Spring 6-2018

Crab Tracker Documentation

Noah Strong
Western Washington University

Margot Maxwell
Western Washington University

Chloe Yugawa
Western Washington University

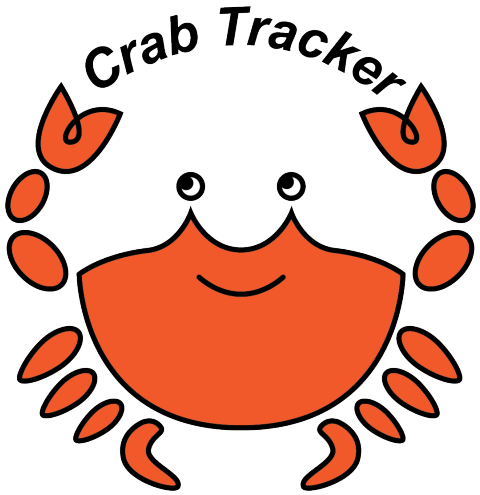
Elizabeth Schoen
Western Washington University

Setup and User Guide

Crab Tracker

Noah Strong

Revision 1.1
May 28, 2018



Contents

1 Introduction **2**

2 Hardware Setup **2**

2.1 Required Hardware Components 2

2.2 Hydrophone Array 2

3 Software Setup **3**

3.1 Loading Software 3

3.1.1 Downloading the source code 4

3.1.2 Flashing code to the Arduino 4

3.1.3 Updating code on the Pi 4

3.2 Configuring Settings 5

3.3 Configuring a new Raspberry Pi 5

1 Introduction

The Crab Tracker project was designed as a cost-effective means of remotely tracking crabs through acoustic signals. Small piezoelectric transmitters can be waterproofed and attached to crabs, and their intermittent signals can then be received by a set of four piezoelectric receivers configured in a square array.

This document will detail the setup and configuration of various elements of the product as they were originally intended to be used. As of the time of this writing, some of the hardware elements of this project are still in development and therefore details about them will not be discussed in this document.

2 Hardware Setup

2.1 Required Hardware Components

The project was initially built with, and officially supports, the following devices:

- Raspberry Pi 3B
- Raspberry Pi 7" Touch Screen
- Arduino Nano

Additionally, there are transmitters and receivers that were custom designed for this product. Specific details about these components are still subject to change as of the writing of this document, and they will be detailed elsewhere as they are formalized. In order to use the Crab Tracker product, it is necessary to fabricate receiver and transmitter microcontrollers per the specifications detailed in those documents. Some of the components must be coated entirely in epoxy for waterproofing.

2.2 Hydrophone Array

The Crab Tracker project uses a set of four receivers to perform triangulation of underwater transmissions. The configuration of the transmitters is extremely important, as an incorrect configuration of the hydrophone array will lead to incorrect triangulation results. The hydrophone array may be attached to a personal watercraft, such as a kayak.

Hydrophones must be placed on a square frame with one hydrophone at each corner. The exact side lengths of the square may vary, but are expected to be roughly 1 meter each. Once put in place, the measurements must be communicated to the software to ensure accuracy of readings. See Figure 1 for an example of the hydrophone configuration. Note that the side lengths shown are 1 meter, but this is only an example.

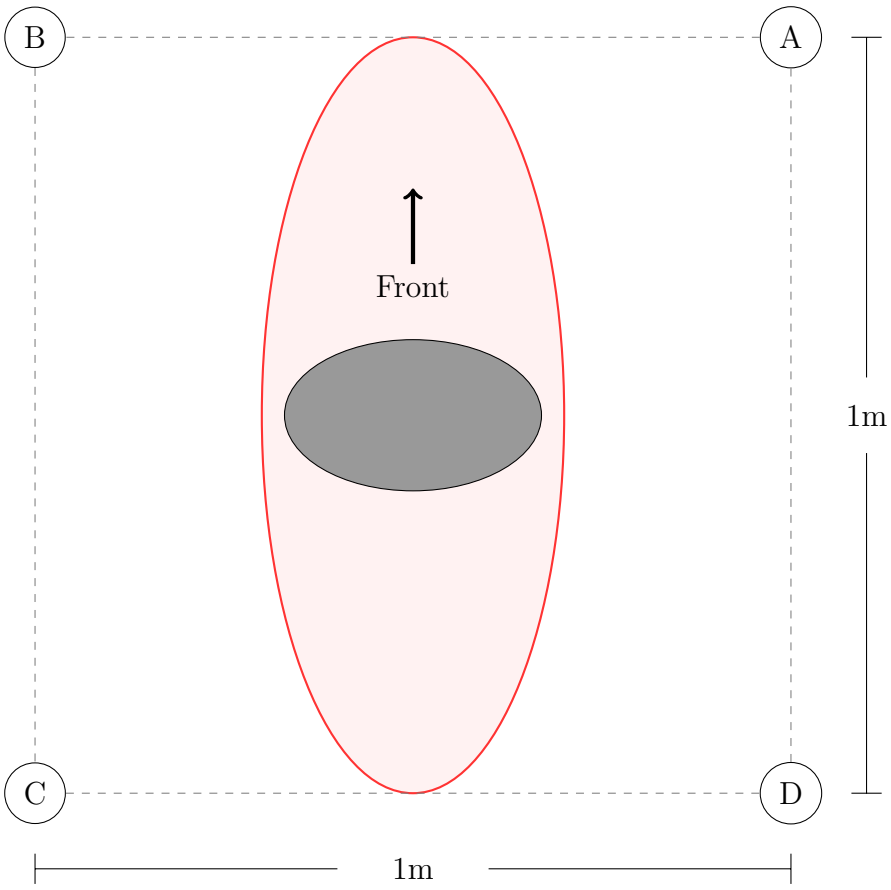


Figure 1: The hydrophone array attached to the kayak.

3 Software Setup

3.1 Loading Software

It may at some point be necessary to add or replace code on one of the devices, such as when new hardware is acquired or the software is updated. To do this, the first step is to download ("clone") the most recent copy of the `master` branch from the Crab Tracker GitHub repository. It is possible to download a ZIP file containing the contents of the repository from the GitHub website. Another option, however, is to use the `git` command-line utility to make a clone. To see if you have Git installed, open a shell/terminal and type `git`. A usage page should be displayed. If not, look online for instructions on how to install Git on your operating system.

The steps shown in the remainder of this section are meant to be run on a UNIX-like operating system (such as Linux, MacOS, or BSD). However, equivalent commands exist for other systems, such as Windows.

3.1.1 Downloading the source code

The following instructions apply to both personal computers (which may be used to upload code to the Arduino), and to the Raspberry Pi that is to be used in the field.

If you have not cloned the repository on this computer before, run:

```
$ git clone https://github.com/cabeese/crab-tracker.git
$ cd crab-tracker
```

If you have already cloned the repository at some point previously, `cd` into the project's directory and run `git pull`.

3.1.2 Flashing code to the Arduino

In order to upload binaries to the Arduino, you will need the Arduino software from their website.

With the software downloaded, plug the Arduino into your computer.

Next, open the `src/arduino-src/arduino-src.ino` file in the Arduino editor. Under the Tools menu, set the *Board* to *Arduino Nano* and set the *Port* to the port you plugged the Nano in to.

Now click the UPLOAD button. The compile and upload process should finish in a few seconds.

That's it!

3.1.3 Updating code on the Pi

To update the source code on the Raspberry Pi, connect the Pi to a power source and to your network (e.g. via an Ethernet cable). Attaching an external keyboard is highly recommended as well, though it is possible to use the on-screen keyboard if you do not have access to a physical one.

With the Pi connected, open up the Terminal app and use the commands listed in Section 3.1.1 to download a copy of the source code.

Next, run the following commands to make the executable file.

```
$ cd crab-tracker
$ cd src/pi-src
$ make
```

The output of the `g++` commands will be shown. When the shell prompt reappears, you can start the program like so:

```
$ ./crab-tracker
```

3.2 Configuring Settings

The main program on the Raspberry Pi has a few configuration options that can be loaded at runtime (i.e. when the program starts up, as opposed to when the source code is compiled into an executable file). The configuration options are located in the `/etc/crab-tracker.conf` file. Use your favorite text editor to make changes to this file as needed. The format for this file is simply a set of line-delimited key/value pairs, where each key is an alphanumeric string and each value is an integer. A single space character separates the key and value. Lines beginning with a hash (`#`) character are comments and will be ignored by the program. A sample version of the configuration file is located at `src/crab-tracker.conf.example`. An updated list of possible configuration parameters can be found in the `src/pi-src/README.md` file.

