CCxx00 Radio Stack

David Moss
Rincon Research Corporation
Tucson, Arizona
mossmoss at gmail dot com

September 3, 2009

Contents

1 Introduction

This document describes the architecture, options, and interfaces integrated into the CC1100 / CC1101 / CC2500 (collectively CCxx00) open-source TinyOS radio stack.

The flexibility of the CCxx00 architecture, combined with well tested and characterized code, makes it a highly desirable choice for reliable wireless sensor network deployments. This document will describe how to setup the TinyOS CCxx00 radio stack to meet your system's target behavior, and discuss the various interfaces you'll encounter in the stack and how to use them most effectively.

The CCxx00 open-source radio stack [5], designed for seamless integration into the TinyOS open-source operating system [1], supports many options and add-on libraries that are useful for commercial and research ad-hoc wireless network deployments. The architecture is very modular, allowing the developer to select options, layers, and behavior that make sense for the target deployment. Developers can choose options such as single radio platforms, multiple radio platforms, low power duty cycling communications, reliable links, acknowledgements, CSMA based channel sharing, duplicate packet suppression, and more. Complex forms of wireless networking can be implemented on top of the radio stack, ranging from simple point-to-point communication to full mesh networks.

2 Structure

The radio stack was designed to be very modular, allowing developers to drop in or select new layers of functionality. If multiple implementations exist for a given directory, it is the developer's job to modify how their platform compiles (using the .platform file perhaps) to select the desired implementation. At a minimum, the compiler must know the location of each main directory in the CCxx00 tree.

2.1 Single Radio Components

There are many components of functionality, some with more than one behavior implementation:

- *am* : Active Message layer. Configures the header, defines the payload, filters packets based on packet type.
- *alarm* : 32 kHz alarm supporting Acks, CSMA, and fast timing.
- *acks* : Acknowledgment layer
  - *DynamicAcks* : Dynamically turn acks on or off for each outgoing packet.
  - *StaticAcks* : Statically keeps acks turned on for every packet, saves memory.
- *crc* : Adds and verifies 32-bit software CRC's at the end of packets.
- *csma* : Carrier Sense Multiple Access / Collision Avoidance

  - *singlesense* : Please see section 5.4.1.
  - *continuoussense* : Please see section 5.4.2.
- *init* : Low-level radio hardware initialization code.
- *interfaces* : All of the CCxx00-specific radio stack interfaces.
- *link* : Packet Link Layer, helps with reliable node-to-node packet transmissions.
- *lpl* : Low power communications implementations, discussed in greater detail in section 5.5.
  - *alwayson* : The radio is always on. This uses the least amount of program memory and has the fastest throughput, but uses the most amount of energy.
  - *bmac* : BMAC [9] low power implementation.
  - *boxmac* : BoX-MAC [8] low power implementation, combining two other layers in the radio stack: BMAC and Packet Link. Use of this implementation implies you also have the BMAC directory compiled in.
  - *enddevice* : This is an end-device implementation that causes the radio to turn on only when a packet is being sent, and turn off when the packet is done sending. This is very efficient on memory footprint, and very efficient on energy consumption. It can be combined with an external add-on layer, *Pending Bit*, to allow your local node to remain active for a period of time after receiving an ack with the *Pending Bit* set.
- *powermanager* : Gracefully enables or disables critical communication layers in the stack where the radio may be in the process of sending or receiving.
- *radios* : Radio hardware configuration. The first CCxx00.h header file that is referenced will become the default radio for the platform.
  - *CC1100* : CC1100 / CC1101 configuration
  - *CC2500* : CC2500 configuration
- *receive* : Low-level packet reception module.
- *select* : Used for forwards-compatibility with multiple radio configurations.
- *spi* : SPI bus implementation, provides access to FIFO's, registers, and strobe commands.
- *splitcontrolmanager* : Manages the SplitControl interface, preventing the radio from turning off when a packet is being sent anywhere in the stack.
- *transmit* : Low-level transmit module. This provides separate interfaces for packet transmission and ack transmission, the difference is how aggressive it tries to transmit. It also physically creates the long preambles for BMAC / BoX-MAC low power communication strategies.
- *unique* : Unique packet reception, prevents duplicate packets from entering the stack. Recommend combining this with an unpublished security layer.

2.2 Single Radio .platform Configuration

Your platform's .platform file will tell the compiler which directories to pull in at compile time. The directories toward the top take precedence over the directories toward the bottom. If a similarly named module appears in more than one directory, the first instance of it is used at compile time, overriding deprecated files. This may be useful if you need to override files in the default CCxx00 stack directories with your own code. Alternatively, you can create a new directory with your own implementations and comment out the old with a #.

The %T at the beginning of every directory automatically references the $TOSDIR environment variable

A typical .platform file that references the ccxx00_single radio stack for the compiler will look like this:

```
%T/chips/ccxx00_single
%T/chips/ccxx00_single/alarm
%T/chips/ccxx00_single/am
%T/chips/ccxx00_single/am/queue
%T/chips/ccxx00_single/crc
%T/chips/ccxx00_single/init
%T/chips/ccxx00_single/interfaces
%T/chips/ccxx00_single/link
%T/chips/ccxx00_single/packet
%T/chips/ccxx00_single/powermanager
%T/chips/ccxx00_single/receive
%T/chips/ccxx00_single/select
%T/chips/ccxx00_single/spi
%T/chips/ccxx00_single/splitcontrolmanager
%T/chips/ccxx00_single/transmit
%T/chips/ccxx00_single/unique

/** Choose static acks to trim memory footprint
    and get an ack for every packet */
%T/chips/ccxx00_single/acks/staticacks
#%T/chips/ccxx00_single/acks/dynamicacks

/** Choose CC1100 / CC1101 radio */
%T/chips/ccxx00_single/radios/cc1100
#%T/chips/ccxx00_single/radios/cc2500

/** Choose continuous sense CSMA to support
    better ack rates and fairness */
%T/chips/ccxx00_single/csma/continuoussense
#%T/chips/ccxx00_single/csma/singlesense

/** Include BMAC to give the option of basic
```

```
    low power communications */
%T/chips/ccxx00_single/lpl
#%T/chips/ccxx00_single/lpl/alwayson
%T/chips/ccxx00_single/lpl/bmac
#%T/chips/ccxx00_single/lpl/boxmac
#%T/chips/ccxx00_single/lpl/enddevice
```

