CC2420 Radio Stack

TEP:	126
Group:	Core Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	David Moss, Jonathan Hui, Philip Levis, and Jung Il Choi
Draft-Created:	5-Mar-2007
Draft-Version:	1.5
Draft-Modified:	2007-06-14
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

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 documents the architecture of the CC2420 low power listening radio stack found in TinyOS 2.x. Radio stack layers and implementation details of the CC2420 stack will be discussed. Readers will be better informed about existing features, possible improvements, and limitations of the CC2420 radio stack. Furthermore, lessons learned from the construction of the stack can help guide development of future TinyOS radio stacks.

1. Introduction

The TI/Chipcon CC2420 radio is a complex device, taking care of many of the low-level details of transmitting and receiving packets through hardware. Specifying the proper behavior of that hardware requires a well defined radio stack implementation. Although much of the functionality is available within the radio chip itself, there are still many factors to consider when implementing a flexible, general radio stack.

The software radio stack that drives the CC2420 radio consists of many layers that sit between the application and hardware. The highest levels of the radio stack modify data and headers in each packet, while the lowest levels determine the actual send

and receive behavior. By understanding the functionality at each layer of the stack, as well as the architecture of a layer itself, it is possible to easily extend or condense the CC2420 radio stack to meet project requirements.

Some details about the CC2420 are out of the scope of this document. These details can be found in the CC2420 datasheet[1].

2. CC2420 Radio Stack Layers

2.1 Layer Architecture

The CC2420 radio stack consists of layers of functionality stacked on top of each other to provide a complete mechanism that modifies, filters, transmits, and controls inbound and outbound messages. Each layer is a distinct module that can provide and use three sets of interfaces in relation to the actual radio stack: Send, Receive, and SplitControl. If a general layer provides one of those interfaces, it also uses that interface from the layer below it in the stack. This allows any given layer to be inserted anywhere in the stack through independant wiring. For example::

```
provides interface Send;
uses interface Send as SubSend;

provides interface Receive;
uses interface Receive as SubReceive;

provides interface SplitControl;
uses interface SplitControl as subControl;
```

The actual wiring of the CC2420 radio stack is done at the highest level of the stack, inside CC2420ActiveMessageC. This configuration defines three branches: Send, Receive, and SplitControl. Note that not all layers need to provide and use all three Send, Receive, and SplitControl interfaces::

```
// SplitControl Layers
SplitControl = LplC;
LplC.SubControl -> CsmaC;

// Send Layers
AM.SubSend -> UniqueSendC;
UniqueSendC.SubSend -> LinkC;
LinkC.SubSend -> LplC.Send;
LplC.SubSend -> TinyosNetworkC.Send;
TinyosNetworkC.SubSend -> CsmaC;

// Receive Layers
AM.SubReceive -> LplC;
LplC.SubReceive -> UniqueReceiveC.Receive;
UniqueReceiveC.SubReceive -> TinyosNetworkC.Receive;
TinyosNetworkC.SubReceive -> CsmaC;
```

If another layer were to be added, CC2420ActiveMessageC would need to be modified to wire it into the correct location.