3.3 Configuring a new Raspberry Pi

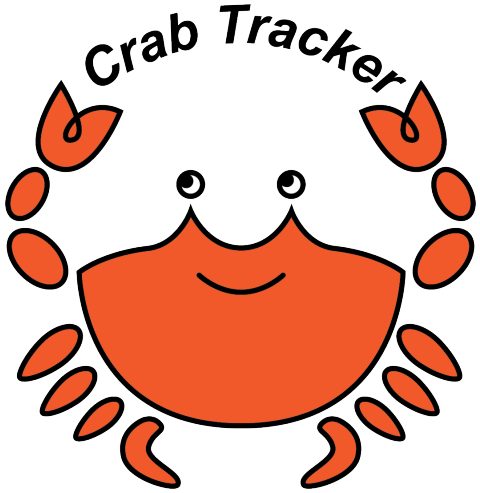
If starting with a brand-new Raspberry Pi or simply a new Micro-SD Card, follow the instructions found in doc/model-B-config.md.

Technical Documentation

Crab Tracker

Noah Strong

Revision 1.4
May 28, 2018



Contents

1 Introduction **2**

2 Background and Overview **2**

2.1 Motivation For The Project 2

2.2 Description of Components 2

3 Electronics Hardware **3**

3.1 Transmitters 4

3.2 Receivers 4

4 Computer Hardware **5**

4.1 Timestamp Recorder 5

4.2 Data Processor 5

4.3 Wiring Diagram 6

5 Software **7**

5.1 Timestamp Recorder Software 7

5.1.1 Storage, Setup and Loop 7

5.1.2 SPI Handling 7

5.1.3 Pin Change Detection and Timestamping 8

5.2 Data Procesesor Software 9

5.2.1 Overview of Files 9

5.2.2 Processing and Structure 10

6 References **11**

Glossary **11**

1 Introduction

The Crab Tracker project was designed as a cost-effective means of remotely tracking crabs through acoustic signals. Small piezoelectric transmitters can be waterproofed and attached to crabs, and their intermittent signals can then be received by a set of four piezoelectric receivers configured in a square array.

Because this product was built with very few "off-the-shelf" components, it is somewhat complex and many of the finer details may be difficult for future collaborators to infer based on the existing documentation. This document aims to provide a technical overview of the project, including the rationale for some of the design choices, in hopes of giving the reader a deeper insight into the inner workings of the product.

2 Background and Overview

2.1 Motivation For The Project

While solutions already exist to aid in the tracking of aquatic animals, they are often prohibitively expensive without significant financial resources. Additionally, many of these products require the use of somewhat larger watercraft and multiple people. The Crab Tracker product has been designed with cost and simplicity in mind, and can be operated by a single person in a small boat such as a kayak. A majority of the components, including both hardware and software, are custom-made. However, much of the product, especially the software, has been designed in a "modular" fashion, meaning that the various components are not tightly coupled. Simply put, future researchers should be able to exchange various components to better fit their needs with relative ease.

2.2 Description of Components

Our solution uses high-frequency audio broadcasts to transmit data about each crab to a central receiving station (likely mounted on a kayak or similar water vessel). Each transmitter has a unique identifier (an integer value) that determines the duration and pattern of its broadcasts. See the iCRAB Transmission Protocol definition for further details. At the receiving station we have four underwater microphones (known as hydrophones) built using piezoelectric receivers which are tuned to detect the signals that the crabs' transmitters broadcast. These hydrophones are arranged in a square shape with equidistant side lengths, and all are to be submerged an equal depth into the water.

As acoustic transmissions reach the hydrophone receivers, two distinct processes must be run. The first process is to detect incoming signals and label them with accurate timestamp values. Once this is done, software must run calculations using these timestamps to decode the unique identifiers of the broadcasting transmitter(s) and to determine the direction from which the sound(s) originated. Because the latter task requires a mild amount of computation time, and because the former task

requires low-latency monitoring of the input devices, these two tasks are to be performed on separate physical computers.

The first of the two devices, referred to as the Timestamp Recorder (4.1), has the simple but important task of applying timestamp labels to incoming data. As of the current revision of this document, we have tested the system with an Arduino Nano, a small, inexpensive microcontroller with a clock speed of 16MHz. Once broadcasts are detected and labeled with a timestamp, the device can send this data to the Data Processor (4.2) that will run various calculations and display crab locations to the user. We originally investigated using the popular Inter-Integrated Circuit (I^2C) protocol for this, but unfortunately I^2C can't be interrupted during a transfer, and therefore the device would be effectively frozen (and thus unable to detect incoming data) any time a transfer was in progress. If any signals were to reach the device during a transfer, their timestamps would be delayed and therefore inaccurate. Instead, we opted to use a similar protocol, Serial Peripheral Interface (SPI). The advantage of SPI is that data can be placed into a separate physical bus on the device when it is ready to be transmitted, and the Data Processor can fetch that data without affecting the detection and timestamp code.

The Data Processor (4.2), which runs calculations and displays results to the user, requires a slightly more powerful computer than the Timestamp Recorder. We opted to use a Raspberry Pi Model 3B for this task, as these machines are inexpensive, reasonably powerful, well-supported, SPI-compatible, and easy to develop on. We also purchased a 7-inch touchscreen made specifically for the Pi as a means for the user to interact with the system in the field. The Pi is directly wired to the Arduino for SPI. See Figure 1 for an illustration of these components.

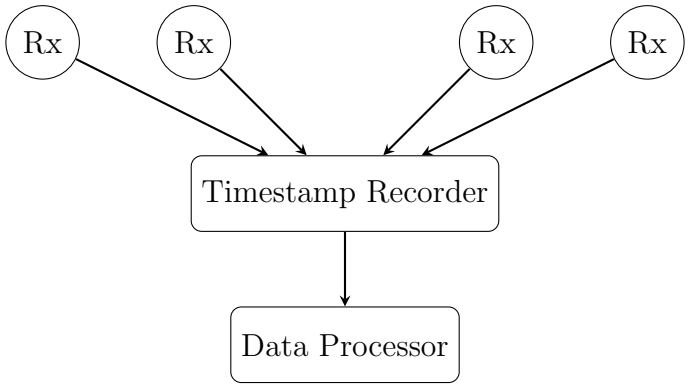


Figure 1: A simple diagram detailing the main components. Each circle labeled Rx is a receiver, consisting of a piezoelectric transducer and some processing circuitry.

For a more detailed version of this figure, see the Wiring Diagram (3) in Section 4.3.

3 Electronics Hardware

While both the Arduino and the Raspberry Pi were off-the-shelf products, the transmitter and receiver hardware was all custom designed and fabricated for use in this project. Clearer specifi-

cations, including detailed diagrams, are given in other dedicated documents, so the information presented here will only be a high-level overview of the main components.

3.1 Transmitters

The transmitters are small piezoelectric transducers that oscillate at an ultrasonic frequency. In our work, this frequency has been in the 40-60kHz range, depending on the resonant frequency of the transducer itself. Though we have not established bounds on our range of potential frequencies, the actual frequency that is selected shouldn't matter so long as all of the hardware is capable of operating at, and appropriately tuned to, the frequency (through filtering, hardware characteristics, and so on). For example, some hardware may limited to roughly 150kHz because the microcontrollers won't have a high enough clock speed to drive the receivers at that rate.

Each transmitter is attached to a small Printed Circuit Board (PCB) designed specifically for the project. The PCB can power the transducer by feeding it a current, effectively turning it off or on as needed. Each board will be programmed with a unique identifier (a numeric value). The ID determines the pattern of pulses that the transmitter should emit. For more details on this, see the iCRAB Transmission Protocol document.