2.3 Multiple Radio Components

The CCxx00 radio stack supports multiple radios. You probably need to prevent radios from turning on simultaneously, because the crystals that power the radios interfere with each other and prevent transmission / reception without very robust filters preventing that interaction. The CCxx00 radio stack was designed to prevent multiple radios from turning on simultaneously.

- *init* : Overrides the CCxx00PlatformInit module, which puts multiple radios to sleep on Platform Init.
- *interfaces* : Provides a CentralWiring interface to support multiple radios.
- *select* : Implements the Radio Select layer, which allows the user to select which radio a packet should be sent through, and which radio is enabled.
- *spi* : BlazeCentralWiring to support switching between one radio's physical interfaces to the next.

2.4 Multiple Radio .platform Configuration

To enable multiple radios, modify your .platform file to include the ccxx00_multiple directories at the top, which will override files from the ccxx00_single directories.

```
%T/chips/ccxx00_multiple/
%T/chips/ccxx00_multiple/init
%T/chips/ccxx00_multiple/interfaces
%T/chips/ccxx00_multiple/select
%T/chips/ccxx00_multiple/spi
```

2.5 CCxx00 Add-ons

The radio stack was designed for flexibility and modularity. Some might even claim it is too flexible, but all of that flexibility goes a long way toward allowing developers to add or modify functionality in the radio stack to meet their system needs.

Of course, there is a balance to strike between flexibility and memory footprint, and our goal was to make the tightest radio stack that is reliable and provides the basic foundation for mesh networking functionality. By placing add-on functionality in a separate space than the basic radio stack, we can keep the program footprint to a minimum while giving the option of including more functionality.

At the time of this writing, the CCxx00 radio stack had the following add-ons available to developers:

- *availableradios* : Discovers which radios are connected to a platform during initialization. This taps directly into the CCxx00PlatformInit module to send a few more commands to the radios on board. The purpose was to verify platform functionality to prevent libraries from accessing non-existent radios on a modular radio platform.
- *dutycycle* : Keeps track of the amount of time the radio has been in some active state vs. how long the platform has been enabled. This is extremely useful for energy consumption estimates.
- *packetstats* : Keeps track of the number of packets sent, received, and overheard. As of the time of this writing, we have yet to implement another statistics interface to provide the number of missed acknowledgments.
- *pendingbit* : This layer watches out for a pending bit enabled in the FCF word of the packet and will activate the radio for a period of time if a pending bit is ever seen. This could be used in conjunction with any low power communication profile except always-on, where it wouldn't make sense because the radio is always active.
- *packetstats* : Keeps track of the number of packets sent, received, and overheard. As of the time of this writing, we have yet to implement another statistics interface to provide the number of missed acknowledgments.

2.6 Radio Add-ons .platform Configuration

To enable add-ons radios, two things must occur. First, you must modify your .platform file to include the ccxx00_addons directories. Second, you must reference the add-on configuration file(s) to pull it in at compile time. The normal coding strategy is to modify your platform's ActiveMessageC.nc configuration to pull in the add-ons and provide their interfaces out of the top of ActiveMessageC.

```
%T/chips/ccxx00_addons/availableradios
%T/chips/ccxx00_addons/dutycycle
%T/chips/ccxx00_addons/packetstats
%T/chips/ccxx00_addons/pendingbit
%T/chips/ccxx00_addons/rssi
```

2.7 Layer Wiring

Layers stack up to form the radio driver, hence the name Ŕadio Stack.ź Commands such as SplitControl, Send, and Receive propagate through each layer serially.

All wiring is done at the highest level BlazeC.nc configuration file. You can change the architecture of the radio stack by adding ́dummyṕass-through configurations for

ActiveMessageC
SplitControlManager
RadioSelect
UniqueReceive
PacketLink
Low Power Communications
Acknowledgments
CSMA
Transmit / Receive / Init
SPI Bus / Central Wiring
Hardware Presentation Layer
Platform Code
Hardware

Table 1: CCxx00 Radio Stack

any given layer, or hacking the BlazeC.nc file (which should be avoided to maintain backwards / forwards compatibility if possible).

The order of these layers is very important to forming a fully functional radio stack.

```
/***************** Send Layers ***************/
AMSend = BlazeActiveMessageC;
BlazeActiveMessageC.SubSend -> RadioSelectC.Send;
RadioSelectC.SubSend -> SplitControlManagerC.Send;
SplitControlManagerC.SubSend -> PacketLinkC.Send;
PacketLinkC.SubSend -> LplC.Send;
LplC.SubSend -> AcknowledgementsC.Send;
AcknowledgementsC.SubSend -> CsmaC;

/***************** Receive Layers ***************/
Receive = BlazeActiveMessageC.Receive;
Snoop = BlazeActiveMessageC.Snoop;
BlazeActiveMessageC.SubReceive -> RadioSelectC.Receive;
RadioSelectC.SubReceive -> UniqueReceiveC.Receive;
UniqueReceiveC.SubReceive -> LplC.Receive;
LplC.SubReceive -> BlazeReceiveC.Receive;

/***************** SplitControl Layers ***************/
SplitControl = RadioSelectC.SplitControl;
RadioSelectC.SubControl -> SplitControlManagerC.SplitControl;
SplitControlManagerC.SubControl -> LplC.SplitControl;
LplC.SubControl -> Ccxx00PowerManagerC.SplitControl;
```

3 Interfaces

There are many available interfaces to access in the CCxx00 stack, and most are located in the ccxx00_radio/interfaces directory. We will cover these interfaces, starting from the core interfaces to extended and advanced interfaces.

