarm parameter.
 *
 * <p>See TEP102 for more details.
 *
 * @param precision_tag A type indicating the precision of this Alarm.
 * @param size_type An integer type representing time values for this Alarm.
 *
 * @author Cory Sharp <cssharp@eecs.berkeley.edu>
 */
interface Alarm<precision_tag, size_type> {
    async command void start(size_type dt);
    async command void stop();
    async command bool isRunning();
    async command void startAt(size_type t0, size_type dt);
    async command size_type getNow();
    async command size_type getAlarm();

    async event void fired();
}
```

Listing 3.1
Code: Alarm Interface

The access to Msp430TimerC is effected via the configuration *MspTimer32khzMapC*. By providing a parameterized interface *MspTimer[uint8_t]*, MspTimer32khzMapC allows to use the seven compare registers of TimerB. Each new instance of the generic configuration Msp430Timer32khzC uses another MspTimer interface with a unique identifier. Since TimerB has only seven registers, not more than seven new Msp430Timer32khzC components can be generated.
By hardware, TimerB provides a counter with a precision of 32 kHz and a width of 16 bit. On the HAL there are components that transform this precision and width. Thus also a timer with the precision of milliseconds (Milli) can be used.

Above the just described mapping and transforming, the generic module *AlarmToTimerC* is used to change the asynchronous Alarm interface into a synchronous Timer interface. The Timer interface consists of the following commands and events:

```
/**
```

```
 * A Timer is TinyOS's general purpose timing interface. For
 * more precise timing, you may wish to use a
 * (platform-specific) component offering an Alarm interface.
 *
 * <p>A Timer is parameterised by its "precision" (milliseconds,
 * microseconds, etc), identified by a type. This prevents,
 * e.g., unintentionally mixing components expecting
 * milliseconds with those expecting microseconds as those
 * interfaces have a different type.
 *
 * <p>See TEP102 for more details.
 *
 * @param precision_tag A type indicating the precision of this Alarm.
 *
 * @author Cory Sharp <cssharp@eecs.berkeley.edu>
 */

interface Timer<precision_tag> {
    command void startPeriodic(uint32_t dt);
    command void startOneShot(uint32_t dt);
    command void stop();
    event void fired();

    command bool isRunning();
    command bool isOneShot();
    command void startPeriodicAt(uint32_t t0, uint32_t dt);
    command void startOneShotAt(uint32_t t0, uint32_t dt);
    command uint32_t getNow();
    command uint32_t gett0();
    command uint32_t getdt();
}
```

Listing 3.2
Code: Timer Interface

The usage of the Timer interface instead of Alarm does not allow a timer-interrupt to preempt tasks and event-handlers anymore. This means, an event generated by a timeout that has been instantiated via a Timer interface will be asynchronous up to the level of *AlarmToTimerC* and will then proceed as synchronous event. This prevents an application using the timer interface from delaying other asynchronous events by running oversized event-handlers. As mentioned above, TimerB provides only seven compare registers. Virtualizing timers allows having more than seven timers running at the same time. The virtualization is done by the generic module *VirtualizeTimerC* which uses the *Timer* interface provided by *AlarmToTimerC*. One generic module VirtualizeTimerC uses only one compare register but provides up to 256 parallel running timers via the parameterized interface *Timer[uint8_t num]* .

3.3.3 Hardware Interface Layer

On the HIL the configuration *HilTimerMilliC* provides access to the lower layers via the interface Timer[uint8_t num]. This interface is wired to the VirtualizeTimerC component and offers up to 256 timers.

On the application layer one normally uses the component *TimerMilliC* for timing. This component is wired to HilTimerMilliC.

3.3.4 Measurements

The Timer interface consists of synchronous commands and events. Thus, for example the command to start a timer or the event generated when a timer is fired is not preemptive. Hence, it could happen that the handling of an event that signals the firing of a timer is delayed by a task that has to run to its end yet. As described above, on lower layers of the timer stack another interface for timing is used. The interface Alarm completely consists of asynchronous commands and events.

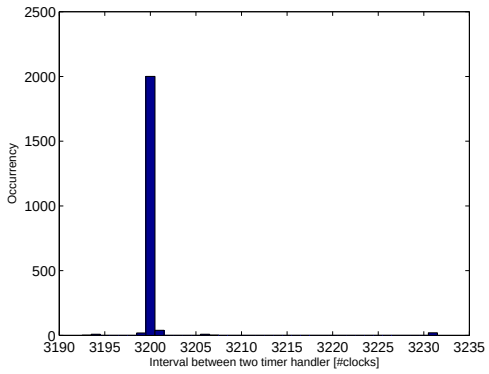


Figure 3-3
Distribution of the interval between the handling of two following timer firings

If wake-up time estimation is implemented, we want to be able to send a packet at a certain point of time to hit the listening interval of the destination node. Thus we have to look at the differences between timing with the Alarm interface and the Timer interface.

3.3.4.1 Evaluation of the Timer Interface

In a first evaluation, a component that only uses one timer provided by the TimerMilliC component has been considered. Always when the timer is fired, it is started anew with a timeout period of 100 binary milliseconds. 1024 binary milliseconds are equal to one second. The timer is started with a synchronous command call from within the event handler that handles a timer firing. Thus, the timer should fire periodically all 3200 clocks of the 32 kHz quartz. Besides the handling of the timer event, the evaluated component has no functionality. This is to prevent from tasks that could run at the same time when a timer event is signaled. 2100 timer firings have been evaluated. In 0.24 % of the considered samples, the timer event has not been handled accurately. The handling has been done one clock too late. There have been no discrepancies larger than one clock. This leads to 0.48 % of the periods that are either one clock longer or one clock shorter than 3200.

Another test component has been executed where three timers were used at the same time and several tasks were posted. Again 2100 timer firings have been evaluated. In 4.67% of the samples, the period between two handlings of a timer firing differed from 3200 clocks. The distribution of the duration of the evaluated periods is show in Figure 3-3. As expected, other timers and tasks running during a synchronous timer event occurs lead to variable intervals between the handling of two following timer firings.

3.3.4.2 Evaluation of the Alarm Interface

To evaluate the Alarm interface, the test components used were quite similar to the components described in Section 3.3.4.1. The only difference was that instead of the Timer interface the Alarm interface has been used. In the component to evaluate a scenario with timers and tasks running, two Timer interfaces and one Alarm

Figure 3-4
Form of transmission of a single packet

interface has been used. 2100 sample periods were measured. Neither in the scenario with only one Alarm nor where tasks and timers were implemented, a period differing from 3200 clocks could have been detected.

3.3.4.3 Summary

To make sure that the handling of a timer event is not delayed, an Alarm interface should be used. In spite of tasks running, the Alarm interface allows handling a timer event without delay and thus leads to a stable period.

3.4 The CC2420 Low Power Listening Radio Stack

The radio stack implements a packet-based asynchronous low power listening [9]. Instead of a long preamble several packets are sent to hit the listening interval of the duty cycling destination node. The transmitter sends the full packet over and over again for twice the receiver's duty cycle period.
The transmission of on single packet is typically of the form shown in Figure 3-4. Before the message is sent, the radio checks the channel for the duration of the *LPL Backoff* period $T_{LPLBackoff}$ to ensure that no other node is transmitting at the same time. After the transmission of the message the transmitter waits for the acknowledgement packet. For this a period $T_{AckWait}$, during which the transmitter waits for an acknowledgement, is defined. If no acknowledgement is received within $T_{AckWait}$ the message can be assumed as not acknowledged and has to be resent.
Thus the channel is only modulated during the actual transmission of the message. During the period $T_{Calm} = T_{LPLBackoff} + T_{AckWait}$ the channel is calm. If several packets are sent in order to execute acknowledged low power listening a period of duration T_{Calm} exists between two packets during which the channel is not modulated. In order to overlap the transmission period of a packet the duty cycling receiver has the sample the channel a moment longer than T_{Calm}.

Figure 3-5 and Figure 3-6 show the wiring of the whole radio stack from the application layer represented by the component *ActiveMessageC* down to the Hardware Presentation Layer. The stack can be divided into different layers, each providing a set of the three interfaces, *Send*, *Receive* and *SplitControl*. A component providing a certain interface to the layer above uses the corresponding interface of the layer below. A message sent via the Send interface provided by the component ActiveMessageC passes through all layers down to the hardware.

3.4.1 Hardware Abstraction of the Send and Receive Interface

As mentioned above, the stack consists of several layers. In the following sections we will have a deeper look at how the two interfaces *Send* and *Receive* are wired

through the radio stack.

3.4.1.1 Send Interface

The Send interface consists of the following commands and events:

```
interface Send {
    command error_t send(message_t* msg, uint8_t len);
    command error_t cancel(message_t* msg);
    event void sendDone(message_t* msg, error_t error);

    command uint8_t maxPayloadLength();
    command void* getPayload(message_t* msg);
}
```

Listing 3.3
Code: Send Interface

An application can use the Send interface of the ActiveMessageC to send a message of the type `message_t` [8]. `message_t` constitutes the standard message buffer in TinyOS2.x. The structure of the type `message_t` is defined in *tos/types/message.h*. Roughly, the `message_t` consists of a header, the data, a footer and additional metadata. The content of the metadata is not sent but used to exchange information between the different layers of the radio stack.
The Send interface is wired via the configuration *CC2420ActiveMessageC* to the module *CC2420ActiveMessageP*, where the CC2420 specific header is composed. The message is passed to the *UniqueSend* layer where a unique sequence number is added to the header of the message.
Then the message is handed on the configuration *CC2420AckLplC*, which is linked to the module *CC2420AckLplP* where the low power listening is implemented. The *AckLpl* layer is responsible for adapting the transmission of a message to the duty cycling of the receiver. Since the receiver is only listening during a short period of time, a message is transmitted several times to guarantee the listening period of the receiver is hit. In order to increase the possibility that a message is received at the destination, the AckLpl layer sends the same message during a period as long as two duty cycle periods. When a send done event is signaled from the layer below, it is checked whether the message has been acknowledged. If not, the message is resent by directly accessing the interface of the component *TransmitC*. If a message has to be sent for the first time, it is handed over from the AckLpl layer to the component *CsmaC*, which is linked to the module *CsmaP*. The module *CsmaP* defines the FCF byte of the IEEE 802.15.4 header. It also determines the random backoff periods.
Finally the message is handled on the *Transmit* layer. This layer directly accesses the SPI bus to interact with the CC2420. First, the *TransmitP* module loads the message into the TXFIFO buffer. If this is done correctly the transmission of the message is released by the command strobe `STXON`. A more detailed description of the component TransmitP is given in Section 3.4.4.

3.4.1.2 Receive Interface

The Receive interface consists of the following commands and events:

```
interface Receive {
    event message_t* receive(message_t* msg, void* payload, uint8_t len);

    command void* getPayload(message_t* msg, uint8_t* len);
    command uint8_t payloadLength(message_t* msg);
}
```

Listing 3.4
Code: Receive Interface

If a message comes in, it is first loaded into the RXFIFO buffer. If a predefined number of bytes is loaded into the buffer, the `FIFOP` pin of the CC2420 is set to high. This change of the `FIFOP` pin is captured and leads to an event signaled to the *ReceiveP* module. In its event handler the receiving is started. First, the content of the RXFIFO buffer is read out into the main memory via the SPI bus. Rather, at the first step, only the length byte is read out. The length field is checked and if it is within the predefined frame, the read is continued. If the message read out is of the data type, a hardware generated acknowledgement packet is sent by calling the `SACK` command strobe. At the end, a receive-done event is signaled to the component above.
The *CC2420CsmaC* component's Receive interface is directly wired to the Receive interface of the component *CC2420TinyosNetworkC* whose Receive interface is implemented in the module *CC2420TinyosNetworkP*. There, the network field of the header is checked. If the network field indicates a `TINYOS_6LOWPAN_NETWORK_ID` identifier, a receive event is signaled to the component *UniqueReceiveC*.
The Receive interface of the UniqueReceiveC configuration is implemented in the module *UniqueReceiveP*, where the sequence number of the packet is checked. On the next higher level, the module CC2420AckLplP implements the Receive interface where a timer is started. Since the duty cycle is suspended in case of receiving a packet, the radio has to be turned off by the AckLpl layer. This is done when the mentioned timer runs out. At the AckLpl layer a receive event is signaled which is handled by the CC2420ActiveMessageC.

3.4.2 Duty Cycling

The component *CC2420DutyCycleC* provides the interface *State*, *CC2420DutyCycle* and *SplitControl*. The two interfaces CC2420DutyCycle and SplitControl are implemented in the module *CC2420DutyCycleP*.
If duty cycling is enabled, the DutyCycleP module periodically turns on the radio to sample the medium in order to check whether another node in the neighborhood is transmitting. The command `isChannelClear()`, which is part of the interface *CC2420Cca*, is called `MAX_LPL_CCA_CHECKS` times to scan the channel. If another node transmitting is detected, the CC2420DutyCyle event `detected()` is signaled. After signaling this event, the CC2420DutyCycleP component stays idle. Duty cycling is only resumed, if a radio stop done event form CC2420CsmaC is signaled. Otherwise, if no other node transmitting is detected, the radio is stopped. The event handler of the radio stop done event starts the timer again and initiates the next sleeping phase. Another firing of the timer leads to starting the radio and wake up procedure begins anew. Depending whether a node transmitting is detected or not, the period between two wake-ups can strongly vary. The period also varies because the clear channel assessment is done in a task and thus can be interrupted by asynchronous commands and events.

3.4.3 Acknowledged Low Power Listening

The configuration *CC2420AckLplC* provides the needed interfaces for acknowledged low power listening. Besides the interfaces Send and Receive, the component also provides the interfaces *LowPowerListening*, SplitControl and State. The SplitControl interface is wired to the DutyCycleC component. Thus, by calling the SplitControl interface of the CC2420AckLplC component, duty cycling is started. The low power listening itself is started directly by the `init()` command of the *MainC* component.
The LowPowerListening interface is used to adjust the duty cycle period. It provides also commands that return the wanted period, for example the current sleep interval. Like the interfaces Send and Receive, the LowPowerListening interface is implemented in the module *CC2420AckLplP*.

If a packet wants to be sent for the first time this is done by using the Send interface of the CC2420TinyosNetworkC. The program flow on the CC2420AckLplP layer continues when the transmitting is completed, this means when a `sendDone()` event is signaled from the component below. If the packet just sent has been acknowledged, the procedure of sending is terminated and a `sendDone()` event can be signaled to the component above. Otherwise, the packet has to be resent. To resend the packet it is not necessary to go again the path down to the hardware over the CC2420TinyosNetworkC component. Since the message is still stored in the TXFIFO buffer, the TransmitP component can be accessed directly. For the purpose of resending a packet already existing in the TXFIFO buffer, the CC2420Transmit interface offers the command `resend()`.
Since the receiving node is only listening for a short listening interval, a packet has to be resent repeatedly at least until the next wake-up of the destination node. Thus, before abort transmitting, the packet has to be sent over a period of at least one duty cycle to guarantee that the listening interval of the destination node can be hit. To increase the possibility to hit the listening interval, the period over which a packet is resent is set to twice the duty cycle period.

3.4.4 The Component TransmitP

3.4.4.1 General

The component TransmitP implements the hardware nearest layer of the CC2420 radio stack. Among other interfaces, it mainly implements the *CC2420Transmit* interface, which consists of the following commands and interfaces:

```
/**
 * Low-level abstraction for the transmit path implementaiton of
 * the ChipCon CC2420 radio.
 *
 * @author Jonathan Hui <jhui@archrock.com>
 * @version $Revision: 1.5 $ $Date: 2007/04/12 17:11:12 $
 */

interface CC2420Transmit {

  /**
   * Send a message
   *
   * @param p_msg message to send.
   * @param useCca TRUE if this Tx should use clear channel assessments
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t send( message_t* p_msg, bool useCca );
```

```
  /**
   * Send the previous message again
   * @param useCca TRUE if this re-Tx should use clear channel assessments
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t resend(bool useCca);

  /**
   * Cancel sending of the message.
   *
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t cancel();

  /**
   * Signal that a message has been sent
   *
   * @param p_msg message to send.
   * @param error notifaction of how the operation went.
   */
  async event void sendDone( message_t* p_msg, error_t error );

