EagleChat

Kevin Ward, Chase Meadors, Sam Bretz, Cris Slaughter

ECEN 4024 – Capstone Design II

5/1/15

1 CONTENTS

2 INTRODUCTION

EagleChat is a small peripheral device that connects via USB to Android smartphones, which act as the interface and controller for the device.

Whenever EagleChat devices are powered on, they act as routers and form an ad-hoc mesh network with other EagleChat devices in the vicinity.

Whenever two EagleChat users have recognized each other as contacts, they can send messages to one another over the mesh network, provided that a route can be found, simply by using the Android application. These messages are fully end-to-end encrypted, and can only be interpreted by the sender and receiver.

The application manages incoming and outgoing messages, informing the user if his or her message fails to reach the destination.

In addition, the application allows two EagleChat users to register each other's contact information simply by scanning a QR code that is displayed in the app, and organizes all of a user's contacts and conversation threads.

EagleChat forms a device network without any central infrastructure. If a device moves to a different location, or is powered down, any nodes depending on it for message transmission will simply attempt to find a new route.

EagleChat delivers simple, secure communication in environments where other conventional mass communication methods such as cell networks and the Internet may be controlled or censored.

This report will detail the hardware and software design of EagleChat. Unless noted otherwise, the lead engineer listed for each section of the report is also the author of that section.

3 OPERATION INSTRUCTIONS

You will need at least two EagleChat devices and two modern Android phones to make use of the functionality.

For each EagleChat device and phone, perform the following steps:

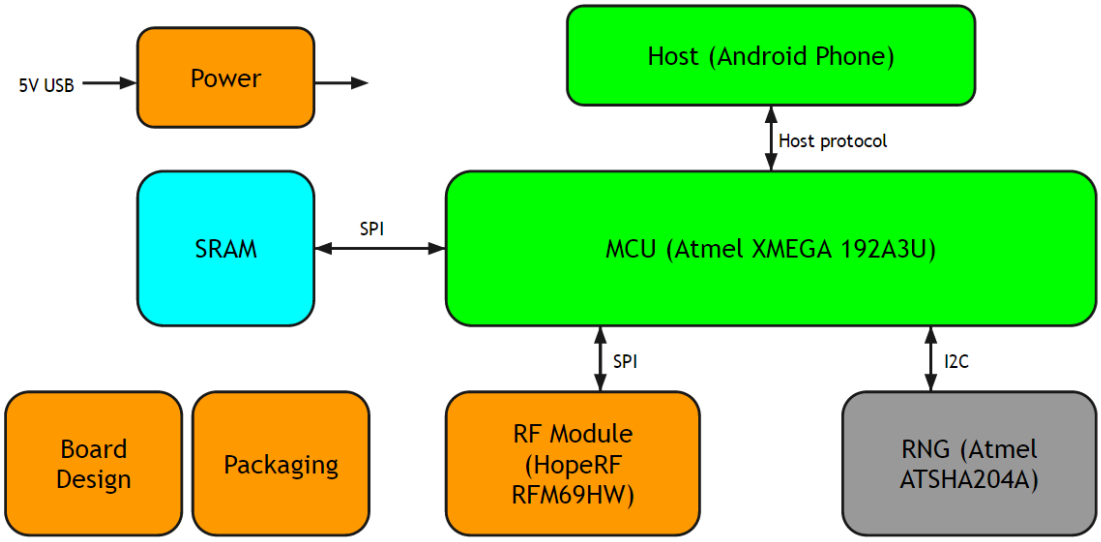
1. Install the EagleChat application, provided with this report
2. Plug the EagleChat board into the phone
3. Launch the EagleChat app – a screen will appear offering to set up a new EagleChat device
4. Enter the following information:
    a. A name that will identify you to other users on the network
    b. A password you will use to login to your device
    c. A network ID that does not conflict with any other ID on the network
5. Press the check-mark button to commit the values and setup your EagleChat device.

Now that the EagleChat devices have been set up, you and your partner need to add each other as contacts:

1. One user will navigate to the “My Details” page, where a QR code is displayed.
2. The other user will click the “+” button to add a contact, and then click “Scan QR Code” to read the QR code on the first user’s screen.
3. The other user will then click the check-mark button to finish adding the contact.
4. Perform this process in reverse so that both users have added the other as a contact.

Now, any user can select that contact from the list and type a message. The application and the EagleChat device will take care of establishing a route, and the other user will receive the message.

4 HARDWARE DESIGN



4.1 MICROCONTROLLER

Lead Engineer: Kevin Ward

The heart of the EagleChat system is the XMEGA 192a3u, an 8-bit microcontroller from Atmel, Inc. The XMEGA line is an advanced RISC architecture CPU, available in a wide variety of configurations featuring different combinations of peripherals and flash and SRAM sizes.

The following features of the XMEGA 192a3u are the most relevant to the EagleChat project [1]:

- Integrated USB controller and driver
- 16 KB SRAM
- Built-in EEPROM
- CRC-16 module
- Real-time clock
- 16-bit timers
- Multiple SPI modules
- Multiples I2C/TWI modules
- Easy to program 8-bit architecture
- Well supported by the Atmel Software Framework

The microcontroller runs the EagleChat firmware, and is responsible for communicating with the host and the radio module, and performs all encryption related tasks.

[1] Atmel, “8/16-bit Atmel XMEGA A3U Microcontroller,” XMEGA A3U datasheet, July 2007 [Revised July 2014]

4.2 RANDOM NUMBER GENERATOR

Lead Engineer: Chase Meadors

Contributors: Kevin Ward

EagleChat requires secure random numbers in order guarantee that the encryption process is cryptographically secure. The Atmel XMEGA chip does not have an on-board RNG, and this chip provides a lightweight, simple, external solution.

We selected the **Atmel ATSHA204A** (http://www.atmel.com/devices/ATSHA204A.aspx), which has an internal high-quality random number generator [1].

The chip communicates via a simple I2C interface, and integrates simply into our application. In addition, we made use of a driver library already made available in the Atmel Software Framework (ASF) [2]. We implemented a wrapper and API on top of the ASF driver, which can be found in the `/sha204` subdirectory of the source code respository. The API provides the following functions:

- Initialize an ATSHA204A chip
- 'Lock' an ATSHA204 chip, which permanently commits the configuration of the device, allowing it to provide random numbers
- Retrieve random numbers from the chip

[1] Atmel.com,. 'ATSHA204A'. N.p., 2015. Web. 1 May 2015.

4.3 RF MODULE

Lead Engineer: Sam Bretz

The communication device of the EagleChat system is the HopeRF RFM69HW, a monolithic device based off of the Semtech SX1231H.

The following are the important features of this chip for EagleChat [1]:

- +20 dBm - 100 mW Power Output Capability
- High Sensitivity: down to -120 dBm at 1.2 kbps
- Low current: Rx = 16 mA, 100nA register retention
- Automatic RF Sense with ultra-fast AFC

We obtained a radio range of about 422' under optimal conditions and configuration settings. I believe range could be increased with fine trial and error tweaking of the radio driver configurations parameters and various antenna counterpoise designs.

[1] HopeRF, "RFM69HW ISM TRANSCEIVER MODULE V1.3" Datasheet
[http://www.hoperf.com/upload/rf/RFM69HW-V1.3.pdf]

4.4 POWER

Lead Engineer: Sam Bretz

We decided to solely power device through the USB cable without any internal batteries. A buffered pin is connected through from the 5V line to tell the MCU if external power has been cut and to prepare the device to be powered down. This device does not consume a lot of power, with the radio module consuming by far the most at 130mA during transmission. A simple high efficiency solution was sought after, yielding a specialized 5V to 3.3V LDO, the TLV1117LV33. With a quiescent current of approximately 150uA at peak current consumption for our device, it was one of the most efficient available while also being one of the cheaper units.

