NUVATION ENGINEERING
ELECTRONIC PRODUCT DESIGN AND MANUFACTURING

SILICON VALLEY HEADQUARTERS
151 GIBRALTAR CT
SUNNYVALE, CA 94089 USA
408.228.5580

WATERLOO DESIGN CENTER
332 MARSLAND DR., SUITE 200
WATERLOO, ON N2J 3Z1 CANADA
519.746.2304

WWW.NUVATION.COM

WNG01: DIGITAL CONTROLLER FOR COOLANT & OIL DOOR ACTUATORS

Mobile Software Design Description

For Michael Malcolm

Last Updated: *September 18, 2019,* ***Revision*** *1.4*

Approvals for Rev 1.4

NUVATION ENGINEERING

Michael Malcolm	Nuvation
_____	_____
Name	Name
_____	_____
Signed	Signed
_____	_____
Title	Title
_____	_____
Date	Date

Revision History

Revision	Date	Description	By
0.1	Nov 27, 2017	Initial Draft	Chris Bell – Nuvation
0.2	Dec 21, 2017	Updates from requirements feedback	Chris Bell – Nuvation
0.3	Dec 22, 2017	Modifications from internal review feedback	Chris Bell – Nuvation
1.0	Nov 19, 2017	Modifications from client feedback	Chris Bell – Nuvation
1.1	Jun 22, 2018	Back documentation	Chris Bell – Nuvation
1.2	Dec. 3, 2018	Added new configuration_service characteristics	Jeff Teng – Nuvation
1.3	Mar. 27, 2019	Added information on the Data Service	Jeff Teng – Nuvation
1.4	Sep 17, 2019	Back-documentation	Chris Bell - Nuvation

Table of Contents

Approvals for Rev 1.4 2

Revision History 3

1 Introduction 10

1.1 Assumptions 10

1.2 Reference Documentation 10

1.3 Related Documentation 10

2 Communications between App and Controller 11

2.1 BLE Services and Characteristics 11

2.2 Advertising and Discovery 12

2.3 Bonding 13

3 Device Requirements 14

3.1.1 Permissions 14

4 Functionality 15

4.1 System Status Indicators 15

4.1.1 BLE Service 15

4.2 Oil System Monitoring 16

4.2.1 BLE Service 16

4.3 Coolant System Monitoring 17

4.3.1 BLE Service 17

4.4 Alert Notifications 17

4.5 Event Logging 18

4.5.1 Post-Flight Log Transfer 18

4.5.2 Controller Timestamps 18

4.5.3 Log Output Format 19

4.5.4 BLE Service 20
4.6 Graphing 20
4.6.1 Graph Example 21
4.6.2 BLE Service 21
4.7 Configuration 22
4.7.1 Set-Point Temperatures 22
4.7.2 WOW Switch 22
4.7.3 Aircraft Registration Number 23
4.7.4 Clear Event Logs 23
4.7.5 BLE Service 23
4.8 Firmware Upgrade 24
4.8.1 BLE Service 24
4.9 General Data Service 25
4.9.1 BLE Service 26
5 User Interface 27
5.1 Styling 27
5.2 Internationalization and Text String Customization 27
5.3 Screens 27
5.3.1 Header 27
5.3.2 Home Screen 28
5.3.3 Details Screens 33
5.3.4 Log Data Screen 36
5.3.5 Actuator Controller Screen 39
5.3.6 Aircraft Configuration Screen 42
5.3.7 Control Parameters Screen 45

6 Developer Guide 47

6.1 Changing the included Controller Firmware Version 47

6.2 Code Structure 48

Connection Process 50

6.3 50

Appendix A – GATT Protocol 51

System Status Service – UUID: 0000 51

Oil Monitor Service – UUID: 0100 51

Coolant Monitor Service – UUID: 0200 52

Configuration Service – UUID: 0300 52

Logs Service – UUID: 0400 53

Firmware Service – UUID: 0500 54

Data Service – UUID: 0600 54

Appendix B – Log File Transfer Protocol 55

Log Service 55

Control Characteristic Values 55

Response Characteristic Values 55

Sequence Flow 56

Process 57

Appendix C – Firmware Upgrade Protocol 58

Firmware Service 58

Control Characteristic Values 58

Data Characteristic Values 58

Response Characteristic Values 59

Control Sequence 59

List of Figures and Tables

Figure 1 - BLE Services and Characteristics Example 12

Figure 2 - Example Real-Time Graph 21

Figure 3 - Data Service Packet Format 25

Figure 4 - Main Screen - Normal Operation 29

Figure 5 - Main Screen - Alert Status 30

Figure 6 - Main Screen - No Bluetooth Connection 31

Figure 7 - Navigation Menu 32

Figure 8 - Coolant Details Screen 34

Figure 9- Oil Details Screen 35

Figure 10 - Log Data Screen 37

Figure 11 - Clear Logs Pop-up 38

Figure 12 - Actuator Controller Screen 40

Figure 13 - Upgrade Firmware 41

Figure 14 - Aircraft Configuration Screen 43

Figure 15 - Aircraft Configuration Screen 44

Figure 16 - Control Parameters Screen 46

Figure 17 - BLE Connection Process Flowchart 50

Figure 18 - Log Transfer Sequence Diagram 56

Table 1 - GATT Profile - system_service 15

Table 2 - GATT Profile - oil_service 16

Table 3 - GATT Profile - coolant_service 17

Table 4 - GATT Profile - logging_service 20

Table 5 - GATT Profile - configuration_service 23

Table 6 - GATT Profile - firmware_service 24

Table 7 - Defined send_data Types 25

Table 3 - Defined receive_data Types 25

Table 9 - GATT Profile - data_service 26

Table 10 - Chracteristic Details - System 51

Table 11 - Chracteristic Details - Oil 51

Table 12 - Chracteristic Details - Coolant 52

Table 13 - Chracteristic Details - Configuration 52

Table 14 - Chracteristic Details - Logs 53

Table 15 - Chracteristic Details - Firmware 54

Table 16 - Chracteristic Details - Data 54

Table 17 - Logs Characteristics Properties 55

Table 18 - Log Transfers - Control Characteristic Values 55

Table 19 - Log Transfers - Response Characteristic Values 55

Table 20 - Firmware Characteristic Properties 58

Table 21 - Firmware Upgrade - Control Characteristic Values 58

Table 22 - Firmware Upgrade - Description of Data Packet 58

Table 23 - Firmware Upgrade - Response Characteristic Values 59

Table 24 - Firmware Upgrade Sequence 59

Glossary

Term	Description
ATP	Acceptance Test Plan
BLE	Bluetooth Low Energy
GATT	Generic Attribute Profile
GUI	Graphical User Interface
JSON	JavaScript Object Notation
PRD	Product Requirements Document
PWP	Project Work Plan
RAM	Random Access Memory
ROM	Read Only Memory
SDD	Software Design Description
TBD	To Be Determined
USB	Universal Serial Bus
WOW	Weight-on-wheels

1 Introduction

This document describes the iOS mobile application that connects to the actuator controller in order to provide real-time monitoring and configuration of the controller. The app is built to run on a current iOS tablet and communicates with the controller via Bluetooth Low Energy (BLE, Bluetooth Smart).

The primary purpose of the app is continuous monitoring of the oil and coolant temperatures as well as the current state of the door actuators. It also has the ability to adjust the configuration of temperature set-points and other features, download log data from the controller, and push firmware updates.

1.1 Assumptions

Security of communications between the iOS app and the controller is not a primary concern; however, security features that are built in to the BLE protocol will be utilized.

Compliance with industry or safety standards is outside of the scope of this document.

1.2 Reference Documentation

Document Name	Location
Digital Controller Design Overview (v3)	\WNG01\ext\docs\Digital Coolant and Oil Actuator Controller v.3.docx

1.3 Related Documentation

Document Name	Location
Digital Controller Firmware SDD	\WNG01\ext\docs\WNG01_FW_SDD.docx

2 Communications between App and Controller

All communications between the controller and the iOS app is performed over Bluetooth Low Energy (BLE).

The controller acts as a BLE **peripheral** and the iPad will be the BLE **central**. As a peripheral, the controller can send out advertising packets but is not responsible for initiating the connection. As the central, the iPad listens for the advertising packets and sets up the connection.

Battery requirements for a BLE central device are much higher than those of a peripheral.

Apple device operating systems use the Generic Attribute Profile (GATT) API to communicate with a connected BLE device. GATT uses the terms **server** and **client** to distinguish between the two connected devices. The client is the device that initiates GATT commands and requests, and receives responses. The controller acts as the GATT server and the iPad acts as the client, making requests to the controller.

2.1 BLE Services and Characteristics

The BLE server advertises one or more **services**, each of which contains one or more **characteristics** (Figure 1 shows an example BLE service list).

A characteristic is a single piece of data that contains a value, with a default maximum size of 20 bytes, as well as optional descriptors, which provide meta-data about the characteristic (the maximum size is increased to 238 bytes if the hardware supports BLE 4.2). Characteristics are used for all data exchanged between the server and client, and can be readable, writable, or both.

A service is a logical grouping of characteristics.

Services and characteristics are identified using 128 bit UUIDs. These UUIDs are passed to the client and stored, and are used for communicating with the server after the discovery phase.

Read characteristics can be read in one of two ways: the client can poll a characteristic on a regular basis to retrieve updated information, or the client can subscribe to a characteristic, allowing the server to push values to the client at the server's discretion. We use subscribed characteristics whenever possible in order to reduce the overhead of constant polling.

Details about the app's specific services and characteristics will be discussed in a later section.