  /**
   * Modify the contents of a packet. This command can only be
   * used when an SFD capture event for the sending packet is
   * signalled.
   *
   * @param offset in the message to start modifying.
   * @param buf to data to write
   * @param len of bytes to write
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t modify( uint8_t offset, uint8_t* buf, uint8_t len );
}
```

Listing 3.5
Code: CC2420Transmit Interface

TransmitP directly accesses the pins and registers of the Chipcon CC2420 through the interfaces *GpioCapture*, *GeneralIO*, *CC2420Fifo*, *CC2420Register* and *CC2420Strobe*. The execution of the `send()` command can roughly be divided into two parts. First, the message to be sent is loaded into the TXFIFO buffer of the CC2420. For doing this, the SPI bus has to be acquired. If the loading is completed, an event is generated, which leads to phase two. If clear channel assessment is enabled, at this point the whole procedure is run through. If clear channel assessment is disabled or already successfully terminated, next TransmitP attempts to send the message, which also requires the SPI bus to be acquired. The transmission of the message is released by a command strobe.

3.4.4.2 Clear channel assessment

Since clear channel assessment takes some time and thus affects accurate sending of a message, we will have a closer look to the procedures implemented in the TransmitP component. Before starting to send a message that has already been loaded into the TXFIFO buffer, an initial backoff timer is set to a random period of time. When the backoff timer is fired, the channel is checked whether it is clear. If the channel is clear, the sending is delayed for another period of about 0.2 milliseconds. This is done to prevent from conflict with a probable acknowledgment. After this period if the channel is still clear, the message is sent. If the channel is used, the component sets the congestion backoff timer.

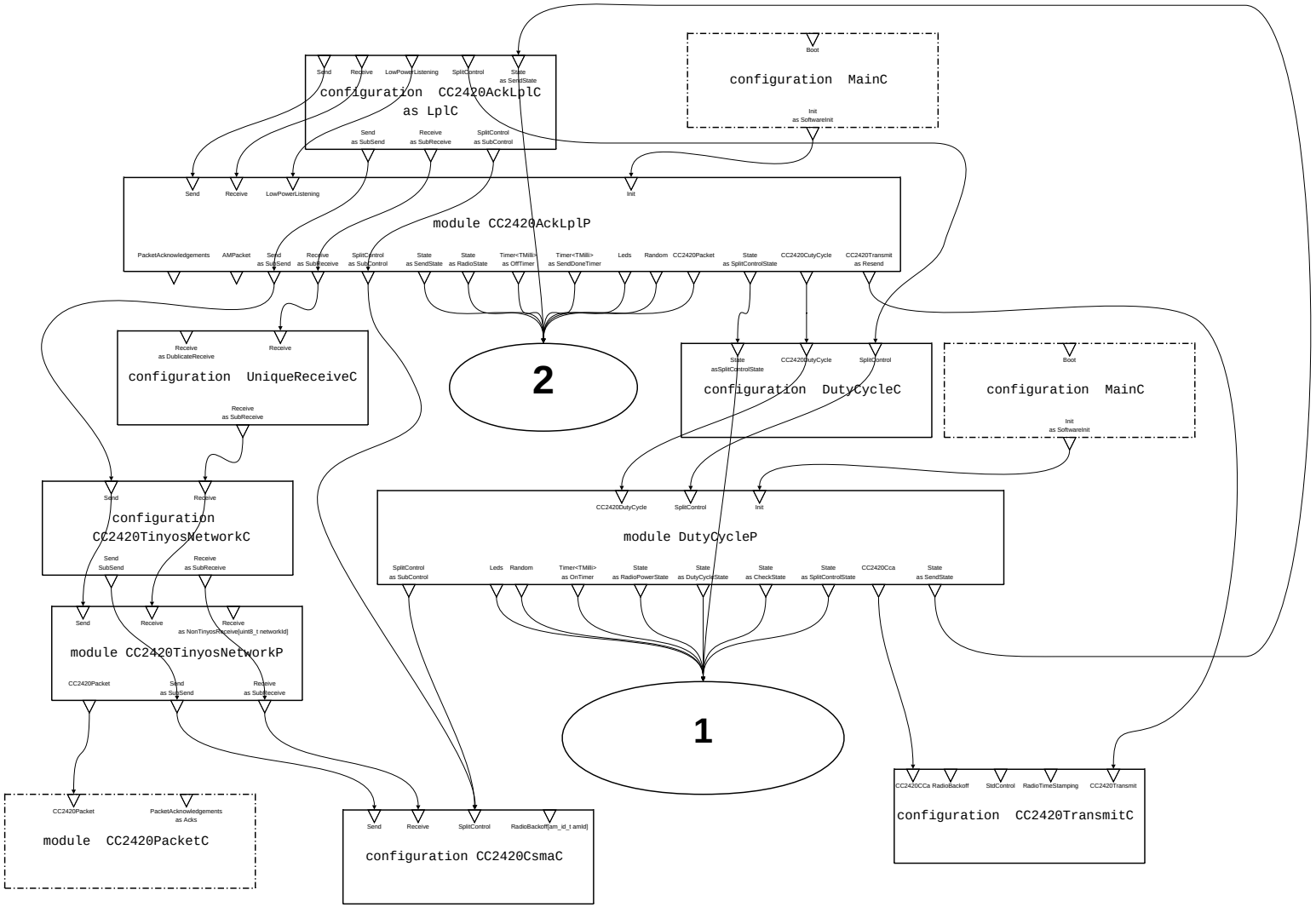


Figure 3-5
Wiring of the upper part of the CC2420 radio stack

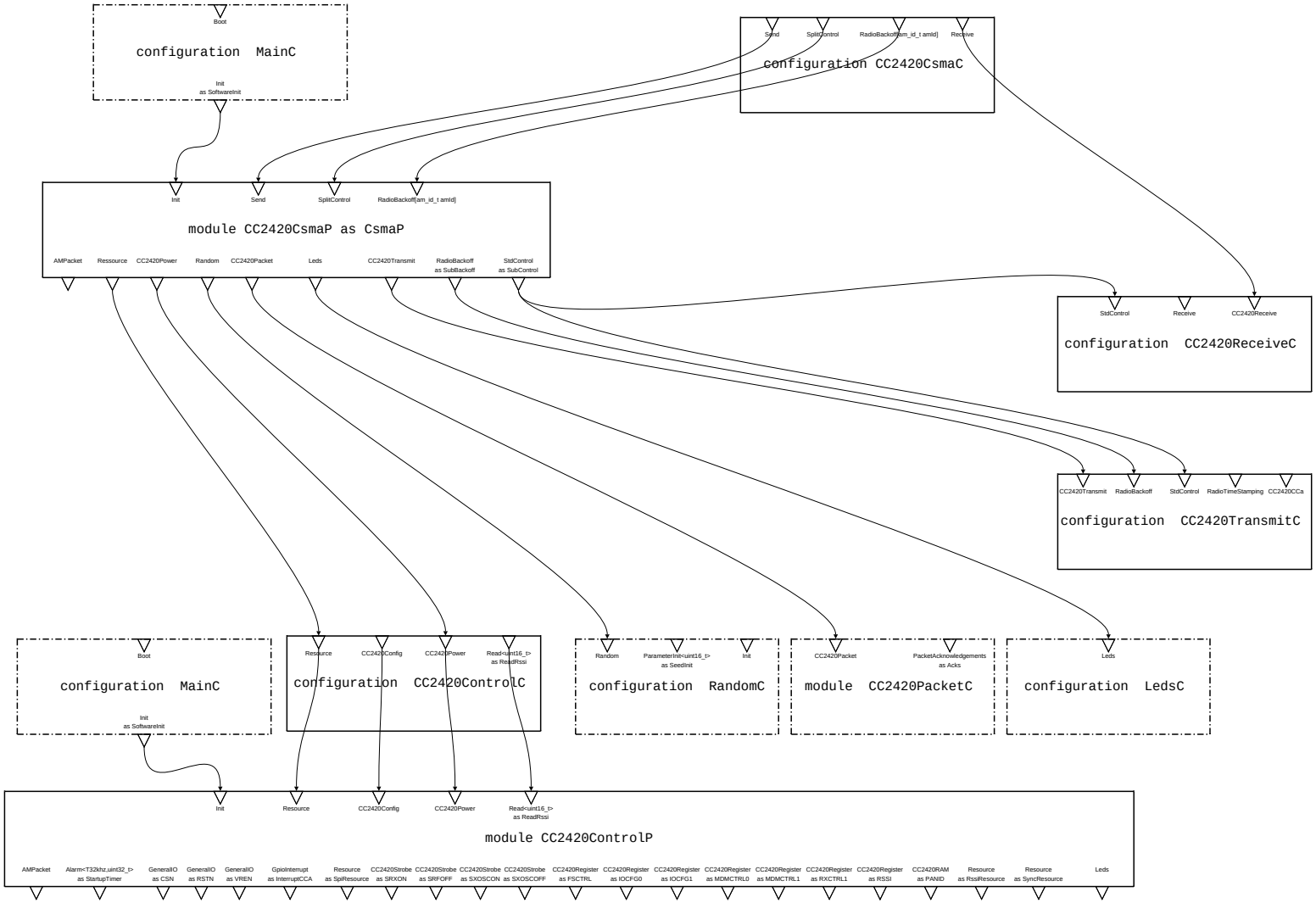


Figure 3-6
Wiring of the upper part of the CC2420 radio stack

4

Synchronous Low Power Listening

4.1 Basic Concept

Our implementation of a *synchronous low power listening* bases on the idea of WiseMAC. By means of the acknowledgment packets, timing information is exchanged between nearby nodes. A node can use this information to estimate the wake-up time of the destination node. Thus the source gets able to synchronize its transmission with the sampling schedule of the destination.

The existing radio stack in TinyOS should be modified as little as possible. Since the CC2420 is packet-based while WiseMAC assumes a bit stream radio, no adaptive preamble can be transmitted. Instead of a preamble, a packet burst is used. This means, the same packet is transmitted several times (Figure 4-1). Packet-based synchronous low power listening reduces the number of packets that have to be sent by starting the transmission only short before the wake-up time of the destination node.
After the transmission of a single packet, which is part of a packet burst, the transmitter waits a short interval. This is because the transmitter wants to allow the receiver of the packet to send back an acknowledgment packet. Hence, the packet burst is only as long as necessary since it is terminated after a successful acknowledged packet.
In order to be able to estimate the destination's next wake-up time, the transmitter needs some information about the sampling schedule of the destination node. This information is exchanged by each acknowledgement carrying 2 bytes with timing information. An adjacent node can use this timing information to determine the relative wake-up schedule offset between its own wake-up and the one of its neighbor.

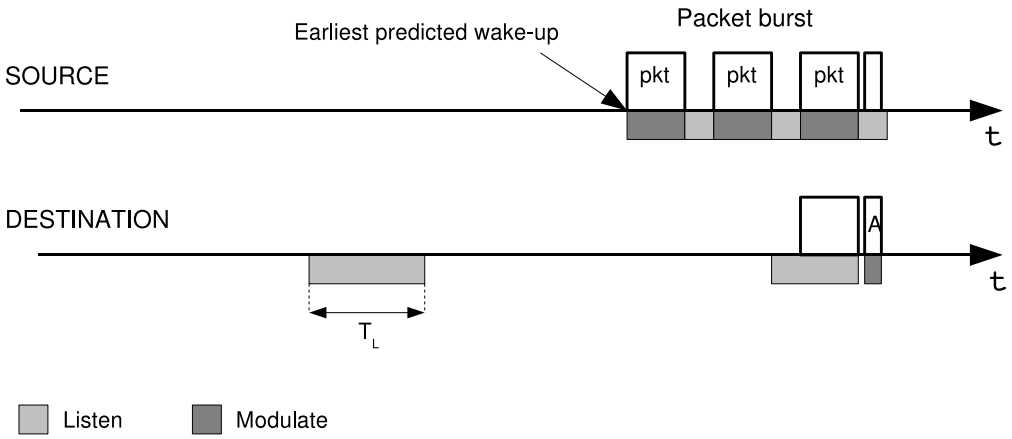


Figure 4-1
Packet-based synchronous low power listening
Instead of a bit-stream preamble a packet burst is used. The length of the burst has to guarantee that the sampling interval (T_L) of the destination is hit.
After each packet the channel is calm for a certain period to allow the reception of a possible acknowledgment packet. The sampling interval has to be larger than this period to guarantee that no burst is missed because the radio waked up between two packets.

4.2 Duty Cycling with a Constant Period

4.2.1 Concept

As described in Section 3.4.2 the current radio stack of the CC2420 in TinyOS already implements duty cycling but with a variable period. To be able to determine the wake-up point of a destination node, all nodes in a network have to sample the medium with a constant period T_W. Hence, the current CC2420 radio stack has to be modified to that effect that a constant wake-up period T_W is given.

The interval between two wake-ups of duration T_W basically consists of two phases. In one phase nothing has to be done and the radio is turned off. This phase we call the *sleep interval*. In the other phase, called *active interval*, the radio is started and the medium is checked for other nodes transmitting. The active interval starts when the wake-up timer is fired and ends when the radio is turned off again.

4.2.1.1 The Active Interval

According to the adjusted value, an alarm fires periodically. This point of time is the wake-up time t_W. In the event handler of the alarm, at first a new alarm has to be started to guarantee a constant wake-up period T_W. Figure 4-2 shows the events and processes going on during the active interval.

The duration of sampling the medium is determined by the time it takes from receiving a data packet and sending back an acknowledgement packet on the one hand and the time needed for the backoff because of the clear channel assessment on the other hand. We call the period in which the channel is sampled either *sampling interval* or *listening interval* T_L.

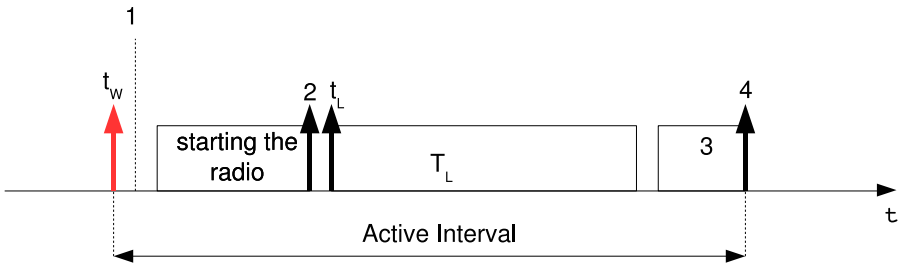


Figure 4-2
Active interval
An alarm firing at the wake-up time t_W initiates the start of the active interval. At first, the alarm for the next wake-up is set (1). Afterwards the radio is started up. The termination of the radio starting is indicated by a start-done event (2).
If the radio is on, sampling of the channel is started at t_L. During the whole sampling interval T_L the channel is checked for a possible modulation.
After the end of the sampling interval, the radio is turned off again (3). The stopping of the radio is signaled by a stop-done event (4).

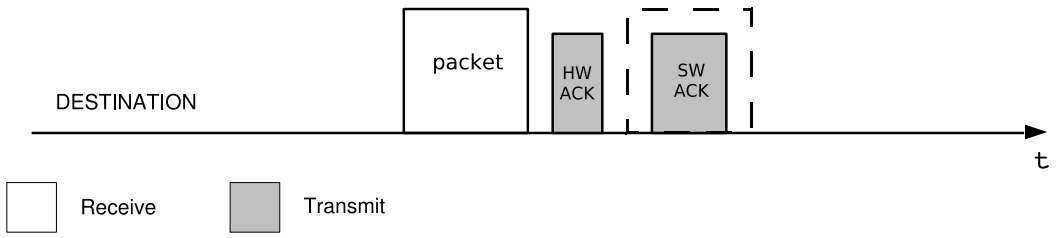


Figure 4-3
Acknowledgment concept
In order to minimize the period between two packets of the packet burst, a combination of a hardware acknowledgment and a software acknowledgment is used. First a fast hardware ack is transmitted and afterwards a software ack, which carries the synchronization information follows.
If no hardware ack is received, the transmitter immediately sends the next packet of the burst without waiting for a software ack.

4.2.2 Implementation

In the asynchronous low power listening radio stack the duty cycling is handled by the component *CC2420DutyCycleP*. In order to achieve duty cycling with a constant period, we have introduced the component *CC2420DutyCycleSyncP*, which bases on the duty cycle component of the asynchronous protocol.
The component of the synchronous protocol uses an alarm instead of a timer interface to initiate the wake-up. At first, the event handler `OnTimer.fired`, which handles the wake-up alarm, sets up the alarm for the next wake-up. By setting up the alarm in the event handler a duty cycling with a stable period can be guaranteed.

4.3 Acknowledgements

4.3.1 Concept

As it is shown in section 5.2, the software generated acknowledgements possess a turnaround time of about 108 $\text{clocks}_{32\text{kHz}}$. Since we do not want to implement an acknowledgement that is slower than the acknowledgement of the current radio stack version, we propose another idea for acknowledging a received packet.

If a packet is received, the new protocol first sends an acknowledgement that is generated and automatically launched by hardware. In the following step, the software generates a second acknowledgement packet, which also carries the synchronization information, and sends this packet to the original transmitter of the data-packet (Figure 4-3). As the original transmitter receives an acknowledgement, it first checks if it is the hardware or the software acknowledgement. In case of a hardware acknowledgement it elongates the countdown of the running timer to permit the reception of a software acknowledgement. Otherwise, in case of a software acknowledgement, the timestamp carried by the acknowledgement packet is used to calculate the offset between the duty cycle of the transmitting and the receiving node.

If no hardware acknowledgement is received within a certain period of time, a timer expires and an event signals to the components on higher layers that the transmission failed. The component on the Low Power Listening layer will then decide whether to send another data packet or to abort the transmission.

4.3.2 Implementation

4.3.2.1 Hardware Acknowledgement

In order to automatically generate and transmit hardware acknowledgements, the `AUTOACK` bit of the `MDMCRL0` configuration register has to be set to one. This is done in the component *CC2420ControlP* when the `CC2420Power.startOscillator()` command is called.

4.3.2.2 Software Acknowledgement

The IEEE standard 802.15.4 defines four different message types. They all have a predefined frame format. Since synchronous low power listening uses acknowledgement packets to transport additional timing information, the defined frame type 3, which specifies acknowledgement packets, cannot be used for a software acknowledgement. If we would use the IEEE 802.15.4 acknowledgement frame format and extend it by two additional bytes for the timing information, the packet would not be accepted by the address recognition performed in hardware. Such an acknowledgement is discarded.