3.2 Receivers

The receivers include small piezoelectric transducers that are able to detect vibrations in the same range as the transmitters. A preamplifier steps up the signal from the transducer before sending this signal along to a band-pass filter. The band-pass filter restricts the frequencies that the hardware will detect, meaning that a lot of irrelevant noise can be discarded. After being filtered, the signal is then amplified. A potentiometer on the board allows for variable amplification. Next, the signal is rectified (effectively producing the absolute value of the incoming sine wave). The low pass filter transforms the signal from a sinusoidal wave into a single pulse. Finally, the Schmitt Trigger acts as a variable threshold, preventing a noisy rising or falling edge from being interpreted as a series of short rising and falling edges. See Figure 2 for details on this pipeline.

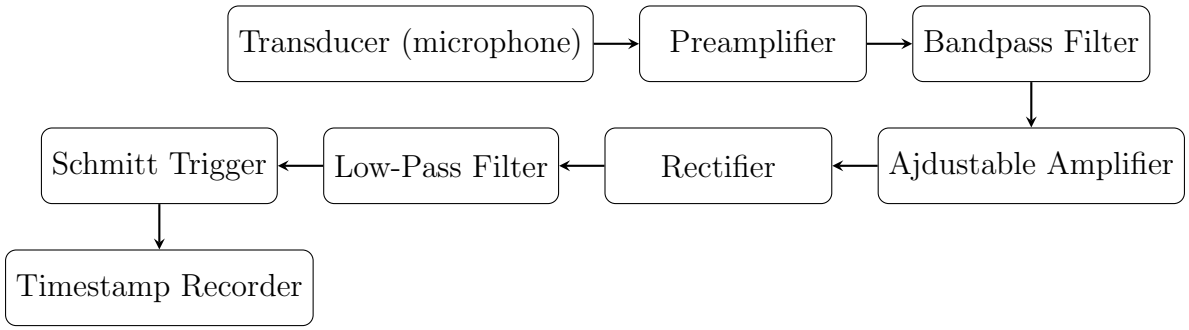


Figure 2: The receiver hardware pipeline.

The resulting digital signal is sent over a wire to the timestamp recorder detailed in section 4.1.

4 Computer Hardware

Signals detected through the receiving hardware described in Section 3 are delivered to more advanced microcontrollers and computers using common General Purpose Input Output (GPIO) pins. Two separate devices are used for the signal processing. The first, referred to in this document as the Timestamp Recorder (4.1), provides a low-latency means of detecting the time of an incoming signal. This device must be able to reliably detect when a signal arrives with sub-millisecond precision. Once timestamped, the data can be transferred to the Data Processor (4.2). This device decodes the ID from each transmission and displays the crab's direction to the user.

4.1 Timestamp Recorder

The timestamp recorder reads a digital signal on a GPIO pin and records the "time" at which the signal changed from low to high or vice versa. In this case, the "time" need not have any meaningful relationship to "wall clock" time, so long as the timestamps have some meaning relative to each other. For example, the time associated with a state change could be measured in microseconds since the board reset. Since the software (described in Section 5.2) is primarily concerned with the time separation between events, the epoch is irrelevant. It is important, however, that the Timestamp Recorder and the Data Processor computers agree on the units used, because some of the later processing will be based on the speed of sound in water, measured in meters per second.

In addition to providing reliable timing measurements when input pin states change, the timestamp recorder must also be able to communicate the timestamped data to another device. Use of a non-blocking protocol such as SPI is necessary here so that the data transfers do not have a large negative impact on the device's ability to continuously detect changes.

4.2 Data Processor

The majority of the processing software exists on a separate device, referred to as the Data Processor. This machine reads timestamped data from the Timestamp Recorder (4.1) and stores it in its own local storage until it has enough data to run some calculations. This device is also the user's primary interface for interacting with the system in most use cases. Equipped with a touch-screen or other means of displaying data, the computer is able to display the unique identification numbers and relative locations of crabs as they are detected.

The software written for this project assumes that the Data Processor is a Linux-based machine with an SPI interface. Specifically, it has been tested on a Raspberry Pi 3B, which is a very suitable device given its low cost, built-in GPIO pins, and adequate processing power.

4.3 Wiring Diagram

This section demonstrates in greater detail how the various components are connected to each other. Note that though we use the official 7" touchscreen built for the Pi, wiring for the screen is not included in the following diagram. In Figure 3, the Pi (the SPI master node), Arduino (SPI slave node), and a receiver are shown. Note that each of the other receivers could be connected in the same way that Receiver 1 is connected; that is, they need to be tied to the common ground and they need their signal pin attached to the appropriate input pin on the Arduino.

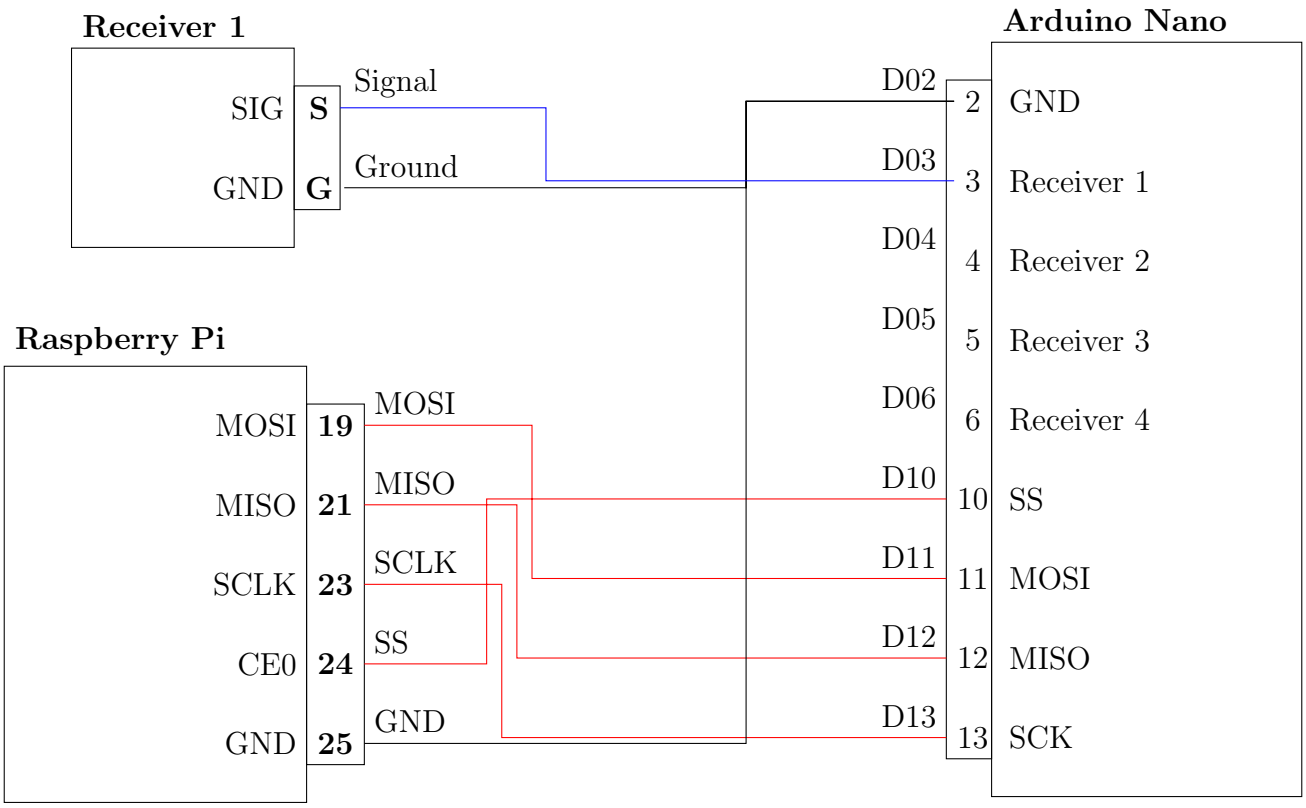


Figure 3: A wiring diagram for the devices at the receiving station.

SPI uses four shared wires. Briefly, the purpose of each is:

- `MOSI` Master-In-Slave-Out – data from the slave to the master (such as timestamps and pin values).
- `MISO` Master-Out-Slave-In – data from the master to the slave (used for flags, in our case).
- `SCLK` The clock. Driven by the master node.
- `SS` or `CE` is the Slave Select or Chip Enable line. SPI supports multiple slave devices, so the master node uses the Chip Enable line to turn on one slave at a time to communicate. In our case, since there is only one slave device, we know that the `CE0` line will always be active during a transfer.