3.1 Basic Core Interfaces

These are the basic interfaces you need to turn on the radio, send, and receive packets.

3.1.1 SplitControl

The SplitControl interface is a split-phase call to activate or deactivate a radio. You must activate the radio before using it, and the radio is only considered ready for use when the SplitControl.startDone() event is signaled.

When active, the radio is expected to transmit or receive packets at any time. When your platform boots, a call to SplitControl.start() should never return FAIL. This can prevent you, the developer, from wasting precious code space trying to test for SUCCESS on SplitControl.start() at boot.

The desired behavior of SplitControl is documented in TEP 115 [4]. Some minor alterations to the implementation of this TEP may have been made in the SplitControlManager component of the radio stack to decrease compiled code size, since the user is generally only interested in SUCCESS or FAIL of enabling / disabling the radio stack.

When low power duty cycling is enabled with BMAC, BoX-MAC, or End Device, the physical radio may be off most of the time, but the radio can still send and receive packets.

3.1.2 AMSend

This is the basic Active Message Send interface, defined by the TinyOS community.

3.1.3 Receive

This is the basic Receive interface, defined by the TinyOS community. One modification from Receive's original design is the fact that the CCxx00 stack will not double-buffer packets at the lowest layer. The reason it was implemented this way is because the low level double buffering functionality has never been proven useful, and the inclusion of it wastes program ROM.

If you need to do double-buffering for any reason, memcpy the inbound packet's contents to your own array, post a task to do processing on it at a later time, and then return a message pointer as usual. Returning any pointer, including NULL, will not alter any behavior in the functioning radio stack.

3.2 Packet Interfaces

These interfaces tap directly into a given packet to retrieve information about the packet's header, payload, or footer.

3.2.1 AMPacket

This interface, defined by the TinyOS community, allows access to standard fields in the header of a packet. These fields include the source address, destination address, packet type, and group ID.

If you set any of these fields and send the packet, the fields will be automatically overwritten by the radio stack. In that sense, the AMPacket interface is more useful for obtaining these fields in a given packet. The AMPacket interface also returns the local node's software address with the address() command.

To set the node's address, use the ActiveMessageAddressC component, located in the tos/system directory in the TinyOS baseline. To make your node remember its software-configured address after a reboot event, you must store and load it from non-volatile memory. One easy option is to learn how to use the Configurator component provided by Rincon in the tinyos-2.x-contrib/rincon tree. This component mirrors small variables normally managed in RAM on your microcontroller's internal non-volatile memory.

3.2.2 Packet

The Packet interface, also defined by the TinyOS community, is mostly concerned with the payload of an inbound or outbound packet.

3.2.3 BlazePacket

This interface provides access to CCxx00-specific fields that may not be found in other radio stacks. This includes commands to look at LQI, RSSI, and the Packet Pending bit.

3.3 Extended Interfaces

3.3.1 PacketLink

Packet Link allows you to configure a packet to be retried for a set number of times before the radio stack gives up sending it. Acknowledgments are automatically set when Packet Link is enabled. In the case where a packet is received by the destination node, but the acknowledgment is dropped, the follow on duplicate packets will automatically be filtered out by the receiving radio stack's UniqueReceive filter.

It is possible to remove the Packet Link layer from the radio stack if that functionality is not going to be used.

See TEP 127 [7] for more details.

3.3.2 PacketAcknowledgements

Yes, this interface is misspelled in an American English sense, due to its history.

If Static Acknowledgments is used, this interface only serves to check the status of an acknowledgment.

If the Dynamic Acknowledgments implementation is used, you must explicitly call this interface to request an acknowledgment before the receiving node will send one back.

3.3.3 SendNotifier

The SendNotifier interface always generates a signal when the radio stack is about to send a packet. The interface is parameterized by AM ID out the top of the CCxx00 stack.

This is useful because it allows other areas of your system to modify outbound packets. For example, the Collection Tree Protocol connects directly into the radio stack and will not allow you to configure outbound packets to be delivered to low power receivers. Instead of modifying the CTP code to do the configuration there, you can add a module into the system that will look for outbound CTP packets and configure them to be delivered to a low power receiver.

If you do not implement the SendNotifier interface anywhere, it should be removed by the compiler which costs no code space.

3.3.4 AckSendNotifier

Same as the SendNotifier interface, but only pertains to acknowledgment frames that are about to be returned. This could, for example, be used to set the Packet Pending bit in an outbound acknowledgment. In the future, we may modify the Receive component to also give the option to prevent sending back an acknowledgment to unauthorized nodes.

3.3.5 LowPowerListening

The CCxx00 radio stack implements the latest LowPowerListening interface. This interface will configure the wake-up intervals for your local node, and configure individual outbound packets to be delivered to a duty cycling receiver. Please see TEP 105 [6] for details on the original implementation.

3.3.6 SystemLowPowerListening

This new interface requires support from components external to the radio stack. You can set the desired wake-up interval for all outbound packets. The external module that does not exist as part of the CCxx00 stack is supposed to provide an AMSend interface, which when you send a packet through it, will configure that packet with default low power communications settings.