However, in order to stay compatible with the IEEE standard, we use one of the reserved types (Section 1.3.1). Frames with a reserved type identifier are accepted by hardware, if the corresponding control bit `RESERVED_FRAME_MODE` in `MDMCTRL0` is set.
The synchronous low power listening protocol introduces the new frame type

IEEE154_TYPE_SYNC_ACK. Additionally to a normal acknowledgement, the new synchronization acknowledgement provides two bytes to transport information about the destinations sampling schedule. The definition of the new frame type is described in the file *IEEE802154.h*.

The transmitted sampling schedule information consists of the difference $\Delta_{W,next} = t_{W,next} - t_{now}$ where t_{now} is the time at which $\Delta_{W,next}$ is calculated and $t_{W,next}$ is the next wake-up time of the node. The calculation is done short before the software acknowledgement packet is written to the TXFIFO buffer.

4.3.2.3 Extension of the CC2420Transmit-Interface

In the current version of the CC2420 radio stack in TinyOS, the transmission of the hardware generated acknowledgement is released by a command strobe that is called from within the ReceiveP component. In order to be able to transmit an acknowledgement that carries a timestamp, the software generates the second (besides the one automatically sent by hardware) acknowledgement. To avoid conflicts by several components that want to access the TXFIFO-Buffer, we decided to implement the generation of the acknowledgement packet in the component CC2420TransmitP. Hence, it can be ensured that CC2420TransmitP is the only component that accesses the TXFIFO buffer.

To allow the component CC2420ReceiveP to access the mentioned generation procedure, a new command is integrated in the existent interface. This new command is called `sendAck()` and needs the sequence number of the packet that has to be acknowledged as argument.

4.3.2.4 Handling an Acknowledgement Packet

The component CC2420TransmitP is also responsible for the handling of a received acknowledgement. When a packet arrives, it first passes the component CC2420ReceiveP over which it gets to the component CC2420TransmitP. This component first checks whether the received packet is an acknowledgement. In case of a hardware acknowledgement the running alarm is stopped and set to the period CC2420_SYNCACK_WAIT_DELAY, which is defined in file *tos/chips/cc2420/CC2420.h*.This period roughly corresponds to the difference between the turnaround time of a complete hardware acknowledgement and a software acknowledgement.

4.4 Wake-up Time Estimation

4.4.1 Concept

Each node in a network samples the channel according its own sampling schedule with the constant period T_W. Since the clocks of the different nodes are not synchronized, there is a relative offset between the different sampling schedules. We call this offset the *sampling offset* Δ_L (Figure 4-4).

The sampling offset Δ_L can be calculated by taking the difference between $\Delta_{W,Dest,next}$, which can be extracted from the received acknowledgement and $\Delta_{W,Own,next}$, which is the time left to the next wake-up of the node itself.

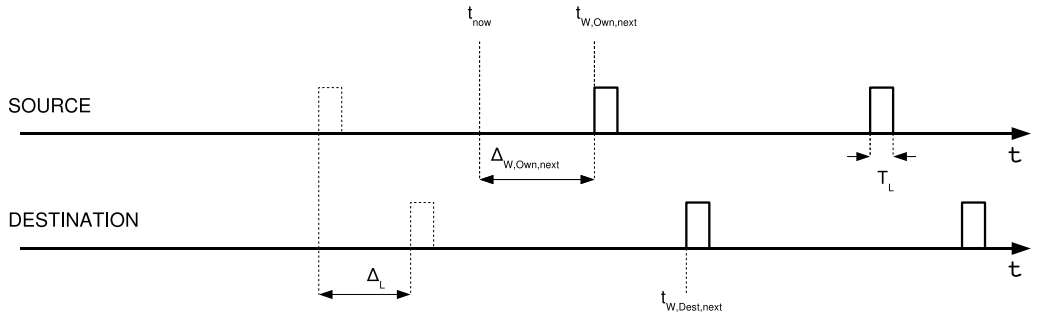


Figure 4-4
Parameters needed for wake-up time estimation
Δ_L: Offset between the two sampling schedules
$\Delta_{W,next,Own}$: Time until the next own wake-up will occur
$t_{W,Dest,next}$: Newt wake-up time of the destination

If a message is to be sent at time t_{now}, the destination's next wake-up time $t_{W,Dest,next}$ can be estimated by using the stored Δ_L and the source node's own next wake-up time:

$$t^{est}_{W,Dest,next} = t_{W,Own,next} + \Delta_L \tag{4.1}$$

4.4.2 Implementation

The calculation of Δ_L is added to the existing Module *CC2420TransmitP* and is done immediately after an acknowledgement packet is received. The generated $\Delta_L = \Delta_{W,Dest,next} - \Delta_{W,Own,next}$ does not exactly correspond to the real sampling offset Δ^{real}_L since there exists a hardware-caused offset between the sent $\Delta_{W,Dest,next}$ and the actual $\Delta^{real}_{W,Dest,next}$ at the point of time when the acknowledgement is transmitted. This offset is compensated when the estimated wake-up time of the destination $t^{est}_{W,Dest,next}$ is used for accurate sending.

The estimation of the destination's next wake-up time is done in the Component *CC2420WakeUpEstP* (Section 4.5.2.3).

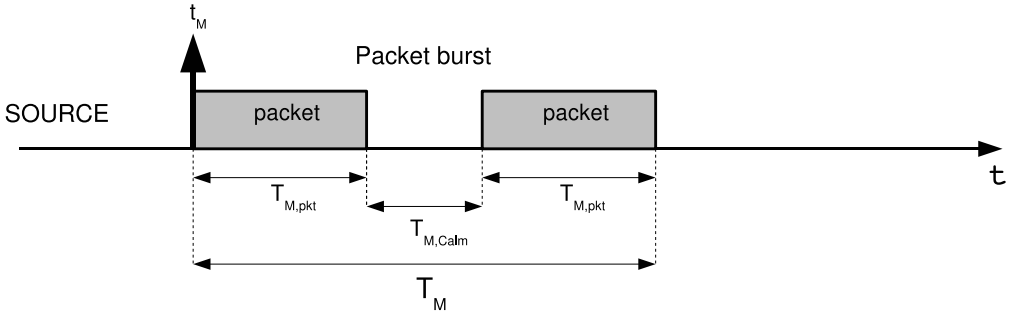
4.5 Accurate Sending

4.5.1 Concept

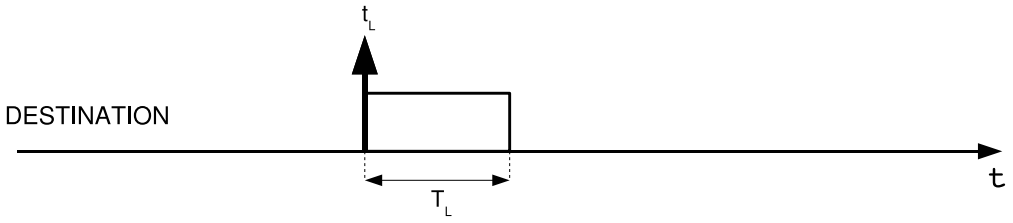
Using the estimated wake-up time t^{est}_W (Section 4.4), a transmitting node has to be sending the packet burst just at the right moment to hit the sampling interval of the destination node.

A receiving node that wakes up at t_W, has to turn on its radio first. Therefore, the destination node actually starts sampling at a certain period after t_W called t_L. The node samples the channel for the duration of T_L (*listening*) and then turns the radio off again.

A source node that transmits a message modulates the channel for the duration of period T_M, which depends on the packet length. T_M can consist of one packet but also of more. T_M has to be long enough in order to guarantee that the packet or the packet burst overlaps the sampling interval of length T_L.
If a packet burst of n_{pkt} packets is used, we call the period in which the channel is modulated by a packet $T_{M,pkt}$, while the period between two packets is called $T_{M,calm}$. Hence T_M of the packet burst is of length $T_M = n_{pkt} * T_{M,pkt} + (n_{pkt} - 1) * T_{M,Calm}$. Since the sampling interval of length T_L is larger than the period between two packets ($T_{M,Calm}$) at least one packet of the burst overlaps the sampling interval (Figure 4-5).



(a) Parameters concerning the transmitter of a message



(b) Parameters concerning the receiver of a message

Figure 4-5
Important parameters of the synchronous low power listening protocol
The modulation interval of a packet burst consists of the actual modulation of the channel ($T_{M,pkt}$) and the period between two following packets of the burst. The length of the packet burst (T_M) has to be long enough to make sure that the sampling interval of the destination is hit.
In order to prevent from missing a packet burst the minimal length of the sampling interval $T_{L,min}$ has to be larger than the maximal time between two packets of the packet burst $T_{M,Calm,max}$. Thus $T_{L,min} > T_{M,Calm,max}$ has to hold.

In order to be modulating the channel at same time as the destination node is sampling, the point of time t_M at which the transmission of the single packet or the packet burst respectively starts has to satisfy the following inequality.

$$t_L - T_M < t_M < t_L + T_L \tag{4.2}$$

However, the clocks might drift which needs to be taken into account. For the further

calculations, the maximal frequency tolerance of the quartz is called θ. In a worst case scenario one clock runs too fast ($f+\theta$) while the other clock goes too slow ($f-\theta$). Thus, the clockdrift θ has to be compensated twice which leads to Expression 4.3.

$$t_L + 2A\theta - T_M < t_M < t_L - 2A\theta + T_L \tag{4.3}$$

A is the age of the stored Δ_L.

From Inequality 4.3 the minimal needed T_M at a certain age of the synchronization information can be derived.

$$T_M > 4A\theta - T_L \tag{4.4}$$

$T_{M,min}$ leads to a minimal number of packets needed for the burst.

$$n_{pkt,min} = \lceil \frac{T_{M,min} + T_{M,Calm}}{2(T_{M,pkt} + T_{M,Calm})} \rceil \tag{4.5}$$

4.5.2 Implementation

4.5.2.1 Basics

We've added the Component *CC2420WakeUpEstP* in the radio stack to allow wake-up time estimation and accurate sending. In order to keep the packet burst as short as possible, the transmission of the first packet has to be deferred to t_M. A message that arrives at the CC2420WakeUpEstP component is first prepared, which means it is loaded into the TXFIFO buffer. This preparation is done by using the Interface *AccSend*, which is provided by the Component *CC2420TransmitC*. If the preparation is terminated, the point of time $t_{SendNow}$ at which the command `sendNow()` of the AccSend interface has to be called is determined. $t_{SendNow}$ is calculated by adding an offset $\Delta_{ModulationPlacing}$ to the estimated wake-up time t_L^{est}. The offset $\Delta_{ModulationPlacing}$ is used to place the modulation interval T_M in the middle of the estimated sampling interval T_L^{est}. $\Delta_{ModulationPlacing}$ can be determined by an empirical evaluation.

4.5.2.2 Increasing T_M

Since we want to keep the sampling interval as short as possible, we increase T_M. This can be done by using two packets as described in Section 4.5.1. If we set T_M to 3ms, which corresponds to the interval between two messages sent, we are on the secure side, since T_M is actually higher. In the case of two packets used, t_M has to be chosen so that T_M is placed in the middle of T_L^{est}.

4.5.2.3 CC2420WakupEstC Component

The configuration *CC2420WakeUpEstC* wires the interfaces used by the module *CC2420WakeUpEstP* to the component *CC2420TransmitP*. The interfaces provided by the CC2420WakeUpEstC component are used by the component CC2420CsmaC. Compared to the initial CC2420 radio stack where the Component *CC2420CsmaC* directly wires to the Component *CC2420TransmitC*, our implementation inserts a sublayer between the CSMA layer and Transmit layer. This sublayer is responsible for deferring the packet burst to keep it as short as possibel.

4.5.2.4 AccSend Interface

The newly introduced interface *CC2420AccSend* consists of the following commands and intefaces:

```
/**
 * Interface which allows accurate sending of packets on the
 * ChipCon CC2420 radio.
 *
 * @author Roman Amstutz <amstutzr@ee.ethz.ch>
 * @version $Revision: 0 $ $Date: 2007/05/21 17:11:12 $
 */

interface CC2420AccSend {

  /**
   * Prepare sending a message
   *
   * @param p_msg message to send.
   * @param useCca TRUE if this Tx should use clear channel assessments
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t prepare( message_t* p_msg, bool useCca );

  /**
   * Send the message which is already in the buffer
   * Use this command only when the event prepareDone is
   * signaled
   *
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t sendNow();


  /**
   * Send the previous message again
   * @param useCca TRUE if this re-Tx should use clear channel assessments
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t resend(bool useCca);

  /**
   * Cancel sending of the message.
   *
   * @return SUCCESS if the request was accepted, FAIL otherwise.
   */
  async command error_t cancel();

  /**
   * Signal that a message has been sent
   *
   * @param p_msg message to send.
   * @param error notifaction of how the operation went.
   */
  async event void sendDone( message_t* p_msg, error_t error );

  /**
   * Signal that the message has been loaded into the buffer
   *
   * @param p_msg messsage to send.
   * @param error notifaction of how the operation went.
   */
  async event void prepareDone( message_t* p_msg, error_t error);