5 Software

Wherever possible, the software for this project was written in such a way that various elements may be altered or entirely replaced without requiring major changes to the rest of the codebase. The hope is that it is flexible enough to be adapted for new scenarios and hardware setups in the future. That said, it has only been tested with the hardware detailed elsewhere in this document, so its portability and modularity is unproven.

In this section, we will examine the software that runs on the two devices described in Section 4.

5.1 Timestamp Recorder Software

The relevant source code can be found here: `src/arduino-src/arduino-src.ino` in the project source or on GitHub.

The Timestamp Recorder (4.1) software has two primary functions: (1) record timestamps of signal changes (i.e. when a pin goes high or low) and (2) transfer the timestamps and pin values to the Data Processor.

The `arduino-src.ino` file is heavily documented, so this section will explore the major ideas without providing extensive detail about individual lines. Specifically, it will examine in broad strokes the data formatting, data transfer, and input pin change detection.

5.1.1 Storage, Setup and Loop

Data is stored in a bounded buffer where each entry contains a timestamp value and the state of the pins at that timestamp (i.e. which pins were high and which were low). The timestamp is a 32-bit `unsigned long` and the pin values are all stored in a single `byte` where each bit corresponds to a single pin. The `pinvals` are shifted such that the lowest order bit corresponds to pin D3. A 1 in position 0 means that pin D3 is high; 0 means low. (Note that a few of the low-order pins on the board are reserved for special purposes, so the receivers are to be attached to pins D3 through D6.)

The `setup()` function sets the GPIO pins on Port D (pins 0-7) as input pins, meaning that another device (in this case, a receiver, as described in section 3) can drive the line high or low. The Timestamp Recorder can detect these changes when the pins are in input mode.

The `loop()` function runs indefinitely and on each iteration it can fill the SPI data register and/or detect and timestamp a change on one of the pins. We will examine both of these primary functions in more detail in the following two sections.

5.1.2 SPI Handling

SPI is a synchronous serial communication interface that is natively supported by both the Arduino and the Raspberry Pi. It supports a single "master" node and multiple "slave" nodes. When the

master node is ready to transfer data, it selects one of the slave devices by making its Chip Enable/Slave Select line active and then begins pulsing the clock. More details about the SPI protocol can be found online.

The Arduino Nano has a built-in SPI bus that supports 8-bit data transfers. On a transfer, the master node provides a clock pulse and on every pulse, one bit is shifted out of the data register (sent to the master) and one bit is shifted in (sent from the master).

On each iteration of the loop, the Arduino checks to see if the "End of Transmission" flag is set, which indicates that a transfer has completed. If it is set, the board can fill the Data Register with the next byte to be sent. The next time the master requests data, the newly-loaded byte will be shifted out to that device.

As mentioned, the built-in SPI hardware supports 8-bit data transfers. However, each segment of data that we wish to transfer contains a 32-bit timestamp and an 8-bit state, meaning that it takes 5 individual transfers to send a segment (or "block") of data. When the master node requests data, the slave sends the pin values on the first transfer, and then sends the timestamp in 8-bit chunks over the course of the next four transfers, starting with the lowest order byte and working up to the highest order byte. Note that this means both the master and slave nodes must be in sync with each other, as there is nothing to indicate what kind of data is sent in a single transfer. The two devices may get out of sync if one is interrupted between transfers, such as when the master's software is stopped unexpectedly. In this case, when the master's software restarts, it will expect to read pin values on the first transfer, even though the slave node may be prepared to send part of a timestamp.

To solve this issue, the SPI master node can send flags to the slave that will alter the slave's state or behavior. Certain values that the master can send will induce specific behaviors on the slave. For example, the `SPI_ECHO_REQUEST` flag, when sent, will cause the slave to send the `SPI_ECHO_RESPONSE` value on the next transfer. This is used when the master's program starts up in order to ensure that the two devices are correctly connected. The `SPI_RESET` flag will re-position the de-facto "read heads", meaning that it will send a new state starting with the pin values on the next transfer. This functionality is utilized to ensure that the two devices stay in sync when one of them restarts while the other continues to run.

5.1.3 Pin Change Detection and Timestamping

Pin change detection is comparatively straightforward. Each time the loop runs, the current state of the relevant pins (D3..D6) is compared with the state recorded from the previous iteration. If the two states differ, this indicates that at least one pin changed from low to high or vice versa. When this is the case, the device stores the new state in a circular buffer along with the current time.

As of the current revision of the document, the timestamp is generated from the `TCNT0` clock, which increments every four microseconds. This is an inadequate granularity for the purposes of this project, so future revisions of the code will instead read from a source that can generate ticks at a higher rate (for example, a clock cycle counter, or a timer set up to increment once every microsecond).

Note that no matter the units used, we should expect to store no more than 32 bits of information for any given timestamp. Therefore, $2^{32} \approx 4.3$ million is the highest value we can represent. Limited precision means that a source that increments frequently will overflow more rapidly than a slower source, which leads to an important trade-off. A 4-millisecond timer will overflow every $4 \times (4.3 \times 10^6)$ milliseconds, or roughly once every 4,772 hours. However, a timer that increments once per microsecond will overflow every 1.1 hours (or roughly 72 minutes). Currently, overflowing timer values are not handled by any part of the software; instead, any such data will automatically be thrown out. In other words, if a transmission is received while an overflow occurs, such that some timestamps have extremely high values while others have extremely low (near-0) values, that transmission will be ignored. No attempt is made to account for the overflow, though to do so is well within the realm of possibility for future work.

5.2 Data Procesesor Software

The relevant source code can be found in multiple files in `src/pi-src/` in the project source or on GitHub.

5.2.1 Overview of Files

The primary functions of the various files are as follows:

- `main.cpp` – The main loop of the program. This file periodically polls the SPI slave device and then runs routines to process any data that is fetched.
- `data_collection.cpp` – When SPI data is read, the functions in this file parse and store the data in meaningful structures. For example, when a rising and subsequent falling edge are detected on a given pin, a `ping` object is stored containing these timestamps. The data collection code also steps through the collected data to determine if enough pings have been collected for the direction algorithm to run. For more details, see Section 5.2.2.
- `direction.cpp` – Using an algorithm written by one of the team's developers, Chloe Yugawa, this code determines the relative direction of a crab given the timestamps of its pings. That is, it performs a Time Difference On Arrival calculation based on ping times. It returns a direction relative to the front of the boat in cylindrical coordinates in the form (r, θ, z). For more information, see the Triangulation Algorithm[5] document.
- `spi.cpp` – This file facilitates and abstracts away the communication with the SPI slave device. Routines in this file can determine if the devices are properly connected and will also ensure that the master and slave devices are in sync with each other.
- `uid.cpp` – Each transmitter broadcasts in a specific pattern based on its unique identifier. The code in this file examines the duration of and delay between two pings and determines the UID(s) that these time values encode.

- `util.cpp` – Various helper methods are located here, including binary printing functions and configuration setting loading.
- `/etc/crab-tracker.conf` – Configuration options are located in this file in the `/etc` directory. Data is stored in key/value pairs and is read on program startup. Parameters include what kind of data should be logged, the distance between the corners of the hydrophone array, and more.

5.2.2 Processing and Structure

Each file contains extensive documentation, so the finer details about various functions and variables will not be duplicated in this document. Instead, we'll take a broad look at how these pieces fit together.

When data is read from the Timestamp Recorder, it is translated from raw pin states into something more meaningful, such as a "ping" (that is, a single broadcast as measured by the rising and falling edges of a signal on a single pin). Pings are collected in a set of buffers, with one buffer for each pin. These buffers are managed by the `data_collection.cpp` code. Since the iCRAB Transmission Protocol mandates that every transmission consists of two pings, no calculations are performed until a pair of pings is detected by all four receivers. Each time a new ping is detected, these arrays are searched for matching pairs. If there are two pings in every buffer, and they all share the same ID and are spaced apart correctly (temporally, based on iCRAB Transmission Protocol specifications), they are all returned as a set of 8 pings to the main loop of the program, which then runs the direction calculation algorithm. If the direction algorithm is able to determine an approximate location for the signal*, the direction and associated ID can be reported to the user.

* Note that the direction algorithm will intentionally fail if it is given impossible data. That is, if the timestamps are too widely separated temporally, it will assume that the measurements are not from the same transmission or are otherwise incorrect and will therefore abort the direction calculation.