Battery life reduction of the phone while being used was not fully measured but it was not noticed during testing and use. Complete analysis highly depends on network node and message density. With the radio module consuming 16mA in receive mode, 130mA in transmit mode, and a transmit to receive duty cycle of 15% this would use approximately 33mAh of capacity over an hour. 15% is estimated to be rather high transmission ratio based on transmission and receive timings. Accounting for the efficiency of the various regulators in both the phone and device this results in approximately 50mA consumption. An average modern phone battery has approximately 2200mAh of capacity.

4.5 BOARD DESIGN

Lead Engineer: Sam Bretz

EagleChat's board design prioritized small size and good RF characteristics. This lead to our final design with a decently small size and acceptable RF characteristics when mounted inside metal packaging to serve as the antenna counterpoise.

- Size: 1.67” x 1.36” (2.27sq in)
- Mounting holes are in a standard 1” x 1” square pattern allowing for ease of mounting.
- Antenna is of a screw-on-PCB design that allows for easy assembly
- Hard wired USB interface to connect to the phone
- Exclusively uses SMD components except for connector headers and antenna

Refer to Appendix B for PCB diagrams.

4.6 PACKAGING

Lead Engineer: Sam Bretz

The ideal packaging solution would meet the following criteria:

- Universal device compatibility
- Non-permanant mounting with ease of removal
- Would not cause the phone to disrupt the RF antenna pattern creating dead reception zones

After heavy design consideration some of these criteria are nearly mutually exclusive. Due to the nature of varying modern phone sizes, making a universal phone case is impractical. The only universal solution to satisfy all three is to have it as a dongle.

The enclosure takes the form of a simple pocketed enclosure with antenna port, stress relieved USB cable slot, and a hole for the reset button.

Material of the enclosure is rather unimportant unless considering durability, while the only electrically stipulated requirement to maximize range is to have a good ground shield the length and width of the enclosure to act as the counterpoise of the antenna. This could be done with a stamped steel shield, vapor deposited metal coating, or by having a solid metal enclosure. We made a single enclosure with aluminum and one out of 3D printed PLA. When considering manufacturability an injection plastic molded enclosure with stamped steel shield would be most practical.

The current 3D printed enclosure specifications are as follows:

- Dimensions including antenna: 3.6” x 1.75” x 0.5”
- Weight: 0.7oz

5 Software Design

5.1 Firmware Architecture

Lead Engineer: Kevin Ward

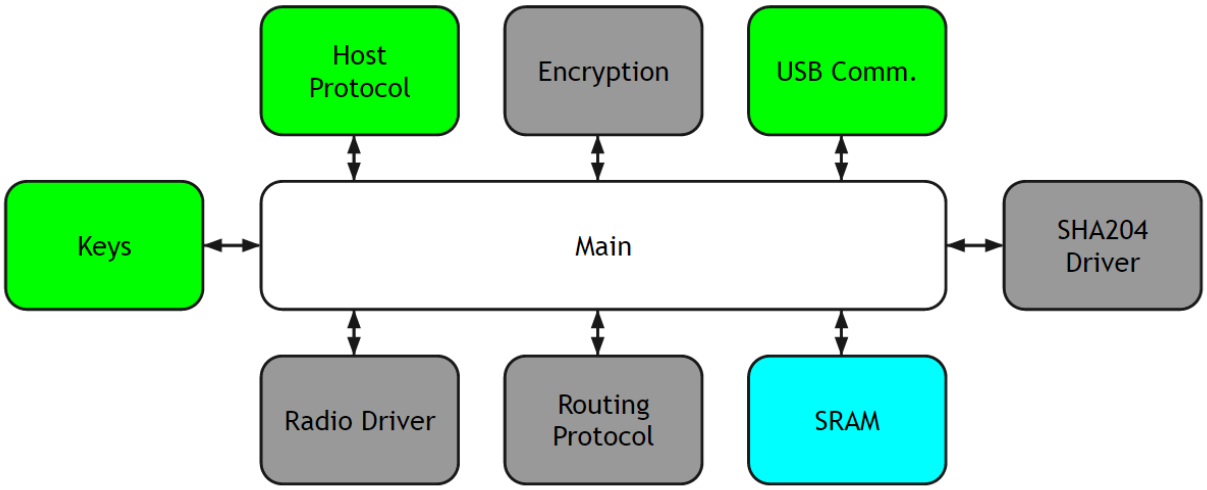
Contributors: Chase Meadors

Overview

The EagleChat firmware is written in C and is compiled using the AVR port of GCC. It is organized into several modules that either provide a hardware driver interface or serve as convenience layers for high layers. At the top layer, the main function in app/main.c is responsible for coordinating the work of the different modules and enforces the expected behavior of the system.

Each module will be discussed in its own section, but here is an overview of the modules composing the firmware:

- `app` -- Top level code that integrates all modules
- `protocol` -- Implements the Peregrine host communication protocol
- `cdc` -- Interface to the CDC USB stack, used for communicating with the host device
- `keys` -- Stores configuration data and encryption keys
- `crypto` -- Encrypts and decrypts data
- `sha204` -- Drives the SHA204p random number generator IC
- `routing` -- Manages the receptions and forwarding of packets
- `radio` -- Drives the RFM69HW radio module and handles sending and receiving data frames



Architecture

The firmware has an asynchronous architecture, where all input and output is handled by interrupts and is buffered until it is ready to be sent/consumed. The routing handling code is run at regular intervals on a timer interrupt, to prevent a long running operation on the main "thread" from causing dropped packets.

The firmware's main loop performs the following actions as quickly as possible:

- Process new incoming messages from the host, which are collected and buffered asynchronously.
- Perform requested actions based on the incoming message, such as queueing packets for transmission through the radio stack
- Process received packets from the network stack, which are collected and buffered asynchronously, and decrypt them

And on a timed regular basis:

- Instruct the routing stack to process its send queue
- Instruct the routing stack to process received frames

In addition, the firmware implements BURN functionality. When instructed by the host phone, the board will erase all contents of the on-board EEPROM, which is where secure keys and other information is stored.

Build system

The firmware is designed to be built on Unix-like systems. The build system is based on the venerable make utility, and uses a base makefile derived from the makefile included with ASF.

Dependencies

The EagleChat firmware has the following build dependencies:

- AVR-GCC 4.2+
- AVR-Libc
- Atmel Software Framework

Build Instructions

After installing required dependencies, copy the file `conf.mk.bak` in the firmware project root to "conf.mk", and edit the file to reference the root directory of a local copy of ASF.

To build the firmware, at the terminal change to the /app directory. Invoke make:

```
eagle-chat-firmware/app$ make
```

The firmware will compile and result in several product files. The firmware binary is contained in `main.hex` and is suitable for upload to the EagleChat peripheral.

Source Code

Our source code repository is located at https://github.com/EagleChat/eagle-chat-firmware, and our development occurs on branch `develop`.

A copy of our production source code is also included on the accompanying CD, under `eagle-chat-firmware`. When we refer to the *source code repository*, we are referring to this folder.

Android application development is located in https://github.com/EagleChat/eagle-chat-app

5.2 USB COMMUNICATION

Lead Engineer: Kevin Ward

Contributors: Chase Meadors

The EagleChat peripheral communicates with its host device through a Communications Device Class (CDC) driver over USB. Our implementation builds on the Atmel Software Framework's CDC implementation. The CDC device class is used for "virtual com ports" and other serial devices. It appears to the host operating system as a serial port, tty, serial modem or similar, and provides a simple byte based communication interface.

From a software perspective, this module is very straightforward. The details of the USB interface (vendor ID, product ID, baud rate) are configured in a header file, `config/conf_usb.h`. The ASF provides several functions for reading and writing to the data stream. We then wrote convenient wrapper functions to accomplish common writing and reading tasks. This code is available in `cdc/cdc`, and includes functions for:

- Writing a string followed by an integer
- Writing a string followed by a byte in hexadecimal form
- Reading a line of bytes from the host

In addition, the cdc module allows the firmware to tell when a connection has been established or closed.

The functions in cdc are used pervasively throughout the firmware. Every module that does any sort of printing or reading is built on top of this module. Host Communication Protocol

Communication between the EagleChat peripheral and its host (Android phone, PC) is structured according to a custom protocol. This protocol is nicknamed Peregrine to distinguish it from other protocols used in this project.

