# HERALD: Integration Guide

## Overview

HERALD is a cross-platform proximity detection solution for iOS and Android devices. It detects all devices within epidemiologically relevant range (8 metres), and offers frequent sampling of distance measurements (at least one sample per 30 seconds) for each device. The solution has been designed to operate indefinitely on screen locked iOS and Android devices in the background without any user interaction. It overcomes all the background operation challenges, especially on iOS devices. HERALD works on iOS 9.3+ and Android 5.0+, making it compatible with 98% of smartphones worldwide. The solution is also payload agnostic, to facilitate integration with existing contact tracing apps, acting as a reliable cross-platform transport for existing device identification data payload, thus it has no impact on approved security and privacy designs to minimise disruption to existing apps.

This integration guide aims to offer an easy to follow recipe for enhancing existing contact tracing apps with HERALD. As an overview, the process involves:

1. Isolating the existing device identification data payload generation code and wrapping it into a **PayloadDataSupplier**.
2. Isolating the existing contact data logging code and wrapping it into a **SensorDelegate**.
3. Remove existing Bluetooth Low Energy (BLE) code for detecting devices, exchanging device identification data, and distance measurement.
4. Replace the existing BLE code with the HERALD **SensorArray** for the **PayloadDataSupplier** and register the **SensorDelegate** for receiving detection and distance measurement events.

The API for both iOS and Android have been kept nearly identical in concept and in code where possible for ease of integration. The integration process can be as quick as one working day for both platforms; this was tested on the original NHS app and also the open source C19X app to ensure the API is practically applicable and easy to use. The API design also considered the architecture of TraceTogether (Singapore), COVIDSafe (Australia), StopCOVID (France) and other apps for ease of integration.

# Integration

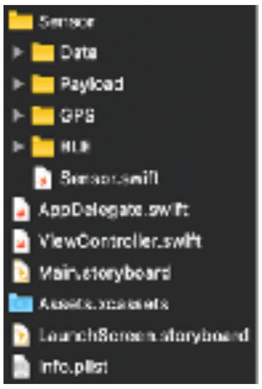
## Run test apps

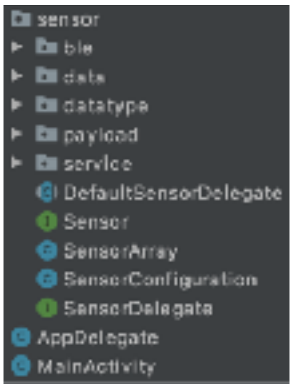
HERALD is delivered as a fully functional test app on iOS and Android. The recommended first step is to clone the iOS and Android GitHub repositories in Xcode and Android Studio, then deploy and test the apps on your own devices to ensure your development environment is compatible and to gain confidence in HERALD. Full details of our development environment and procedure for deploying and testing these test apps are available on the GitHub repositories.

## Understanding the code

The iOS and Android test app code have similar packages and structure for ease of understanding and integration. All the code required for integration into your contact tracing app can be found in the **sensor** package. Files and resources outside of this package are only used by the test app, and not required for integration.

The starting point for integration is the **Sensor** protocol/interface and the class **SensorArray** that implements this protocol/interface. A typical app will negotiate device registration data with a server (e.g. unique identifier and shared secret), then instantiate its device identification payload data generator (e.g. for encrypting the unique identifier for sharing with other devices over BLE). HERALD can then be used as the transport for this encrypted data by instantiating a **SensorArray** with the instantiated data generator as a **PayloadDataSupplier** (see **Payload** package) and registering a **SensorDelegate** (see **sensor** package on Android or **Sensor.swift** on iOS) to receive proximity detection data for recording and processing on-device or centrally.





For reference, the purpose of the other packages are as follows. For iOS, the **Data** package contains a range of loggers for on-device logging of application and detection data for debugging and testing. Most of these loggers are optional in a production environment and can be disabled by modifying the **SensorArray** initialisation code. The **Payload** package contains the interface **PayloadDataSupplier** for integration with your own device identifier data generator. This interface is called on every read payload request from another device, thus offering the opportunity to mutate your payload to maintain security and privacy. For variable length payloads, you will also need to implement the data parser method for partitioning a concatenation of payloads into individual payloads. This is necessary as the payload sharing code transmits multiple payloads in a single transmission to enable efficient detection across iOS and Android devices. The **GPS** package provides the optional location monitoring capability. The default implementation uses the location monitoring capability, not to monitor user location, but to enable beacon ranging which is necessary for iOS background operation. Finally, the **BLE** package contains the Bluetooth LE based proximity sensor that makes up the core of HERALD. All the configuration data for BLE operation can be found in the struct **BLESensorConfiguration**, for instance the service, characteristic and manufacturer UUID for the beacon service, and also the advert refresh interval.