3.4 Advanced Interfaces

The following extremely advanced interfaces are not normally used, but do allow for very detailed control over the operation of the radio stack.

3.4.1 Csma

Our CCxx00 radio stack implements CSMA/CA to share the channel fairly with other nearby nodes. When a packet enters a CSMA layer, this interface will signal a request to see if any part of the application even wants to use clear channel assessments to avoid packet collisions. In some point-to-point isolated cases, it may make sense to disable clear channel assessments. This will allow your node's transmissions to occur as fast as possible.

The CSMA layer, as well as the effects of its backoffs, are discussed in section 5.4.

3.4.2 Backoff as InitialBackoff

This interface, parameterized by AM ID, will allow your application to configure initial CSMA backoff settings for the current outbound packet. Initial backoffs occur for every outbound packet when CSMA is enabled. Modifying the duration of an initial backoff could be useful, for example, if the outbound packet is part of a large block of data that needs to be burst through to another point very quickly.

Modifying the backoff settings effectively changes the priority of a packet on the channel. Shorter backoffs equal higher priority.

3.4.3 Backoff as CongestionBackoff

Congestion backoffs occur if your node attempts to transmit, but finds the channel busy. You can modify the packet's priority on the channel by increasing or decreasing its backoff from the default values the radio stack selects.

3.5 Multiple Radio Interfaces

These interfaces are only used when the ccxx00_multiple behavior is compiled in.

3.5.1 SplitControl as BlazeSplitControl[radio_id_t radioId]

This SplitControl interface will allow the user to attempt to turn a different radio on or off. Remember, only one radio is allowed to be turned on at a time.

3.5.2 RadioSelect

This interface will allow the user to select which radio a packet is supposed to be sent through, and also informs the application to know which radio the packet came from.

4 Hardware Presentation Layer

The CCxx00 radio stack can port to any platform, with general IO connections to any general IO port. The hardware presentation layer (HPL), which is platform-specific and outside of the radio stack, provides interfaces that allows the stack to interoperate with hardware.

There are several main files the CCxx00 radio stack expects to find. Typically, the developer will place these files in a directory such as /tos/platforms/yourplatform/chips/ccxx00 to declare they are platform-specific configurations related to the CCxx00 chip.

It is your responsibility to implement these configurations and interfaces for your platform, and modify your .platform file so the compiler knows where to find the HPL directory. The radio stack itself will reference these configuration files, which will cause them to be pulled in at compile time.

4.1 HplCC1100PinsC / HplCC2500PinsC

This file is pulled in by the CC1100ControlC or CC2500ControlC. It provides GeneralIO and GpioInterrupt interfaces which allow the radio stack to interact with the radio's CSN and GDOx digital IO lines.

The signature for this hardware presentation layer configuration looks like this:

```
configuration HplCC1100PinsC {

  provides interface GeneralIO as Csn;
  provides interface GeneralIO as Gdo0_io;
  provides interface GeneralIO as Gdo2_io;

  provides interface GpioInterrupt as Gdo2_int;
  provides interface GpioInterrupt as Gdo0_int;

}
```

4.2 HplRadioAlarmC

This is a generic configuration that creates a new instance of the Alarm component. Each new instance should create a unique client for the alarm, where one client's start() command will not eventually result in a different client's fire() event.

```
generic configuration HplRadioAlarmC() {

  provides interface Init;
  provides interface Alarm<T32khz,uint16_t> as Alarm32khz16;

}
```

An example implementation for an MSP430 platform could look like this:

```
implementation {

  components new Alarm32khz16C();

  Init = Alarm32khz16C;
```

```
  Alarm32khz16 = Alarm32khz16C;

}
```

4.3 HplRadioSpiC

The HplRadioSpiC configuration simply forwards TinyOS SPI-bus interfaces to the appropriate SPI bus module.

```
configuration HplRadioSpiC {

  provides interface Resource;
  provides interface SpiByte;
  provides interface SpiPacket;

}
```

An example implementation for an MSP430 platform could look like this:

```
implementation {

  components new Msp430Spi0C() as SpiC;
  Resource = SpiC;
  SpiByte = SpiC;
  SpiPacket = SpiC;

}
```

5 Layers and Architecture Details

In this section, we'll comment on some of the layers and behavior in the radio stack, and clarify which alternative implementations for a single layer may be better for your system than others.

5.1 SPI Bus Implementation

The CCxx00 SPI bus was built from the beginning to support access to every register, strobe, and FIFO, which would allow the developers to experiment with all available options.

There are two layers of SPI bus arbiters built into the system. First, there is the SPI bus arbiter for the entire platform, which is built into TinyOS. On top of this arbiter is a CCxx00 driver specific arbiter which allows individual layers in the radio stack to control the SPI bus exclusive of each other. In order for an individual layer to access the SPI bus, it only has to call one request() command, which is provided by instances of BlazeSpiResourceC.

Multiple layers can request access to the SPI bus at a single time. The CCxx00 SPI bus arbiter will grant access in a round robin fashion. When there are no more users, the CCxx00 SPI bus arbiter will signal a request to see if it should temporarily maintain control of the platform's SPI bus. This is useful, for example, when the CCxx00 driver is waiting for an acknowledgment but no layers necessarily need to be interacting with the radio hardware.

When access is SPI granted to a layer in the radio stack, it is up to that layer to clear / set the chip select (CSN) line before attempting to interact with the radio.