6 References

Unless otherwise noted, the documents referenced in this section are published on GitHub, or else in the same location as this PDF.

1. Strong, N. *Transmission Protocol – iCRAB.* 2018. PDF.
2. Strong, N. *RFC 1: Collisions in Delay-Based ID Encoding Protocol.* 2017. PDF.
3. Strong, N. *RFC 2: Boolean Value Encoding in Transmission Protocol.* 2018. PDF.
4. Yugawa, C. *RFC Stats.* 2017. PDF.
5. Yugawa, C. *Triangulation Algorithm.* 2017. PDF.

Glossary

GPIO General Purpose Input Output. 5, 7

I^{2}C Inter-Integrated Circuit. 3

iCRAB Transmission Protocol Our custom encoding protocol for transmitters. See Reference 1 for complete documentation. 2, 4, 10

PCB Printed Circuit Board. 4

SPI Serial Peripheral Interface. 3, 5–9

Transmission Protocol — iCRAB

Noah Strong

January 27, 2018 – v1.1

Contents

1 Introduction **2**

2 Background **2**

3 iCRAB Protocol Definition **3**

3.1 Overview 3

3.2 Requirements, Rationale and Related Information 3

3.3 Preamble to Definitions and Specifications 4

3.4 iCRAB Definitions and Specifications 6

4 References **7**

A Glossary of Terms **7**

1 Introduction

The Crab Tracker project aims to provide a simple, efficient, reliable, and cost-effective method for tracking crabs underwater. There are no accepted standards that we're aware of for achieving the results we hope to achieve, and to base our work too heavily off the work of existing products would violate the clauses in the licenses of those products that protect against reverse engineering. For these reasons and more, we must define our own technologies and protocols. Central to the project is the protocol that will be used to relay information from transmitters (attached to crabs) to the central receiver (affixed to a water-going vessel, such as a kayak). Documented herein is that protocol, as well as the motivations and requirements for many of the decisions behind it. As of this writing, **the protocol is still subject to change.** We may find shortcomings or other problems with the protocol during the prototyping stage of the product, at which point adjustments will be made. This document will be updated as needed to reflect these changes, and should always be treated as the official documentation for the protocol.

One of the major requirements of this project is the ability for each individual transmitter to be uniquely identifiable. Therefore, we must encode the device's unique identifier (herein referred to as the ID or UID) in each signal that the device broadcasts. Additionally, we may want to encode some accelerometer data to indicate that a given transmitter has stopped moving. This data will be encoded as a boolean value, which we refer to as `inert`.

Finally, because all transmitters transmit at the same audio frequency and their broadcasts are uncoordinated (no transmitter is aware of the broadcasts of adjacent transmitters), it is possible for two or more transmissions to overlap. In the event that two or more broadcasts are received simultaneously, the receiving software should always be able to detect this collision. Simple implementations of an encoding protocol can lead to situations in which collisions are not detectable, but the protocol proposed in this document aims to prevent the possibility of undetectable collisions. For a further discussion on how collisions may arise, proposed solutions, and other background information, please see RFC 1[1].

2 Background

For more background on some of the challenges, motivations, and decisions that have come up in the creation of this protocol, see RFC 1[1] (a discussion of collision possibilities and remedies) and RFC 2[2] (an examination of ways to encode a boolean value).

To satisfy the requirements of this project, we are designing a new protocol. This protocol will encode the UID of each transmitter in such a way that collisions (multiple simultaneous transmissions by different transmitters) can be detected by a receiving station and discarded. The protocol is a simple series of HIGH and LOW audio signals operating at a predefined frequency. (The specific frequency to be used will be documented elsewhere on the hardware engineering side of things.) The series of "pings" and the separation between them will be organized in a specific pattern based on the UID of the given transmitter. We'll generally refer to this pattern as the Unique Transmission Pattern (or UTP for short).

Additionally, we may want to have the ability to detect when a transmitter has stopped moving, possibly because the crab molted its shell or died. In this case, we want each broadcast to not only encode the UID of the transmitter, but also some boolean value.

Given the nature of the protocol and the project itself, we have named this specification the *id-correlated rhythmic audio broadcast* protocol, or iCRAB for short.

3 iCRAB Protocol Definition

3.1 Overview

At the core of this protocol is a single burst of information which is transmitted repeatedly on some randomly-varying interval. All transmitters have a unique identifier (an integer number) associated with them, and each burst of data will encode that identifier. The burst, hereafter referred to as a unique transmission pattern (UTP), will consist of two pings (short, continuous transmissions of the carrier frequency), separated by some delay time d. The duration of the pings and the delay time d will be functions of the transmitter's UID. The interval between UTPs will be random, and each transmitter will recalculate the interval time after each UTP according to a shared formula. See Figure 1 for an example of a UTP.

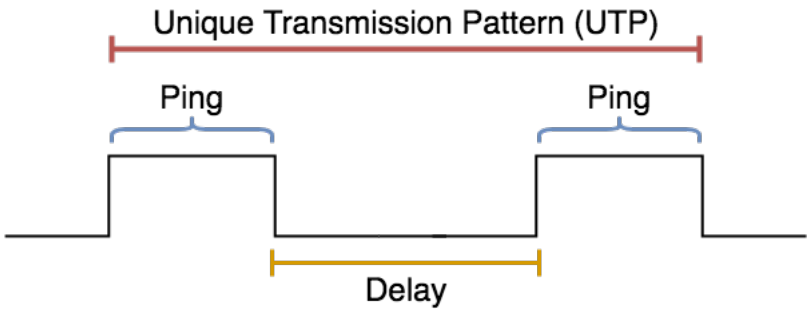


Figure 1: A Unique Transmission Pattern

3.2 Requirements, Rationale and Related Information

Based on the rough estimates given to us by our colleague in Electrical Engineering early in the project's history, we will initially comply with the following constraints.

1. The minimum ping duration should be 1 millisecond.
2. The "step size" (i.e. smallest difference in duration between any two pings) should be 0.1 milliseconds.
3. The minimum delay duration should be 10 milliseconds.
4. The "step size" for the delay should also be 0.1 milliseconds.

The remainder of this section provides details about requirements of the protocol, some reasoning for our choices, and other relevant information. However, this is meant to be a general look at the protocol; all specifics, including timing information, ID encoding ranges, and other constants, are detailed in Section 3.4. The motivation for leaving the information in this section (Section 3.2) somewhat unspecific is that requirements may change as we test and prototype our technology, and we'd prefer to have one single source of truth for all numerical values so as to reduce confusion.

Each transmitter will periodically broadcast a signal that the receiving system can detect. The receiving station will process the data encoded in the broadcast and also determine the direction, relative to the system, of that transmitter. The UTP broadcasts should be frequent enough for location updates to be practical to the user, while also being infrequent enough that the expected number of collisions is very low. For more information on collision statistics, see RFC Stats[3].

Observe, however, that if every transmitter transmitted on a fixed interval, then we could theoretically encounter a situation in which two transmitters transmit at almost exactly the same time for the entire duration of their deployment. In other words, every single one of their transmissions would collide until one of the transmitters depleted its battery charge, and the receiver would never be able to reliably determine the location of either one. To remedy this, we will instead randomly vary the interval between broadcasts. That way, if two transmitters happen to transmit at the same time in one instance, they won't necessarily collide the second time around. The random adjustment will need to be recalculated for each interval in order to reduce our chances of collisions.

The iCRAB protocol is expected to support only a finite number of transmitter UIDs. The upper bound selected is fairly arbitrary but based on the client's expected needs for the project's research.

Finally, this protocol is also expected to be able to encode (along with the ID of a given transmitter) a boolean value. For the purposes of the project, this value will be `true` if and only if a given transmitter is inert. This generally indicates that the crab died or molted its shell. We will encode this boolean value by effectively doubling the number of IDs that can be encoded. Half of the IDs are to be used when the boolean value is `false`, and the other half are used when it is `true`. Specifically, values in the range $[0, MAX\_UID)$ are to be used when `inert=false`. There is no additional mapping when IDs are in this range. For all values in the range $[MAX\_UID, 2 * MAX\_UID)$, the value encoded is actually MAX_UID bigger than the ID that is encoded, and `inert=true`.