For Android, the **BLE**, **Data** and **Payload** packages are identical to iOS. The **Datatype** package contains classes for implementing the type aliases in iOS, as aliases are not supported in Java. The **Service** package contain the foreground and notification services that are mandatory for continuous background operation on Android.

## Insert HERALD

The process for inserting HERALD into your app is similar in concept for both iOS and Android. It involves copying the HERALD library from the test app into your app and making the library accessible from your app. This section presents the procedure for inserting the library for iOS and Android.

### Android

1. Open your app project for integration with HERALD in Android Studio.
2. Copy the **HERALD** subfolder from the HERALD test app project into your project.
3. Open **settings.gradle** and insert the following line.

```
include ':HERALD'
```

4. Select **File > Sync Project with Gradle Files**, the **HERALD** module should now appear in your project.

5. Open **build.gradle** for your app and insert the following line.

```
implementation project(':HERALD')
```

6. Select **File** > **Sync Project with Gradle Files**, the **HERALD** module is now available for use in your project.

7. Copy **requestPermissions** and **onRequestPermissionsResult** methods from the MainActivity of the HERALD test app into your main app activity, along with the **permissionRequestCode** static variable; call the **requestPermissions** method on initialisation of your app to ensure all the required permissions are requested and granted for HERALD to function.

8. Select **Build** > **Make Project** to ensure HERALD is able to coexist with your existing app code, resolve any naming clashes and compilation errors now before proceeding to integration.

### iOS

1. Open your app project for integration with HERALD in Xcode.

2. Copy the **HERALD** subfolder from the HERALD test app project into your app project in Finder.

3. Drag and drop **HERALD.xcodeproj** from the **HERALD** subfolder in your project from Finder into Xcode under your project, the **HERALD.framework** product is now available for use in your app.

4. Open your app configuration, go to **General** > **Frameworks, Libraries, and Embedded Content**, then drag and drop the **HERALD.framework** product to this list to add the framework as dependency.

5. Select **Product** > **Clean Build Folder**, then **Product** > **Build For** > **Testing** to ensure HERALD is able to coexist with your existing app code, resolve any naming clashes and compilation errors now before proceeding to integration

6. The **HERALD** framework is now available for use in your app by importing the **HERALD** module in your app code with the following line at the top of your code.

```
import HERALD
```

## iOS integration

### 1. Implement a PayloadDataSupplier

Contact tracing apps enable the exchange of a globally unique device identifier between devices for contact logging. The encoding of device identifier into binary data is usually a key element in the security and privacy design, thus HERALD will simply act as a transport for your binary data to avoid changing the fundamental designs elements of your app.

The **PayloadDataSupplier** protocol is the API for integrating your device identifier payload. The protocol has two functions. The first function takes a timestamp and returns the binary payload data for transmission to another device. This function is called upon every device identifier exchange for all devices, thus making it possible to mutate the payload on every exchange and encrypt the payload based on time. For reference, **PayloadTimestamp** and **PayloadData** are simply type aliases for **Date** and **Data**. HERALD supports both fixed and variable length payloads up to 510 bytes. The performance reported in literature is based on a 129 byte payload.

```
func payload(_ timestamp: PayloadTimestamp) -> PayloadData
```

The second function is used to parse raw data into a list of payloads. This function is required because HERALD uses Android devices to relay payloads between iOS devices as one of the two mechanisms for enabling iOS background detection. Internally within HERALD, upon discovery of an iOS device (A), it will concatenate several recently seen iOS device payloads (e.g. B,C,D) and write the concatenated binary data to the iOS device (A). This parser function is then called at (A) to retrieve the separate payloads (i.e. B,C,D), thus enabling A to discover B,C,D while they are all in background mode. Please note, HERALD also includes a second mechanism that enables direct detection between iOS background devices without relying on an Android device.

```
func payload(_ data: Data) -> [PayloadData]
```

Implementation of this second function is optional for apps that use a fixed length payload, as a default implementation (see extension to **PayloadDataSupplier**) calls the first function to obtain an example payload to establish the payload length for partitioning concatenated data into individual fixed length payloads. For variable length payloads, you will need to implement your own logic for partitioning concatenated payload data, e.g. by recognising start and end byte patterns. If this is too challenging for your payload format, the Android code can be modified to share on payload at a time instead.