5.2 Blaze Central Wiring

The CCxx00 radio stack supports individual as well as multiple radios. The central wiring component simply brings all of the hardware and configuration interfaces for each individual radio into the same location. In this manner, the BlazeCentralWiringC component simply acts as a façade to connect with more specifically named components like CC1100ControlC, CC2500ControlC, etc.

5.3 Acks

5.3.1 General notes on acknowledgments

The ack layer actually sits on top of CSMA and matches the source address, destination address, and data sequence number (DSN) of any incoming acknowledgment with corresponding fields in the outbound packet.

When a packet enters the Acknowledgment layer, the packet is quickly forwarded onto the CSMA layer. The acknowledgment timeout window does not start until the packet has actually been transferred, which is indicated by the sendDone() event coming back from CSMA. At that time, the ack layer starts its wait timer, which is really a 32 kHz alarm for more accuracy, to wait for an ack to be received by the radio.

The CSMA layer will attempt to release the SPI bus resource, and the CCxx00's personal SPI bus arbiter will ask any listeners if it should release the SPI bus for the entire radio. The ack layer will refuse to let go of the SPI bus for the chip. This allows ack timing constraints to be met. Imagine what would happen if we started waiting for an ack, but the flash chip driver took over the SPI bus and didn't release it until after our ack window: the ack would be dropped. This strategy prevents that scenario from occurring.

The acknowledgment wait period is in units of jiffies (1 jiffy is 1/32768 clock ticks per second, approximately 30.51 microseconds) and depends on your data rate. Its default value is a #define in the Acknowledgements.h header file, which can be overridden elsewhere to be compatible with your chosen data rate. For very fast data rates, the ack wait period could be on the order of 200-300 jiffies. At slow data rates, it could be as long as 2000 jiffies or more. The longer the ack wait period, the more likely you will get a successful ack back. The shorter the ack wait period, the faster your throughput.

To discover a good ack wait period for your given data rate, you calculate it, or setup a test to transmit packets from a receiver to a transmitter and increase the ack

wait period until acknowledgments are being reliably received. Another method, which I prefer, is modify the Blaze.h header file to set ENABLE_BLAZE_TIMING_LEDS to 1 and monitor the LED lines on a logic analyzer across several platforms that are all transmitting and requesting acknowledgments. This lets the developer actually see on the logic analyzer when a platform is transmitting, when it is waiting for an ack, and when the other platform is actually sending the ack. We should see the ack wait period be slightly longer than the amount of time it takes the receiving platform to transmit back an ack.

There are two things that might cause an ack to drop. First, collisions. Because there is a slight gap in modulation between the time the transmission ends and the time the acknowledgment begins to transmit, another node may jump in and start transmitting on the channel before or during the ack. This could push the ack outside the bounds of the ack window, or straight up collide. This could potentially be improved by leaving the original transmitter in TX mode (instead of going back into RX mode) and leaving TX mode manually. The purpose of this would be to remove the gap in modulation between the packet transmission and the ack transmission.

The second issue that has been observed is the fact that the CCxx00 radio has a problem with managing its internal state, or may see false energy on the channel when there is none. A receiver tells the radio to go into TX mode to send the ack, but the physical radio refuses. Eventually, the driver gives up and reboots the radio, which has been found to be the only solution. The reboot is transparent to other areas of the system, but causes the ack to be dropped because it misses its window.

5.3.2 Dynamic Acks

- Advantages: Very flexible. Fully implements the PacketAcknowledgements interface. Allows acknowledgments to be requested or not requested on a per-packet basis.
- Disadvantages: Uses more ROM to support flexibility.

5.3.3 Static Acks

- Advantages: Smaller memory footprint. All packets request acknowledgments.
- Disadvantages: Cannot prevent a packet from being acknowledged, which could decrease throughput if you don't need to acknowledge on a per-packet basis.

5.4 CSMA

CSMA/CA allows your node to share the channel fairly with other nodes. This is the main throughput bottleneck in the system, which impacts throughput even more so than your data rate in most cases.

The Csma.h files define the amount of time, in jiffies, the backoff events should occur. Backoff periods are also randomized to some degree to prevent two nodes from attempting to transmit simultaneously. If you need to increase your node-to-node throughput, decrease the minimum backoff periods. Beware though: decreasing the

backoff periods too much in a congested setting will cause collisions that will actually decrease total throughput.

5.4.1 Single Sense

Single Sense CSMA mimics the 802.15.4 CSMA implementations, like those found in the CC2420 stack. Its strategy is to backoff for a random period of time, and then perform a clear channel assessment one time immediately before taking over the channel and transmitting.

- Advantages: Low memory footprint. Faster throughput.
- Disadvantages: Can cause serious collisions and problems in a congested network. The implementation is considered ŕude. Completely incompatible with packetized low power communication wake-up transmissions, as we've seen in BoX-MAC low power communication strategies and CC2420 implementations in the past. Nodes can jump in very easily between the end of a packet transmission and the beginning of an ack transmission, killing acks. Only use this CSMA implementation if you expect isolated nodes in a non-mesh setting.

5.4.2 Continuous Sense

Continuous sense was designed and implemented in the CCxx00 stack specifically to fix the problems with the original single sense CSMA implementation that was instigated by 802.15.4.

Continuous sense truly behaves more like humans communicate: Listen for a period of time, if someone is heard talking during a listen period, back off for longer but grow a little more impatient. Don't jump in talking immediately as another entity finishes a sentence. Never interfere with acknowledgments.

The backoffs start with long durations, which gives the node a lower priority on the channel. If energy is heard on the channel at any point during a backoff, an energy interrupt fires and a new backoff period must be performed. A packet will only begin transmitting when an entire backoff period expires without energy on the channel at any point during that time. The instantaneous clear channel assessments of the single sense implementation are all done away with, and there is no way to interrupt an acknowledgment unless the minimum backoff settings are too low or the previous transmission on the channel was out of range.