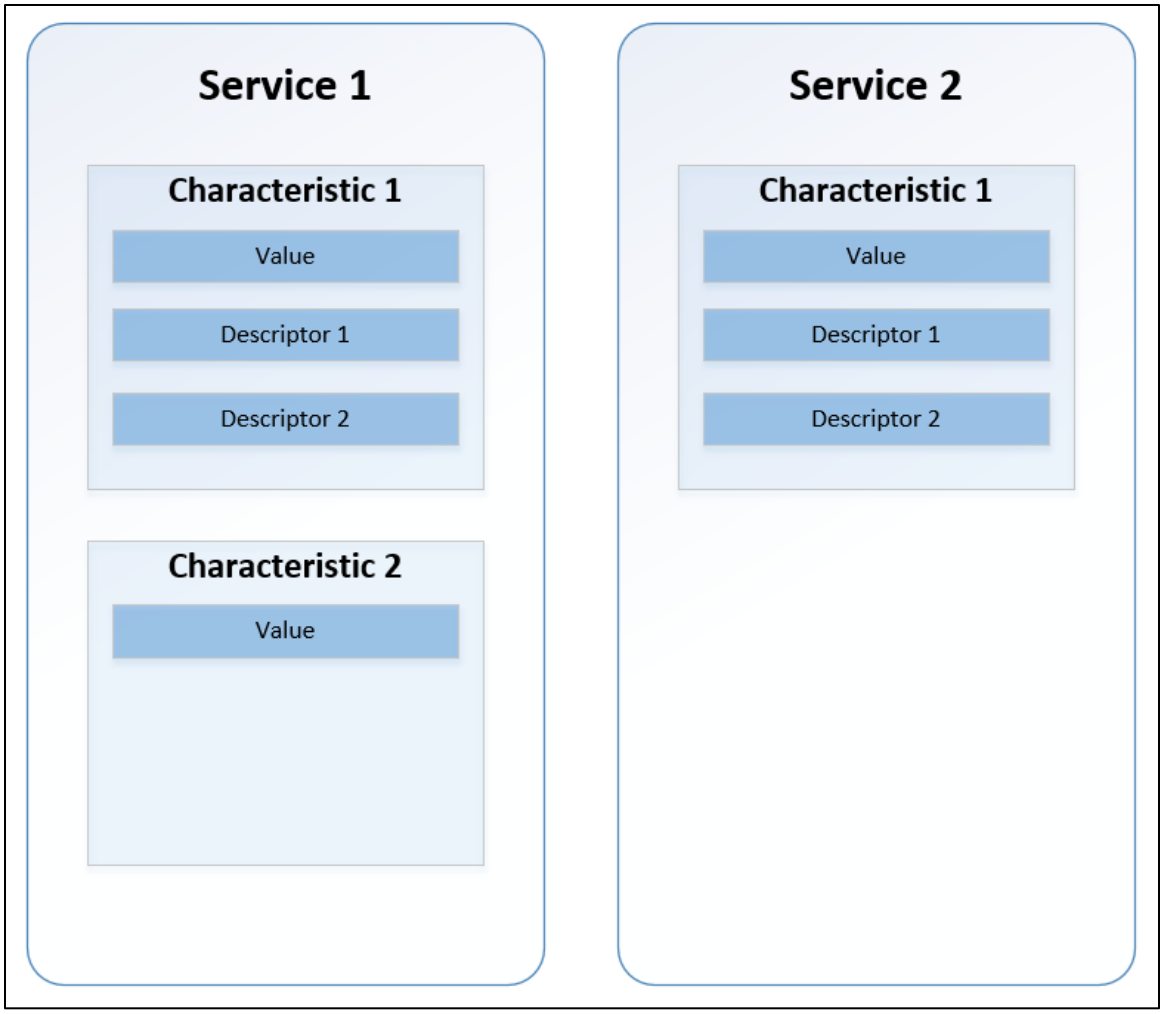

Figure 1 - BLE Services and Characteristics Example

2.2 Advertising and Discovery

In order to initiate the BLE connection process, the controller needs to be put in a "discoverable" mode, during which it begins broadcasting advertising packets at set intervals. This occurs whenever the controller is powered on and not connected to a central.

The advertising packets contain a UUID that uniquely identifies one of the BLE services that the controller offers. The app only scans for devices advertising this particular UUID.

At the same time, the app begins scanning for a peripheral with that same list of services. No user interaction will be required during this process. Once discovered, the app will initiate a connection to the controller, and the controller will stop advertising. At this point, the app can begin requesting details about the various services, and can start reading from and writing to characteristics.

In the event that the app loses connection to the controller for any reason, the app will immediately begin scanning for the controller again and will automatically attempt to reconnect when the controller is found again.

2.3 Bonding

Once the central is connected to the peripheral, it can start accessing the peripheral's characteristics, so long as the characteristics are not marked as "secure". However, at this point, the connection is not encrypted, and the iOS device will not "remember" the controller, so it will need to re-discover it the next time it's brought in range.

The central can initiate a bonding request to the peripheral, which will trigger an exchange of encryption keys and allow all future communications to be secured. It will also cause the iPad's operating system to remember the peripheral device, allowing it to automatically connect the next time it's in range. The bonding request will trigger a "request to pair with device" notification to the iPad user, which will need to be accepted before the bonding can take place.

Currently, there are no requirements for establishing a secure communication channel between the iOS device and the controller. However, a bonding strategy may still be considered in the future in order to take advantage of the "free" encryption.

3 Device Requirements

Since all communications between the controller and the iOS device are performed over BLE, the app will only function on devices that fully support BLE.

BLE 4.0 is fully supported on all versions of the iPad Mini and iPad Air, as well as on the iPad 3rd generation devices and all subsequently released iPads. It is fully supported on the iPhone 4s and newer. Devices must be running at least iOS version 5 in order to use BLE.

BLE 4.2 is supported on the iPhone 6 and newer and the iPad Mini 4 and newer. It is currently not expected that the controller will support BLE 4.2, so there will be no advantage to targeting iOS devices that support it.

This app currently targets all versions of the iPhone and iPad Mini that support BLE.

3.1.1 Permissions

iOS does not require specifying any special permissions to allow the app to use BLE. All apps are given a set of basic permissions including BLE access. When submitting the app to the Apple App Store, it should be targeted to iOS version 5 at minimum. This can be enforced by adding the value "bluetooth-le" to the "UIRequiredDeviceCapabilities" key in the Info.plist file. This enforces that the app requires the device to support BLE, and won't allow the app to be installed on any device that does not support it.

At run time, when the app attempts to access Bluetooth services, the app will prompt the user to allow it. The user only needs to do this once, after which the app will also be able to access those services.

In order to allow the app to work properly while it's in the background, certain background modes need to be enabled. The following background mode capabilities will need to be set in the info.plist file:

"**Uses Bluetooth LE accessories**" allows the app to continue to listen for BLE characteristic data while the app is in the background.

"**Audio, AirPlay, and Picture in Picture**" will allow the app to play audio notifications while in the background.

4 Functionality

This section describes specific functions of the app in detail.

4.1 System Status Indicators

In order to avoid the extra overhead added by frequent polling, the app subscribes to a BLE service served by the controller. The controller determines when to send updated data to the app. If required, the app can still request an update from the characteristic while subscribed to it.

The app always displays the current status of the BLE connection between the app and the controller. While the BLE connection is in a disconnected state, the app locks out access to the controller configuration settings. It also clearly shows that the system is not receiving sensor data by greying out the temperature and door gauges, and by not displaying any values in the gauges.

The controller has its own temperature sensor and reports this to the app. If the controller's temperature goes above the accepted range, it will send a high temp alert to the app, and the app will display a warning message. The accepted temperature range is managed by the controller firmware. The alert will also be used to communicate any other system-level alert statuses.

4.1.1 BLE Service

Service Name: system_service

Characteristics:

Name	Value	Properties
controller_temp	int [Celsius, 1/100th degree], timestamp	Read
alert_bitmask	bitmask [controller temp high, watchdog tripped]	Read
engine_power	boolean [0:off, 1:running]	Read
set_timestamp	UNIX timestamp	Write
flight_number	int [flight number]	Read
set_reg_name	ASCII aircraft registration	Write, Read

Table 1 - GATT Profile - system_service

4.2 Oil System Monitoring

One of the primary features of the app is to display the current status of the oil system. The controller monitors the oil temperature and the current position of the air intake door for the oil cooling system.

The oil door system can be switched between 'automatic' mode and 'manual' mode by toggling a lockout switch in the cockpit. The mobile app does not have the capability to change operating modes; however it displays the current mode on the main screen.

The app displays the current oil temperature on the Main and Oil Details screens in degrees Celsius. It displays the position of the air intake door on the Main and Oil Details screens.

If the oil temperature exceeds the acceptable range, the app displays a warning to the user. Alert statuses are determined by the controller; the app receives a notification of the alert type. The acceptable temperature range is managed by the controller's firmware.

In order to provide data for the app's real-time graphing functions, the oil system temperature readings are cached locally.

4.2.1 BLE Service

Service Name: oil_service

Characteristics:

Name	Value	Properties
system_mode	int [0:manual, 1:manual opening, 2:manual closing, 3:auto]	Read
temp	int [Celsius, 1/100th degree]	Read
door_position	int [percentage open]	Read
alert_bitmask	bitmask [high temp, oil actuator malfunction]	Read

Table 2 - GATT Profile - oil_service

4.3 Coolant System Monitoring

The app displays the current status of the engine coolant system. This works identically to the oil system. The controller reports the current temperature of the coolant and the position of the coolant air intake door, and the app display them on the Main and Coolant Details screens. Alerts are displayed if the app is notified of an alert by the controller.

4.3.1 BLE Service

Service Name: coolant_service

Characteristics:

Name	Value	Properties
system_mode	int [0:manual, 1:manual opening, 2:manual closing, 3:auto]	Read
temp	int [Celsius, 1/100th degree]	Read
door_position	int [percentage open]	Read
alert_bitmask	bitmask [high temp, oil actuator malfunction]	Read

Table 3 - GATT Profile - coolant_service

4.4 Alert Notifications

In the event that the controller notifies the app of an alert state, such as a high temperature warning or an actuator malfunction, the app will notify the user in several different ways. While the app is in the foreground there will be error text displayed at the bottom of the Main screen. The app will also issue a system notification through the operating system and will use Text-To-Speech to speak the alert aloud, in English. The system notifications and audio alerts work whether the app is in the foreground or background.

Some alert states also have specific visual representations. A temperature error will cause the temperature readings to be displayed in flashing red text. An actuator error will display an error icon prominently on the door gauge.

As long as the alert status is still in effect, the alert visuals will remain and the audio notification will play approximately every 3 seconds.

4.5 Event Logging

The controller maintains a log of all event and temperature data in its local memory. The logs record timestamped temperature readings from the oil, coolant and controller sensors at regular intervals, determined by the controller firmware. Event data is recorded at irregular intervals, as events occur. The controller is able to store the entire log history in its local memory, though the log data can be manually cleared via the app.

The user has the ability to request a log transfer from the controller to the app. As the app receives log data for each flight, it parses the data and stores the log records locally. Subsequent log transfers will only request data that isn't already stored in the local logs.

Exact details of the log transfer process are discussed in **Appendix B** of this document and in the **Digital Controller Firmware SDD**.

Once the log data transfer is complete, the app will generate a CSV file of all of the log data and store it on the iPad's internal storage.

4.5.1 Post-Flight Log Transfer

At the end of each flight, when the controller notifies the app the engine has been turned off, the app will immediately request a log transfer from wherever it last left off. This process is initiated in the background, but its progress is still displayed on the settings page and the transfer can be cancelled by the user. Automatic log transfers dramatically speed up the log transfer process when the user manually requests a log transfer. If the user always keeps the app connected to the controller during flights, then log transfer requests will be nearly instantaneous since all data will already be stored locally and will not need to be sent over BLE.

