# Buoy Management System

## Buoy Sensor Device

The device will contain 2 main parts. An ECB and an ECU.

### The ECB

The ECB will receive data via 4 fiber ports (0, 1, 2, 3). The ECB will share data via an ethernet port. The ECB does not persist data. Instead, on demand it will give the requestor values from all ports in the form of a custom TCP packet. The device can respond to 200 requests per second.

The ethernet port will be bound to a fixed IP address and port as it is not reachable externally and DNS is not available. 127.0.0.1 should not be used to avoid collisions with the ECU and any potential loopback logic.

The TCP packet will contain an integer length (based on the number of ports) and a decimal reading of each port. If a fiber port is not connected to the device it will return NaN. In this way a device can have N fiber ports and report their values without configuration. For instance:

| Length | Port 0 | Port 1 | Port 2 | Port 3 |

| ------ | ------ | ------ | -------| ------ |

| 4 | 100.00 | NaN | 100.00 | NaN |

The decimal values are the depth in feet of the water beneath the buoy.

### The ECU

The ECU will be a device similar in capability to a Raspberry Pi device. It will run a Linux OS.

The ECU will host a firmware written in [ASP.NET](http://asp.net) / C# / Blazor Server. The firmware will be responsible for collecting data from the ECB, calculating nominal sea level, transmitting the data to a remote service, queuing data that is unable to be sent to the remote service, allowing technicians to “peak” at data in real time, and maintaining its own configuration.

Data Collection: The firmware will contain a “BuoySensorReaderService”. This service will be a BackgroundService. Each second it will connect to the ECB and gather current readings. For each port that returns a value (not NaN) a packet of data is formed. To later calculate “Wave Amplitude” the service will keep a collection of recent readings in memory. These readings will be averaged to give a nominal sea level. To avoid false averages sea levels will be NaN until there is at least 60 seconds of data. This packet will contain the depth reading, sea level, a globally unique id, and a timestamp. This packet will be “published” to an internal dispatcher.

public Guid Id { get; init; }

public int Port { get; init; }

public double Depth { get; init; }

public double? SeaLevel { get; init; }

public long ReadingOn { get; init; }

The “BuoySensorPacketDispatcher” will respond to dispatch events. It will first attempt to push this packet to a remote service for aggregation and management. If the remote service is unavailable the packet will be stored in a local SQL Lite database for retry later. The dispatcher will also send readings to the local user interface if a user is watching. This allows technicians to see data in real time.

A “BuoyPacketRetryService” will poll the database on a regular interval (default 5 minutes). If any packets are in the database the service will attempt to send them to the remote service. If any fail to send the retry process is stopped until the next interval. Upon successful sending of a stored packet the packet is removed from the database. This process continues until an internet connection is established and all packets have been sent successfully.

The “BuoyPacketRetryService” will have 2 unwanted side effects.

1. Retried packets will be sent out of chronological order. The timestamp in the packet must be trusted.
2. The database will grow if there is no internet connection. Eventually packets will need to be ejected to prevent the disk filling and causing device failure.

To maintain the database size a “BuoyPacketDatabaseService” will purge older packets. Multiple strategies can be implemented to choose when to purge packets. In no particular order.

1. Monitor database size
2. Monitor disk space
3. Monitor row count (not recommended as knowledge of row byte size is needed)

In any strategy, we are likely in this state due to an ongoing internet outage. We need to be proactive enough to keep the system from crashing and to stay ahead of new data ingestion.

## Buoy Sensor Server

Given there will be up to 1000 buoys and 250 devices it makes sense to centrally manage data warehousing, aggregation, reporting, and alerting.

The server will contain 2 main parts. An API and a UI. These should be separate services given the different usage contexts, scalability, and release requirements.

### The API

The API will be written in [ASP.NET](http://asp.net) / C# / Web API.

The API will authenticate and receive packets from all the sensor devices. Given that the devices are known to have frequent network interruptions, packets may be received out of chronological order. It will be necessary to trust the timestamps on the packets.

The API must be able to ingest 1000 records per second 24 x 7. The lightest touch should be applied to each record as it comes in to avoid resource exhaustion and bottlenecking.

Given the rate of data feed and the need to validate we do not receive packets twice it would be prudent to use a queue to receive the packets and process them asynchronously.

Deployments of the API could be considered from 2 perspectives.

1. The devices buffer buoy packets and will resend packets if the server is unavailable. Short restarts may be tolerable. Costs less.
2. Alerts may be time sensitive, so no downtime is allowed. Consider A / B deployment strategies. Costs more.

### The UI

The API will be written in [ASP.NET](http://asp.net) / C# / Blazor Server.

The UI will authenticate users, provide user interfaces for managing user alert preferences, viewing and downloading buoy data.

Deployments of the UI could be considered from 2 perspectives.

1. Use cases may have windows of little to no active user usage. Short restarts may be tolerable. Costs less.
2. If usage is constant and no downtime is allowed. Consider A / B deployment strategies. Costs more.

#### Shared Libraries

The API and UI will share a library stack. These libraries will be “services”, “core”, and “utilities”.

* The “services” layer will manage all incoming and outgoing data queuing, validation, mapping, and routing.
* The “core” layer will contain models, repositories, and messaging IO.
* The “utilities” library will contain constants, enums, extensions, and other useful tools.

#### Data Storage

With 1000 buoys deployed this service will receive ~2.6B readings per month. While these records are narrow, storage and query limitations should be considered carefully.

#### Other Considerations

The requirement for “alerting” is too vague. How are alerts delivered, email, text, voice, notifications? Are alerts rate-limited? Should rules be put in place to alert once for a geographical area even though multiple buoys are in the area. Imagine a hurricane triggering 100 buoys for 5 hours. Managing alerts at the device is extremely limiting and puts too much configuration into the device. Imaging needing to update a phone number or email address or an alert threshold on 250 devices.

Given the buoys only communicate the depth of water beneath them this is not enough information to know a “wave height” alone. There are multiple possible methods for calculating wave heights. Two I know of:

Follow previous peaks and troughs (highs and lows). This method is problematic as we must know the current high or low as consecutive waves could be smaller than previous. This method also requires the completion of wave form to know the actual height. This creates a slight delay in alerting.

Calculate nominal sea level by averaging previous depth readings. This method is problematic as initial data will be inaccurate. In addition, tidal changes will change nominal sea level over time. So, nominal sea level is a constantly changing target. However, once a current nominal sea level is obtained, we need only see a peak or trough (amplitude) half the alert height to trigger an alert. If seconds count...

Given the known network interruptions allowing the server to calculate the nominal sea level will result in frequent inaccurate sea level windows following device reconnection. I chose to calculate the sea level at the device and transmit this data to the server. With the depth and nominal sea level, wave amplitudes and heights are easily calculated. I did not calculate wave height and amplitude on the device as it would only decrease offline storage space.

I chose not to allow direct downloads from the device. Instead, I chose to push data from the device to a server. Preserving large amounts of data on a space limited device and providing external access to 100’s of devices is problematic. It is also a very bad user experience. Devices should be good at what they do and nothing else.

I chose not to put any authentication in the device user interface. I didn’t want to take the time to add a local login screen and credentials.

I did not take the time to air gap the device code. Checking all scripts, images, and style sheets exist on the local device.