- Advantages: Very robust, helps nodes share the channel more effectively. Much fewer collisions, much greater ack success rate in a network. Much more compatible with packetized wake-up transmissions in BoX-MAC type low power communication strategies.
- Disadvantages: Slightly larger memory footprint, which I find to be an acceptable trade-off for the increase in reliability.

5.5 Low Power Communications

The always-on, BMAC, BoX-MAC, and End Device implementations were briefly described in section 5.5. There are a few more things to note about low power communications.

5.5.1 BMAC

BMAC [9] is configured by your application layer to wake up periodically and sample the channel for energy. It is a bit more sophisticated than blind energy sampling, and can actually prevent itself from locking up in an infinite loop while the channel is being jammed by another device.

When BMAC wakes up to sample the channel for nearby wake-up transmissions, it goes through two phases to save energy and prevent jamming. First, it resets a counter and increments the count each time a sample confirms a wake-up transmission on the channel. Every sample that observes no wake-up transmissions on the channel decrements the counter, and if the counter reaches 0, the radio turns off. The initial value of this counter is selected to turn off the radio if the first sample returns false.

If energy is being continuously heard on the channel, this counter will increment to the point where it exceeds a predefined threshold. Going over this threshold causes BMAC to change sampling strategies. The next phase of sampling looks for for a preamble quality threshold (PQT) instead of clear channel assessments. This action prevents most types of jamming. If the signal on a channel is not that of a wake-up preamble, the counter will decrement to 0 and the radio will shut off.

5.5.2 BoX-MAC

The BoX-MAC [8] implementation is a combination of the best attributes of two other low power communication strategies: BMAC and XMAC. BMAC uses a long wake-up preamble during its transmission, which allows the channel to be continuously modulated and the receive checks to be extremely efficient. XMAC uses a packet-based wake-up transmission instead of a long preamble, which allows acknowledgments within the wake-up transmission. The wake-up transmission can be cut short, which increases throughput and saves energy at the transmitter.

Normally, the acknowledgment gaps within a packetized wake-up transmission cause quick and efficient receive checks in BMAC to miss the energy on the channel. The BoX-MAC implementation in the CCxx00 stack handles this a bit differently: Each wake-up packet actually has a long preamble, which allows the receive checks on the receiver side to remain extremely efficient because the probability of waking up in an acknowledgment gap is low.

BoX-MAC simply ties together the BMAC implementation with the Packet Link layer by using the SendNotifier interface to modify the packet as it enters the radio stack. It is up to the developer to wire the SendNotifier interface from the radio stack to the BoX-MAC layer, because the developer may want to make initialize outbound packets using the SendNotifier interface before the BoX-MAC layer takes over:

```
BoxmacC.SendNotifier -> BlazeC.SendNotifier;
```

The BoX-MAC layer makes several modifications to a packet:

1. Looks at the low power communication settings of a packet. If the destination's wake-up interval is above a certain value, BoX-MAC will divide the wake-up interval by a factor of 4.
2. Multiplies the Packet Link retries by 4 to make up for the shortened preamble length in each individual outbound copy of the packet.
3. Disables clear channel assessments in the CSMA layers for all copies of the packet except the first copy.

5.5.3 Wake-on Radio

Wake-on Radio [2] was attempted early on in the development of the CCxx00 stack. After fighting it for some time, we had a basic implementation working on a desk. Taking it outside proved we couldn't communicate more than the range of a desk. This was probably the way the implementation was exercising the radio, and not the radio itself.

Some severe non-range related issues were also observed where a radio would clearly enter WoR mode and never come out of it. Obviously if the radio doesn't do its job, the microcontroller has no way of knowing anything is wrong.

We abandoned WoR after discussing it with TI. Software controlled radio duty cycling is more accurate and easier to control. Eventually, the last remnants of the original WoR implementation were removed from the stack.

Reliability always wins over a slight increase in energy efficiency.

5.6 Radios

The radios can be configured to enable or disable address filtering, personal area network ID filtering, and auto acknowledgments at compile time using the preprocessor flags:

```
NO_ACKNOWLEDGEMENTS

NO_ADDRESS_RECOGNITION

NO_PAN_RECOGNITION
```

These settings can also be modified through the BlazeConfig interface.

All registers are stored in RAM so they may be updated at any time before burst initialization of the radio.

5.7 Unique Packet Filter

The unique packet filter keeps track of a short history of recently seen data sequence numbers (DSN) and source addresses. If a received packet's source address and DSN matches historical data, it is deemed to be a duplicate and is dropped.

Historically, the DSN field is set at the top level of the radio stack and is always incremented linearly for each outbound packet. Lower layers in the radio stack, such as packet link or low power communications, may try sending multiple copies of a packet which can be identified as duplicates by a receiver because the DSN doesn't change.

This linearly increasing DSN implementation leaves nodes open to attack, when no other security means are employed. Someone only needs to walk into the network, sniff a packet, and rebroadcast the packet with a DSN incremented by one. Receiving nodes then filter out packets from the good node, while only accepting the attacker's packets.

To prevent this kind of simple attack, the CCxx00 stack uses a random DSN for each outbound packet. This is more unpredictable and can help prevent an attacker from capture the channel.

6 CCxx00 Register Settings

Chip register values can be modified by altering the CC1100.h or CC2500.h files. We used SmartRF studio combined with manual hacking to come up with the register values found in these files.