4.5.2 Controller Timestamps

The digital controller does not have a continuous source of power, so it can't maintain a real-time clock. The controller instead logs events in "flight time" and "flight number". Flight time is the time in seconds from the last time a new flight was initiated. Flight number is an integer that increments each time a new flight is started.

Every time the controller connects to the app over BLE, the app calls the "setTimestamp" characteristic, passing it a UNIX timestamp. The controller logs this timestamp as a "BLE connection" event. Later, when the user downloads log data to a computer, this timestamp can be used in combination with the flight time to extrapolate the calendar time for each other log entry during that flight.

At connection, the app also calls the "getFlight" characteristic, which retrieves the flight number from the controller. To the app, an interruption in BLE communications will appear the same as the controller powering off at the end of the flight. The app uses the flight number in order to tell if data received after a new BLE connection is from a new flight or an ongoing one.

4.5.2.1 Flight Number

The controller determines when to increment the flight number. A new flight is considered to have started after the engine power status changes from off to on and it remains there for a period of at least 30 seconds.

4.5.3 Log Output Format

When the user manually initiates a full log transfer, the app will request any missing log records from the controller and add them to the local data store. It will then output all log data as a comma-separated CSV file to the app's document store on the device. It will also create a separate log file for system logs. These logs can be transferred to a computer using iTunes.

Each record in the CSV output file will have the following fields, separated by comma:

- Row ID
- Flight number
- Flight time
- Event type
- Event value

The first field it outputs for each log entry is a Row ID, which is an integer that increments for every log record over the lifetime of the controller. It is only reset if the controller logs are fully wiped. This field keeps all log records in chronological order.

Flight Number and Flight Time are output for each log record. A Flight Number of "0" represents an event that was recorded while the engine was off.

Event Type is a text field that describes the Event Value, such as "Engine Start" or "Coolant Door Position (revolutions)".

4.5.3.1 System Logs

System logs are logs created by the iOS app itself as opposed to flight logs, which are retrieved from the controller. System logs track some events such as BLE connections and flight number changes, user initiated changes, and any errors encountered by the app.

System logs are written to the same location on the device as flight logs. Whenever the user initiates a manual log transfer, as well as each time the app is started, the system log CSV file will be written. System logs are organized by timestamp.

4.5.4 BLE Service

Service Name: logging_service

Characteristics:

Name	Value	Properties
log_control	mixed	Write
log_data	data	Read
log_response	mixed	Read

Table 4 - GATT Profile - logging_service

See Appendix B for a detailed description of the Log Transfer protocol

4.6 Graphing

As the app receives temperature and door position data from the controller, it uses it to populate real-time graphs of the flight. Each system, oil and coolant, has its own graph. These graphs are viewable in-flight on the details pages of the app.

Since the app will always have access to a real-time clock, the graphs are shown with the actual time, not the "flight time" from the controller. The graphs continue to append data as long as the flight number remains the same. When the app receives data from the controller with a new flight number, the old data is cleared from the graphs and graphing of the new flight will begin.

The graphs always show the data from the most recent flight, even after disconnecting from the controller or restarting the app.

While a flight is in-progress, the graphs are in a "rolling" state. New data points come in from the right side and push the graph to the left. Once 8 minutes of flight data (480 data points) have been recorded, each new data point will drop the oldest data point off the left side of the graph. This is done in order to reduce the strain on the device hardware, since the graphing framework redraws every point on the graph each time a new point is added. This data point limit can be adjusted in the app code; however, the value chosen must take into account the capabilities of the iOS devices being targeted for the app.

Once the engine status switches to the off state, indicating the end of the flight, the graphs are reloaded using all data points recorded during the flight.

Graphs can be pinched in order to zoom in or out and dragged in order to scroll from side to side.

4.6.1 Graph Example

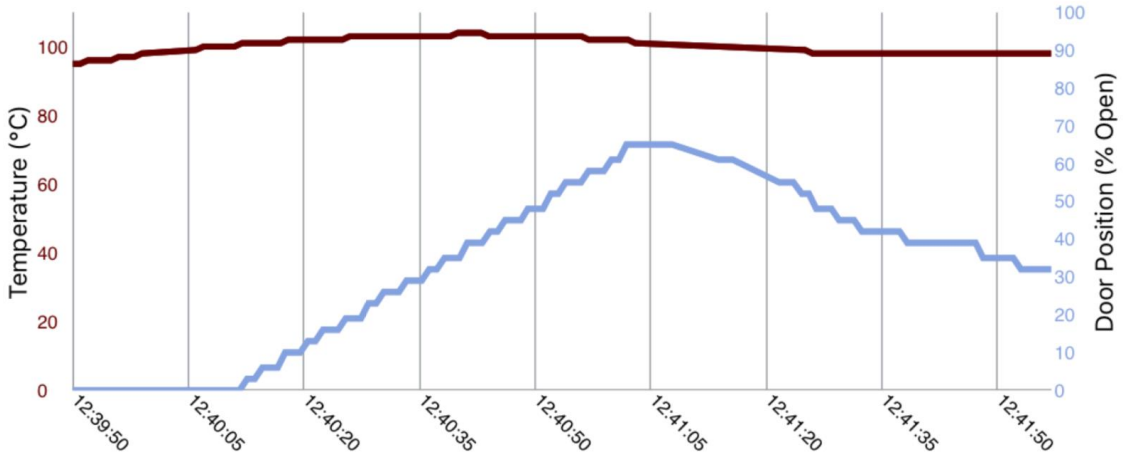

Figure 2 - Example Real-Time Graph

Above is an example of the real-time graph for the coolant system during a flight. The x-axis shows the current time. The app has two y-axes, one for the temperature and one for the door position. The two lines are color-coded to match their respective y-axis.

4.6.2 BLE Service

Graphs use data collected from other services.

4.7 Configuration

The app allows the user to set various configuration options for the system; these options are managed on various configuration screens that re accessible from the navigation menu.

The following configurations can be made from the app:

- Set oil temperature set-point
- Set coolant temperature set-point
- Enable/disable WOW switch
- Set/change aircraft registration number
- Clear controller event logs
- Calibrate oil door
- Calibrate coolant door
- Test alarm lights
- Upgrade controller firmware

Additionally, the following options are configurable when the controller is in maintenance mode and during beta testing. They will be used for fine-tuning the actuator control algorithm:

- Oil proportional gain
- Oil integral gain
- Oil differential gain
- Oil hysteresis threshold
- Coolant proportional gain
- Coolant integral gain
- Coolant differential gain
- Coolant hysteresis threshold
- PID control loop interval

4.7.1 Set-Point Temperatures

The controller will attempt to keep the oil and coolant systems at their respective set-point temperatures. Modifying the set-points will adjust these target temperatures. Changing the set-point temperature does not change the safe operating and/or warning temperature ranges. These are defined according to engine specs.

4.7.2 WOW Switch

Toggling this switch will enable/disable the Weight on Wheels feature. This feature is discussed in detail in WNG01_FW_SDD.docx.

4.7.3 Aircraft Registration Number

The aircraft registration number is an ASCII identifier, up to 8 characters, that identifies the aircraft that the controller is installed in. Once set, this identifier is embedded in the controller's BLE advertising packets under the standard GATT characteristic "**Device Name**", UUID: "**2A00**".

4.7.4 Clear Event Logs

The option to clear the event logs will send a notification to the controller that all past event logs in memory should be erased. This will be a destructive, non-recoverable operation, which is only meant to be done if the engine is overhauled or replaced. Therefore, selecting this option will first prompt the user with a warning and require confirmation before proceeding.

4.7.5 BLE Service

Service Name: configuration_service

Characteristics:

Name	Value	Properties
oil_temp_setpoint	int [Celsius, 1/100th degree]	Write, Read
coolant_temp_setpoint	int [Celsius, 1/100th degree]	Write, Read
clear_logs	none	Write
calibrate_door	int [1:oil, 2:coolant]	Write
oil_proportional_gain	int	Write, Read
oil_integral_gain	int	Write, Read
oil_differential_gain	int	Write, Read
oil_hysteresis_threshold	int	Write, Read
coolant_proportional_gain	int	Write, Read
coolant_integral_gain	int	Write, Read
coolant_differential_gain	int	Write, Read
coolant_hysteresis_threshold	int	Write, Read
temp_adc_filter_window	*uint*	*Write, Read Deprecated*
ctrl_loop_interval	uint	Write, Read

Table 5 - GATT Profile - configuration_service

Some of the configuration functions were added after the GATT profile was finalized and instead make use of the General Data Service described later.

4.8 Firmware Upgrade

The iOS app has the ability to push new firmware to the controller. Since the firmware images for the controller are small, the firmware image file is included directly in the app package on the iTunes App Store. This has several advantages. The user is not required to manually download the firmware from another location. We do not need to create a web server or find a third-party hosting site for distributing the firmware. Finally, it makes it easier to ensure that only valid firmware files are installed and it prevents malicious firmware from being hosted on third-party sites.

Firmware is stored in the app as a base64-encoded JSON file. The steps for encoding and building the firmware files are located later in the document in the Developer Guide section. The JSON file contains two separate firmware images; during a firmware upgrade, the controller will request one of the two images. The file also contains a CRC value for each of the firmware images.

When the app is loaded, and any time the user accesses the Actuator Controller screen, the app will poll the "firmwareVersion" characteristic in order to determine what firmware version is currently running on the controller. The installed and available firmware versions are displayed to the user. If the available version is different from the installed version, the user is given the option to perform an update.

Firmware upgrades take several minutes to complete and the user is prevented from navigating in the app while an upgrade is in progress. However, the user does have the option to cancel an in-progress upgrade, which terminates the file transfer.

At the end of a firmware file transfer, a CRC is sent, which the controller uses to validate the integrity of the firmware file. A detailed breakdown of the firmware upgrade process is located in Appendix C - Firmware Upgrade Protocol.

4.8.1 BLE Service

Service Name: firmware_service

Characteristics:

Name	Value	Properties
firmware_version	major.minor.patch	Read
firmware_control	mixed	Write
firmware_data	data	Write
firmware_response	mixed	Read

Table 6 - GATT Profile - firmware_service

4.9 General Data Service

The General Data Service was added into the GATT profile in order to allow for future development without the need to upgrade the GATT profile stored on the controller's BLE module. The data service consists of two generic characteristics, one for reading data from the controller and one for writing to it.

Each characteristic has a 20 byte value length. The first 2 bytes make up a 16-bit integer that denotes the type of packet being sent, the remaining 18 bytes contain the data (**Error! Reference source not found.**). The exact makeup of the data depends on the function it's being used for.