2.1 Layer Descriptions

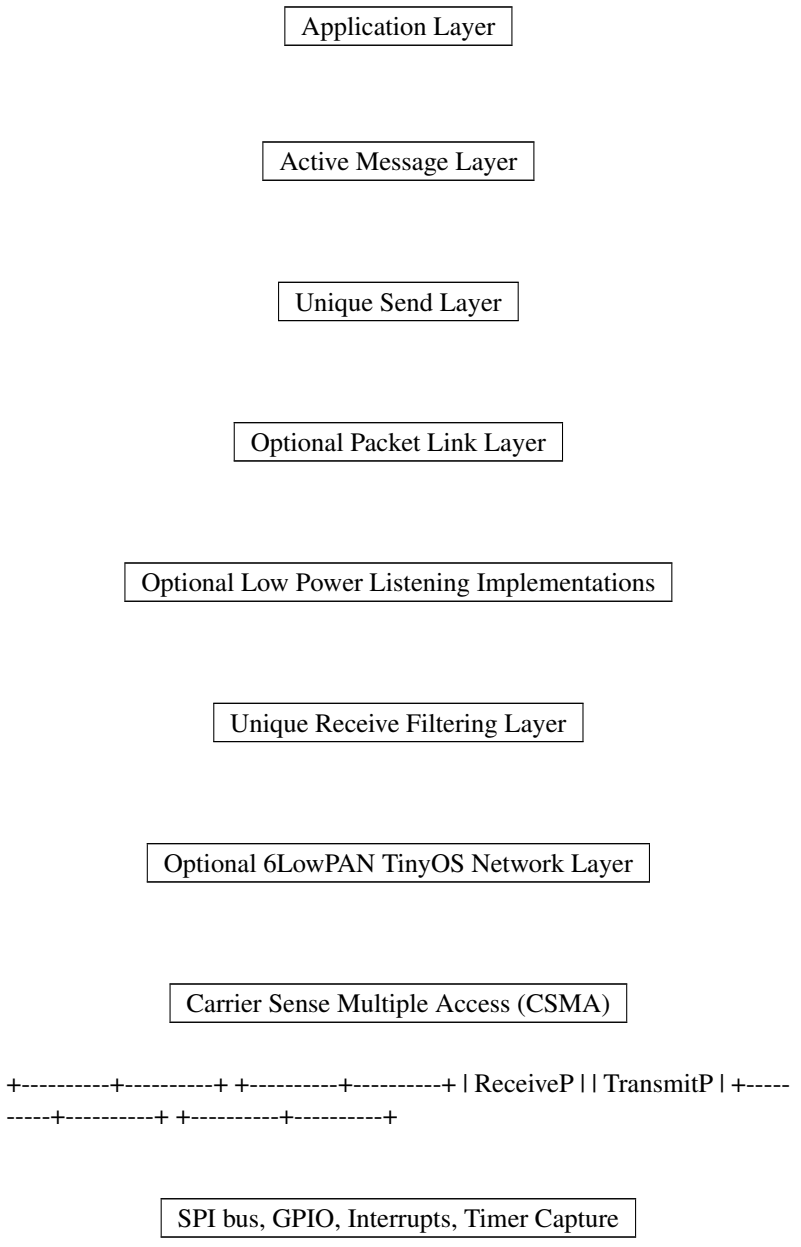
The layers found within this radio stack are in the following order:

- ActiveMessageP: This is the highest layer in the stack, responsible for filling in details in the packet header and providing information about the packet to the application level[2]. Because the CC2420 radio chip itself uses 802.15.4 headers in hardware[1], it is not possible for the layer to rearrange header bytes.
- UniqueSend: This layer generates a unique Data Sequence Number (DSN) byte for the packet header. This byte is incremented once per outgoing packet, starting with a pseudo-randomly generated number. A receiver can detect duplicate packets by comparing the source and DSN byte of a received packet with previous packets. DSN is defined in the 802.15.4 specification[3].
- PacketLink: This layer provides automatic retransmission functionality and is responsible for retrying a packet transmission if no acknowledgement was heard from the receiver. PacketLink is activated on a per-message basis, meaning the outgoing packet will not use PacketLink unless it is configured ahead of time to do so. PacketLink is most reliable when software acknowledgements are enabled as opposed to hardware auto acknowledgements.
- CC2420AckLplP / CC2420NoAckLplP[4]: These layers provide asynchronous low power listening implementations. Supporting both of them is CC2420DutyCycleP. The DutyCycleP component is responsible for turning the radio on and off and performing receive checks. After a detection occurs, DutyCycleP is hands off responsibility to LowPowerListeningP to perform some transaction and turn the radio off when convenient. Low power listening transmissions are activated on a per-message basis, and the layer will continuously retransmit the full outbound packet until either a response from the receiver is heard or the transmit time expires.

  The AckLplP implementation supports acknowledgement gaps during the low power listening packetized preamble, which allows transmitters to stop early but penalizes receive check lengths. AckLplP low power listening is optimal for high transmission, long receive check interval networks.

  The NoAckLplP implementation does not support acknowledgements during the packetized preamble. It continuously modulates the channel, allowing the receiver to perform the smallest possible receive check. NoAckLpl low power listening is effective for low transmission, short receive check interval networks.
- UniqueReceive: This layer maintains a history of the source address and DSN byte of the past few packets it has received, and helps filter out duplicate received packets.
- TinyosNetworkC: This layer allows the TinyOS 2.x radio stack to interoperate with other non-TinyOS networks. Proposed 6LowPAN specifications include a network identifier byte after the standard 802.15.4 header[5]. If interoperability frames are used, the dispatch layer provides functionality for setting the network byte on outgoing packets and filtering non-TinyOS incoming packets.
- CsmaC: This layer is responsible for defining 802.15.4 FCF byte information in the outbound packet, providing default backoff times when the radio detects a channel in use, and defining the power-up/power-down procedure for the radio.