You may want to modify the register settings to change the default channel or frequency, alter the data rate, modify the modulation, etc. Here is the process:

1. Open up SmartRF studio. Select the CC1101, for example.
2. Configure the main settings on the first screen like you want them: frequency, data rate, output power, modulation, manchester, crystal frequency, etc.
3. Go to File, Export CC1101 registers, and save the settings in a .txt file.
4. Manually translate the new register settings into the CC1100.h file.

There are some register settings you probably shouldn't alter, because parts of the radio stack were designed expecting some functionality to be setup on the radio. Typically, registers that have been manually configured have a comment associated with them. To help you know which registers are ok to touch and which may not be ok to touch, we've created a list below.

OK TO TOUCH

- `CC1100_DEFAULT_CHANNEL` (translates into a register)
- `CC1100_DEFAULT_FREQ2` (translates into a register)
- `CC1100_DEFAULT_FREQ1` (translates into a register)
- `CC1100_DEFAULT_FREQ0` (translates into a register)
- `CC1100_PA` (translates into a register)
- `CC1100_CONFIG_SYNC1`

- CC1100_CONFIG_SYNC0
- CC1100_CONFIG_FSCTRL1
- CC1100_CONFIG_FSCTRL0
- CC1100_CONFIG_MDMCFG4
- CC1100_CONFIG_MDMCFG3
- CC1100_CONFIG_MDMCFG2
- CC1100_CONFIG_MDMCFG1
- CC1100_CONFIG_MDMCFG0
- CC1100_CONFIG_MCSM2
- CC1100_CONFIG_FOCCFG
- CC1100_CONFIG_BSCFG
- CC1100_CONFIG_AGCTRL2
- CC1100_CONFIG_AGCTRL1
- CC1100_CONFIG_AGCTRL0
- CC1100_CONFIG_WOREVT1
- CC1100_CONFIG_WOREVT0
- CC1100_CONFIG_WORCTRL
- CC1100_CONFIG_FREND1
- CC1100_CONFIG_FREND0
- CC1100_CONFIG_FSCAL3
- CC1100_CONFIG_FSCAL2
- CC1100_CONFIG_FSCAL1
- CC1100_CONFIG_FSCAL0
- CC1100_CONFIG_RCCTRL1
- CC1100_CONFIG_RCCTRL0
- CC1100_CONFIG_FSTEST
- CC1100_CONFIG_PTEST
- CC1100_CONFIG_AGCTST

- CC1100_CONFIG_TEST2
- CC1100_CONFIG_TEST1
- CC1100_CONFIG_TEST0

DO NOT TOUCH

- CC1100_CONFIG_IOCFG2
- CC1100_CONFIG_IOCFG1
- CC1100_CONFIG_IOCFG0
- CC1100_CONFIG_FIFOTHR
- CC1100_CONFIG_PKTLEN
- CC1100_CONFIG_PKTCTRL1
- CC1100_CONFIG_PKTCTRL0
- CC1100_CONFIG_ADDR
- CC1100_CONFIG_CHANNR
- CC1100_CONFIG_FREQ2
- CC1100_CONFIG_FREQ1
- CC1100_CONFIG_FREQ0
- CC1100_CONFIG_MCSM1
- CC1100_CONFIG_MCSM0

7 CCxx00 Radio Stack History Manifesto

Below are some personal comments on my observations leading up to and through the development of the CCxx00 radio stack.

7.1 Bluetooth

Rincon originally pursued Bluetooth technology for basic wireless networking. This was abandoned as wireless sensor networks came to light and we realized Bluetooth was too power hungry and too short range for our intended applications.

7.2 CC1000 Nightmares

Working with TinyOS, the team from Rincon began down the path like everyone else in the TinyOS community: working with off-the-shelf, rather flakey hardware featuring the CC1000 radio at 915 MHz. Flaws in the hardware design of these off-the-shelf platforms caused mismatches in the matching network of the radio, and this was coupled into poor antennas. Communication was horrible. Frustrations ran high.

7.3 Tmote Sky

The CC1000 platform was soon abandoned as the tmote platform was released, featuring the CC2420 2.4 GHz 802.15.4 compliant radio. Rincon adopted this new off-the-shelf tmote platform for deliverable systems, and made significant architecture changes to the radio stack going into TinyOS 2.x while doing so. We decided early on to contribute infrastructure back to the open source community, since our business was to sell systems and solutions, not infrastructure. Life was good.

7.4 2.4 GHz is the devil

Basic testing revealed 2.4 GHz is an extremely poor choice for communication. We wanted hardware to work on the ground, but we could only get a parking space distance or two on a good day.

This led me to realize there are exactly two technical reasons why 2.4 GHz would make sense for any system: antenna size, and multi-country ISM band usage without hardware modification. Two non-technical reasons include the availability of off-the-shelf hardware, and marketing. The Intel culture has brainwashed everyone to think that more gigahertz is better. In the case of RF propagation, it's just the opposite: lower frequencies are better. The tribulations people experienced with the CC1000 radio seemed to encourage the wireless sensor network community to shy away from sub-GHz comms.

Your operating frequency is mutually exclusive of your data rate, which is another common misconception I hear quite often. To prove my point, the CC1100 radio has twice the data rate as the 2.4 GHz 802.15.4 radios, and it operates down to 300 MHz. Case closed.

The downsides to 2.4 GHz communication should be obvious. WiFi interference is everywhere. Portable phones moved away from 900 MHz to 2.4 GHz, and continue to move to other bands. 2.4 GHz is in the microwave band which allows those signals to be absorbed by water (your body) very easily. 2.4 GHz is absorbed by the ground and walls. It does not propagate well, and therefore is not a good choice for wireless sensor networks in general.

If you can think of any more reasons why 2.4 GHz would make sense for any given system, please email me.