Type 1	Type 2	Data 1	Data 2	Data 3	…	Data 15	Data 16	Data 17	Data 18

Figure 3 - Data Service Packet Format

Currently-defined functions making use of the data service are listed in the following two tables:

Type Code	Description	Values	Notes
0x01	Full Status	None	Requests a status update
0x02	WOW state	0: Disabled, 1: Enabled	Set the enabled state of the WOW feature
0x05	Test Alarm Lights	None	Trigger an alarm light test
0x06	*Oil Pre-bias*	*int [Celsius, 1/100th degree]*	*Sets the oil pre-bias DEPRECATED*
0x07	*Coolant Pre-bias*	*int [Celsius, 1/100th degree]*	*Sets the coolant pre-bias DEPRECATED*

Table 7 - Defined send_data Types

Type Code	Description	Values	Notes
0x02	WOW state	0: Disabled, 1: Enabled	Read the enabled state of the WOW feature
0x03	WOW Status	0: not active, 1: active	Read whether the aircraft is currently in a WOW state
0x04	Log Clear Complete	None	Indicates that the controller has finished clearing logs
0x06	*Oil Pre-bias*	*int [Celsius, 1/100th degree]*	*Reads the oil pre-bias DEPRECATED*
0x07	*Coolant Pre-bias*	*int [Celsius, 1/100th degree]*	*Reads the coolant pre-bias DEPRECATED*

Table 8 - Defined receive_data Types

4.9.1 BLE Service

Service Name: data_service

Characteristics:

Name	Value	Properties
send_data	varies	Write
receive_data	varies	Read

Table 9 - GATT Profile - data_service

5 User Interface

This section describes prototype designs of the user interface screens that make up the app.

5.1 Styling

The app was developed with a minimal level of styling. A graphic designer can be consulted if it's determined that a more enhanced level of styling is required, but this is currently out of the scope of this project.

5.2 Internationalization and Text String Customization

No internationalization of text is required in this application; all text will be displayed in English. This is consistent with aviation practices, where all communication and documentation is generally performed in English.

5.3 Screens

5.3.1 Header

The top of the screen is a small header containing a Bluetooth connectivity indicator and a button that links to the Configuration screen.

5.3.1.1 Bluetooth Connectivity

When connected to the controller, the aircraft registration number is displayed in the header. When disconnected, it is replaced with the words "Not Connected".

While disconnected from the controller, the app does not receive any sensor readings. In order to make this obvious to the user, all digital readouts are replaced with question marks and the gauges do not show a reading. The app also disables all configuration controls, such as those for adjusting set-points and updating firmware.

5.3.1.2 Menu

A dropdown navigation menu is accessible on all main screens.

5.3.2 Home Screen

This screen is displayed when the app is loaded. It provides an overview of the current state of the system. There is a navigation bar link on all pages other than this one that will return the app to the Home screen.

During normal operation, while the app is connected to the controller and receiving sensor readings, this screen displays two main sections. The top section shows the temperature of the oil and coolant systems and also has indicators showing the operating mode of each system (Manual, Manual Opening, Manual Closing, Auto). The lower section displays the current position of the coolant and oil air-intake doors.

Temperature is displayed using an analog gauge accompanied by a digital readout for each system. Temperatures are always displayed in degrees Celsius. Each digital gauge has a colored range that indicates the recommended safe operating temperature, indicated in green. Another colored range, in red, covers temperatures on the gauge that are above the warning temperature of the system.

Tapping anywhere on the top section will switch to one of the details screens.

The door positions are consolidated into a single custom gauge. This gauge is a rough representation of the actual underbelly of the aircraft, with moving door indicators positioned similarly to the real ones. The gauge is not overly stylized though; it is kept simple for ease of readability. Underneath each door in the gauge is a digital readout of the door's position, shown as the number of actuator revolutions from closed, overtop the percentage of its full arc ("revolutions"/"percent open").

5.3.2.1 Alerts

Any alerts reported by the controller cause a message to be displayed in a third section on the bottom of the screen. This section is not displayed unless there's an alert.

Wherever appropriate, alerts are also displayed on the gauges. During a high temperature alarm, the temperature gauge will read in the red zone on the gauge and the digital temperature reading will turn red. An actuator failure will display a failure icon overtop of the door in the gauge. These alerts will still be accompanied by a message at the bottom of the page.