For example, if a signal encodes the value 42, the receiver should report this as transmitter 42 with `inert=false`. However, if the value 542 is received, assuming that $MAX\_UID = 500$, the receiver should interpret this as transmitter 42 but with `inert=true`.

For more background on the topic of boolean value encoding, see RFC 2[2].

3.3 Preamble to Definitions and Specifications

Each Unique Transmission Pattern (UTP) will be formed by a ping, a delay, and another ping, in that order. Each transmitter will broadcast a UTP and will then wait for some interval before transmitting a UTP again. This process loops indefinitely throughout the transmitter's lifetime.

Defined in Table 1 are the constants that we will use for the various aspects of this protocol. This section of the document will be updated as needed if these values change.

Table 2 lists the various mathematical formulae we will use for encoding. Some functions are passed a single integer value, which is a UID.

Between each UTP broadcast, every transmitter should wait for some amount of time. As mentioned previously, this interval should include some random variation that is recalculated each time. The length of each interval should be a number of seconds in the range $[MIN\_INTERVAL,\ MAX\_INTERVAL]$.

Finally, we formally define the behavior of a transmitter in pseudo-code (see Listing 1.) The functions *HIGH* and LOW cause the physical transmitter to begin or cease transmitting, respectively. The *sleep()* function simply causes execution of the code to stop for a given number of milliseconds.

Section 3.4, on the following page, aims to include relevant constants, formulae, and other definitions that developers will need while programming the components of this project. This section is our single source of truth for the values used in the iCRAB protocol, and it will be updated as needed throughout the prototyping and testing phases of the project.

The remainder of this page is intentionally left blank.

3.4 iCRAB Definitions and Specifications

Constant	Value
MIN_INTERVAL	15 s
MAX_INTERVAL	20 s
MAX_ID	499
MIN_PING_DUR	1.0 ms
MIN_DELAY_DUR	10.0 ms
STEP_SIZE	0.1 ms

Table 1: Constants to be used for the iCRAB Protocol

Name	Definition
ping(id)	$(id \times STEP\_SIZE) + MIN\_PING\_DUR$
delay(id)	$(id \times STEP\_SIZE) + MIN\_DELAY\_DUR$

Table 2: Formulae to be used for the iCRAB Protocol

```
1  void doPing(int id){
2    HIGH()
3    sleep(ping(id))
4    LOW()
5  }
6
7  void loop(id){
8    doPing(id)
9    sleep(delay(id))
10   doPing(id)
11   sleep(interval())
12 }
```

Listing 1: Transmitter Behavior

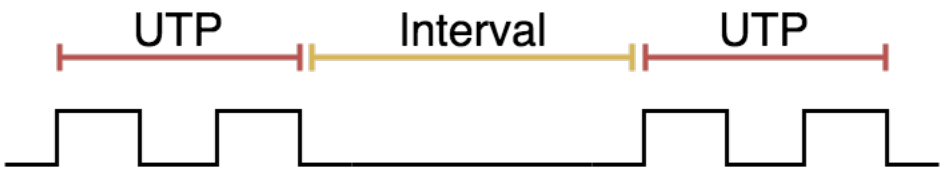


Figure 2: For reference, pictured are two UTPs separated by an interval

4 References

1. Strong, N. *RFC 1: Collisions in Delay-Based ID Encoding Protocol.* 2017. PDF. `https://github.com/cabeese/crab-tracker/blob/master/doc/rfc1/RFC1.pdf`

2. Strong, N. *RFC 2: Boolean Value Encoding in Transmission Protocol.* 2018. PDF. `https://github.com/cabeese/crab-tracker/blob/master/doc/rfc2/RFC2.pdf`

3. Yugawa, C. *RFC Stats.* 2017. PDF. `https://github.com/cabeese/crab-tracker/blob/collision-statistics/doc/rfc1/RFC1_stats.pdf`

A Glossary of Terms

Delay: in the context of ID encoding, the space between the falling edge of one **ping** and the rising of the next within a single **UTP**.

Delay Time (d): the duration (generally in milliseconds) of a given **delay**.

iCRAB (id-correlated rhythmic audio broadcast) protocol: the protocol designed by the members of the Crab Tracker project and described in detail in this document.

Inert: A transmitter will be marked as **inert** if it is determined that the transmitter has not moved "enough" in a given period of time. This definition is subject to change based on hardware constraints, and its complete definition is outside the scope of this document.

Interval: the time between two consecutive broadcasts of **UTP**. Measured by the distance between the final falling edge of one ping and the first rising edge of the next.

Ping: a single, continuous transmission of signal.

Ping Duration: the length of time between the rising and falling edges of a continuous transmission (a **ping**).

Unique Transmission Pattern, UTP: a sequence of two **ping**s separated by some **delay** used to encode the unique identifier of a transmitter.

RFC 1: Collisions in Delay-Based ID Encoding Protocol

Margot Maxwell Lizzy Schoen Noah Strong Chloe Yugawa

v1.2 – November 19, 2017

Contents

1 Introduction **2**

2 Original Protocol **2**

3 Possibility for Collisions **3**

3.1 Example Scenarios 3

3.1.1 Close Proximity, Similar Start 3

3.1.2 Close Proximity, Offset Broadcasts 3

3.1.3 A Series of Overlapping Broadcasts 4

3.1.4 Mutually Distant Transmitters 5

4 Potential Solutions **5**

4.1 Use Two Frequencies To Differentiate Pings 5

4.2 Attempt to Detect Anomalies Through Statistical Reasoning 5

4.3 Detect Falling Edges of Transmissions 6

4.4 Adjust Ping Length (p) Relative to Delay (d) Time 6

5 Statistical Probability of Collisions **7**

6 Discussion **7**

1 Introduction

The protocol proposed for this project encodes a unique identification number in the transmitted signal by the delay between two consecutive pings. While this method is easy to implement in hardware, and works well in the case of only a signal transmitter, the addition of multiple transmitters within range of a receiver brings about the potential for collisions between different signals. In some cases, the collisions may be undetectable, leading to incorrect reporting of nearby crab IDs. Completely solving this problem will require a change to the underlying protocol. However, the likelihood of such collisions may be low enough that we can move forward with this known flaw in the protocol.

2 Original Protocol

In the current iteration of our detection and identification protocol, every transmitter will transmit at the exact same frequency, somewhere around 40kHz. This is ideal from a hardware perspective, as the piezoelectric equipment can be tuned to work on a single frequency with a high degree of accuracy.

Every transmitter will periodically send out two quick "pings," each separated by some delay d. The value of d will encode the ID of the transmitter. For example, $d = 42$ms may correspond to ID 30, while $d = 50.5$ms may correspond to an ID of 38. Note that these numbers are only for illustration purposes. Because the receiving hardware can easily detect the two pings, it can measure the value of d by calculating the time difference between the rising edges of the consecutive signals. See Figure 1 for an illustration.

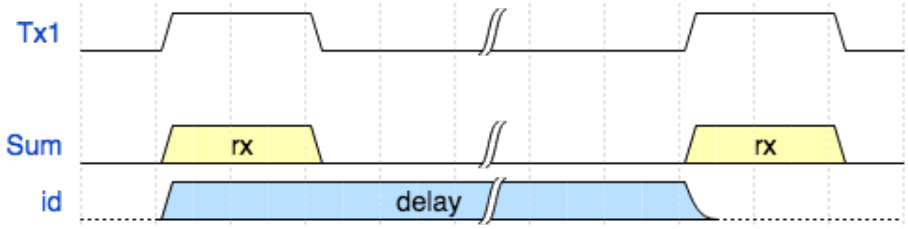


Figure 1: A single transmitter ($Tx1$), the input received by the receiver (Sum), and the calculated delay d based on the time measured between the rising edges of the two pings from $Tx1$.

Each transmission of an ID by a transmitter (that is, the sequence of a ping, a delay, and another ping) will happen regularly around some average interval. Because there is no synchronization between transmitters, the interval between each broadcast will vary randomly within a given range. For example, the delay may be 30 seconds, ± 5 seconds, with the random variation recalculated after every broadcast.

The motivation behind the randomly-varying schedule is to decrease the likelihood of simultaneous transmissions by two different receivers. However, the random interval only functions to reduce the likelihood of two *consecutive* overlapping transmissions. Without an inter-receiver collision-detection solution, there will always be the possibility that two transmissions overlap.