7.5 Dual-radio platforms

We started getting requirements in to have a dual-radio platform. This obviously couldn't be purchased off-the-shelf, so we had to spin our own. One radio should continue to operate at 2.4 GHz because of the small antenna size, while the other radio would operate the network at a much lower frequency. The CC1100 / CC2500 radios were selected for this platform because they are pin and chip compatible and only require one radio driver to operate.

We ported TinyOS to SoftBaugh development boards and built a basic radio stack to facilitate testing of our custom hardware after assembly. The directory for this basic, non-fully functional stack was labeled 'ccxx00'. The hardware platforms being developed were dubbed 'blaze', and we needed a place to store a fully functional dual-radio platform stack. It seemed logical to call the dual-radio stack 'blaze' to match up with the hardware it was going on. The Blaze radio stack was born, and files labeled throughout the stack maintain this heritage.

7.6 Iterations

We did several revisions of the CCxx00 stack. We blazed our own path (pun intended) of designing how a dual-radio platform should look in TinyOS. TUnit was created just before the development of the CCxx00 stack, so unit tests were employed through development, starting with basic SPI bus communication and verification of the radio registers. The first revision got basic communication up and running, but no low power communications implementations. Some areas of the radio stack had issues and needed to be redone after the first phase.

The second iteration fixed the architectural issues and started adding in wake-on radio. As described in the wake-on radio section, we got it working but found it didn't work too well. The chances of us getting it to work 100% were slim given the amount of time we had for development, so we abandoned it and built BMAC.

BMAC worked great on these platforms, so we also tried a few other low power communications layers. The good ones still exist.

The third iteration tried to knock out bugs dealing with the radio hardware. Using SPI bus analyzers, we were able to see our platform telling the radio to enter TX mode or RX mode, but the radio would refuse and go to IDLE. This caused us to rearchitect even more of the stack to be able to reboot the radio while in the middle of transmission or reception logic. Quite annoying.

We tried to get a layer built into the low levels of the stack to throttle back communications to the TX/RX FIFO's. This would have allowed the radio stack to communicate packets greater than the length of the 64 byte FIFO's, but never got it working 100% because we didn't have the hours to finish it.

At some point, we figured out how to get the radio to properly transmit at low data rates, which we were not able to do before because our previous versions didn't burst-init some TEST registers. This spawned a fourth revision of the radio stack tailored to these low data rates. The general rule is: every 50% decrease in data rate results in a 50% increase in range. Revisiting 802.15.4 which claims to be low data rate, we see that 500, 250, or even 40 kbps is definitely not a low data rate. 1.2 kbps is. 5 kbps

maybe. Who ever used dial up internet access in the '90s with a 250 or 500 kbps data rate? Not me.

Our networks now operate at the lowest frequency we can get our hands on, and at a low data rate. We see 700-800 feet on the ground, and up to 1 mile with nodes in the air at 10 kBaud Manchester GFSK. When we lowered the data rate to 1.2 kBaud Manchester GFSK, it was more like 4 kilometers with one node on the roof and the other in my hand. No external PA's or LNA's. We did have a large ground plane, though, made out of 4 wires that formed an X with the monopole wire antenna sticking up through the middle. That large RF ground plane helped our range a lot.

7.7 Polish

The goal of the fifth iteration of the radio stack development was to decrease the code size as much as possible. In doing this, we used module_memory_usage [10], a script developed by Cory Sharp, to see which modules were sucking up the most program ROM. Turns out every time you use a 32-bit alarm, it creates an instance of TransformAlarmC which eats away memory, and the radio stack had several of these. Swapping it out with a 16-bit alarm didn't change functionality, but did significantly decrease memory footprint.

We also extracted the parameterized interfaces throughout the stack, originally designed for the dual-radio platform. Now, multiple radios can be selected by letting the stack dynamically switch the wiring in the Central Wiring component.

Other modifications were made, unnecessarily functionality was separated, and the stack became a radio stack that could be compiled for single- or multi-radio platforms.

7.8 CCxx00 Issues

The CC1100 radio has some issues, but once we worked past them, the radio outperformed other radios in the wireless sensor networking community. One issue is the problems on the digital logic side, where the state sometimes refuses to change. Another issue is with the wake-on radio implementation and how it sometimes just stops working. The radio sometimes registers a busy channel while doing a clear channel assessment, even though the channel is definitely shown to be clear on the spectrum analyzer. The errata is thick [3]. For these reasons and others, the radio stack took a long time to develop. We may have put in a little bit extra ROM trying to work around some of these issues, but it turned out not to be too bad after a few revisions.

References

[1] TinyOS Community. http://docs.tinyos.net.

[2] Texas Instruments. CC1101 datasheet. http://www.ti.com/lit/gpn/cc1101.

[3] Texas Instruments. CC1101 errata. http://www.ti.com/litv/pdf/swrz020b.

[4] Kevin Klues, Vlado Handziski, Jan-Hinrich Hauer, and Phil Levis. TEP115: Power management of non-virtualised devices. http://www.tinyos.net/ tinyos-2.1.0/doc/html/tep115.html.

[5] David Moss. CCxx00 implementation. http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x-contrib/blaze/.

[6] David Moss, Jonathan Hui, and Kevin Klues. TEP105: Low power listening. http://www.tinyos.net/ tinyos-2.1.0/doc/html/tep105.html.

[7] David Moss and Phil Levis. TEP127: Packet link layer. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep127.html.

[8] David Moss and Philip Levis. BoX-MACs: Exploiting physical and link layer boundaries in low-power networking, 2008.

[9] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. *SenSys*, 2004.

[10] Cory Sharp. Module memory usage script. http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-1.x/contrib/ SystemC/scripts/.