- TransmitP/ReceiveP: These layers are responsible for interacting directly with the radio through the SPI bus, interrupts, and GPIO lines.



3. CC2420 Packet Format and Specifications

The CC2420 Packet structure is defined in CC2420.h. The default I-Frame CC2420 header takes on the following format::

```
typedef nx_struct cc2420_header_t {
  nxle_uint8_t length;
  nxle_uint16_t fcf;
  nxle_uint8_t dsn;
  nxle_uint16_t destpan;
  nxle_uint16_t dest;
  nxle_uint16_t src;
  nxle_uint8_t network;  // optionally included with 6LowPAN layer
  nxle_uint8_t type;
} cc2420_header_t;
```

All fields up to 'network' are 802.15.4 specified fields, and are used in the CC2420 hardware itself. The 'network' field is a 6LowPAN interoperability specification[5] only to be included when the 6LowPAN TinyosNetwork layer is included. The 'type' field is a TinyOS specific field.

The TinyOS T-Frame packet does not include the 'network' field, nor the functionality found in the Dispatch layer to set and check the 'network' field.

No software footer is defined for the CC2420 radio. A 2-byte CRC byte is auto-appended to each outbound packet by the CC2420 radio hardware itself.

The maximum size of a packet is 128 bytes including its headers and CRC, which matches the 802.15.4 specifications. Increasing the packet size will increase data throughput and RAM consumption in the TinyOS application, but will also increase the probability that interference will cause the packet to be destroyed and need to be retransmitted. The TOSH_DATA_LENGTH preprocessor variable can be altered to increase the size of the message_t payload at compile time[2].

4. CSMA/CA

4.1 Clear Channel Assessment

By default, the CC2420 radio stack performs a clear channel assessment (CCA) before transmitting. If the channel is not clear, the radio backs off for some short, random period of time before attempting to transmit again. The CC2420 chip itself provides a strobe command to transmit the packet if the channel is currently clear.

To specify whether or not to transmit with clear channel assessment, the CC2420TransmitP requests CCA backoff input through the RadioBackoff interface on a per-message basis. By default, each packet will be transmitted with CCA enabled.

If layers above the CSMA layer wish to disable the clear channel assessments before transmission, they must intercept the RadioBackoff.requestCca(...) event for that message and call back using RadioBackoff.setCca(FALSE).

4.2 Radio Backoff

A backoff is a period of time where the radio pauses before attempting to transmit. When the radio needs to backoff, it can choose one of three backoff periods: initialBackoff, congestionBackoff, and lplBackoff. These are implemented through the RadioBackoff interface, which signals out a request to specify the backoff period. Unlike the CsmaBackoff interface, components that are interested in adjusting the backoff can call back using commands in the RadioBackoff interface. This allows multiple components to adjust the backoff period for packets they are specifically listening to adjust.

The lower the backoff period, the faster the transmission, but the more likely the transmitter is to hog the channel. Also, backoff periods should be as random as possible to prevent two transmitters from sampling the channel at the same moment.

InitialBackoff is the shortest backoff period, requested on the first attempt to transmit a packet.

CongestionBackoff is a longer backoff period used when the channel is found to be in use. By using a longer backoff period in this case, the transmitter is less likely to unfairly tie up the channel.

LplBackoff is the backoff period used for a packet being delivered with low power listening. Because low power listening requires the channel to be modulated as continuously as possible while avoiding interference with other transmitters, the low power listening backoff period is intentionally short.

5. Acknowledgements

5.1 Hardware vs. Software Acknowledgements

Originally, the CC2420 radio stack only used hardware generated auto-acknowledgements provided by the CC2420 chip itself. This led to some issues, such as false acknowledgements where the radio chip would receive a packet and acknowledge its reception and the microcontroller would never actually receive the packet.

The current CC2420 stack uses software acknowledgements, which have a higher drop percentage. When used with the UniqueSend and UniqueReceive interfaces, dropped acknowledgements are more desirable than false acknowledgements. Received packets are always acknowledged before being filtered as a duplicate.