## 2. Instantiate a SensorArray

HERALD has been designed as a sensor array for proximity detection, rather than an exclusively BLE based solution. A sensor array combines multiple different technologies for recording person-to-person and person-to-environment encounters, i.e. BLE and ultrasound for encounters with people, and GPS and static beacons for encounters with places. The current implementation includes BLE and GPS sensors. Work is underway for investigating the feasibility of ultrasound, static beacons, and other sensors.

The **SensorArray** class is initialised with the **PayloadDataSupplier** implemented in the previous step. Once initialised, the **SensorArray** will be activated and deactivated according to Bluetooth state, i.e. proximity detection will commence or cease upon Bluetooth power on or off. For clarity, if Bluetooth is already on when the **SensorArray** is initialised, proximity detection will be immediately active, and contact events shall be distributed to all registered **SensorDelegates**. For initial integration testing, given two devices, the application log should show your payload being exchanged across the devices.

### Sensor selection

The initialisation code in **SensorArray** defines the sensors that make up the sensor array and also the loggers for writing sensor data to files on the local device. The current implementation includes the BLE and GPS sensors. The former is essential for proximity detection with BLE. The latter is required to enable iOS background detection in isolation without relying on Android devices. The GPS sensor is currently configured as a beacon ranging location sensor that is seeking a fictional beacon; this enables iOS devices to discover each other while both devices are running in the background, when the screen is lit, even for an instant. Bluetooth behaviour on iOS devices is influenced by screen ON events, where background adverts are fully read by a background scanner only when the screen is ON. Considering the average time between device pickup (screen ON) is only 10 minutes during the working day, this offers a reliable secondary discovery method for backgrounded iOS devices when there are no Android devices in the vicinity. Please note, **ConcreteGPSSensor** can be configured as a complete location monitor by adjusting the desired accuracy and distance filter to enable recording of person-to-environment encounters. The current default parameters essentially disables location monitoring while maintaining location update functionality. The combination of beacon ranging and location update is necessary for enabling background detection.

### Logging

Several loggers have been included in the **SensorArray** initialisation code. These were used during development to automate testing and analysis of efficacy, by measuring and logging detection, continuity, and sampling rates in plain text CSV files on device for download and processing by analysis scripts. The **ContactLog** logger offers comprehensive recording of all the detect, measure, read, share and visit events and associated data for all encountered devices. Details about the different events can be found in the **SensorDelegate** section. The **contacts.csv** file generated by this logger contains the time for all distance measurements in raw RSSI for all encountered devices (measure events), and also the mapping between ephemeral device identifiers and payload data (read events), thus making it possible to establish exposure duration and proximity for all encountered devices. The raw data is used during formal testing for measuring **detection** and **continuity** rates for assessing **efficacy**.

The **StatisticsLog** logger offers summary statistics (i.e. count, mean, standard deviation, min, max) for the sampling rate, in elapsed time between samples, for each encountered device. The **statistics.csv** file generated by this logger provides the raw data for estimating **completeness**, which in simple terms is the difference between the actual start and end time of an encounter and the recorded times. Accurate exposure timings enable accurate infection risk estimation, and this is particularly important when the epidemiological risk model is based on cumulative exposure.

The **DetectionLog** logger records key information about the device (i.e. name, model, operating system version), its own payload and all the payloads that it has detected. For formal testing, a test **PayloadDataSupplier** has been created for publishing a fixed and consistent payload for each device (see **identifier** function in **AppDelegate**), thus the **detection.csv** file generated by this logger provides the raw data required for measuring **detection** rate, i.e. did the device detect all other devices in the vicinity.

Finally, the **BatteryLog** logger records power source and battery level data at 30 second intervals to measure total power usage of the device over time. The **battery.csv** file provides a continuous log for assessing power drain.

All four loggers in the **SensorArray** initialisation code are optional in a production app. In addition, HERALD uses **SensorLogger** for detailed logging across all the code to a plain text file **log.txt** for debugging and analysis. This is also optional in a production app. Log level can be adjusted by changing **BLESensorConfiguration.logLevel**.