5.3.2.2 Screen – Normal Operation

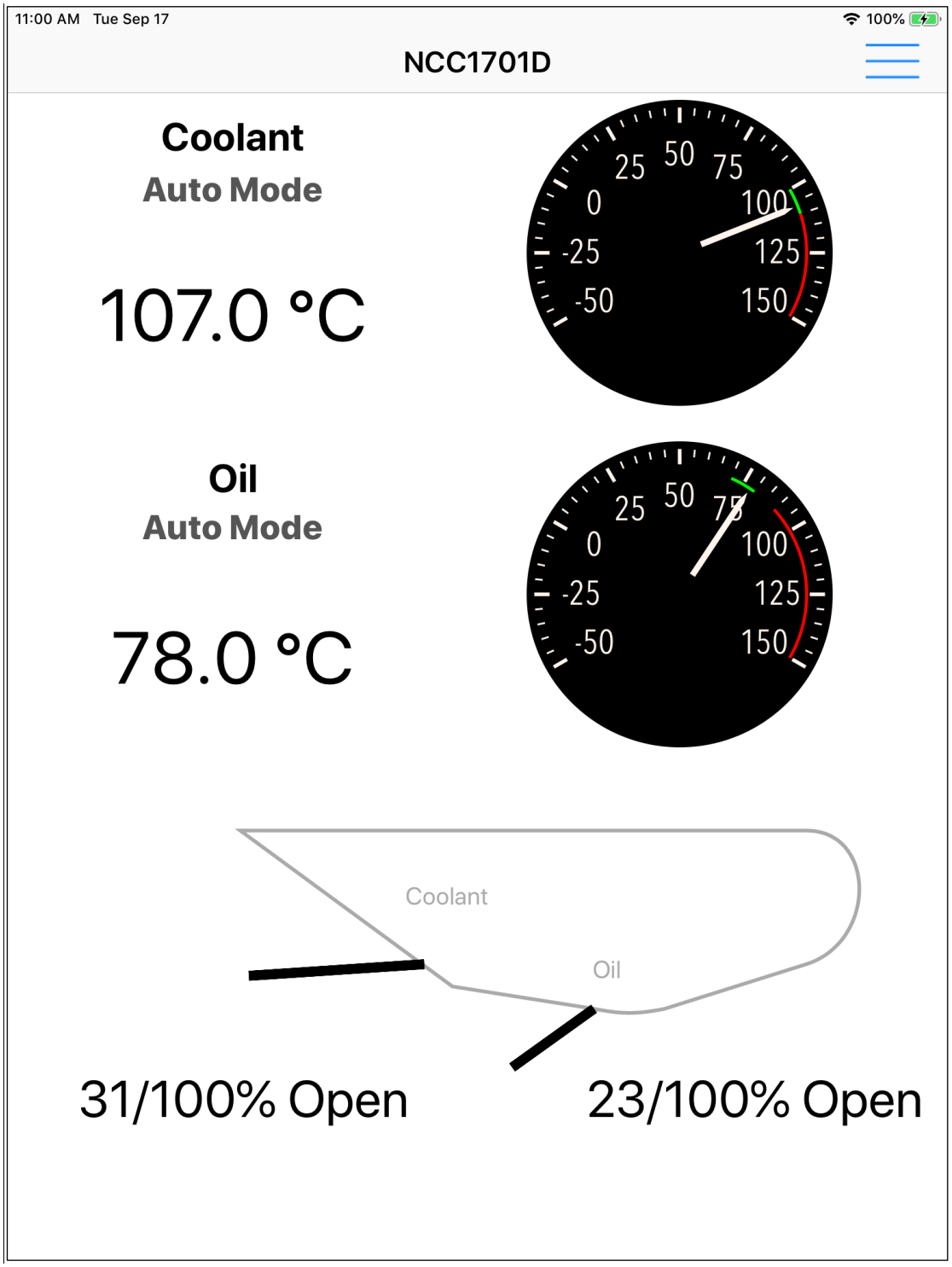

Figure 4 - Main Screen - Normal Operation

5.3.2.3 Screen – Main Screen Alert Status

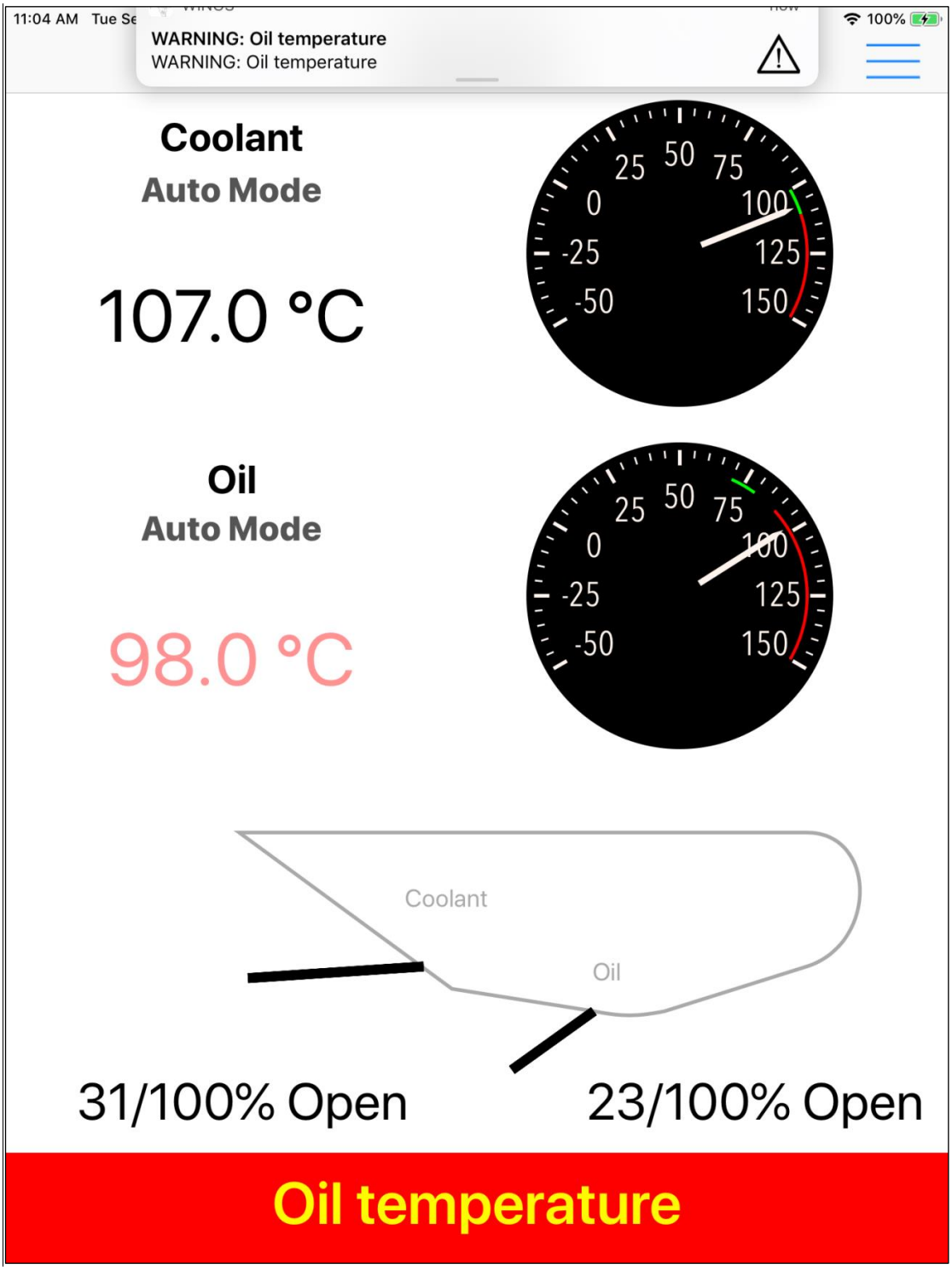

Figure 5 - Main Screen - Alert Status

5.3.2.4 Screen – No Bluetooth Connection

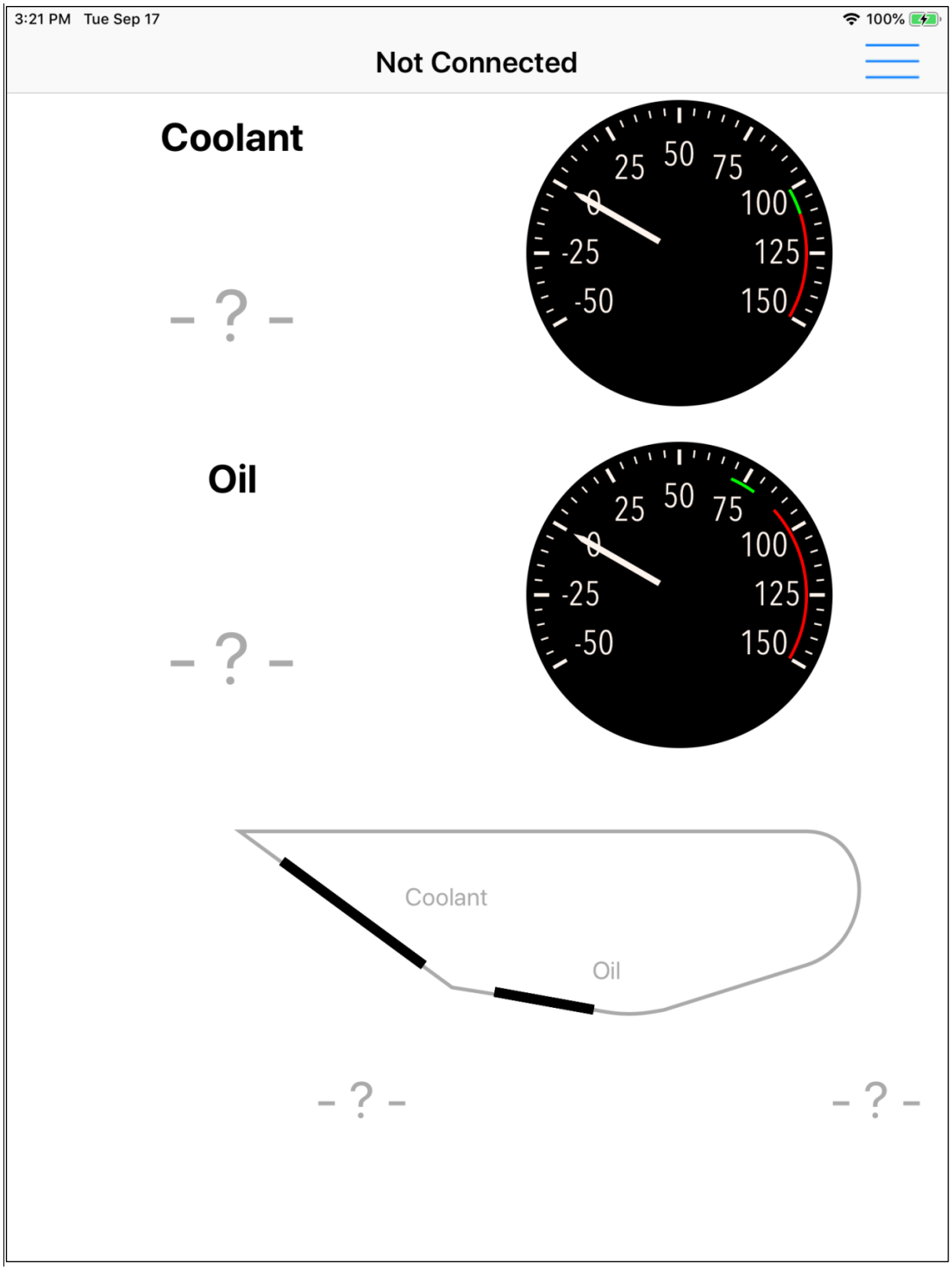

Figure 6 - Main Screen - No Bluetooth Connection

5.3.2.5 Navigation Menu

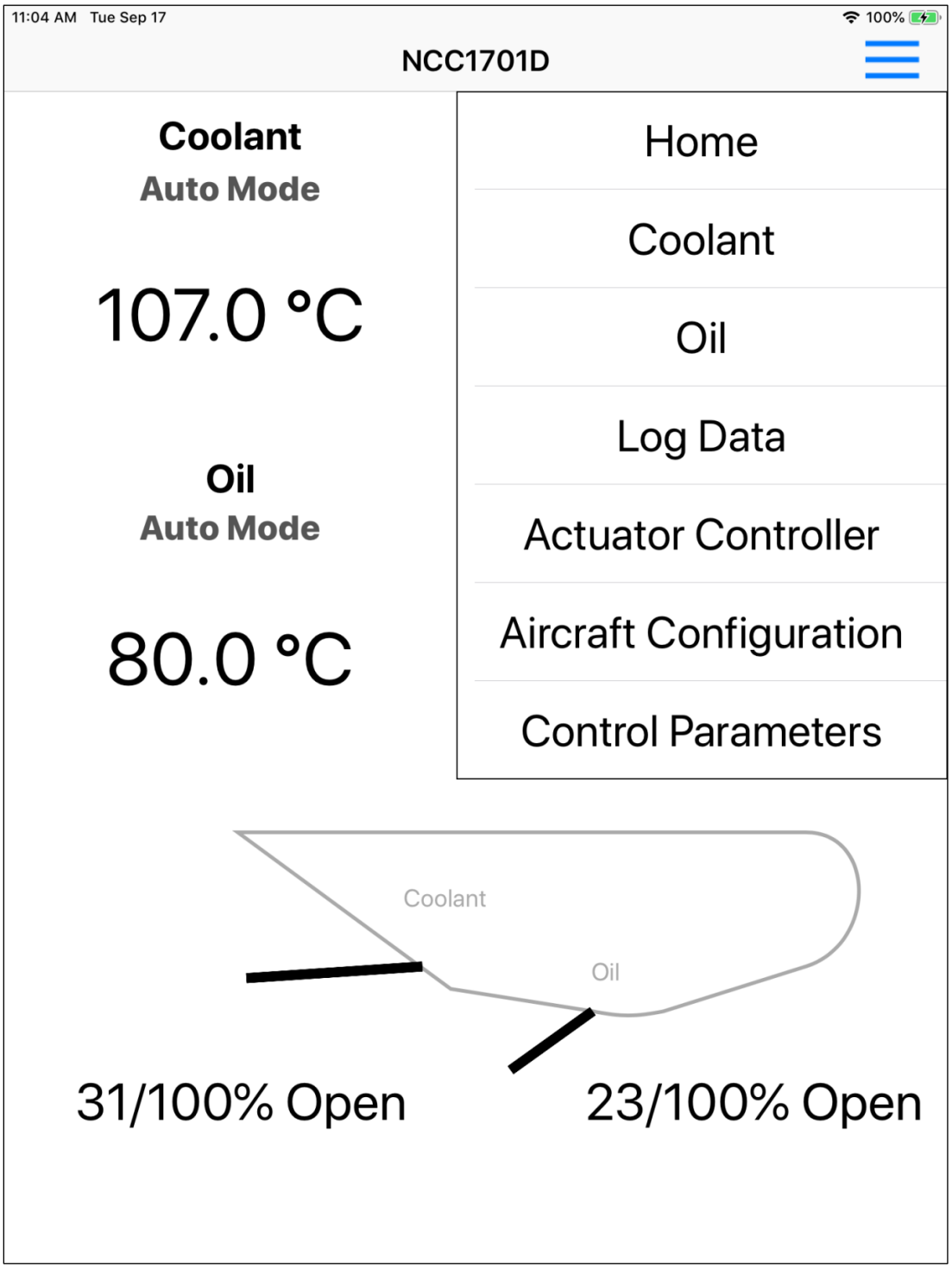

Figure 7 - Navigation Menu

5.3.3 Details Screens

The oil and coolant systems each have a details screen, which is accessed from the main page by tapping on their respective sections or from the navigation menu.

The details screen has a larger version of the temperature display and a digital readout of the door position. It also has a graph showing temperature and door position data during a flight. The graph updates in real-time as new sensor data is sent from the controller. Two-finger pinching the graph allows the user to narrow the time range displayed on the graph.