The Peregrine protocol is based on commands from the host and replies from the peripheral. All commands and replies are composed of printable ASCII characters and are terminated with a newline character (0x0A).

Commands follow this basic scheme:

Command

`[command byte][0x0A]`

For example, issuing a command to generate an encryption key-pair is simply

`k[0x0A]`

Command with arguments:

`[command byte]:arg1:arg2 ...[0x0A]`

For example, passing a message to be broadcast would look like this:

`s:123:This is the text of the message.[0x0A]`

Replies follow a similar scheme. Replies are always prefixed with '`x`', and are followed by a colon-delimited messages. Successful execution of a command is indicated with "`x:OK`". If the command cannot be completed or fails, this will be indicated by "`x:FAIL`" or "`x:FAIL:[Error description]`".

Binary data is always transmitted by first encoding it in printable hex-format (e.g. 0x12 is sent as '1' followed by '2'). It should be interpreted by first parsing it back into raw binary. The length of the resulting array will be half of the length of the hex-encoded string.

The exception to binary data encoding is in commands requiring the address of another node. These are encoded in printable decimal form (e.g. `123` as "`123`").

See the table on the following page for a complete summary of available commands and replies.

Command	Format	Description	Replies

Send message	`s:[addr]:[message]`	Queues the contents of message string to be encrypted and transmitted to the node with the specified address.	`x:OK`
			`x:FAIL:INVALID: NO ADDRESS`
			`x:FAIL:INVALID: NO CONTENT`
			`x:FAIL:No public key entry for that node`

Set public key for contact	`p:[addr]:[64 hex]`	Stores the 32-byte public key for node with specified address and computes the shared key	`x:OK`
Generate keys	`K`	Generates a new keypair	`x:OK`
Set ID	`i:[2 bytes hex]`	Sets this node's network ID	`x:OK` `x:FAIL:Node Id must be < 255`
Set password	`h:[60 bytes hex]`	Sets the password hash to be used for authentication	`x:OK`
Authenticate	`a:[60 bytes hex]`	Attempts to log-in to the peripheral	`x:OK` `x:FAIL`
Commit configuration	`c`	Saves the peripheral's set up data and confirms that it has been fully set up	`x:OK` `x:FAIL:All components not configured`
Erase all data	`BURN`	Erases the peripheral's non-volatile memory section and resets the MCU.	No reply, immediate execution of command
Get public key	`g:p`	Gets the peripheral's public key	`x:[64 characters hex]`
Get network ID	`g:i`	Gets the peripheral's network ID	`x:[2 characters hex]`
Get status	`g:s`	Gets the peripheral's configuration status	`x:[2 characters hex]`
Print routing table	`d:r`	Debugging command that prints the peripheral's current routing table entries	Text describing the current state of the routing table

Message received

In addition, when a message has been received and decrypted by the peripheral, it will notify the host by sending a "message received" message, using the following format:

`r:[2 hex characters](address):[message string]`

where address is the network ID of sender, and message string is the decrypted contents of the message

Implementation

The app/host_rx.c and app/host_tx.c modules buffer input and output messages until they are ready to be consumed/sent; app/protocol.h and protocol.c contain protocol definitions and convenience functions for formatting replies; and app/main.c is responsible for interpreting and responding to commands appropriately.

5.3 RADIO DRIVER

Lead Engineer: Sam Bretz

Contributors: Kevin Ward, Chase Meadors

We based our Radio software on an existing Arduino library that drives the RFM69 family of HopeRF radio modules. The library was open source, written and provided by Low Power Lab [1].

The library provides a simple API that approximately implements network functionality up to the data link layer (layer 2 in the OSI model).

The API automatically handles all radio physical settings, frame headers, and acknowledgements (ACKs).

Since the library was for the Arduino platform, we made extensive changes in order to make a custom port for our platform. We also added new features. Some of our changes were:

- Re-implementing the send functions to enable data sizes over 61 bytes – a limit that was imposed by the original library. This involved changing the send functions to use the on-board FIFO in the RFM69 in a different way
- Re-implemented CRC functionality to work with the new send functions
- Fixed bugs relating to incorrect reading of the radio RSSI values

In addition, we then wrapped the radio library in a higher level wrapper that integrated nicely into the rest of our application.

The custom RFM69 library can be located in `/radio/RFM69.h` and `/radio/RFM69.cpp`

The radio wrapper and API can be located in `/radio/radio.h` and `/radio/radio.cpp`

[1] GitHub,. 'Lowpowerlab/RFM69'. N.p., 2015. Web. 1 May 2015.

5.4 ROUTING PROTOCOL

Lead Engineer: Chase Meadors

Contributors: Kevin Ward

EagleChat's radio driver provides full data link layer functionality (layer 2 of the OSI model) to guarantee reliable point-to-point acknowledged transmission of frames from one point to another. Our goal in designing the routing protocol was to engineer a reliable implementation of the "IP Layer" (layer 3 of the OSI model) that manages routes and the transmission of packets from any arbitrary node to another.

Our protocol was influenced by researching **AODV** – Ad-hoc on-demand distance vector routing [1]. AODV is a mesh routing protocol used in ZigBee, and contains many advanced features to improve network efficiency and reliability.

Since there were no implementations of AODV or similar protocols that would integrate into our project in the time allotted, we designed a custom routing protocol that contains the most basic ideas from AODV.

Our routing protocol is based on packets with headers containing the following information:

- Source (node id)
- Destination (node id)
- Type (enumeration)

There are four packet types:

- CONTENT – a normal packet containing a message
- RRQ – route request packet
- RUP – route update packet
- FAIL – route failure

In addition, RRQ and RUP packets have the following additional information in their header:

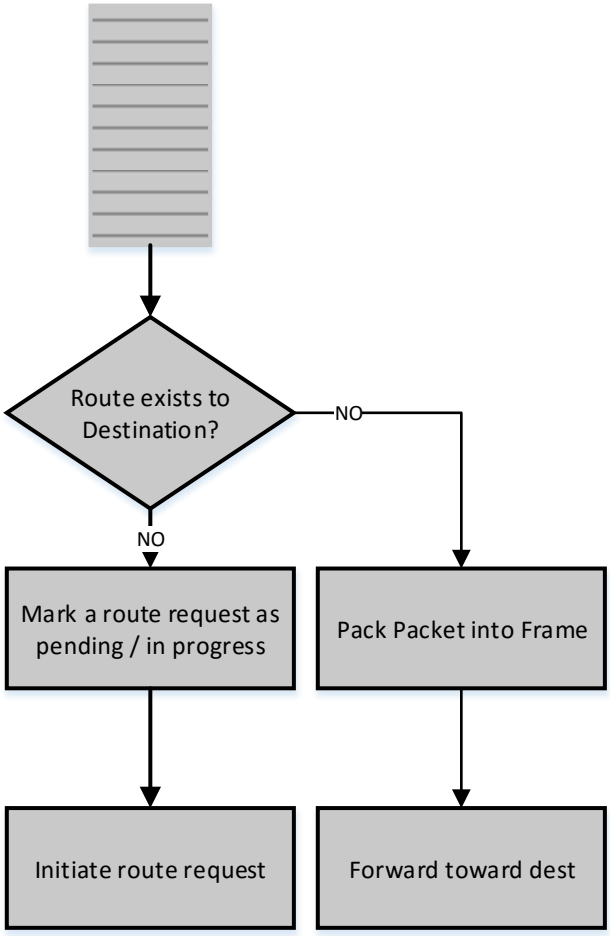
- RRQ ID – a unique ID that identifies a particular route request
- Hop count – A value tracking the length of the node path that the RRQ travels

Nodes also posses a routing table whose entries have the following information for each destination:

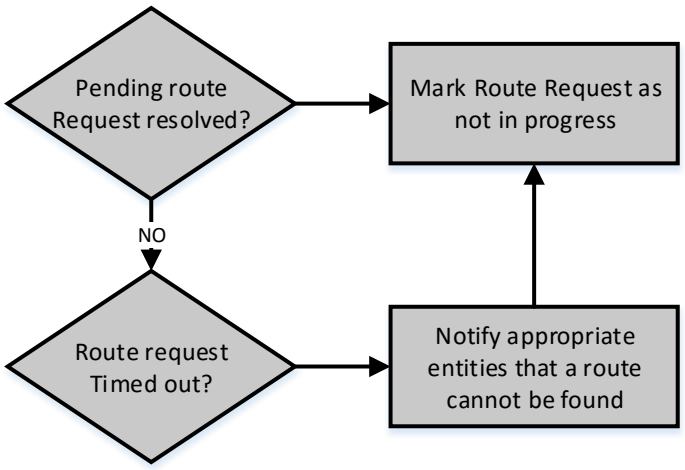
- Next Hop – a node ID that represents the next hop to the destination
- Original RRQ ID – a number specifying the RRQ that generated this entry
- Failures – A number keeping track of the number of times transmissions using this entry failed

The next few figures demonstrate the three major processes occurring in the routing module:

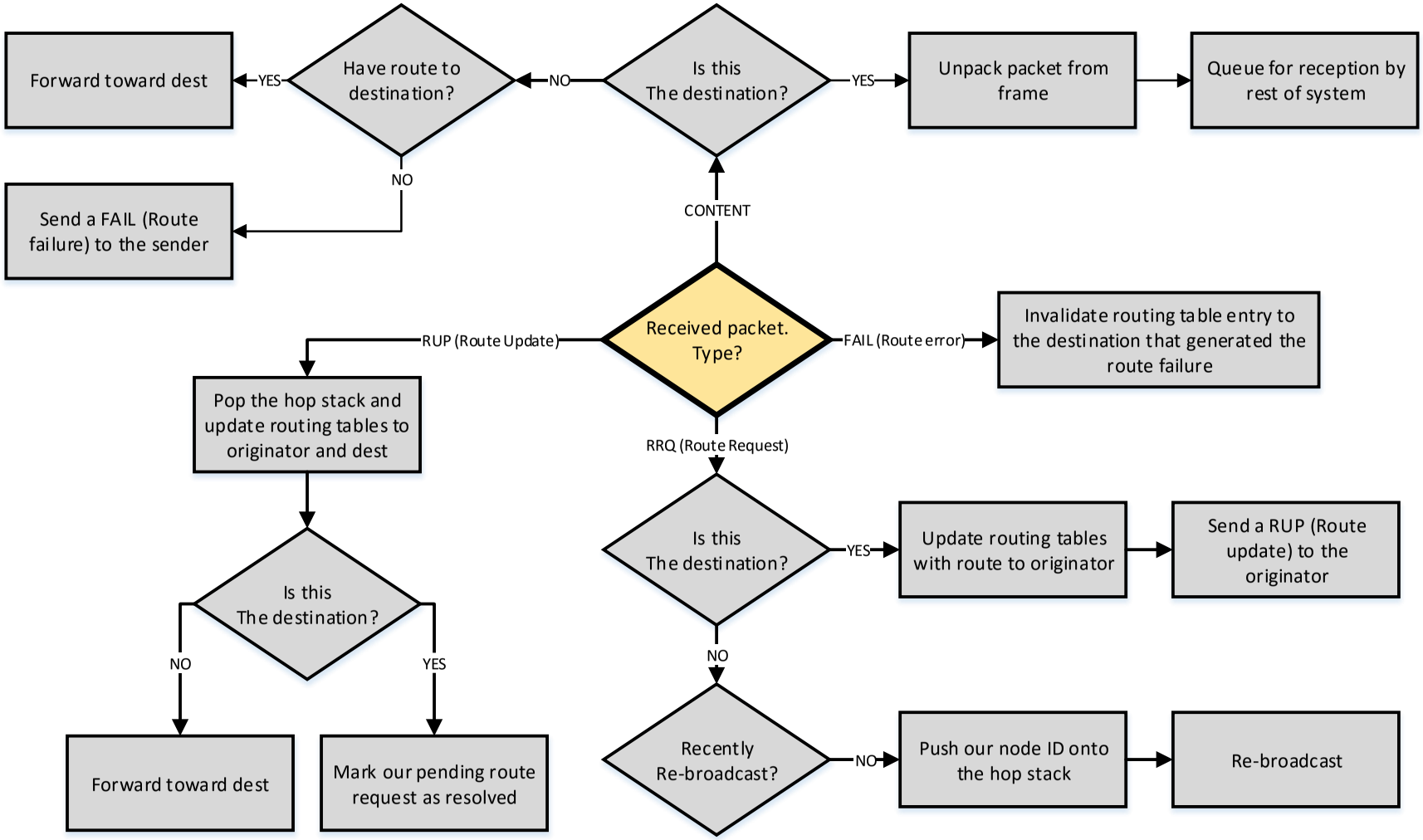
When a new Packet is queued for transmission by the rest of the system, the following process occurs:



In addition, the following process that manages route requests occurs ‘constantly’ (whenever serviced)



The following page shows the process that occurs upon frame reception:


Forward toward dest
YES
Have route to destination?
NO
Is this The destination?
YES
Unpack packet from frame
Queue for reception by rest of system
NO
Send a FAIL (Route failure) to the sender
CONTENT
Received packet. Type?
RUP (Route Update)
FAIL (Route error)
Invalidate routing table entry to the destination that generated the route failure
Pop the hop stack and update routing tables to originator and dest
RRQ (Route Request)
Is this The destination?
NO
YES
Forward toward dest
Mark our pending route request as resolved
Is this The destination?
YES
Update routing tables with route to originator
Send a RUP (Route update) to the originator
NO
Recently Re-broadcast?
NO
Push our node ID onto the hop stack
Re-broadcast

Finally, a brief description of the route request process from a holistic perspective:

Suppose that node A queues up a packet addressed to node B, but fails to have a route (as demonstrated in the diagram above). Then:

1. Node A creates a new RRQ packet, initializing hop count to 0, and broadcasts it.
2. Every other node in the vicinity receives the RRQ, and checks if they have recently forwarded the RRQ with this ID. If not, they push their own ID onto the stack in the packet contents and re-broadcast the packet
3. When the destination hears the RRQ, it initiates a RUP, which will be sent in reverse order along the hop stack built up inside the RRQ
4. Each node that hears the RUP on the way back will update their routing tables.

As part of routing development, I also developed a visualization tool and simulator of the algorithm built in JavaScript and HTML5. Kevin Ward also collaborated on this project. The simulator contains a complete “port” of our routing algorithm, and simulates its behavior in large dense networks of nodes.

The demonstration can be viewed at http://cemulate.github.io/mesh-routing.

The source repository is, of course, also on GitHub and can be viewed at http://github.com/cemulate/mesh-routing.

5.5 ENCRYPTION

Author: Chase Meadors

Contributors: Chase Meadors (Lead), Kevin Ward

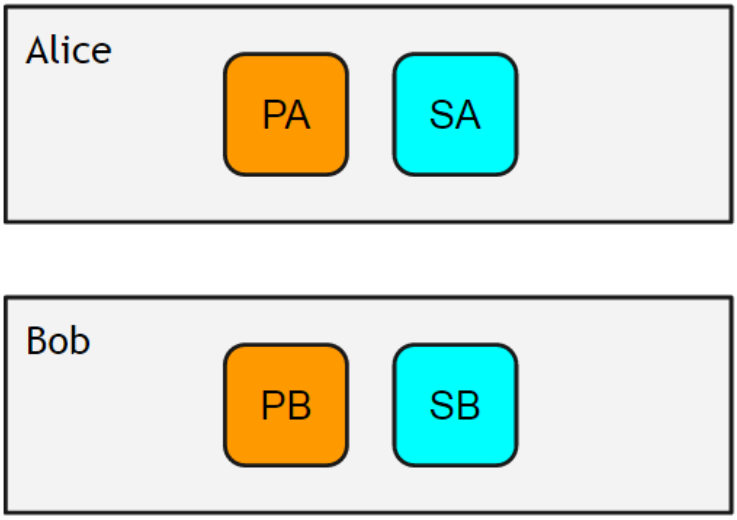
EagleChat uses encryption techniques to guarantee the safe sending of messages between registered partners in the system. One of the core design goals in this project was to provide cryptographically secure communication. Indeed, any information travelling over the wireless channel in our system is fully encrypted, and can only be decrypted by the sender and receiver for which the information is addressed.