}
```

Listing 4.1
Code: AccSend Interface

The interface allows executing the functionality of the normal Send interface split into two parts. In a first phase, the message is loaded into the TXFIFO buffer by calling the command `prepare()`. In the second phase, the message loaded into the buffer can be accurately sent by calling `sendNow()`. Accurately sending does not mean that the message leaves immediately when the command is called, but the delay is deterministic, since it does not depend on the length of the packet and all the function calls involved are asynchronous.

5

Design Evaluation

This section presents the performance analysis of the synchronous low power listening protocol and lists comparisons with the asynchronous protocol.

5.1 Duty Cycling

A synchronous low power listening protocol, which bases on wake-up time estimation, requires a duty cycle with a constant wake-up time period. Otherwise, the wake-up time of the receiver would not be deterministic. The stability of the new implemented synchronous duty cycling is therefore evaluated in this section.

As mentioned in Section 3.4.2, the original asynchronous implementation of the CC2420 low power listening radio stack of TinyOS changes to a sleep interval after the listening interval is terminated. The length of the listening interval depends on whether another node transmitting is detected or not. The sleep interval starts, when the radio is stopped. This results in a wake-up period that can strongly vary.
In order to allow wake-up time estimation, the component *CC2420DutyCycleSyncC* whose interfaces are implemented in the module *CC2420DutyCycleSyncP* provides a duty cycling with a stable period. This stability of the duty cycle period has been evaluated considering 4500 duty cycles. In only 0.22% of the evaluated cycles, the period differs from 3200 clocks. The difference is always only one clock (Figure 5-1). Thus, the duty cycle period can be assumed as constant.

5.2 Acknowledgements

In the context of acknowledgement packets, we define the period that it takes from sending a message out of the TXFIFO buffer until the corresponding acknowledgement is received as *turnaround time*. This period is of importance since it determines the acknowledgement waiting period $T_{AckWait}$, which again affects the lower bound of the sampling interval T_L.
To determine the lower bound of the wait period between two packets ($T_{AckWait}$), we are going to evaluate the resulting turnaround time for the three different types

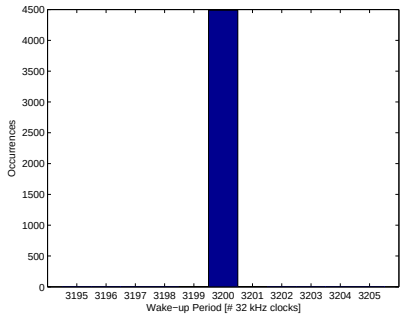


Figure 5-1
Distribution of the Wake-up Period

of acknowledgments. These are hardware acknowledgments (*HW Ack*), software controlled hardware acknowledgments (*SWHW Ack*) (both Section 2.2.2) and software acknowledgments (*SW Ack*). For the evaluation one node is listening all the time while the transmitting node tries to send a message every second.

5.2.1 Hardware Acknowledgement

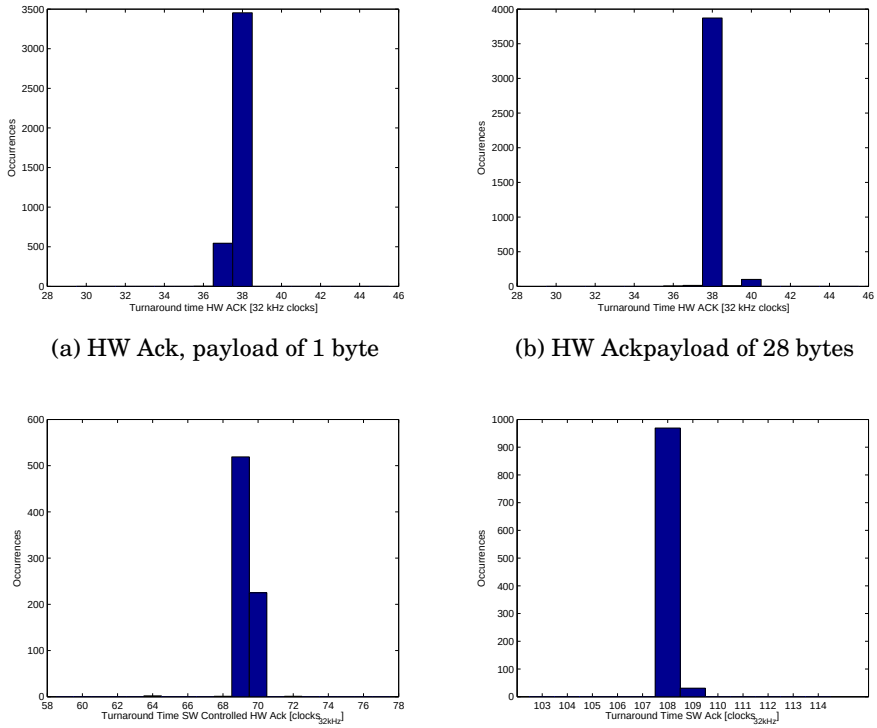
The CC2420 supports acknowledgements that are completely generated and released by hardware. In this case, the software does not have to care about the transmission of an acknowledgement. The drawback of using hardware acknowledgements is that it could lead to packets that get acknowledged by the hardware but are never recognized by the software.

To evaluate the turnaround time of a hardware acknowledgement a test component that periodically (every second) transmits a packet is used. To study the influence of the packet length, packets of different payloads have been used. The results are listed in Table 5-1 and show a slightly increased turnaround time of 0.19 $\text{clocks}_{32\text{kHz}}$ in average if additional bytes are sent.

Figure 5-2 shows the distributions of the turnaround times using variable packet length. Since the maximal payload length of the implemented radio stack is set to 28 bytes, the maximal expected turnaround time is limited to 40 $\text{clocks}_{32\text{kHz}}$.

5.2.2 Software controlled Hardware Acknowledgements

The original, asynchronous CC2420 radio stack in TinyOS uses hardware acknowledgements that are initiated by software. 98.42% out of 760 messages sent have been acknowledged. The resulting turnaround time can be seen in Table 5-1 while the distribution is shown in Figure 5-2.
The turnaround time of software controlled hardware acknowledgment is about 30 $\text{clocks}_{32\text{kHz}}$ higher than the one of hardware acknowledgments.



(a) HW Ack, payload of 1 byte

(b) HW Ackpayload of 28 bytes

(c) SWHW Ack

(d) SW Ack

Figure 5-2
Distribution of the turnaround time for different acknowledgment types

5.2.3 Software generated acknowledgement

With a maximal measured value of 109 $\mathrm{clocks}_{32\mathrm{kHz}}$ (Table 5-1), software generated acknowledgments show the longest turnaround time. This result is not surprising since it takes additional time to firstly generate the acknowledgment packet in software and load it into the TXFIFO buffer.

5.2.4 Summary

The evaluation clearly shows that hardware acknowledgments have the shortest turnaround time. Sending an acknowledgment completely generated in software takes more time than sending a software controlled hardware acknowledgment.
Thus the synchronous protocol, which uses hardware acknowledgments in a first step, shows a decreased $T_{AckWait}$ compared to the asynchronous protocol, which uses software controlled hardware acknowledgments.
But the synchronous protocol uses the acknowledgment packets to carry synchronization information, which cannot be included in standard hardware acknowledgment packets. Thus, software acknowledgments have to be used combined with hardware acknowledgments.

5.3 Wake-up Time Estimation

The wake-up time estimation is based on the synchronization information in the acknowledgment packet. With the help of this scheduling information, the transmitter is able the transmit a message exactly during the time slot when the destination node is active.
In order to exactly determine the next wake-up time of the destination, the synchronization information has to be as accurate as possible. If the time interval between the generation and the interpretation of the synchronization information is not constant, it is not possible to predict the next wake-up time of the destination.
In this section the accuracy of the timestamp and the wake-up time estimation is evaluated.

5.3.1 Accuracy of the Timestamp

As mentioned above, each acknowledgement packet carries a timestamp that indicates the time difference between the point of time short before the acknowledgement is loaded into the FIFO buffer and the next wake up time. Knowing this period of an adjacent node, a node is able to calculate the mentioned offset.

The problem that occurs concerns the accuracy of the described timestamp. It takes a certain period form generating the timestamp until the acknowledgement can be evaluated at its receiver. In order to being able to reconstruct the point of time when the stamp has been generated, the period from stamping to evaluate should be constant.
Before a message can be transmitted by the CC2420, it has to be loaded into the FIFO buffer and afterwards it can be sent by calling the corresponding command strobe.
The time from building the timestamp until the TXFIFO write-done event occurs $T_{StampToWrDone}$ has been evaluated. 2800 samples have been considered. Figure 5-3 shows the distribution of the duration of mentioned interval of time. We found following results:

Mean	σ	Min	Max
22.65 $\text{clocks}_{32\text{kHz}}$	0.51 $\text{clocks}_{32\text{kHz}}$	22 $\text{clocks}_{32\text{kHz}}$	26 $\text{clocks}_{32\text{kHz}}$

$T_{StampToWrDone}$ is part of the interval from taking the timestamp until the start of frame delimiter is sent $T_{StampToSend}$. This period is of importance for a node which want to interpret an acknowledgement for wake-up time estimation. We have evaluated $T_{StampToSend}$ of 1640 messages. The results are shown below:

Mean	σ	Min	Max
40.69 $\text{clocks}_{32\text{kHz}}$	0.54 $\text{clocks}_{32\text{kHz}}$	38 $\text{clocks}_{32\text{kHz}}$	44 $\text{clocks}_{32\text{kHz}}$

5.3.2 Arrival Offset

in order to verify that our wake-up time estimation is correct, we have measured the *arrival offset* (Figure 5-4). In order to measure this period an always listening receiver is needed. Otherwise the packet would not be detected, if it is not sent within the sampling interval. Therefore the component *CC2420DutyCycleSyncP* has been

Ack Type	**Payload** [byte]	**Turnaround Time** [$\text{clocks}_{32\text{kHz}}$]				**# Samples**
		Mean	σ	Min	Max	
HW	1	37.86	0.34	36	38	4000
HW	28	38.05	0.33	36	40	4000
SWHW	1	69.29	0.55	64	72	760
SW	1	108.03	0.17	108	109	1000

Table 5-1: Turnaround time for different acknowledgment types

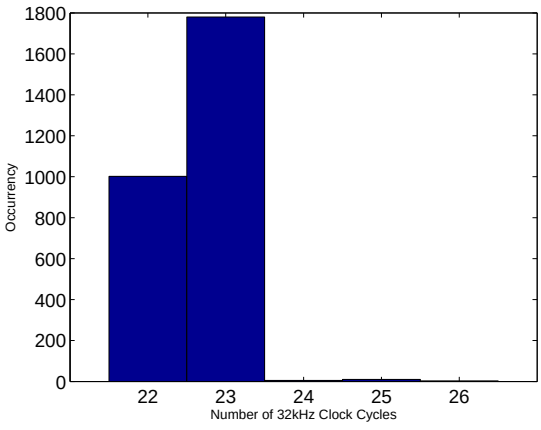


Figure 5-3
Distribution of the time from timestamping until TXFIFO write done event