5.3.3.1 Screen – Coolant Details

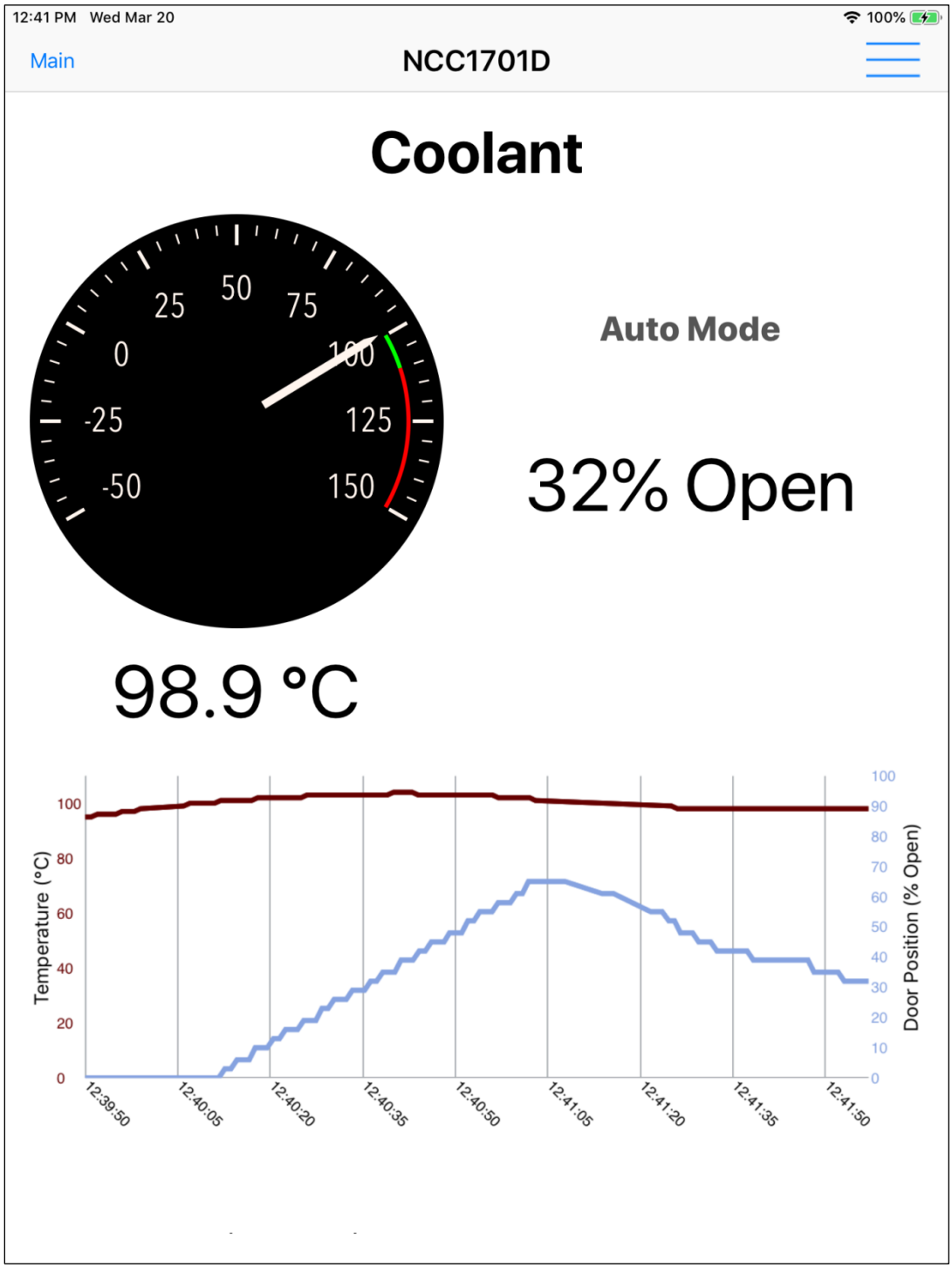

Figure 8 - Coolant Details Screen

5.3.3.2 Screen – Oil Details

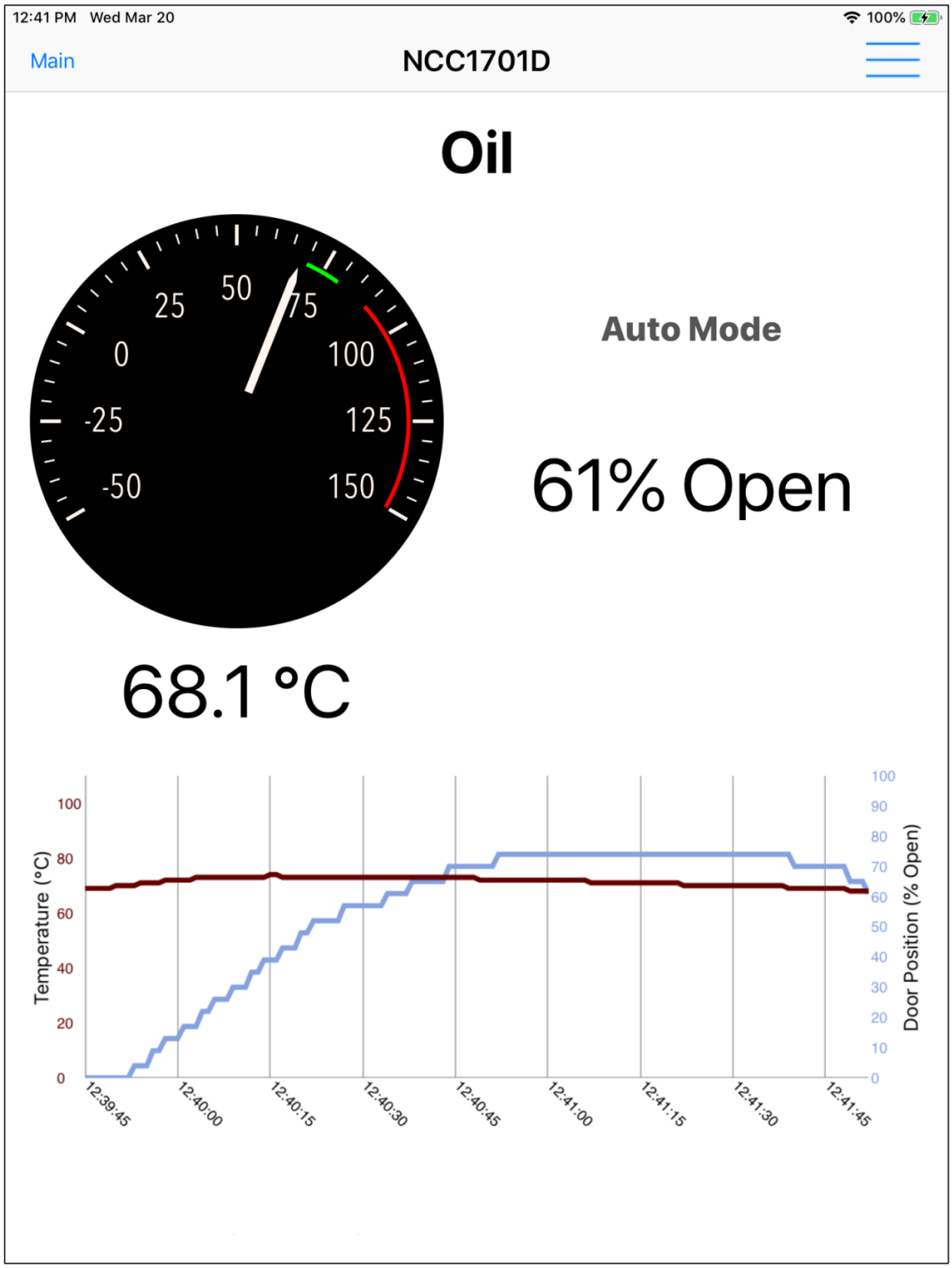

Figure 9- Oil Details Screen

5.3.4 Log Data Screen

From this screen the user can retrieve logs from the controller, observe the progress of a log transfer, or cancel a log transfer. Clearing logs is also accessed from this screen, though it will pop-up a confirmation window with additional information first.