We use Elliptic Curve cryptography, a form of public-key-based encryption. To accomplish our implementation, we used **AVR NaCl** (http://munacl.cryptojedi.org/atmega.shtml). AVR NaCl is an implementation of the more well-known C library NaCl (http://nacl.cr.yp.to/), for the Atmel microprocessor architecture.

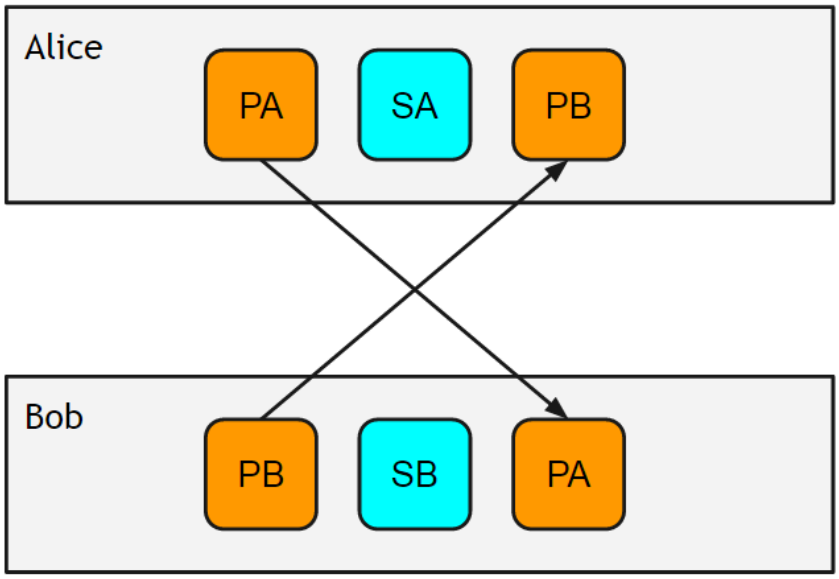
NaCl consists of three major components [1]:

- The Curve25519 Diffie-Hellman key exchange function. This functionality takes advantage of the mathematics of Elliptic Curves (actually using the particular curve, Curve25519, which is recognized as 'strong') to compute shared secrets between two communication partners.
- The Salsa20 stream cipher. This algorithm is responsible for encrypting the message using the shared secret.
- The Poly1305 message-authentication code, the algorithm allowing the authentication (and thus decryption) of messages using the shared secret.

The following illustrates the process used in EagleChat to secure communication between two partners, Alice and Bob.



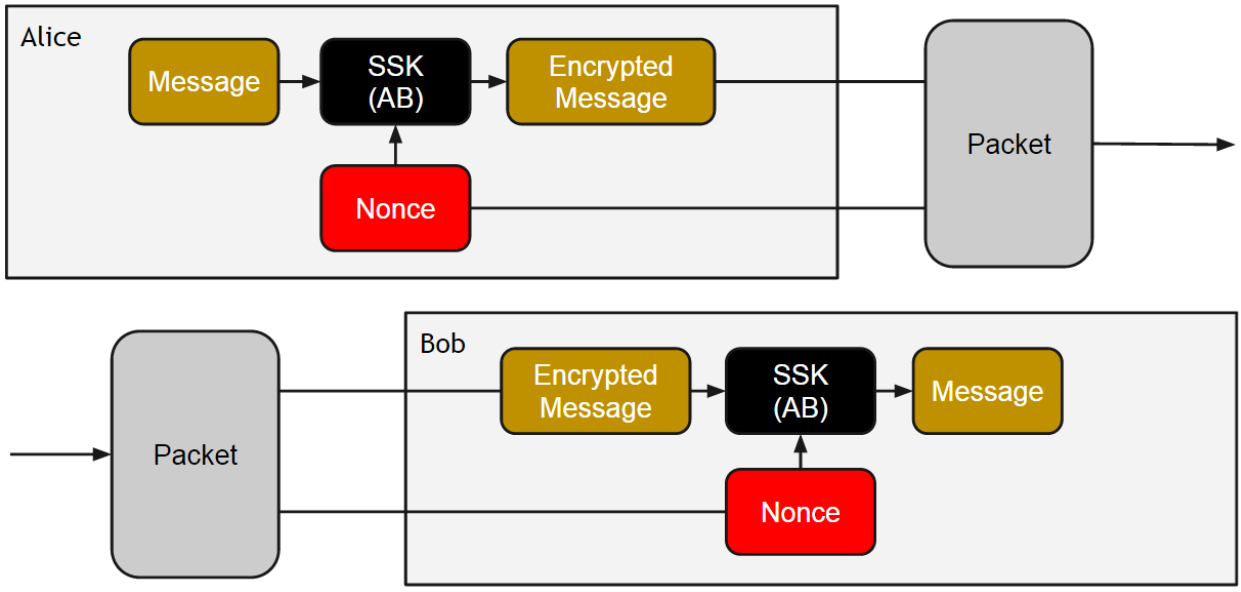
Alice and Bob each possess a public and secret key, that are unique and securely generated at device setup.



Alice and Bob exchange public keys whenever they register as contacts with each other.

Alice and Bob are now able to compute an (identical) shared secret key, using their own secret key and the partner's public key. Alice and Bob both compute this key at the time of contact registration, and store it.

Alice and Bob now have all the information they need to communicate with each other. The following picture shows what happens when Alice sends a message to Bob:



Alice uses the shared secret key, and a randomly generated nonce, to secure the message, and then bundles this information together into the message she sends to Bob.

Bob can use the nonce and his (identical) shared secret key to decrypt the message.

This encryption system is made cryptographically secure by incorporating the ATSHA204 random number generation chip (detailed in its corresponding section) to generate keys and nonces.

The encryption implementation and API can be found in the `/crypto` subfolder of the source repository.

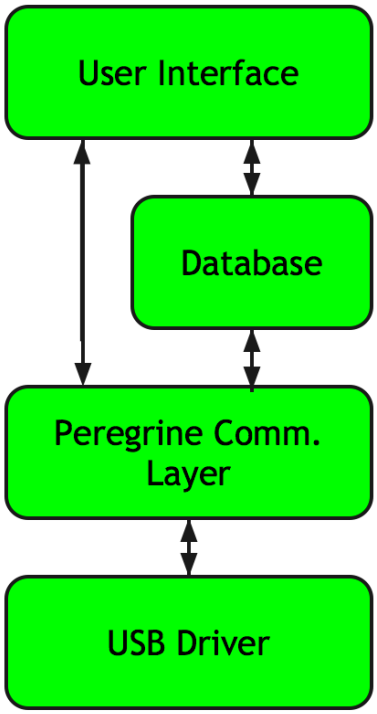
[1] Bernstein, Daniel, Tanja Lange, and Peter Schwabe. Securing Communication. 1st ed. 2013. Web. 30 Apr. 2015.

5.6 ANDROID APP

Lead Engineer: Kevin Ward

The EagleChat Android App serves as the user interface for the EagleChat system, as well as fulfilling other important functions. The EagleChat app is responsible for:

- Text message entry
- Text message display
- Contact management
- Public key exchange and update
- Source of content packets
- Destination of content packets



The app consists of several layers which handle a different area of functionality. The block diagram to the right illustrates this separation of concerns.

User Interface

The topmost layer of the EagleChat application is the user interface. In Android applications, the user-facing part is usually a subclass of the Android Activity class. An "activity" displays a user interface on the phone or tablet device, handles events and inputs generated by the user (e.g. screen taps, keyboard entry), and manages the app lifecycle (e.g. creation, destruction). As such, most of the EagleChat consists of various activity subclasses that allow to do one of several tasks. The tasks a user can perform with the app include:

- Configuring an EagleChat peripheral
- Displaying a QR code representing the user's contact information
- Adding another user's contact information to app's contact database
- Choosing a contact to start a conversation
- Sending text messages
- Receiving text messages

Each of these tasks are represented by a dedicated activity subclass. The workflow is designed to be as familiar and intuitive as possible, and should be self explanatory to anyone who has used a text messaging app and registered for a service (e.g. website, app) before. Detailed instructions on how to use the app are included in the Operation Instructions section.