3 Possibility for Collisions

As discussed above, when two transmitters are present in the receiving range, and there is no synchronization between the two (i.e. they may transmit their IDs at any time, regardless of when the other transmits), it is possible for the two transmissions to be broadcast at similar times, thereby overlapping. The signals may overlap in countless different ways depending on when both transmissions start and where the transmitters are in relation to each other and to the receiver.

In some cases, collisions may cause the data to be some so skewed that it will be easy to detect and throw away. However, in a more likely and more dangerous situation, the overlap of data will lead to inaccurate results and be extremely difficult to detect.

3.1 Example Scenarios

Below are a few possible scenarios that may occur in practice or in testing environments. Note that this is by no means an exhaustive list, but instead a selection of cases chosen to demonstrate potential issues.

We'll define some notation for the sake of convenience and clarity. If $Tx1$ is a transmitter, then d_1 is the delay time between the two pings for $Tx1$; that is, d_1 encodes the unique ID for $Tx1$. We'll also use B_1 to denote the first ping from $Tx1$ (the **B**eginning of the encoding) and E_1 to denote the second (**E**nd of the encoding) ping.

3.1.1 Close Proximity, Similar Start

Situation: Two transmitters in close proximity to each other transmit at very nearly the same time. Their transmissions line up so that the first ping from $Tx2$ starts as the first ping from $Tx1$ is broadcasting. If the delay times happen to be very similar for the two transmitters, it is possible that the same effect is observed in reverse on the ending pings. That is, the second ping for $Tx2$ begins just before the second ping for $Tx1$. See Figure 2 for an illustration.

Potential Effect: Should this happen, the receiver will register two pings, just as it would if only one transmitter broadcasts. However, the measured delay $d_{measured}$ will not equal d_1 or d_2. Thus the software will report the ID corresponding to $d_{measured}$, and not the ID of either of the transmitters actually broadcasting. Worse still, there is no way to detect when this happens.

3.1.2 Close Proximity, Offset Broadcasts

Situation: As in the previous example, suppose there are two transmitters in close proximity to each other. However, in this situation, suppose the two transmissions are broadcast at similar times so that they are interleaved. That is, the receiver will see B_1, B_2, E_1, E_2, in that order. See Figure 2 for an illustration.

Potential Effect: In this case, the receiver will detect two different delay times, which may seem normal. However, these delays are once again wrong – they do not correspond to any transmitter

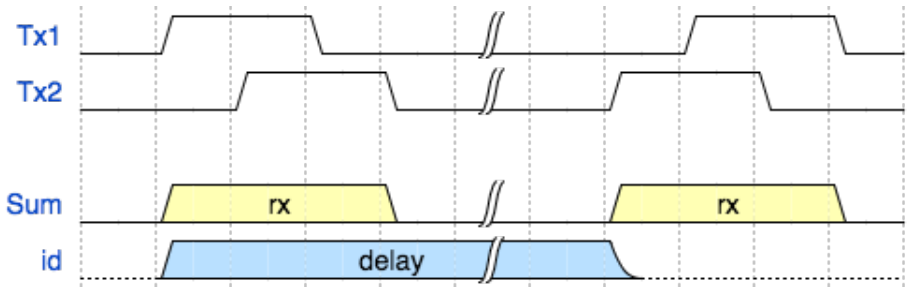


Figure 2: An example collision. The computed delay d is measured as the time difference between the rising edge of $Tx1$'s first ping and $Tx2$'s second ping.

actually transmitting. The software will then report that there are two signals coming from a specific direction, but the IDs that it thinks these correspond to will be incorrect. As in the previous example, there is no way to detect when this happens, so the software would unknowingly present incorrect information.

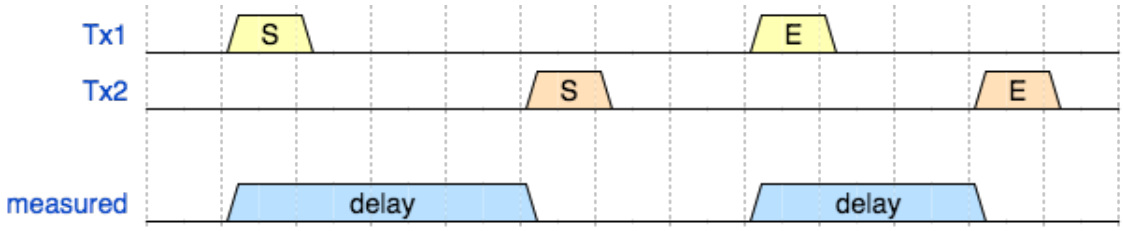


Figure 3: Another example collision. Note that in this situation, two distinct delays are measured. While it is possible for them to both correlate to valid IDs, neither delay matches the delays actually encoded by $Tx1$ or $Tx2$.

3.1.3 A Series of Overlapping Broadcasts

In an extreme case, it may be possible for multiple transmitters to all begin broadcasting at very similar times such that all of their pings overlap and are registered as a single (extremely long) ping. For instance, if n transmitters send out pings perfectly sequentially, it is conceivable that the receiver receives B_1, B_2, ..., B_n in order with so little temporal separation that no falling edges are registered by the receiving hardware. In other words, the receiver hears one long, solid tone, as opposed to a series of quick pings. It is theoretically possible that this single perceived tone is so long that the ending pings of one or more transmitters may be hidden.

This is an extreme case, but is nonetheless a possibility given significant transmitter density in the receiving range, sufficiently long pings, and sufficiently low delay d_n values for the transmitters in play.

3.1.4 Mutually Distant Transmitters

In general, though the above situations are quite possible, it is more likely that two or more transmitters in the receiving range will be in different directions from the receiving station. When two or more transmitters that are in different areas around the receiver broadcast at similar times, the overlaps will interfere in even more complex ways. Such scenarios are much more difficult to illustrate, but suffice it to say that they will also cause problems with invalid or confusing data. As before, some collisions can be detected and compensated for with a sufficiently complex algorithm, but not all cases are solvable with the current protocol.

4 Potential Solutions

Some early solutions that do not entirely change the original protocol have been proposed. They are discussed briefly below. This section may be expanded in the future as we discuss more options.

4.1 Use Two Frequencies To Differentiate Pings

One proposed solution is for every transmitter to have the ability to broadcast at two different frequencies, and use one for the first ping and another for the second. For example, the first ping may be 40 kHz and the second one could be 42 kHz.

Pros:

- With some added computation, we could compensate for some overlaps, such as the one described in example 3.1.2. Though some extreme cases, such as 3.1.1, would still lead to undetected incorrect results, the majority of overlap cases could be fixed.

Cons:

- This method would not scale to > 2 transmitters overlapping. Situations analogous to 3.1.2 but with 3 or more transmitters would still cause problems. However, giving our initial estimates of crab densities, this may be highly unlikely.

4.2 Attempt to Detect Anomalies Through Statistical Reasoning

Another proposal is to make the software smart enough to remember where a signal comes from every time a transmission is received. It keeps track of where it thinks all the known transmitters are and compares that data with incoming data whenever a transmitter is detected.

Further investigation into this suggestion is needed. However, one initial problem with that is that in order for such a technique to be effective, the software must know the location and movement of the receiver (via GPS or a similar technology).

4.3 Detect Falling Edges of Transmissions

Our original protocol assumes that the receiving system only detects the rising edges of every transmission. Adding the ability to detect the falling edges may help counteract some of the aforementioned problems. Specifically, it would potentially allow the system to detect when two nearly-simultaneous (though not truly simultaneous) transmissions overlap, as this technique would give the system the ability to detect when a ping lasts longer than the expected ping length. Unfortunately, such scenarios are only a small subset of the problems identified above.

Additionally, adding this ability may be somewhat difficult from a hardware perspective. It would require precise tuning of the hardware integrator, which could be difficult since the effective amplitude of each ping can vary based on a number of factors (including transmitter strength, aquatic conditions, objects interfering with the signal, and more). It was also mentioned at one point that communicating a falling-edge event to the receiving computer is not entirely simple.

4.4 Adjust Ping Length (p) Relative to Delay (d) Time

In our original protocol, the duration of every transmitter's ping was fixed. This solution proposes that we instead allow the ping's duration to vary on a per-transmitter basis. The ping length should be a function of the delay time, and therefore also a function of the unique ID, so that every unique ID has an associated unique ping duration p and unique delay time d. For example, we might require that $p = \frac{1}{2}d$; that is, every ping is half as long as the delay between rising edges of ping pairs.