## 3. Implement a SensorDelegate

All sensor events are distributed to all sensor delegates that have been registered with the **SensorArray** via the **add:delegate** function. The **SensorDelegate** protocol includes a set of optional callback functions for receiving event information. For integration, the simplest option is to implement a single high level API function for receiving regular distance measurements along with payload data. This function is called immediately for each measurement taken for every device, where the payload data has already been acquired by HERALD.

```
func sensor(_ sensor: SensorType, didMeasure: Proximity, fromTarget: 
TargetIdentifier, withPayload: PayloadData)
```

The **sensor** parameter will always be **BLE** given the **SensorArray** is only using BLE for proximity sensing. The **didMeasure** parameter contains the distance measurement, which is the raw RSSI value. The **fromTarget** parameter can be ignored as it contains the ephemeral device identifier which has no meaning outside of HERALD. The **withPayload** parameter is the payload data for the device. This is the encoded device identification data for processing by your app. In short, the **didMeasure** and **withPayload** parameters provide the data for infection risk estimation. A timestamp is not included in the callback function, as it is called immediately for every distance measurement, thus the time of function call according to the time measurement method in your app is the timestamp. It is anticipated that this function will feed the contact log in your app. The raw data provides the basis for identifying contacts and estimating exposure level according to your RSSI calibration data and risk model.

### Other delegate functions

In HERALD, a device has an ephemeral identifier (i.e. BLE device address) and an actual identifier (e.g. registered permanent device identifier for contact tracing) encoded in the payload data. The former is part of the BLE specification and it is used in HERALD for identifying physical devices and caching payload data to avoid frequent connect and read requests for efficiency. Please note, this address will change at a variable rate on all devices, and it can rotate as quickly as once every few seconds on some devices (e.g. Samsung A10). The impact of frequent address change has been compensated in HERALD by including a rotating pseudo address in the advert on Android devices.

In addition to the high level API function for integration, the **SensorDelegate** protocol offers a set of low level callback functions for receiving all the detailed detection events. The benefit of using this low level API is that all data is available, thus enabling the production of a more complete detection timeline for risk assessment. For instance, HERALD will start taking regular distance measurements for a target device upon detection, but there could be a short delay before the payload is successfully read for the device. The high level API will only start reporting distance measurements after the payload is read, whereas the low level API enables retrospective attribution of payload to historic distance measurements based on the ephemeral identifier, thus offering a more precise start time for an encounter and also additional distance measurements for risk estimation. In practice, this is unlikely to change the overall risk estimate given the payload is typically read long before the target device reaches close proximity.

The typical sequence of events from initial device detection to regular distance measurements is as follows:

1. New device comes within detection range, this will generate a **didDetect** event for the device with an ephemeral target identifier.

```
func sensor(_ sensor: SensorType, didDetect: TargetIdentifier)
```

2. Regular distance measurements shall be taken upon detection, this will generate a series of **didMeasure** events for the device to provide raw RSSI measurements. At this point, the measurements are unattributed, as the target identifier is not the payload data, thus there is insufficient information for contact tracing or risk estimation, i.e. we know the device has been near another device, but we don't know the registered device identifier for attribution.

```
func sensor(_ sensor: SensorType, didMeasure: Proximity, fromTarget: 
TargetIdentifier)
```

3. HERALD will attempt to establish connection between the two devices to acquire the payload data. Upon successful connection and data acquisition, it will generate a **didRead** event to enable association of target identifier with payload data for attribution of historic and future distance measurements.

```
func sensor(_ sensor: SensorType, didRead: PayloadData, fromTarget: 
TargetIdentifier)
```

4. Once the payload of a target device has been acquired, all future distance measurements will generate a **didMeasure** event followed by a **didMeasure:withPayload** event.

```
func sensor(_ sensor: SensorType, didMeasure: Proximity, fromTarget: 
TargetIdentifier)
```

```
func sensor(_ sensor: SensorType, didMeasure: Proximity, fromTarget: 
TargetIdentifier, withPayload: PayloadData)
```