In addition, the application provides a “BURN” feature, which erases all contents of the database (in addition to the index file of the database), and then instructs the connected EagleChat device to BURN as well, effectively “factory resetting” the device.

Database

All data about sent and received text messages and contacts is centralized in an SQLite database [1]. Android components are designed to access a data set through a "content provider", an object conforming to a specific interface that exposes a standard set of data manipulation methods [2]. As such, our database is wrapped in a content provider interface. This allows decoupling between the activities which modify and consume messages and contacts and the exact method in which they are stored.

Peregrine Communication Layer

This layer implements the Peregrine Host Protocol specification as described in the Host Communication section. It is responsible for translating the desired action of another part of the app, for example, retrieving the peripheral's public key, into the message format required by the protocol. These protocol messages are sent to the USB driver to then be transmitted to the peripheral. The Peregrine layer also receives responses from the peripheral and interprets them.

The Peregrine layer is implemented as an Android service and several helper classes. An Android service is an application component that runs in the same process as the activities of the application, but has no user interface. The service is started when an EagleChat peripheral is connected to the host device and acts as the sole manager of the connection.

The service registers with the Database for notifications about new text messages, which are initially marked as unsent. When notified of new messages available for sending, the service formats the message data and transfers it to the peripheral.

When the peripheral receives and decrypts a packet, this data is transferred to the host's USB driver and then to the Peregrine service. The service unpacks the data, and if the data represents a text message, inserts a new row into the database. The service will also send an acknowledgement, or ACK, message back to the sender.

The service expects to receive an ACK from the destination for each message that is sent, and maintains a list of un-ACKed messages. After a timeout to allow for network latency, if a message has not been ACKed it will be resent. Once ACKed, the message's database row is updated to reflect that it has been successfully sent and received by the destination node.

Text message format

The app includes extra data about a text message, in addition to the destination address and the message contents. This extra content is a sequence number, a 32-bit integer that is stored along with the text message in the app's database and serves to distinguish one message from another. This number is encoded in base64 format before transmission, and is decoded from base64 to a 32-bit integer by the destination upon reception.

For example, a text message with sequence number 1 and contents "check out my message, bro", would be encoded like this:

```
mAAAAAQ==check out my message, bro
```

The initial 'm' lets the receiver know this packet should be interpreted as a text message, the sequence "AAAAAQ==" is the 32-bit sequence in base64 format, and the rest of the string is the message contents. The sequence number is encoded in base64 to avoid the byte representing a newline to appear accidentally, as that would cause the following contents to be interpreted as a new transmission.

ACK message format

When the service acknowledges a received text message, it sends an ACK message which references the sequence number of the received message back to the sender.

Following on the previous example, the receiver of a message with sequence number 1 would reply with an ACK message like this:

```
aAAAAAQ==ACK
```

Where the sequence number is encoded as before, but the initial character is 'a' to indicate the contents should be interpreted as an acknowledgement. The content of the message after the sequence number is not important, but having a human readable message is useful for debugging purposes.

USB Driver

The final component of the EagleChat application is the interface between the application and the Android host's USB system. For this purposes, we use the open source library "usb-serial-for-android"

[1] Developer.android.com, 'android.database.sqlite | Android Developers', 2015. [Online]. Available: http://developer.android.com/reference/android/database/sqlite/package-summary.html. [Accessed: 01- May- 2015].
[2] Developer.android.com, 'Content Providers | Android Developers', 2015. [Online]. Available: http://developer.android.com/guide/topics/providers/content-providers.html. [Accessed: 01- May- 2015].
[3] *usb-serial-for-android*. https://github.com/mik3y/usb-serial-for-android, 2015.

6 PROJECT CONSTRAINTS AND CONSIDERATIONS

All engineering projects must address certain factors external or tangential to the design of the system, such as legal or political implications, as well manufacturability and environmental impact. The EagleChat project was developed with the following constraints in mind.

Legal

As a product that uses encryption, EagleChat may be subject to export restrictions as defined by the Bureau of Industry and Security [1]. However, we believe that as a project that simply uses an open source encryption that is not distributed with the source, we believe the project would not actually fall into a restricted category. The library AVR-NaCl could conceivably be restricted from being made available to those in other nations, which could prevent successful replication of the project outside of the United States. We have purposely chosed to make use of an existing library without modifications to avoid falling into th category of restricted encryption software.

Social

As EagleChat is a fully open source and freely available design, it is intended to be equally available to all people. We attempted to create a communication platform that is not controlled by any specific entity, be that a government, corporation, or social group. We believe that EagleChat does not favor any social group, ethnicity, or nation over another.

Political

There is currently a strong debate in the United States [2] and the U.K. [3] over the use of encryption to protect the privacy of communications. The law enforcement agencies of these countries imply that they have the right to inspect all the data and communications associated with their citizens, while many in the technology industry believe that all citizens have the right to protect their digital assets from prying eyes. As EagleChat is designed to allow secure, tamper-proof, private communication, we have clearly taken sides with those who believe digital privacy is our right.

Manufacturability

EagleChat uses exclusively SMD components, and has a low component count. One of our goals when designing the project was that the boards be inexpensive and easy to assemble – even by a private party. In addition, in case the products could not be purchased, the components could be distributed as a small kit – able to be put together by a single person.

Economics

The components of one board, as detailed in the cost section of the report, cost about 25 dollars in total. We feel that the boards are quite economic and affordable by hobbyists.

[1] Bureau of Industry and Security, 'Commerce Control List - TELECOMMUNICATIONS AND “INFORMATION SECURITY”', 2012.
[2] G. Gross and G. Gross, 'Obama administration's encryption concerns meant to start a debate', *PCWorld*, 2015. [Online]. Available: http://www.pcworld.com/article/2896355/obama-administrations-encryption-concerns-meant-to-start-a-debate.html. [Accessed: 01- May- 2015].

[3] S. Landau, 'What David Cameron Doesn’t Get', *Lawfare*, 2015. [Online]. Available: http://www.lawfareblog.com/2015/01/what-david-cameron-doesnt-get/. [Accessed: 01- May- 2015].

A. TEAM MEETING MINUTES

1/15/15

- First day of class
- team assignments and discussions
- class expectations

1/20/15

- Discussed crytpo methods: Elliptic curve diffie-Helman and microcc
- Libraries: libsodium (crytpolibrary)
- Discussed final day of proposal prepration
- Also discussed major systems of project: RF, IR, Crypto, Application Layer, Packet, and USB stack
- RF communicator: nRF240L01+Module
- Discussed possibilities of how to maintain anonymity so that if it's determined messages are sent, it's not clear who the messages came from.
- ways to exchange QR codes

1/22/15

- Meeting cancelled

1/26/15

- Deliverables? What will we work towards?
- avoid unclarity with them
- What will be accomplished by next week? Board order, parts order, code widgets
- Top level flowchart
- Stumbling blocks
- RF ground plane
- problems with USB driver implementation
- How will we do prototyping

1/29/15

- How will we do prototyping
- Present app flowchart

- Protocols used/to use?
- AODV
- mesh network-existing
- board order
- ordering from mouser
- build chain setup
- difficulties with Atmel
- uploading code to board
- Next week
    - parts in
    - boards and some testing
    - packaging dev
    - Final report

2/2/15

- mesh network-existing
- Discussed more about crypto methods
- Elliptic curve diffie-Helman and microcc
- Discussed utilization of app
- Discussed expected issues with implementation of radio
    - Size
    - Range
    - Interference

2/5/15

- Discussed utilization of app
- Talked planned implentation of routing
- problems that may arise
- system utilized for routing
- how many nodes to use
- Discussed USB and difficulties with CDC
- Discussed future packaging ideas
    - dongle that attaches to phone
    - container that looks similar to mint container
    - velcro application
- updates on status of boards