5.3.4.1 Screen – Log Data

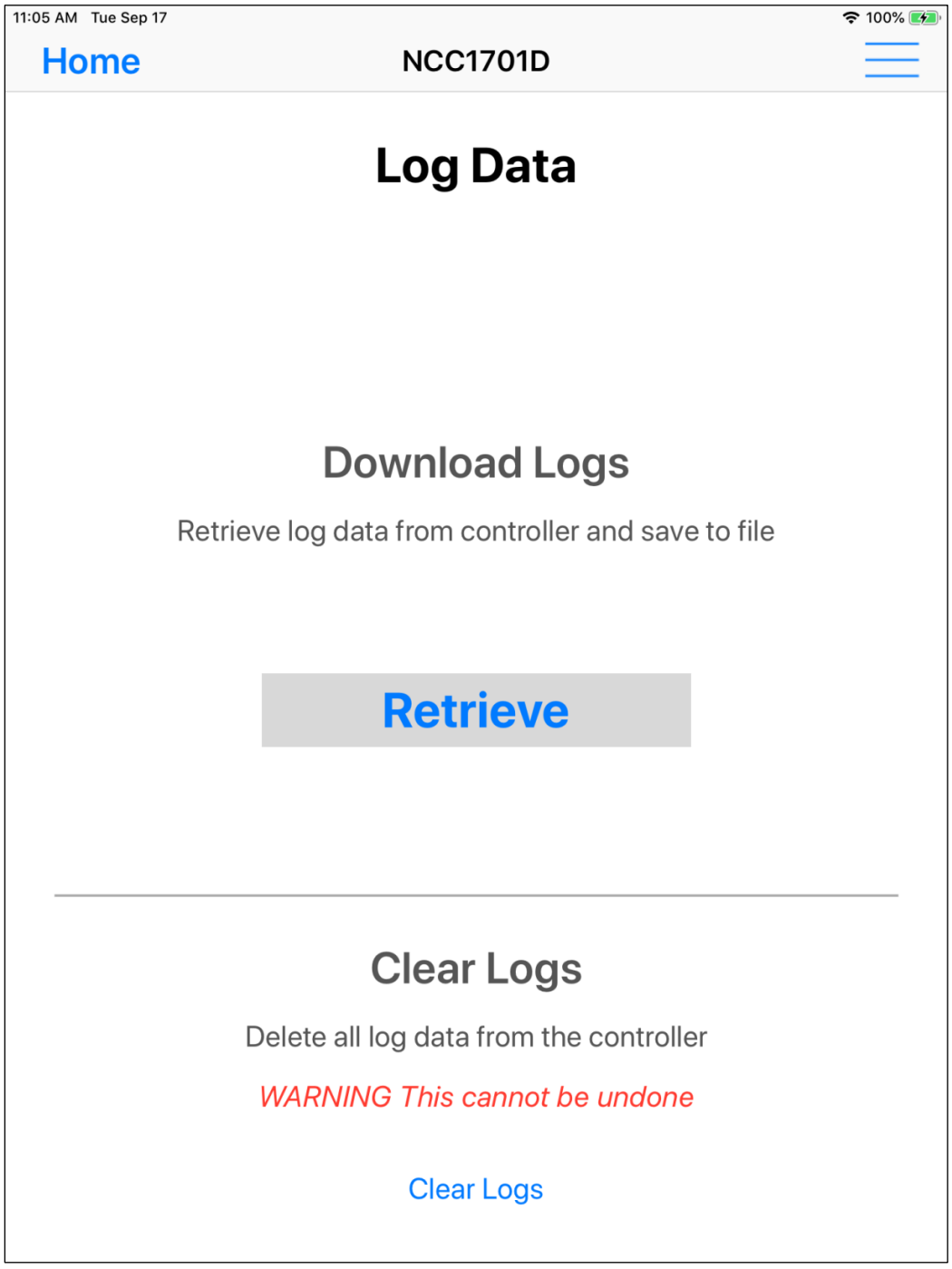

Figure 10 - Log Data Screen

5.3.4.2 Screen – Clear Logs

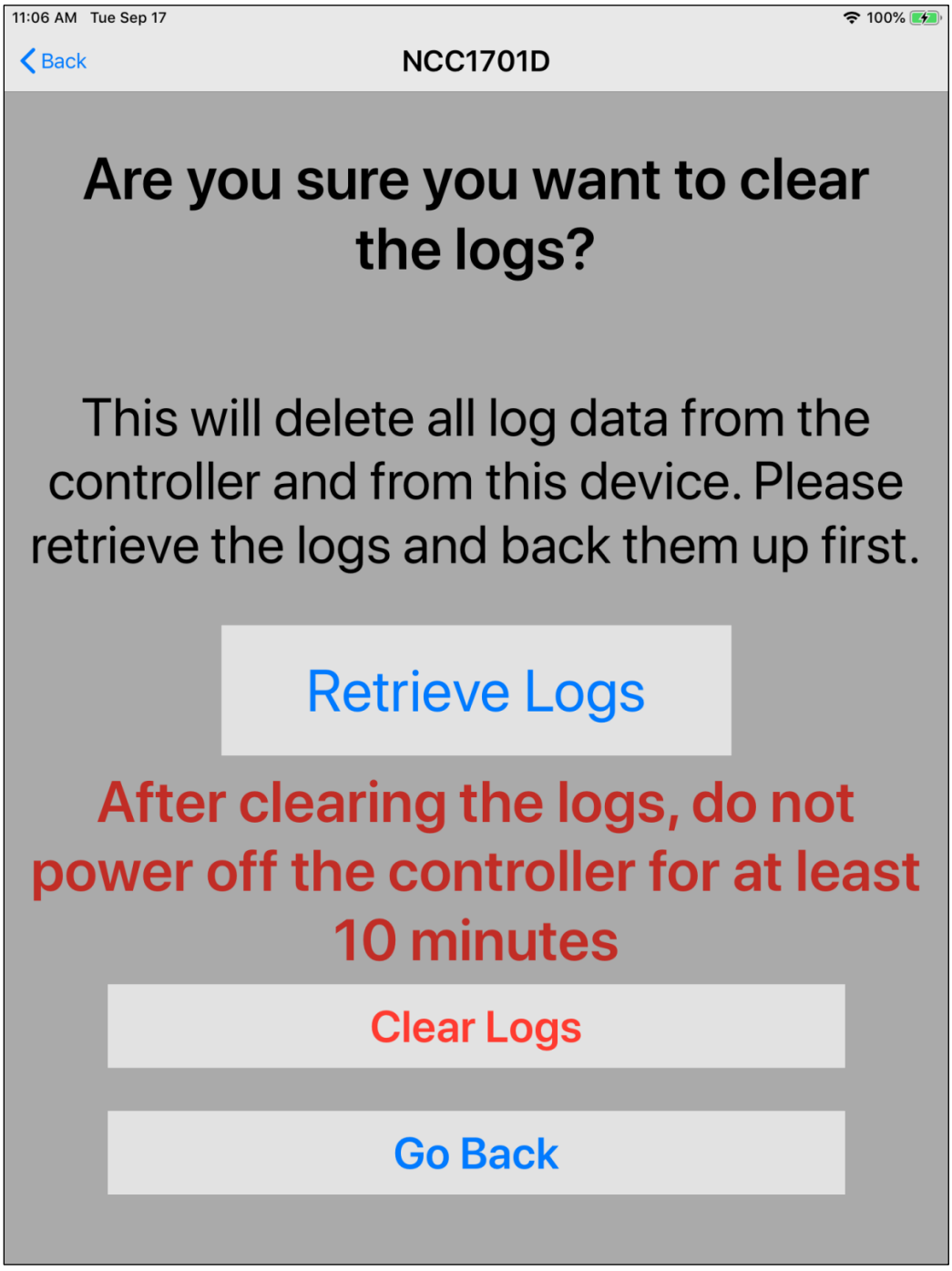

Figure 11 - Clear Logs Pop-up

5.3.5 Actuator Controller Screen

This is the controller management screen. It displays the current board temperature of the controller, in green if it's within a safe range, red if it's too hot.

The Test Alarm Lights button will trigger the controller to execute a test of the connected alarm lights in the cockpit.

The Controller Firmware section displays the current available firmware that is included with the app as well as the firmware version currently installed on the controller. If they do not match, the Upgrade Firmware button is enabled, which will bring the user to the Firmware Upgrade screen.

At the bottom of the screen is displayed the current version and build number of the iOS app.