The **SensorDelegate** protocol also offers callback functions for reporting sensor status and location updates (if enabled). A state change for any of the sensors in the sensor array will result in a **didUpdateState** event, where the first parameter describes the sensor that changed state and the second parameter describes the new state. State **on** means the sensor is active and fully operational (e.g. Bluetooth is powered on). State **off** means it is available on the device but inactive (e.g. Bluetooth is available but powered off). State **unavailable** means the sensor is either unsupported on the device or no longer working (e.g. Bluetooth is unsupported).

```
func sensor(_ sensor: SensorType, didUpdateState: SensorState)
```

Location sensing enables person-environment (rather then person-person) contact tracing, to isolate people who may have been in contact with contaminated surfaces (e.g. at an open air market), or a dense crowd of people containing infectious individuals (e.g. music festival where detection rate may not be 100% due to density and people not carrying their phones). The current implementation includes a GPS sensor that can be configured for fine grained location monitoring (disabled by default). A future implementation may include other location sensors such as static beacons. All location detections will generate a **didVisit** event where the location parameter can be a GPS coordinate or place description depending on the sensor.

```
func sensor(_ sensor: SensorType, didVisit: Location)
```

The last remaining callback function in the **SensorDelegate** protocol is the **didShare** function. This is unlikely to be used by any app, but it has been included to make all internal data available for evaluation and analysis. As mentioned before, HERALD has two mechanisms for enabling iOS background detection, one of which relies on relaying payload data via an Android device in the vicinity. When an Android device shares the payloads of recently seen iOS devices via characteristic write to an iOS device, it will generate a **didShare** event. The **sensor** parameter will be **BLE** as payload sharing only occurs in the BLE sensor. The **didShare** parameter is a list of payloads extracted by the **payload:data** function in **PayloadDataSupplier**; these are the payloads of recently seen iOS and non-transmitting Android devices, that were shared by an Android device in the vicinity. The target identifier parameter is the ephemeral identifier of the Android device that shared the data.

```
func sensor(_ sensor: SensorType, didShare: [PayloadData], fromTarget: 
TargetIdentifier)
```

This **didShare** callback function is only of interest to apps that would like to understand the exact provenance of payload data, e.g. to understand the level of reliance on Android devices for iOS detection. Every call to this function is followed by a series of **didMeasure** calls for each shared payload. For simplicity, consider two iOS devices (A,B) and one Android device (C). Let's also assume the Android device (C) has already detected and read payload of A and B, thus generated the **didDetect**, **didMeasure** and **didRead** calls. The Android device C knows A and B are iOS devices, thus it will share the payloads of recently seen iOS devices to A and B. In other words, it will share the payload of B with A, and vice versa.

The typical sequence of events at A when C shares the payload of B is as follows:

1. Android device (C) shares the payload of iOS device (B), this generates a **didShare** event, where the list of payload data contains the payload of B, and the target identifier is the identifier of C as that is the origin of the shared data.

```
func sensor(_ sensor: SensorType, didShare: [PayloadData], fromTarget: 
TargetIdentifier)
```

2. Internally, payload sharing data includes both payload and RSSI data. More specifically, the distance (RSSI) between C and A as measured by C is transmitted to A along with the shared payloads. As such, A can infer its distance to the shared device B based on the distance between A and C. With this in mind, for each shared payload, HERALD will generate a series of **didRead** and **didMeasure** events. The **didRead** event offers the individual payload, as if A detected B directly (rather than via C). The target identifier in this instance is a generated ephemeral identifier for B (as the device address is unknown). For information, if the device address becomes available in the future, it will be used in this call. The **didRead** event is followed by a **didMeasure** event, reporting the RSSI value between A and C as approximation for the distance between A and B. Once again, the target identifier is the same as that in the **didRead** event. Finally, a **didMeasure:withPayload** event is generated which combines the information in the previous two events for ease of integration.

```
func sensor(_ sensor: SensorType, didRead: PayloadData, fromTarget: 
TargetIdentifier)
```

```
func sensor(_ sensor: SensorType, didMeasure: Proximity, fromTarget: 
TargetIdentifier)
```

```
func sensor(_ sensor: SensorType, didMeasure: Proximity, fromTarget: 
TargetIdentifier, withPayload: PayloadData)
```

## Android integration

Android integration is practically identical to iOS as they share the same API. Please refer to the iOS integration guide for design details. For consistency, the description for Java methods will use the iOS function notation, e.g. **payload:data** refers to the **payload(Data data)** method.