2/9/15

- block diagram revamp
- document with data/measurements that support operation of subsystem
- how to break up subsystems
- power consumption and battery life
- think about specifications
    - range of 100-200ft
    - 21dB antenna

2/12/15

- Finalized specifications for presentation
- Continued working on code for radio and crypto
- Discussed issues with board costs
- Worked on budget and Sam presented costs for production to group

2/16/15

- Meeting cancelled

2/19/15

- App presentation
    - Burn Function
    - Adding Contacts
    - Sending and receiving messages
    - layouts

2/23/15

- Presented specifications
- Presented board
- Early build app overview
- Overview of QR code and adding new contacts
- Burn function
- Cipher used Salsa 20
- Prove encryption key erasure
- look around kernel to see deleted and overwritten file
- USB
- Be able to explain what to do/have working prototype next week

2/26/15

- updates on status of boards
- Discussed what to talk about during prototype demonstration
    - SHA
    - USB
    - Radio
    - Packaging
- Did more work on coding together for networking and routing
- Started to write some of the data necessary for supplemental documentation for design proposal
- Discussed updated board design and how SRAM will be implemented

3/2/15

- Discussed crypto
- Discussed salt library
- Discussed board layout issues
- Went over demonstration
    - have phone talk to board
    - board will receive message from terminal and show encrypted text
    - Prove that when it decrypts, a stored version of data is being sent
    - radios communicating
- Discussed converting from C to C++ with library

3/5/15

- updates on status of boards
- Prepared final paperwork for prototype demonstration
- Demonstrated prototypes

3/9/15

- Meeting cancelled

3/12/15

- updates on status of boards
- Working to discover encryption overhead
    - 30 bytes enc. overhead, 1 byte sender, 1 byte receiver, 16-bit CRC, and 16-bit packet number/flags/extras.

- Discussed characteristic for design of final board and packaging
- clearance for antenna and end of screw on bottom of casing
- mounting hole diameter for antenna screws
- Discussed problems implementing acks
- perhaps receiver has to have delay in main loop for sender to get acks

3/16/15

- Spring break

3/19/15

- Spring break

3/23/15

- Considered how future implementation of AODV documents will be done for project
- Succesfully got acks working
    - Problem was value used to store wait time for ack was not of correct type and also true time to send ack is too much
- Working on possible reference implementation of AODV
- Discussed date for new board orders

3/26/15

- Continued work on implementation of FIFO
- Discussed sending packets longer than 66 bytes
- Discussed CRC chip problems
- when sending long packets, large amounts of time reported to set CRC okay register
- possible that use of outside CRC check will be used
- Issues sending packets and data over PuTTY, so talked about using microcom instead
- Discussed problems implementing main as a pure C file and how radio file must be cpp file due to accessing of class meth-ods

3/30/15

- updates on status of boards
- Issues with sending packets
- dropped packet percentage lowered
- Layer progress

    - at level 2
- Discussed progress on SRAM and issues setting correct pin assignments

4/6/15

- Made progress on routing
- working on notification that packet has been sent successfully between nodes
- Discussed possible distances or sending of packets

4/9/15

- updates on status of boards
- discussion for SSK use
- sent over the network to the receiver (because it's calculated from the sender's private key and the receiver's public key)?
- Use nodes public key to compute using a private key?
- more FIFOs implemented for host messages
- More disscussion about how to code read and write functions for SRAM
- Discussed app demo for test run presentation

4/13/15

- Discussed routing progress
- Talked about issues with SRAM implementation
- Data being sent twice when it should be sent once
- seeing erroneous output
- Gave updates on group members
- Discussed moving code to C++

4/16/15

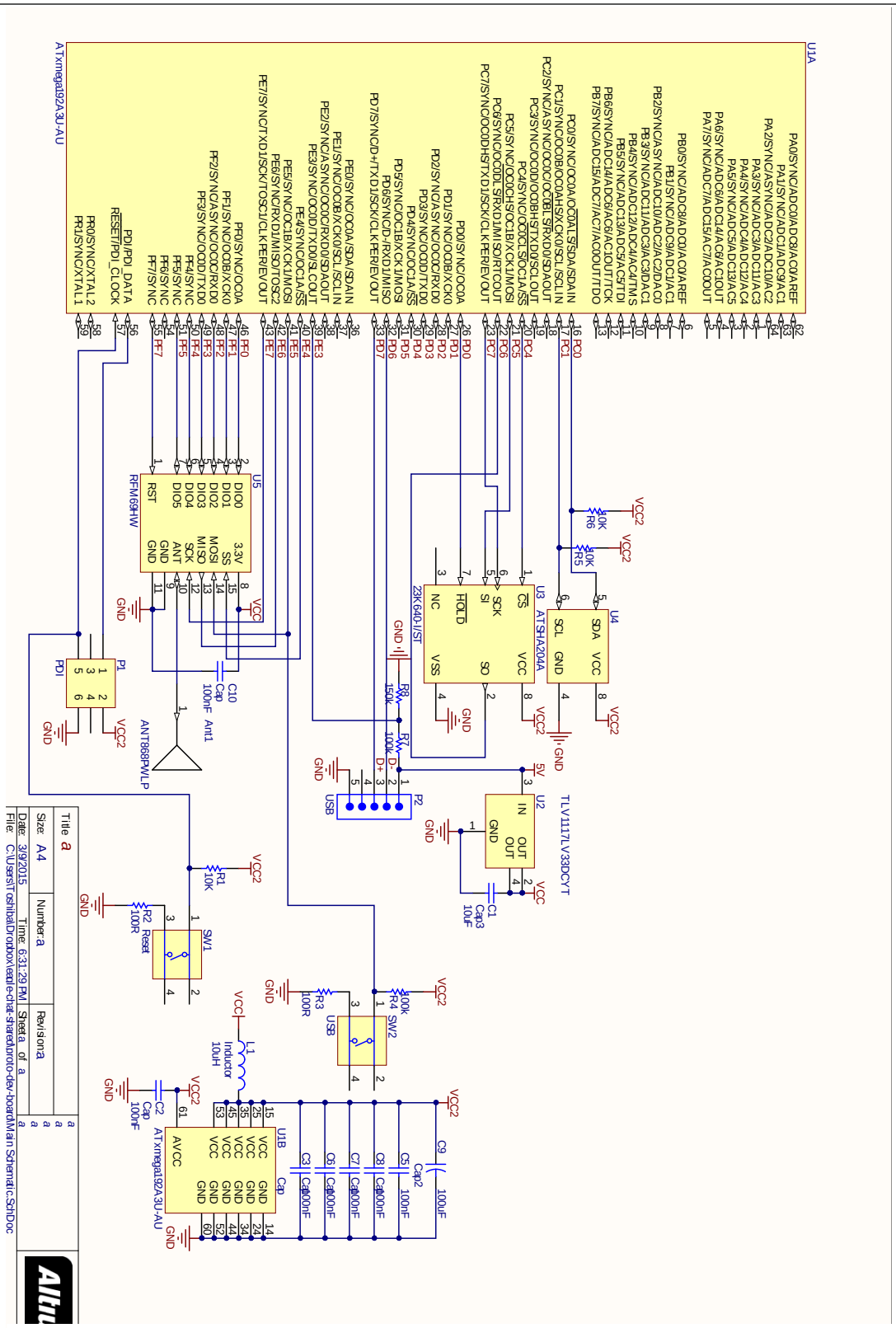
- Trial Project Presentation
- Tested range of devices