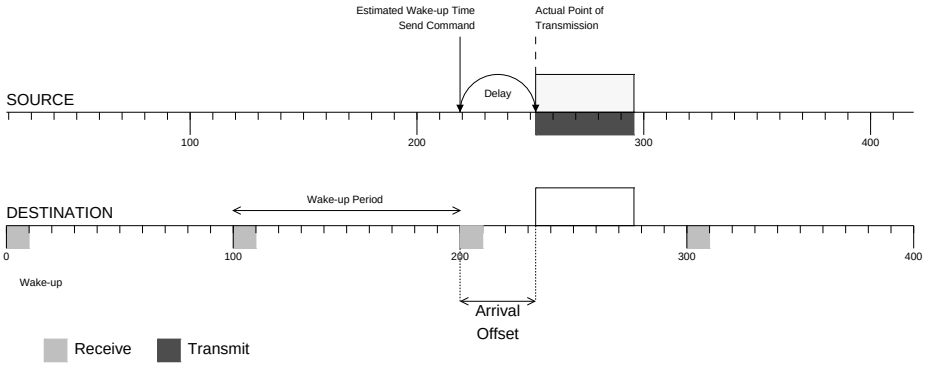


Figure 5-4
The arrival offset is the period between the point when the receiver starts listening and the point when the packet of the transmitter arrives. It is the delay of the arriving packet that is caused by the hardware on one hand and the offset of the timestamp on the other hand.

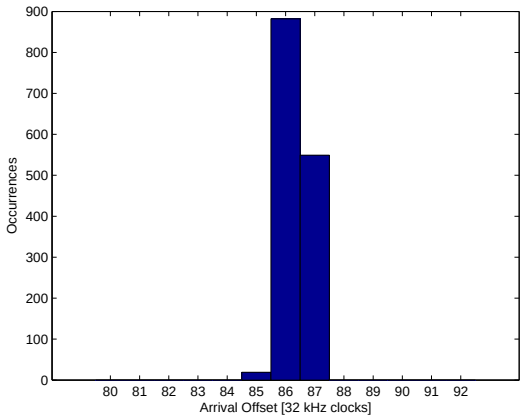


Figure 5-5
Distribution of the arrival offset

modified to that effect that the radio is always left on, but nevertheless a timer indicates the wake-up time.

Figure 5-5 shows the results of a test run where the transmitter sends the message at the point of time at which it assumes the destination node to wake-up. The arrival offset differs from 86 by maximal 1 $\text{clocks}_{32\text{kHz}}$. Allowing a tolerance of $\pm 1\text{clock}_{32\text{kHz}}$, the arrival offset can be assumed to be 86 $\text{clocks}_{32\text{kHz}}$.

5.4 Protocol Performance

Finally, we are evaluating the synchronous low power listening protocol.

5.4.1 Receiver: Sampling Interval T_L

The duration of the sampling interval T_L, which is part of the active interval, is an important parameter since the radio consumes the more energy the longer T_L is.

Duty cycling causes the radio of a possible receiver to be powered off most of the time. A node that wants to transmit a packet to other nodes has to be modulating the channel during the short interval in which the destination is sampling the channel (T_L). This means, the transmitter has to ensure that its modulation interval T_M overlaps the listening interval T_L of a receiving node. Hence, several packets are sent in short intervals after each other. After each packet sent, the transmitter waits for a certain period of time ($T_{AckWait}$) to check whether the packet has been acknowledged.

If no acknowledgement arrives within $T_{AckWait}$, the packet is resend. Hence, there is a period between the sending of two packets in which the channel is not modulated. We call this period *resend calm interval* $T_{M,Calm}$. The maximal occurring $T_{M,Calm}$ determines the minimal allowed T_L. If $T_{L,min}$ would be smaller than $T_{M,Calm,max}$ it could happen, that the destination node misses the channel modulation of the source T_M since it exactly samples between the two packets transmitted shortly after each other (Section 4.5).

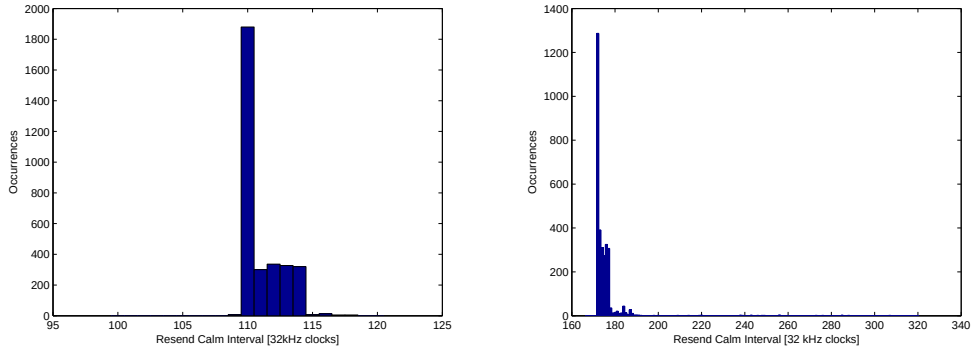
For the synchronous LPL, the evaluation of 3200 sendings, with $T_{AckWait} = 64\text{clocks}_{32\text{kHz}}$, has presented a $T_{M,Calm,max}$ of 118 $\text{clocks}_{32\text{kHz}}$, while the mean value lies at $111.1\text{clocks}_{32\text{kHz}}$. The distribution of T_{Calm} is shown in Figure 5-6. The variance is caused by the random *LPL backoff*, which is in maximum 10 $\text{clocks}_{32\text{kHz}}$.
The resulting values and distribution of the evaluation of 3200 sendings in the asynchronous listening protocol can also be seen in Table 5-2 and Figure 5-6 respectively. It can be seen that the synchronous LPL protocol reduces $T_{M,Calm}$ by at least 64 $\text{clocks}_{32\text{kHz}}$.

Consequently, the sampling interval T_L of the synchronous protocol could be decreased by 64 $\text{clocks}_{32\text{kHz}}$

Protocol	**Resend Calm Interval** $T_{M,Calm}$ $[clocks_{32kHz}]$			
	Mean	Standard Deviation	Minimum	Maximum
Sync	111.07	1.51	109	118
Async	176.12	12.81	167	320

Table 5-2: Values of the resend calm interval $T_{M,Calm}$ for synchronous and asynchronous LPL

In order to determine t_M according Inequality 4.3, we have to know T_L. For this reason 50000 sampling intervals have been evaluated. The resulting values are shown in Table 5-3 while the distributions for a different number of CCA pin samplings can be seen in Figure 5-7. In general it can be said that the standard deviation is only small and all values lie within 4 $\text{clocks}_{32\text{kHz}}$. Hence, for our requirements T_L can be assumed as constant.

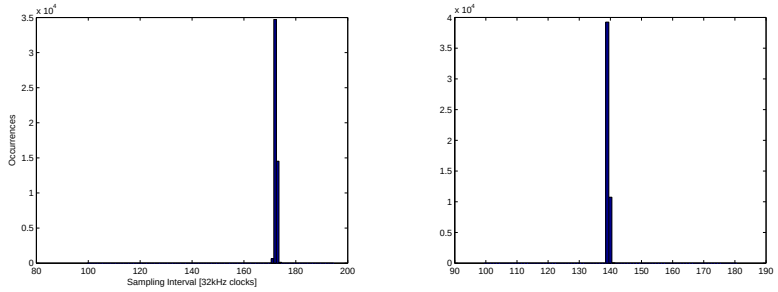


(a) Synchronous LPL (b) asynchronous LPL

Figure 5-6
Distribution of $T_{M,Calm}$

# CCA Checks	Sampling Interval T_L $[(clocks)_{32kHz}]$			
	Mean	Standard Deviation	Minimum	Maximum
500 (async protocol)	172.28	0.48	171	174
400 (sync protocol)	139.22	0.41	139	141

Table 5-3: Values of the sampling interval T_L



(a) 500 samplings (b) 400 samplings

Figure 5-7
Distribution of the sampling interval T_L for a different number of CCA pin samplings

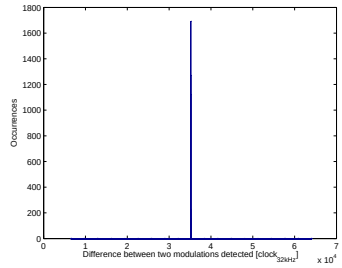


Figure 5-8
Distribution of period between two modulations detected

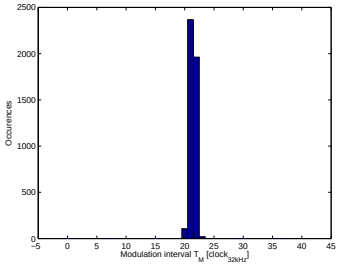


Figure 5-9
Distribution of T_M

5.4.2 Transmitter: Modulation Interval T_M

The duration of the modulation interval T_M depends on the length of the message sent. The length consists of the Synchronization Header, the Physical Header, the MAC Header, the Payload and the MAC Footer. In order to make a general estimation of the optimal start of sampling t_M, we have to know the minimal occuring T_M. This can be done by sending a packet of 1 byte payload. The transmission of 4500 message has been evaluated. We have measured T_M at the destination node by looking at the period over which the CCA pin sampling detected a modulation of the channel. Figure 5-8 shows the distribution of the periods between two detected modulations. Only the samples with a difference from the previous modulation of the channel between 35190 $\text{clocks}_{32\text{kHz}}$ and 35210 $\text{clocks}_{32\text{kHz}}$ can originate from the messages sent by the transmitting node. This is because the transmitting node sends a message always 1 second after the transmission of the previous message which results in a constant period. Thus the detected transmissions can not be caused by the transmitter. They are caused by noise that is detected by the CC2420 or other sources that use the same frequency as the CC2420.
If we only consider samples that exhibit a difference between each other of about 35200 $\text{clocks}_{32\text{kHz}}$, a minimal duration of $T_{M,min} = 20\text{clocks}_{32\text{kHz}}$ and a maximal duration of $T_{M,max} = 23\text{clocks}_{32\text{kHz}}$ is obtained. All the values of the evaluation are shown below:

Mean	σ	Min	Max
21.43 $\text{clocks}_{32\text{kHz}}$	0.55 $\text{clocks}_{32\text{kHz}}$	20 $\text{clocks}_{32\text{kHz}}$	23 $\text{clocks}_{32\text{kHz}}$

Figure 5-9 shows the distribution of T_M.
The duration of the modulation interval of about 21 $\text{clocks}_{32\text{kHz}}$ can be explained as follows:
The length field of a message with 1 byte payload shows a packet length of 13 bytes. This length consists of the payload, the MAC header and the MAC footer (Section 1.3). By adding the preamble of 4 bytes, the SFD of 1 byte and the length field of another byte, overall 18 bytes are transmitted. With a bit rate of 250 kbps, this leads to a needed time for the transmission of the packet of about 19 $\text{clocks}_{32\text{kHz}}$.

5.4.3 Comparison of the Synchronous and the Asynchronous Low Power Listening

The goal of the synchronous duty cycling implementation has been to estimate the wake-up time of the receiver. The information about the wake-up time of the receiver can be used to reduce the number of packets that are sent for each transmission. Thus, the number of packets sent per transmission can be used to compare the performance of the synchronous and the asynchronous protocol. The fewer packets are needed until an acknowledgement is received, the less energy is consumed.

The protocol parameters are shown in the following table:

Parameter	**Sync protocol**	**Async protocol**	**Description**
T_W	100 ms	100 ms	Wake-up period
$T_{AckWait}$	64 $\text{clocks}_{32\text{kHz}}$	128 $\text{clocks}_{32\text{kHz}}$	Time after which a resend is triggered if no acknowledgement is received
`MAX_LPL_CCA_CHECKS`	400	500	Maximal number of `CCA` pin samples executed before a listening phase is terminated

In order to determine the start of the of the packet burst t_M, the values $T_L = 120\text{clocks}_{32\text{kHz}}$ and $T_M = 100\text{clocks}_{32\text{kHz}}$ have been applied to equation 4.2. The packet burst consisted of two packets. An overview of the results can be seen in Table 5-4.

A component that sends messages in random intervals between 5 and 10 seconds has been used for a first evaluation. 230 transmissions have been executed. The synchronous low power listening protocol needs to send only one packet for each of the 230 transmissions. Hence, the only packet sent with wake-up time estimation exactly matches the listening interval of the receiver.
On the other hand the asynchronous low power listening protocol needs in average 9.34 packets per transmission (standard deviation 5.36).

Another evaluation has been made in which the messages were sent in random intervals between 0.5 and 1 second.