### 1. Implement a PayloadDataSupplier

For fixed length payloads, extend the **DefaultPayloadDataSupplier** class to wrap your payload data generator. This class implements the **PayloadDataSupplier** interface and offers a default implementation for **payload:data** method. The default method calls the **payload:timestamp** method to obtain an example payload for establishing the payload length for convenience. A production solution that uses a fixed length payload should override this method for efficiency.

For variable length payloads, please implement the **PayloadDataSupplier** interface and provide your own parser to separating the raw data (byte array of concatenated payload data) into individual payloads.

### 2. Instantiate a SensorArray

The **SensorArray** class constructor requires the application context and a **PayloadDataSupplier** as parameters. The former is necessary for access to the device file system (e.g. logging to plain text files) and Bluetooth functionalities. The latter provides the payloads for exchange with other devices. Like the iOS **SensorArray**, the constructor defines a collection of sensors for proximity detection. At this stage, only the BLE sensor is included in the Android **SensorArray** as location services are not required for background operation. A future version will include an optional GPS sensor like iOS for location monitoring if required.

Logging functions are identical to iOS and the output CSV files are also identical in content and format, thus the data can be analysed using common analysis and visualisation scripts across the two platforms. Once again, the optional logging functions include contact, statistics, detection, and battery logs. Like iOS, the **ConcreteSensorLogger** class provide all the logging functions across HERALD, thus log level is adjustable at **BLESensorConfiguration.logLevel**. The log files can be downloaded from the device using the **Android File Transfer** utility (see **readme.md** file on GitHub repository for details on log data download).

The key differences in the Android constructor are the initialisation of **ConcreteSensorLogger** and **ForegroundService**. The former is required to set the reference to the application context, thus enabling access to the file system for writing log files. The latter is essential for enabling background operation of BLE functions. The foreground service does not perform any actual function, but is a requirement for background operation on Android. The **ForegroundService** class call the **NotificationService** class to create a visible notification, thus granting the app permission for continuous background operation. These are mandatory requirements on Android. If your app has a constantly visible notification, **NotificationService** can be disabled in **ForegroundService** to avoid showing two notifications for one app.

### 3. Implement a SensorDelegate

The **SensorDelegate** interface define exactly the same set of callback functions as the iOS **SensorDelegate**, offering the same logic and data. The **DefaultSensorDelegate** class implements all the callback functions in this interface, thus an application can extend this class and override a subset of the callback functions for simplicity. Once again, simple integration involves overriding the **didMeasure:withPayload** method to receive distance measurements along with target device payload data.

```
void sensor(SensorType sensor, Proximity didMeasure, TargetIdentifier fromTarget, PayloadData 
withPayload);
```

All other delegate functions are called under the same circumstances and sequence as the iOS functions. Please refer to the iOS **SensorDelegate** description for details.

## Integration guide for example apps

### StopCovid : France

This app has a **ProximityPayloadProvider** for generating a **ProximityPayload** for each **didReceiveRead** request. This is conceptually identical to the **PayloadDataSupplier** in HERALD, thus integration can be achieved by wrapping the **ProximityPayloadProvider** into a **payload:timestamp** function and wrapping the **BluetoothProximityPayload** as **PayloadData.** Given the fixed length payload, no further action is required for integration.

For contact logging, the current code uses a **BluetoothCentralManagerDelegate** to receive detected payloads. This is conceptually similar to the **SensorDelegate** in HERALD, thus integration will involve redirecting existing delegate methods to the **SensorDelegate** callback functions to receive and record detected payloads and distance estimates.

### COVIDSafe : Australia

This app has an **EncounterMessageManager.shared.getAdvertisementPayload** function for generating a **Data** payload for each **didReceiveRead** request. The same payload is reused until expiry time. This is conceptually identical to the **PayloadDataSupplier** in HERALD, thus integration can be achieved by wrapping this into a **payload:timestamp** function, using the timestamp for expiry checks, and wrapping the **Data** payload as **PayloadData.** The code suggests this is a variable length payload, thus additional work will be required for implementing the **payload:data** function for partitioning concatenated payloads. If this is challenging, it is trivial to modify the Android code in HERALD to ensure only one payload is shared at a time.

For contact logging, the current code uses an **EncounterRecord** to save received payload and distance data to an on-device database. The same code can be used within a **SensorDelegate** to acquire payload and distance measurement data for storage.