Low Power Listening

TEP:	105
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	David Moss, Jonathan Hui, Kevin Klues

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

This TEP describes the structure and implementation of the TinyOS 2.x link layer abstractions. The architecture is designed to allow each radio type to implement its own low power strategy within the Hardware Adaptation Layer (HAL), while maintaining a common application interface. The history and strategies for low power listening are discussed, as well as expected behavior and implementation recommendations.

1. Introduction

Asynchronous low power listening is a strategy used to duty cycle the radio while ensuring reliable message delivery since TinyOS 1.x [MICA2].

While a CC1000 or CC2420 radio is turned on and listening, it can actively consume anywhere between 7.4 to 18.8 mA on top of the power consumed by other components in the system [CC1000],[CC2420]_. This can rapidly deplete batteries. In the interest of extending battery lifetime, it is best to duty cycle the radio on and off to prevent this idle waste of energy. In an asychronous low power message delivery scheme, the duty cycling receiver node saves the most energy by performing short, periodic receive checks. The power consumption burden is then placed on the transmitter node, which must modulate the radio channel long enough for the recipient?s receive check to detect an incoming message. A synchronous low power message delivery scheme takes this idea a step further by allowing the transmitter to only transmit when it knows the destination node is performing a receive check.

2. Background

2.1 Early TinyOS 1.x CC1000 Low Power Listening Implementation

TinyOS 1.x introduced low power listening on the CC1000 radio, but never introduced a similar scheme for the CC2420 radio in the baseline. The CC1000 radio had the following low power listening commands, provided directly by CC1000RadioIntM::

```
command result_t SetListeningMode(uint8_t power);
command uint8_t GetListeningMode();
command result_t SetTransmitMode(uint8_t power);
command uint8_t GetTransmitMode();
```

The uint8_t 'power' mode parameter was initially defined as follows::

```
//Original CC1000 Low Power Listening Modes
Power Mode 0 = 100% duty cycle
Power Mode 1 = 35.5% duty cycle
Power Mode 2 = 11.5% duty cycle
Power Mode 3 = 7.53% duty cycle
Power Mode 4 = 5.61% duty cycle
Power Mode 5 = 2.22% duty cycle
Power Mode 6 = 1.00% duty cycle
```

There were several issues with this interface and implementation. First, setting up a low power network was cumbersome. The low power listening commands had to be directly wired through CC1000RadioIntM, and called while the radio was not performing any transactions. Second, each node in a network was expected to have the same radio power mode. Finally, the pre-programmed duty cycles were not linear and offered a very limited selection of options.

In this low power listening implementation, the transmitter mote would transmit a packet that consisted of an extremely long preamble. This preamble was long enough to span a complete receive check period. On the receiver?s end, the radio would turn on and read bits from the radio. If a preamble sequence was detected in the incoming bits, the receiver?s radio would remain on for the full duration of the transmitter?s preamble and wait for the packet at the end.

This original low power listening scheme was rather inefficient on both the transmit and receive end. On the receive end, turning on the radio completely and reading in bits typically cost much more energy than necessary. The transmitter's long preamble could end up costing both nodes to have their radios on much longer than required, sending and receiving useless preamble bits.

2.2 CC1000 Pulse Check Implementation

Joe Polastre and Jason Hill developed a better receive check implementation in the CC1000 ?Pulse Check? radio stack for TinyOS 1.x, while maintaining the same interface. This implementation took advantage of a Clear Channel Assessment (CCA) to determine if a transmitter was nearby.

In this implementation, the CC1000 radio did not have to be turned on completely, so it consumed less maximum current than the previous implementation. The radio on-time was also significantly reduced, only turning on long enough for a single ADC

conversion to occur. If energy was detected on the channel after the first ADC conversion, subsequent ADC conversions would verify this before committing to turning the radio receiver on completely.

In this implementation the receiver's efficiency dramatically improved, but the transmitter still sent a long, inefficient preamble. Energy consumption used to transmit messages was still high, while throughput was still low.