4/20/15

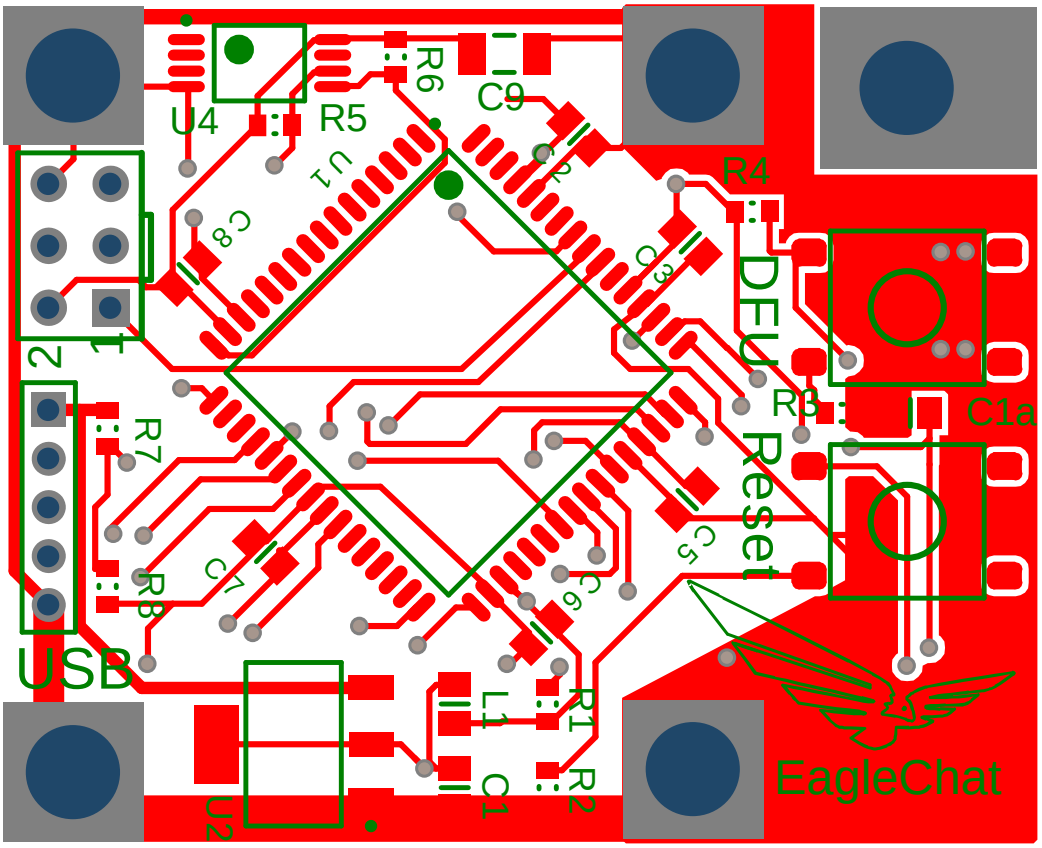
- Project Demo
- Discussed presentation
    - What it will do?
- Routing reported finished

- Discussed casing
- Discussed polishing application
- fixing bugs and adding new features
- Working on high level Ack functionality
- Range test results roughly 240ft
- Future security holes

B. SCHEMATICS AND LAYOUTS

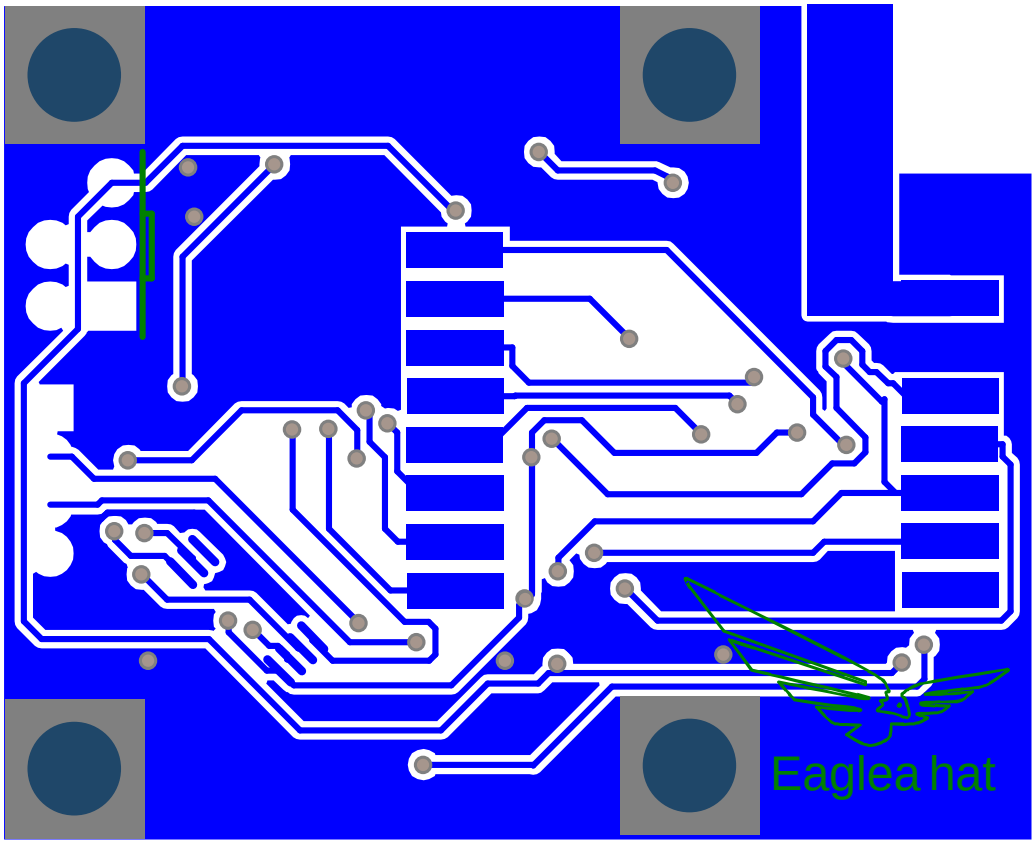


EagleChat Board v1.0 Schematic



EagleChat Board v1.0 Top Layer

[APPENDIX C]



EagleChat Board v1.0 Bottom Layer

C. Cost and Bill of Materials

Bill of Materials for EagleChat Board v1.0

Mfr. #	Manufacturer	Description	Part #	Qty.
GRM31CE70G107ME39L	Murata	SMD/SMT 1206 100uF 4volts X7U + - 20%	C9	1
ADTSM63SVTR	Apem	Tactile Switches SW TACT SPST	SW1, SW2	2
ANT-868-PW-LP	Linx Technologies	Antenna 1/4 Wave 868MHz	ANT1	1
VJ0805G106KXQTW1BC	Vishay	SMD/SMT 0805 10uF 10volts X5R 10%	C1	1
VJ0805Y104MXXAC	Vishay	SMD/SMT .1uF 25volts X7R 20%	C2, C3, C5-8, C10	6
MLZ2012N100L	TDK	Fixed Inductor 500mA 10uH	L1	1
WCR0603-10KFA	TT Electronics	SMD 0603 10 KOhm 1% Tol AEC-Q200	R1, R5, R6	3
CRCW0603100RFKEA	Vishay	SMD 1/10watt 100ohms 1%	R2, R3	2
CRCW0603100KFKEA	Vishay	SMD 1/10watt 100Kohms 1%	R4, R7	2
CRCW0603150KFKEA	Vishay	SMD 1/10watt 150Kohms 1%	R8	1
ATxmega192A3U-AU	Atmel	MCU AVR8 192KB FLSH 16KB	U1	1
TLV1117LV33DCYR	Texas Instruments	LDO Voltage Regulators 1A 3.3V	U2	1
23K640-I/ST	Microchip	SRAM 64K 8K X 8 2.7V	U3	1
ATSHA204A-XHDA-T	Atmel	CryptAuthEE SHA256 TWI	U4	1

Prototyping costs

- Electronic ICs and passive components - $79.34
- RFM69HW radio module x 4 - $19.10
- PCBs x 6 - $29.00

Total: $127.44

Final design construction costs

- Electronic ICs and passive components - $94.29
- RFM69HW radio module x 6 - $27.70
- PCB x 6 - $22.90

Total: $144.89

Component expenses per unit

- Electronic ICs and passive components - $15.72
- RFM69HW radio module - $4.62
- PCB - $3.82

Total: $24.15

Total project cost: $286.80