Use the PacketAcknowledgements or PacketLink interfaces to determine if a packet was successfully acknowledged.

5.2 Data Sequence Numbers - UniqueSend and UniqueReceive

The 802.15.4 specification identifies a Data Sequence Number (DSN) byte in the message header to filter out duplicate packets[3].

The UniqueSend interface at the top of the CC2420 radio stack is responsible for setting the DSN byte. Upon boot, an initial DSN byte is generated using a pseudo-random number generator with a maximum of 8-bits (256) values. This number is incremented on each outgoing packet transmission. Even if lower levels such as PacketLink or LowPowerListening retransmit the packet, the DSN byte stays the same for that packet.

The UniqueReceive interface at the bottom of the CC2420 radio stack is responsible for filtering out duplicate messages based on source address and DSN. The UniqueReceive interface is not meant to stop sophisticated replay attacks. ’

As packets are received, DSN and source address information is placed into elements of an array. Each subsequent message from previously logged addresses overwrite information in the element allocated to that source address. This prevents UniqueReceive’s history from losing DSN byte information from sources that are not able to access the channel as often. If the number of elements in the history array runs out, UniqueReceive uses a best effort method to avoid replacing recently updated DSN/Source information entries.

6. PacketLink Implementation

PacketLink is a layer added to the CC2420 radio stack to help unicast packets get delivered successfully. In previous version of TinyOS radio stacks, it was left up to the application layer to retry a message transmission if the application determined the message was not properly received. The PacketLink layer helps to remove the reliable delivery responsibility and functional baggage from application layers.

6.1 Compiling in the PacketLink layer

Because the PacketLink layer uses up extra memory footprint, it is not compiled in by default. Developers can simply define the preprocessor variable PACKET_LINK to compile the functionality of the PacketLink layer in with the CC2420 stack.

6.2 Implementation and Usage

To send a message using PacketLink, the PacketLink interface must be called ahead of time to specify two fields in the outbound message's metadata::

```
command void setRetries(message_t *msg, uint16_t maxRetries);
command void setRetryDelay(message_t *msg, uint16_t retryDelay);
```

The first command, setRetries(..), will specify the maximum number of times the message should be sent before the radio stack stops transmission. The second command, setRetryDelay(..), specifies the amount of delay in milliseconds between each retry. The combination of these two commands can set a packet to retry as many times as needed for as long as necessary.

Because PacketLink relies on acknowledgements, false acknowledgements from the receiver will cause PacketLink to fail. If using software acknowledgements, false acknowledgements can still occur as a result of the limited DSN space, other 802.15.4 radios in the area with the same destination address, etc.

7. Asynchronous Low Power Listening Implementation

Because the Low Power Listening layer uses up extra memory footprint, it is not compiled in by default. Developers can simply define the preprocessor variable LOW_POWER_LISTENING to compile the functionality of the Low Power Listening layer in with the CC2420 stack.

7.1 Design Considerations

The CC2420 radio stack low power listening implementation relies on clear channel assessments to determine if there is a transmitter nearby. This allows the receiver to turn on and determine there are no transmitters in a shorter amount of time than leaving the radio on long enough to pick up a full packet.

The transmitters perform a message delivery by transmitting the full packet over and over again for twice the duration of the receiver's duty cycle period. Transmitting for twice as long increases the probability that the message will be detected by the

receiver, and allows the receiver to shave off a small amount of time it needs to keep its radio on.

Typically, the transmission of a single packet takes on the following form over time:

LPL Backoff	Packet Tx	Ack Wait

To decrease the amount of time required for a receive check, the channel must be modulated by the transmitter as continuously as possible. The only period where the channel is modulated is during the Packet Transmission phase. The receiver must continuosly sample the CCA pin a moment longer than the LPL Backoff period and Ack Wait period combined to overlap the Packet Transmission period. By making the LPL backoff period as short as possible, we can decrease the amount of time a receiver's radio must be turned on when performing a receive check.

If two transmitters attempt to transmit using low power listening, one transmitter may hog the channel if its LPL backoff period is set too short. Both nodes transmitting at the same time will cause interference and prevent each other from successfully delivering their messages to the intended recipient.