2.3 Possible Improvements

Low power listening is a struggle between minimizing energy efficiency and maximizing throughput. In an asynchronous low power listening scheme, several improvements can be made over earlier implementations. One improvement that could have been made to earlier implementations is to remove the long transmitted preamble and send many smaller messages instead. For example, the transmitter could send the same message over and over again for the duration of the receiver's receive check period. The receiver could wake up and see that another node is transmitting, receive a full message, and finally send back an acknowledgement for that message. The transmitter would see the acknowledgement and stop transmitting early, so both nodes can perform some high speed transaction or go back to sleep. Useless preamble bits are minimized while useful packet information is maximized. Incidentally, this is a good strategy for CC2420 low power listening. This strategy certainly improves energy efficiency and throughput, but further improvements may be possible by employing a synchronous delivery method on top of this type of asynchronous low power listening scheme.

Improvements can also be made to the original low power listening interfaces. For example, instead of pre-programming power modes and duty cycles, a low power listening interface should allow the developer the flexibility to deploy a network of nodes with whatever duty cycle percentage or sleep time desired for each individual node. Nodes with different receive check periods should still have the ability to reliably communicate with each other with little difficulty.

3. Interfaces

3.1 Low Power Listening Interface

The LowPowerListening interface MUST be provided for each radio by the platform independent ActiveMessageC configuration.

In some implementations, low power listening may have an option to compile into the radio stack for memory footprint reasons. If low power listening is not compiled in with the stack, calls to LowPowerListening MUST be handled by a dummy implementation.

The TEP proposes this LowPowerListening interface::

```
interface LowPowerListening {
  command void setLocalSleepInterval(uint16_t sleepIntervalMs);
  command uint16_t getLocalSleepInterval();
  command void setLocalDutyCycle(uint16_t dutyCycle);
  command uint16_t getLocalDutyCycle();
  command void setRxSleepInterval(message_t *msg, uint16_t sleepIntervalMs)
  command uint16_t getRxSleepInterval(message_t *msg);
```

```
    command void setRxDutyCycle(message_t *msg, uint16_t dutyCycle);
    command uint16_t getRxDutyCycle(message_t *msg);
    command uint16_t dutyCycleToSleepInterval(uint16_t dutyCycle);
    command uint16_t sleepIntervalToDutyCycle(uint16_t sleepInterval);
  }
```

setLocalSleepInterval(uint16_t sleepIntervalMs)

- Sets the local node?s radio sleep interval, in milliseconds.

getLocalSleepInterval()

- Retrieves the local node?s sleep interval, in milliseconds. If duty cycle percentage was originally set, it is automatically converted to a sleep interval.

setLocalDutyCycle(uint16_t dutyCycle)

- Set the local node?s duty cycle percentage, in units of [percentage*100].

getLocalDutyCycle()

- Retrieves the local node?s duty cycle percentage. If sleep interval in milliseconds was originally set, it is automatically converted to a duty cycle percentage.

setRxSleepInterval(message_t *msg, uint16_t sleepIntervalMs)

- The given message will soon be sent to a low power receiver. The sleepIntervalMs is the sleep interval of that low power receiver, in milliseconds. When sent, the radio stack will automatically transmit the message so as to be detected by the low power receiver.

getRxSleepInterval(message_t *msg)

- Retrieves the message destination?s sleep interval. If a duty cycle was originally set for the outgoing message, it is automatically converted to a sleep interval.

setRxDutyCycle(message_t *msg, uint16_t dutyCycle)

- The given message will soon be sent to a low power receiver. The dutyCycle is the duty cycle percentage, in units of [percentage*100], of that low power receiver. When sent, the radio stack will automatically transmit the message so as to be detected by the low power receiver.

getRxDutyCycle(message_t *msg)

- Retrieves the message destination?s duty cycle percentage. If a sleep interval was originally set for the outgoing message, it is automatically converted to a duty cycle percentage.

dutyCycleToSleepInterval(uint16_t dutyCycle)

- Converts the given duty cycle percentage to a sleep interval in milliseconds.

sleepIntervalToDutyCycle(uint16_t sleepInterval)

- Converts the given sleep interval in milliseconds to a duty cycle percentage.