In order to analyze the synchronous low power listening protocol we look at 39000 samples. In only 3 cases more than 1 packet has been need for the transmission. Two of these 3 cases directly follow each other. The transmission of more than one packet has been caused by a missing acknowledgement which forced the protocol back to the asynchronous mode. In the third case two packets were needed instead of one.
The asynchronous low power listening protocol exhibits in average 8.94 packets per transmission (standard deviation 5.34). 4900 samples have been evaluated.
Figure 5-10 shows the distribution of the number of packets per transmission over a randomly chosen segment of time.

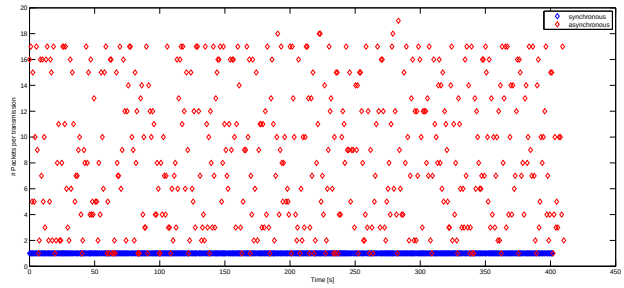


Figure 5-10
Comparison of the message generation between the asynchronous and the synchronous protocol
Each point indicates the number of packets used of one transmission. The synchronous protocol shows a clear enhancement compared to the asynchronous protocol.

Protocol	Packet frequency [s^{-1}]	Mean number of packets of the burst	# Samples
Sync	0.1 - 0.2	1	230
Sync	1 - 2	1.01	39000
Async	0.1 - 0.2	9.34	230
Async	1 - 2	8.94	4900

Table 5-4: Comparison of the number of packets per transmission

6

Conclusion

6.1 Achievements

The task of this thesis consists of an enhancement of the existion TinyOS 2.x radio stack, which already implements a packet-based low power listening. We managed to implement a wake-up time estimation on the basis of the existing radio stack.
The result is a prototype of a MAC protocol that works deterministic. The proper working has been verified by various measurements and analyses. The prototy decreases the number of packets needed per transmission and and reduces the duration of the sampling interval.

The following components of the TinyOS 2.x radio stack have mainly been changed or were added:

- CC2420DutyCycleSyncC (replaces CC2420DutyCycleC)
- CC2420DutyCycleSyncP (replaces CC2420DutyCycleP)
- CC2420Transmit (extended)
- CC2420TransmitP (extended)
- CC2420Receive (extended)
- CC2420ReceiveP (extended)
- CC2420WakeUpEstC (added)
- CC2420WakeUpEstP (added)
- CC2420AccSend (added)

Smaller changes have also been made in other components.

6.2 Summary

Since the whole implementation bases on an existing protocol, at the beginning, at lot of time has been spent on analysing the original code. If one component has to be changed, the whole influence on other components has to be considered in order to prevent from affecting the propper working of the protocol.
The implementation work has been an interesting challenge. Various aspects such as theoretical concepts of sensor network MAC protocol or the component-based programming language nesC had to be considered. The implementation of the protocol has been an optimal opportunity to learn a lot about embedded devices and sensor networks.
At the end, an evaluation took place. The meassurements verified the deterministic working of the protocol and showed an obvious enhancement of the original asynchronous protocol.

A

Aufgabenstellung

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik (TIK)

Sommersemester 2007

SEMESTERARBEIT

für
Roman Amstutz

Betreuer: Andreas Meier

Ausgabe: 26. März 2007
Abgabe: 2. Juli 2007

Wake-up Time Estimation for a Wireless MAC Protocol

Einleitung

Ein drahtloses Sensor Netzwerk (WSN—Wireless Sensor Network) besteht aus einer Vielzahl von kleinen resourcenbeschränkten Knoten welche mit Funkmodul und Sensoren bestückt sind. Diese werden in der Umwelt (z.B. in einem Haus) verteilt und erstellen möglichst autonom ein Netzwerk. Ein solches Netz ermöglicht den Knoten Sensor-Messungen auszutauschen und diese Daten gemeinsam zu verarbeiten. Nach einer Vision von Stankovic et al. [?] soll dies die 'nahtlose Integration von Rechner mit der Umwelt mit Hilfe von Sensoren und Aktoren ermöglichen'.

Ein Anwendungsszenario für solche WSNs ist die Sammlung der Sensordaten in einem designierten Knoten (Senke/Zentrale), welcher anhand der erhaltenen Information Entscheidungen treffen muss. Vielfach sind die Knoten physikalisch weiträumig verteilt, was der Mehrheit der Knoten verunmöglicht direkt mit der Senke zu kommunizieren. Stattdessen müssen die Daten über mehrere Knoten/Hops zur Senke geschickt werden.

Ein zentraler Aspekt solcher Sensornetzwerke ist die Funkkommunikation. Einerseits ist diese aufgrund des geteilten Mediums Luft nicht immer zuvelässig und vorhersehbar [?, ?], und andererseits wird für den betrieb des Funkmodules (z.B. CC1000 [?] oder CC2420 [?]) sehr viel Energie benötigt. Um Energie zu sparen wurde eine Vielzahl verschiedener "Medium Access Control (MAC)" Protokolle Entwickelt, die dies basierend auf der Idee vom regelmässigen An- und Ausschalten (Duty Cycles) tun. Vor allem für sehr Energiearme Anwendungen hat sich herausgestellt, dass das Prinzip von "Low Power Listening" kombiniert mit dem Senden einer sehr langen Preamble (B-MAC [?]) als sehr geeignet herausgestellt. Dieses Prinzip wurde dann von WiseMAC [?] verfeinert indem die asynchrone Kommunikation synchronisiert und dadurch die Preamble markant verkürzt wurde. Das versenden von langen Preamble ist aber nicht mit allen Funkmodulen möglich. Vor allem neuere Module, wie der CC2420, sind Paket basiert und erlauben dies nur noch sehr beschränkt. Dies führte zu weiteren MAC Protokollen wie X-MAC [?] und eines welches zur Zeit unter TEP126 [?] von der TinyOS-2.x Gemeinde diskutiert und implementiert wird.

Das TEP126 hat aber den grossen Nachteil, dass es das Energiesparpotential durch Synchronisation, wie in WiseMAC vorgeschlagen, nicht nutzt. In dieser Arbeit soll nun basierend auf der Asynchronen Kommunikation des TEP126 ein zusätzlicher synchroner Modus (wie in WiseMAC vorgeschlagen) implementiert werden. Zwei wichtige Aspekte die dabei untersucht werden sollen sind die Hardwarebedingte Verzögerung beim Versenden der Pakete sowie die unterschiedlich schnell laufenden Uhren (clock drift) der verschiedenen Knoten.

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest [?]. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze, Systeme, und MAC Protokolle vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie auch nach relevanten neueren Publikationen. Vergleichen Sie bestehende Konzepte anderer Universitäten. Prüfen Sie welche Ideen/Konzepte Sie aus diesen Lösungen verwenden können. Im Speziellen studieren sie [?], [?] und [?].
3. Die Applikation soll auf dem Tmote Sky [?] entwickelt werden. Arbeiten Sie sich in die Softwareentwicklungsumgebung (TinyOS-2.x) der Knoten ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (Versionskontrolle, Bugtracker, online Dokumentation, Mailinglisten, Application Notes, Beispielapplikationen). Schauen Sie dazu insbesondere das TinyOS Webpage, sowie die Datenblätter des MSP430 und des CC2420 an.
4. Machen Sie sich mit der Ansteuerung des CC2420 auf dem Tmote Sky mit TinyOS-2.x vertraut. Schauen Sie insbesondere auf den genauen zeitlichen Ablauf vom Senden und Empfangen von Paketen bzw. wie dieser beeinflusst werden kann.
5. Erstellen sie basierend auf der asynchronen TEP126 Implementation [?] ein Konzept für einen synchronen Modus.
6. Setzen Sie dieses Konzept um, d.h. implementieren Sie die Applikation auf dem Tmote Sky. Analysieren Sie dazu die genauen zeitlichen Abläufe insbesondere auch die Clockdrifts der verschiedenen Power Modes.
7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Semesterarbeit

Allgemeines

- Der Verlauf des Projektes Semesterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Sie verfügen über einen PC mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern.
- Sie führen ein Researchtagebuch in welchem sie die Fortschritte täglich protokollieren.

Abgabe

- Geben Sie vier unterschriebene Exemplare des Berichtes, das Researchtagebuch sowie alle relevanten Source-, Object und Konfigurationsfiles bis spätestens am 2. Juli 2007 dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Bibliography

[1] Wei Ye, John Heidemann, and Deborah Estrin. *An Energy-Efficient MAC Protocol for Wireless Sensor Networks*. Proc. IEEE INFOCOM Conf., 2002.

[2] J. Polastre, J. Hill, and D. Culler. *Verstile Low Power Media Access for Wireless Sensor Networks*. Proc. 2nd ACM Conf. Embedded Networked Sensor System (SenSys 2004), pp. 95-107, ACM Press, New York, 2004.

[3] A. El-Hoiydi and J. Decotignie. *WiseMac: An Ultra Low Power MAC Protocol for Multi-hop Wireless Sensor networks*. Proc. 1st Int'l Workshop Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS 2004) (S. Nikoletseas and J. Rolim, eds.), vol 3121 of Lecture Notes in Computer Science, pp. 18-31, Springer, Berlin, June 2004.

[4] *IEEE 802.15.4*. http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf.

[5] Chipcon, CC2420. *2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*, 2003.

[6] D. Culler et al. *TinyOS: An operating system for Networked Sensors*. http://www.tinyos.net.

[7] Philip Levis. *TinyOS Programming*, June 2006.

[8] Philip Levis. *TinyOS 2.x:TEP111*. TinyOS Developer List.

[9] D.Moss, J. Hui, and Philip Levis. *TinyOS 2.x:TEP126*. TinyOS Developer List, March 2007.