To allow multiple transmitters to transmit low power listening packets at the same time, the LPL backoff period needed to be increased greater than the desired minimum. This increases the amount of time receiver radios need to be on to perform a receive check because the channel is no longer being modulated as continuously as possible. In other words, the channel is allowed to be shared amongst multiple transmitters at the expense of power consumption.

7.2 Minimizing Power Consumption

There are several methods the CC2420 radio stack uses to minimize power consumption:

1. Invalid Packet Shutdown

   Typically, packets are filtered out by address at the radio hardware level. When a receiver wakes up and does not receive any packets into the low power listening layer of the radio stack, it will automatically go back to sleep after some period of time. As a secondary backup, if address decoding on the radio chip is disabled, the low power listening implementation will shut down the radio if three packets are receive that do not belong to the node. This helps prevent against denial of sleep attacks or the typical transmission behavior found in an ad-hoc network with many nodes.

2. Early Transmission Completion

   A transmitter typically sends a packet for twice the amount of time as the receiver's receive check period. This increases the probability that the receiver will detect the packet. However, if the transmitter receives an acknowledgement before the end of its transmission period, it will stop transmitting to save energy. This is an improvement over previous low power listening implementations, which transmitted for the full period of time regardless of whether the receiver has already woken up and received the packet.

3. Auto Shutdown

   If the radio does not send or receive messages for some period of time while low power listening is enabled, the radio will automatically turn off and begin duty cycling at its specified duty cycle period.

4. CCA Sampling Strategy

   The actual receive check is performed in a loop inside a function, not a spinning task. This allows the sampling to be performed continuously, with the goal of turning the radio off as quickly as possible without interruption.

8. CC2420 Settings and Registers

To interact with registers on the CC2420 chip, the SPI bus must be acquired, the chip selecct (CSn) pin must be cleared, and then the interaction may occur. After the interaction completes, the CSn pin must be set high.

All registers and strobes are defined in the CC2420.h file, and most are accessible through the CC2420SpiC component. If your application requires access to a specific register or strobe, the CC2420SpiC component is the place to add access to it.

Configuring the CC2420 requires the developer to access the CC2420Config interface provided by CC2420ControlC. First call the CC2420Config commands to change the desired settings of the radio. If the radio happens to be off, the changes will be committed at the time it is turned on. Alternatively, calling sync() will commit the changes to the CC2420.

RSSI can be sampled directly by calling the ReadRssi interface provided by CC2420ControlC. See page 50 of the CC2420 datasheet for information on how to convert RSSI to LQI and why it may not be such a good idea[1].

9. Cross-platform Portability

To port the CC2420 radio to another platform, the following interfaces need to be implemented::

```
// GPIO Pins
interface GeneralIO as CCA;
interface GeneralIO as CSN;
interface GeneralIO as FIFO;
interface GeneralIO as FIFOP;
interface GeneralIO as RSTN;
interface GeneralIO as SFD;
interface GeneralIO as VREN;

// SPI Bus
interface Resource;
interface SpiByte;
interface SpiPacket;
```

```
// Interrupts
interface GpioCapture as CaptureSFD;
interface GpioInterrupt as InterruptCCA;
interface GpioInterrupt as InterruptFIFOP;
```

The GpioCapture interface is tied through the Timer to provide a relative time at which the interrupt occurred. This is useful for timestamping received packets for node synchronization.

If the CC2420 is not connected to the proper interrupt lines, interrupts can be emulated through the use of a spinning task that polls the GPIO pin. The MICAz implementation, for example, does this for the CCA interrupt.

10. Future Improvement Recommendations

Many improvements can be made to the CC2420 stack. Below are some recommendations:

10.1 AES Encryption

The CC2420 chip itself provides AES-128 encryption. The implementation involves loading the security RAM buffers on the CC2420 with the information to be encrypted - this would be the payload of a packet, without the header. After the payload is encrypted, the microcontroller reads out of the security RAM buffer and concatenates the data with the unencrypted packet header. This full packet would be uploaded again to the CC2420 TXFIFO buffer and transmitted.

Because the CC2420 cannot begin encryption at a particular offset and needs to be written, read, and re-written, use of the AES-128 may be inefficient and will certainly decrease throughput.

10.2 Authentication

In many cases, authentication is more desirable than encryption. Encryption significantly increases energy and decreases packet throughput, which does not meet some application requirements. A layer could be developed and added toward the bottom of the radio stack that validates neighbors, preventing packets from invalid neighbors from reaching the application layer. Several proprietary authentication layers have been developed for the CC2420 stack, but so far none are available to the general public.