3.2 Split Control Behaviour

Low power listening MUST be enabled and disabled through the radio?s standard SplitControl interface, returning exactly one SplitControl event upon completion. While the radio is duty cycling, it MUST NOT alert the application layer each time the radio turns on and off to perform a receive check or send a message.

3.3 Send Interface Behaviour

Attempts to send a message before SplitControl.start() has been called SHOULD return EOFF, signifying the radio has not been enabled. When SplitControl.start() has been called by the application layer, calls to Send MUST turn the radio on automatically if the radio is currently off due to duty cycling. If a message is already in the process of being sent, multiple calls to Send should return FAIL.

The Send.sendDone(?) event SHOULD signal SUCCESS upon the successful completion of the message delivery process, regardless if any mote actually received the message.

3.4 Receive Interface Behaviour

Upon the successful reception of a message, the low power receive event handler SHOULD drop duplicate messages sent to the broadcast address. For example, the CC2420 implementation can perform this by checking the message_t?s dsn value, where each dsn value is identical for every message used in the delivery.

After the first successful message reception, the receiver?s radio SHOULD stay on for a brief period of time to allow any further transactions to occur at high speed. If no subsequent messages are detected going inbound or outbound after some short delay, the radio MUST continue duty cycling as configured.

4. Low Power Listening message_t Metadata

To store the extra 16-bit receiver low power listening value, the radio stack?s message_t footer MUST contain a parameter to store the message destination?s receive check sleep interval in milliseconds or duty cycle percentage. For example, the low power listening CC2420 message_t footer stores the message's receive check interval in milliseconds, as shown below [TEP111].:

```
typedef nx_struct cc2420_metadata_t {
  nx_uint8_t tx_power;
  nx_uint8_t rssi;
  nx_uint8_t lqi;
  nx_bool crc;
  nx_bool ack;
  nx_uint16_t time;
  nx_uint16_t rxInterval;
} cc2420_metadata_t;
```

5. Recommendations for HAL Implementation

In the interest of minimizing energy while maximizing throughput, it is RECOMMENDED that any asynchronous low power listening implementation use clear channel assessment methods to determine the presence of a nearby transmitter. It is also RECOMMENDED that the transmitter send duplicate messages continuously with minimum or no backoff period instead of one long message. Removing backoffs on a continuous send delivery scheme will allow the channel to be modulated sufficiently for a receiver to quickly detect; furthermore, enabling acknowledgements on each outgoing duplicate packet will allow the transmit period to be cut short based on when the receiver actually receives the message.

Asynchronous low power listening requires some memory overhead, so sometimes it is better to leave the added architecture out when it is not required. When it is feasible to do so, it is RECOMMENDED that the preprocessor variable LOW_POWER_LISTENING be defined when low power listening functionality is to be compiled in with the radio stack, and not defined when low power listening functionality shouldn?t exist.

It is RECOMMENDED that the radio on-time for actual receive checks be a measured value to help approximate the duty cycle percentage.

6. Author's Address

David Moss
Rincon Research Corporation
101 N. Wilmot, Suite 101
Tucson, AZ 85750

phone - +1 520 519 3138
email ? dmm@rincon.com

Jonathan Hui
657 Mission St. Ste. 600
Arched Rock Corporation
San Francisco, CA 94105-4120

phone - +1 415 692 0828
email - jhui@archedrock.com

Kevin Klues
503 Bryan Hall
Washington University
St. Louis, MO 63130

phone - +1-314-935-6355
email - klueska@cs.wustl.edu

7. Citations

[MICA2] “MICA2 Radio Stack for TinyOS.” http://www.tinyos.net/tinyos-1.x/doc/mica2radio/CC1000.html

[TEP111] TEP 111: message_t.

[CC1000] TI/Chipcon CC1000 Datasheet. http://www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf

[CC2420] TI/Chipcon CC2420 Datasheet. http://www.chipcon.com/files/CC2420_Data_Sheet_1_3.pdf