This method creates some redundancy be encoding the unique ID up to three times in a single transmission (the first ping's duration, the delay length, and the second ping's duration). Though this redundancy doesn't provide us any improved accuracy or detection, it would allow the software to reliably determine whenever a collision occurs. A further discussion of why this method allow for collision detection is forthcoming.

This solution does have some potential difficulties associated with it. The biggest is that we would need well-tuned detection hardware so that we can reliably note the rising a falling edges of every signal. In practice, the software should probably compensate for some inaccuracy here depending on how successful our tuning efforts are. For example, weaker signals may not register strongly enough for the hardware integrator to report accurate rising and falling edges. Fortunately, as Dr. Lund points out, we could instead opt to have the system compare the ping lengths across the receiving hydrophones. Regardless of signal strength, transmitter distance, or other factors that may cause the measurement of the ping duration to differ from the actual ping duration, the ping duration measured by all four receivers should be nearly identical. That is, the signal won't decay by a significant amount in the short travel from one end of the receiving station to the other. If the measured ping lengths differ between any of the hydrophones, we can be confident that a collision occurred.

5 Statistical Probability of Collisions

For a discussion of the statistical likelihood of collisions occurring, see the *RFC Stats* document written by CY.

6 Discussion

All involved parties seem to agree that this problem is worth some attention, despite its expected low likelihood. We will move forward with implementing a falling-edge based solution related to the ideas proposed in section 4.4. Specific implementation details will be written up in a new document as they are decided on.

RFC 2 - Boolean Value Encoding in Transmission Protocol

Noah Strong

January 12, 2018 – v1.1

Contents

1 Introduction **2**

2 Background **2**

3 Requirements and Concerns **2**

4 Potential Solutions **2**

4.1 Add Additional Ping 3

4.2 Adjust Duration of Both Pings 3

4.2.1 Benefits 3

4.2.2 Drawbacks 3

4.3 Adjust Duration of Only One Ping 3

4.3.1 Benefits 4

4.3.2 Drawbacks 4

5 Discussion **5**

1 Introduction

We would like to find a way to encode a binary value in each transmission along with the UID. The encoding must not interfere with our current collision detection work and should not require a massive overhall of the transmission protocol as it currently stands.

2 Background

Patrick has requested that we look into adding the ability for a transmitter to broadcast some additional detail about the crab it is paired with. Specifically, Patrick is interested in attaching accelerometers to the tagged crabs so that we can determine if a given crab is "inert," perhaps because it has molted its shell or because it has died. We would encode this as an additional binary value in the transmission signal.

Patrick has stated that this is a highly desirable feature, but not absolutely critical to the end product.

3 Requirements and Concerns

We want to be able to encode both a UID and some binary value in each transmission without losing the work we've done to ensure that all collisions are detectable. Additionally, in the event that we decide not to include this feature in the final product, the changes made to the iCRAB protocol must not interfere with our original goals. That is, if we scrap this requirement, it should cost little or no extra work to continue on our original path.

We have already created a protocol that will encode a UID and is robust against collisions (that is, they are detectable). For more information on how this works, see RFC 1 Section 4.4. Some methods for adding binary encoding may interfere with the this collision detection work, and we would prefer that such solutions be avoided. In other words, collisions should still be detectable, at least in most cases, even if addition information is encoded in the signal.

4 Potential Solutions

In the case that the given boolean value that we wish to express is **false**, we may broadcast the original signal with no adjustments. We next propose alterations we could make in the case that the given boolean value we with to express is **true**.

4.1 Add Additional Ping

In this solution, a single UTP would be represented as a sequence such as P_P_P where P is a ping and _ is a delay.

While this would work in the general case, it adds some additional complexity to the receiver's detection code. Worse, it may cause collisions to be undetectable under certain circumstances. We will not discuss these situations, but they are easily discoverable and therefore left as an exercise to the interested reader. (Hint: if a collision occurs such that the first or third ping is affected, the receiver may reasonably assume that it detected two collision-free pings and would therefore report a standard, non-inert transmissions.)

4.2 Adjust Duration of Both Pings

One variation of this is essentially double the number of IDs we can encode. Half of the IDs are unchanged. The other half of IDs (eg the larger IDs, or the even-numbered IDs, or something along those lines) are reserved for transmitters to use when their boolean value has become TRUE.

For example, suppose we have 500 unique IDs, but choose to only assign the even-numbered IDs. The odd-numbered IDs could then be used for existing transmitters when their boolean value B is TRUE. Then transmitter 42, for example, would transmit the number 42 while b was false. Once b becomes true, that transmitter will start transmitting the value 43. Since 43 is an odd number, the receiver could easily deduce that transmitter 42's b value was TRUE.

Alternatively, suppose we have, for example, 500 IDs, and we assign 0-499 to transmitters. We then use values in the range 500-999 only when b is TRUE. That is, transmitter 42 would transmit 42 when b = FALSE and 500+42=542 when b = TRUE.

4.2.1 Benefits

This method requires very little adjustment to the encoding protocol. Instead, it would require only a small adjustment to how we interpret ID values as they are detected.

4.2.2 Drawbacks

Unfortunately, this method does double the number of IDs we need to be able to encode, which means that the max UTP duration will be much higher. This could potentially lead to more collisions. This solution is also potentially confusing, as the encoding is not obvious and it relies to some degree on magic numbers.

4.3 Adjust Duration of Only One Ping

In this scenario, we would make the length of one ping considerably larger than the other ping in a UTP. This long tone (herein referred to as an **inert tone**), like the delay and second ping, would

encode the UID of the transmitter (that is, its length would be determined by the UID). However, it would also indicate that the given transmitter was in the inert state, as it would be longer than a standard ping for that UID. See Figure 1 for an illustration.

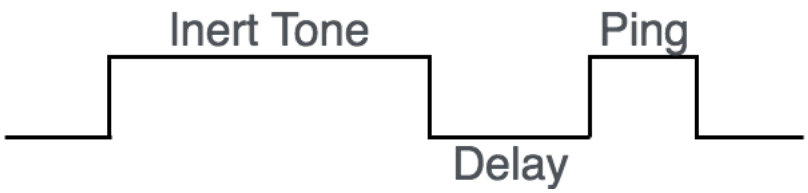


Figure 1: An example of a UTP with an "inert tone" in place of the first ping.

In this solution, we would still be able to detect collisions because the pings and delay would have mathematically-determined values, and any collision could cause one or more of those values to be altered, meaning that they would no longer be copacetic.

4.3.1 Benefits

This method does not require a change to *standard* UTP formatting, and removal of the boolean value encoding requirement would not affect previous work.

This method is resistant to collisions, so long as the appropriate restrictions are placed on the duration of each inert tone.

A UTP with an inert tone is clearly distinguishable from a standard UTP. This means that there is no arbitrarily-defined distinction between standard and inert UTPs. For instance, a simple visual inspection of a given UTP could quickly and easily determine if a UTP came from an inert transmitter or not.

Because the inert tone is considerably longer than most other transmissions, this method may cause the battery in the transmitter to die sooner. Since there is no need to repeatedly record the location of a transmitter that does not move, this may conveniently lead to the inert transmitter ceasing its transmissions earlier than it otherwise would. This is speculation and will depend on the actual hardware performance, though, so it is only a small consideration.

4.3.2 Drawbacks

The total duration of a UTP with an inert tone would be considerably longer than a standard UTP, which means we may have more collisions.

This method will also require additional processing and calculations by the receiver, though the same may be said of *almost* any other alternative.

5 Discussion

The development team has come to the consensus that option 4.2 is the most advisable choice for this project. The appeal of this method is largely its simplicity: it requires no additional calculations on the part of the receiver, as it utilizes the existing encoding scheme that will already be implemented.

The only thing left to determine is how divide the UTP space between the two values. The suggestion of this author is to use the first MAX_ID values exactly as they otherwise would be used (with the boolean value FALSE) and to reserve the values in the range $(MAX\_ID,\ 2 \times MAX\_ID]$ for use when the boolean value is TRUE. However, *this is not a formal definition*; see the official iCRAB Transmission Protocol Definition document for accurate details.