A solid authentication layer would most likely involve the use of a neighbor table and 32-bit frame counts to prevent against replay attacks. Once a neighbor is verified and established, the node needs to ensure that future packets are still coming from the same trusted source. Again, some high speed low energy proprietary methods to accomplish this exist, but encryption is typically the standard method used.

10.3 Synchronous Low Power Listening

A synchronous low power listening layer can be transparently built on top of the asynchronous low power listening layer. One implementation of this has already been done on a version of the CC1000 radio stack. Moteiv's Boomerang radio stack also has a synchronous low power listening layer built as a standalone solution.

In the case of building a synchronous layer on top of the asynchronous low power listening layer, a transmitter's radio stack can detect when a particular receiver is performing its receive checks by verifying the packet was acknowledged after a sendDone event. The transmitter can then build a table to know when to begin transmission for that particular receiver. Each successful transmission would need to adjust the table with updated information to avoid clock skew problems.

The asynchronous low power listening stack needs to be altered a bit to make this strategy successful. Currently, duty cycling is started and stopped as packets are detected, received, and transmitted. The stack would need to be altered to keep a constant clock running in the background that determines when to perform receive checks. The clock should not be affected by normal radio stack Rx/Tx behavior. This would allow the receiver to maintain a predictable receive check cycle for the transmitter to follow.

If the synchronous low power listening layer loses synchronization, the radio stack can always fall back on the asynchronous low power listening layer for successful message delivery.

10.4 Neighbor Tables

Moteiv's Boomerange Sensornet Protocol (SP) implementation is a good model to follow for radio stack architecture. One of the nice features of SP is the design and implementation of the neighbor table. By providing and sharing neighbor table information across the entire CC2420 radio stack, RAM can be conserved throughout the radio stack and TinyOS applications.

10.5 Radio Independant Layers

The best radio stack architecture is one that is completely radio independant. Many of the layers in the CC2420 stack can be implemented independant of the hardware underneath if the radio stack architecture was redesigned and reimplemented. The low power listening receive check strategy may need a hardware-dependant implementation, but other layers like PacketLink, UniqueSend, UniqueReceive, ActiveMessage, Dispatch, etc. do not require a CC2420 underneath to operate properly. The ultimate TinyOS radio stack would be one that forms an abstraction between radio-dependant layers and radio-independant layers, and operates with the same behavior across any radio chip.

10.6 Extendable Metadata

Layers added into the radio stack may require extra bytes of metadata. The PacketLink layer, for example, requires two extra fields in each message's metadata to hold the message's max retries and delay between retries. The low power listening layer requires an extra field to specify the destination's duty cycle period for a proper delivery.

If layers are not included in the radio stack during compile time, their fields should not be included in the message_t's metadata.

One version of extendable metadata was implementing using an array at the end of the metadata struct that would adjust its size based on which layers were compiled in and what size fields they required. A combination of compiler support in the form of unique(..) and uniqueCount(..) functions made it possible for the array to adjust its size.

Explicit compiler support would be the most desirable solution to add fields to a struct as they are needed.

10.7 Error Correcting Codes (ECC)

When two nodes are communicating near the edge of their RF range, it has been observed that interference may cause the packet to be corrupted enough that the CRC byte and payload actually passes the check, even though the payload is not valid. There is a one in 65535 chance of a CRC byte passing the check for a corrupted packet. Although this is slim, in many cases it is unacceptable. Some work arounds have implemented an extra byte of software generated CRC to add to the reliability, and tests have proven its effectiveness. Taking this a step further, an ECC layer in the radio stack would help correct corrupted payloads and increase the distance at which nodes can reliably communicate.

11. 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

Philip Levis
358 Gates Hall
Stanford University
Stanford, CA 94305-9030

phone - +1 650 725 9046
email - pal@cs.stanford.edu

Jung Il Choi

<contact>
phone -
email -

12. Citations

[1] TI/Chipcon CC2420 Datasheet. http://www.chipcon.com/files/CC2420_Data_Sheet_1_3.pdf
[2] TEP111: message_t
[3] IEEE 802.15.4 Specification: http://standards.ieee.org/getieee802/802.15.html
[4] TEP105: Low Power Listening
[5] TEP125: TinyOS 802.15.4 Frames