5.3.5.1 Screen – Actuator Controller

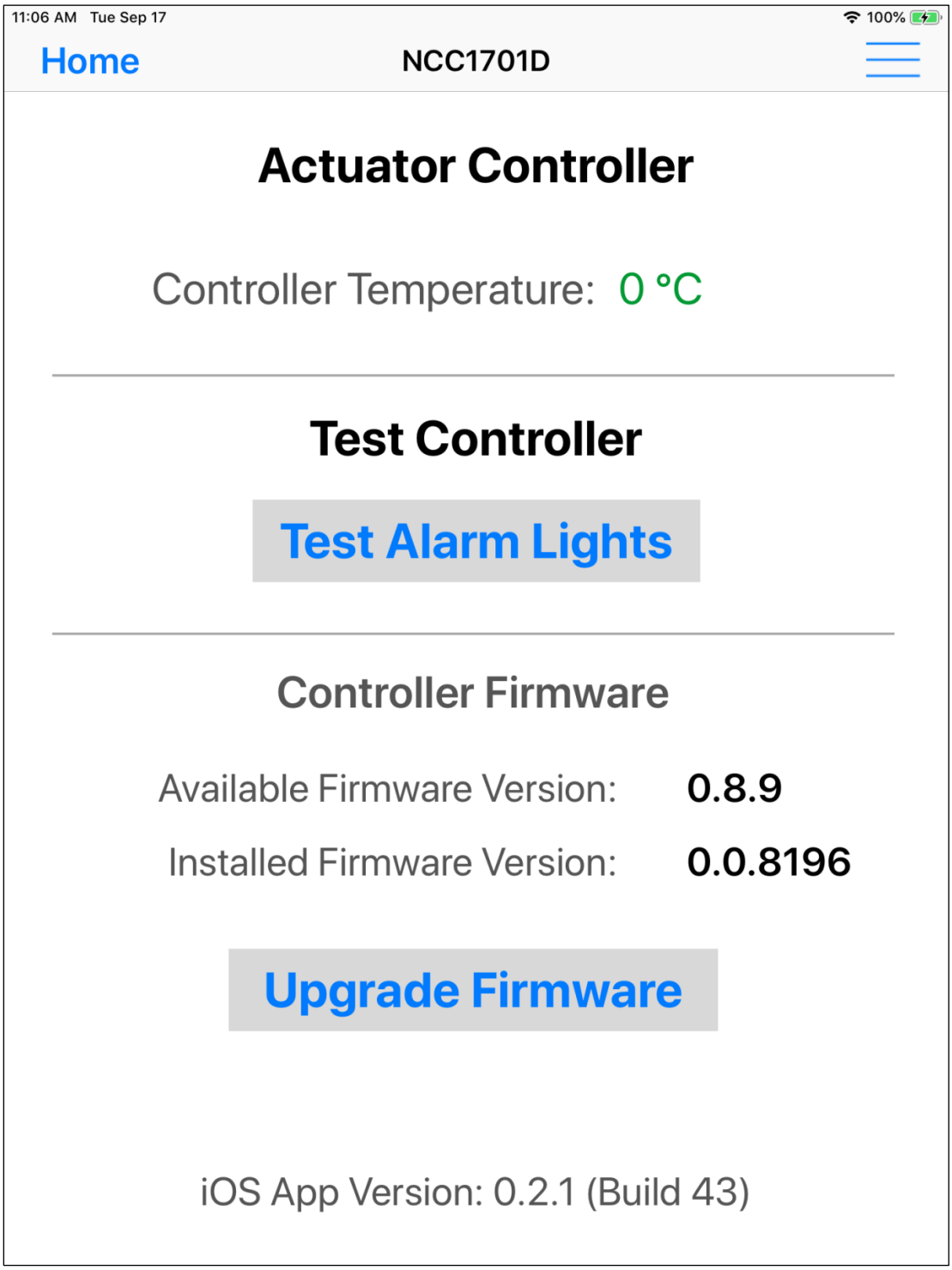

Figure 12 - Actuator Controller Screen

5.3.5.2 Screen – Firmware Upgrade

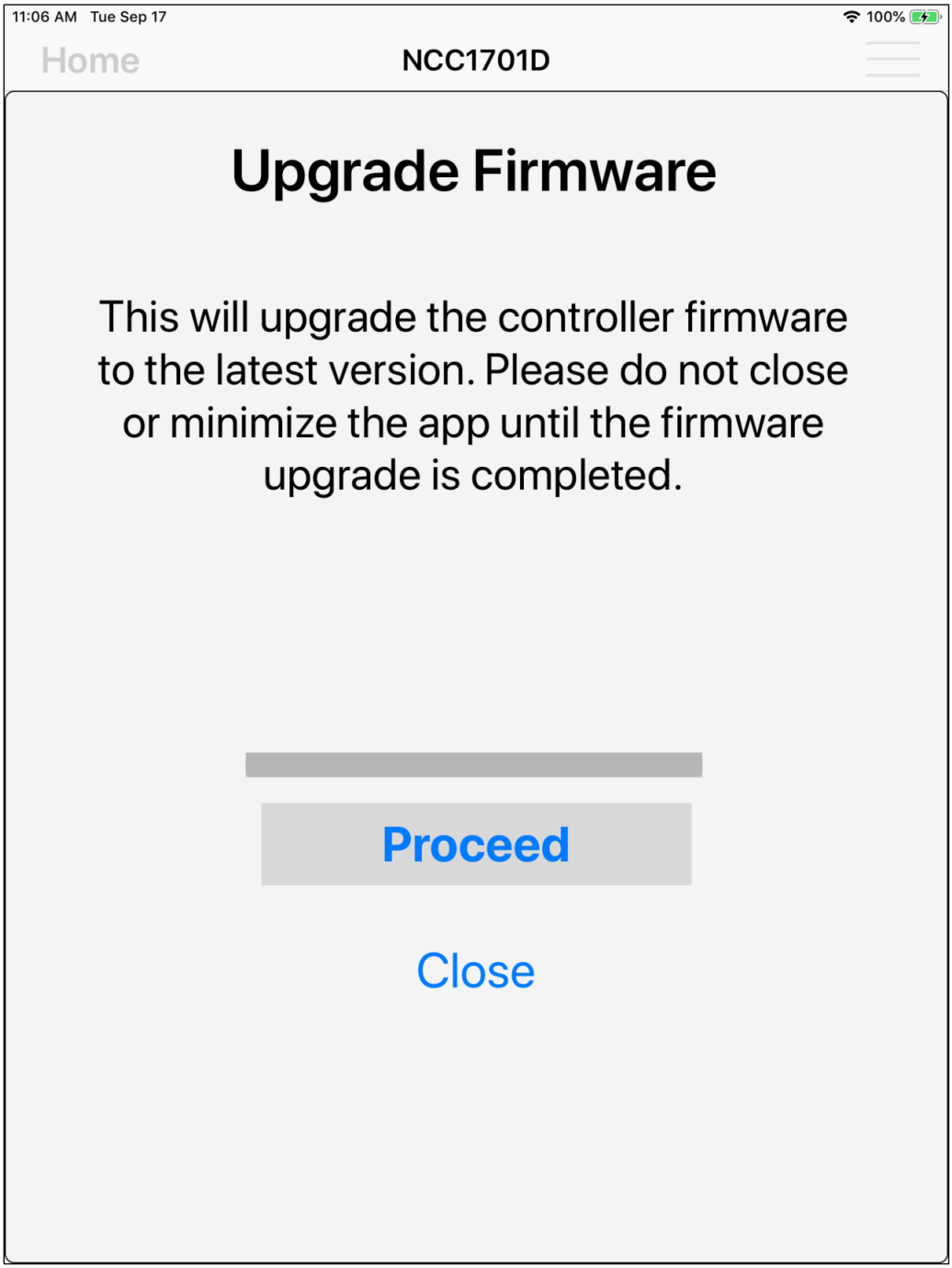

Figure 13 - Upgrade Firmware

5.3.6 Aircraft Configuration Screen

This screen contains the various configuration options available.

The aircraft registration number is set here. After a new value is entered into the text box, a confirmation dialog will appear. Accepting it will cause the controller to reboot, applying the new registration number.

Calibration functions for the two air intake doors can be triggered from here. The WOW feature can be enabled or disabled and the set-point temperatures can be modified as well.

5.3.6.1 Screen – Aircraft Configuration

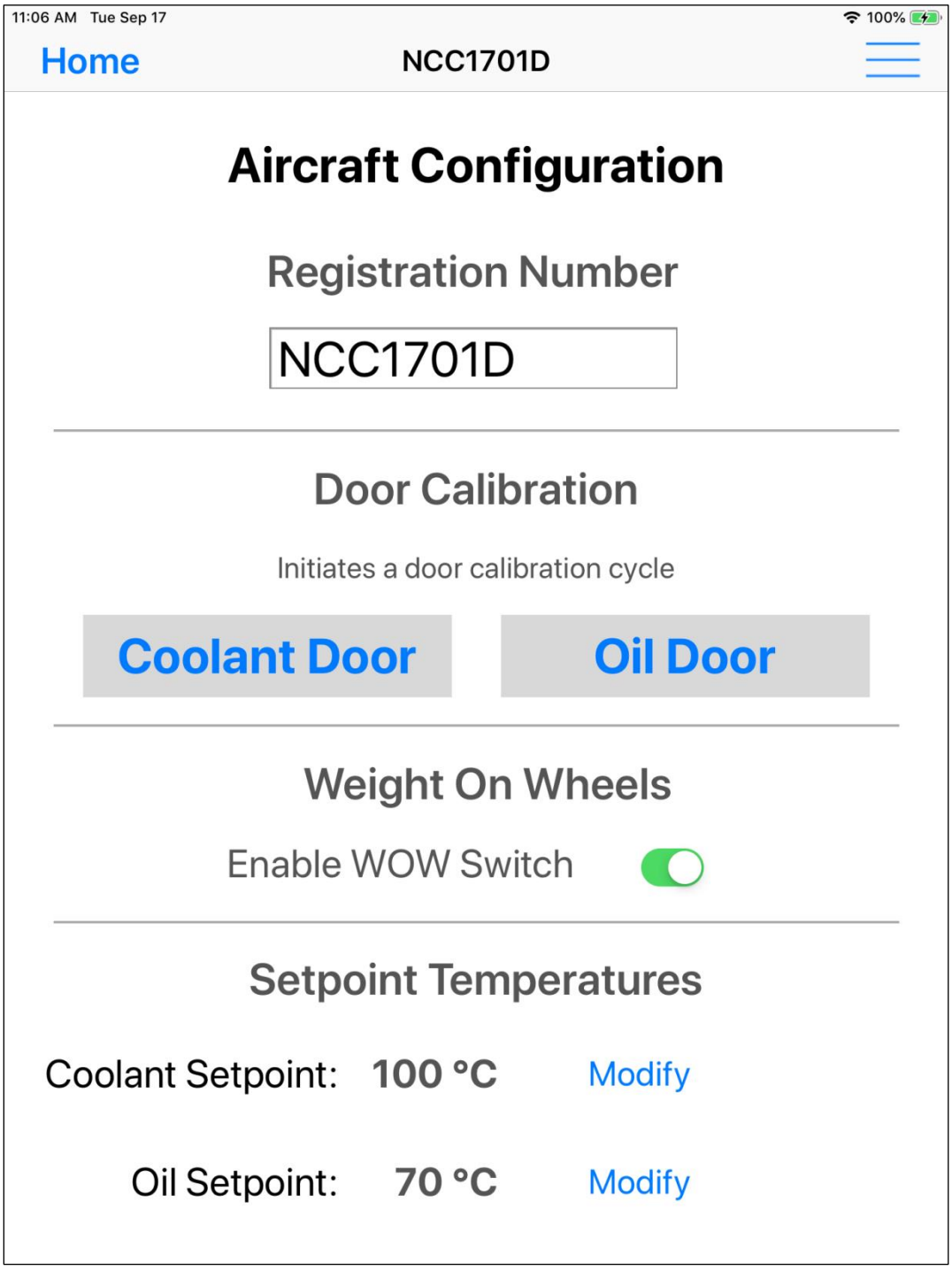

Figure 14 - Aircraft Configuration Screen

5.3.6.2 Modifying Set-Point Temperature

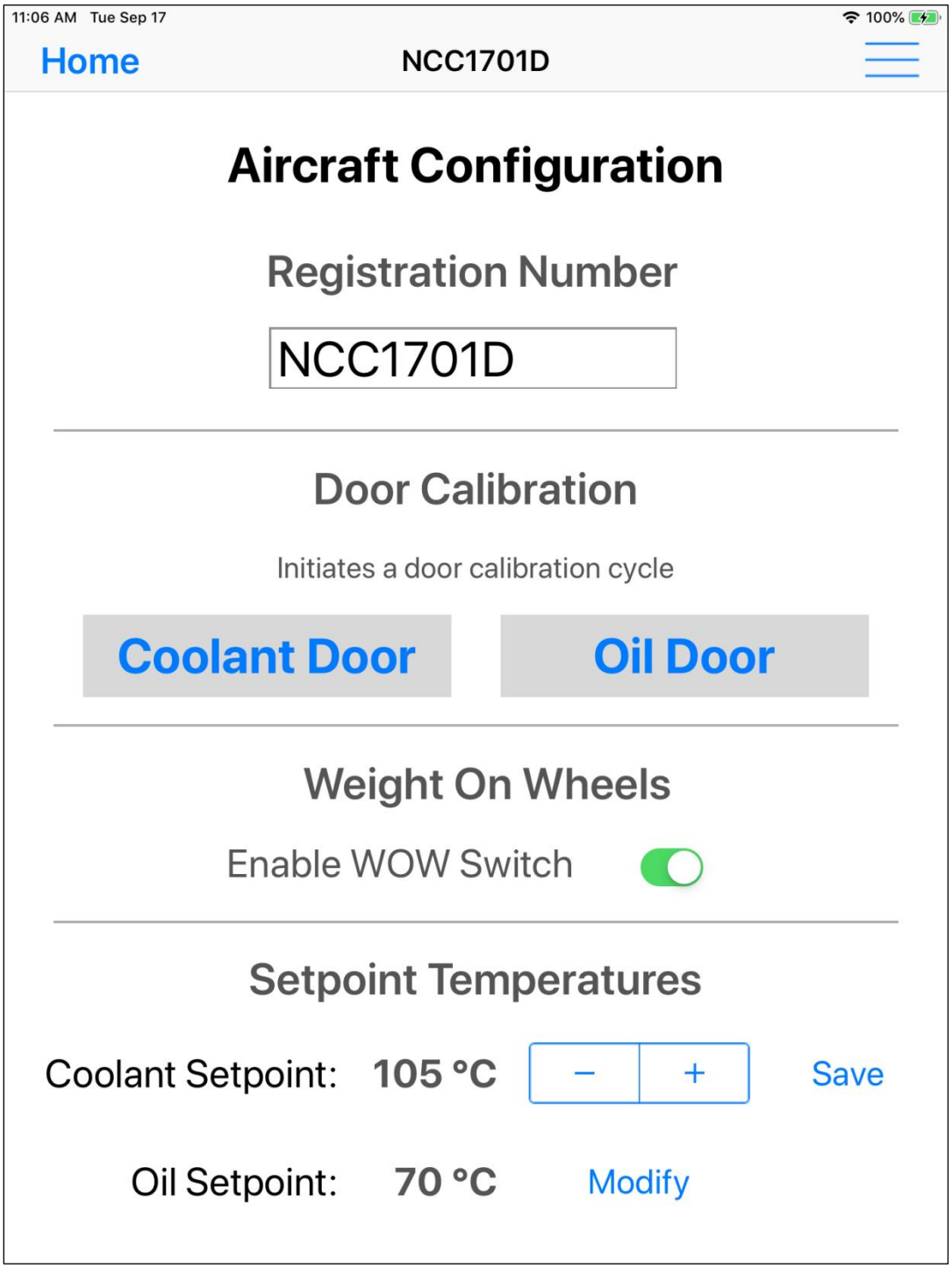

Figure 15 - Aircraft Configuration Screen

5.3.7 Control Parameters Screen

This screen allows the actuator control algorithm to be fine-tuned by adjusting its various parameters. Clicking the Modify Parameters button will enable all of the inputs, then hitting the Save Parameters button will send all of the parameter values to the controller.

This screen is only accessible when the controller is in service mode or the app is in development mode.

5.3.7.1 Screen – Control Parameters

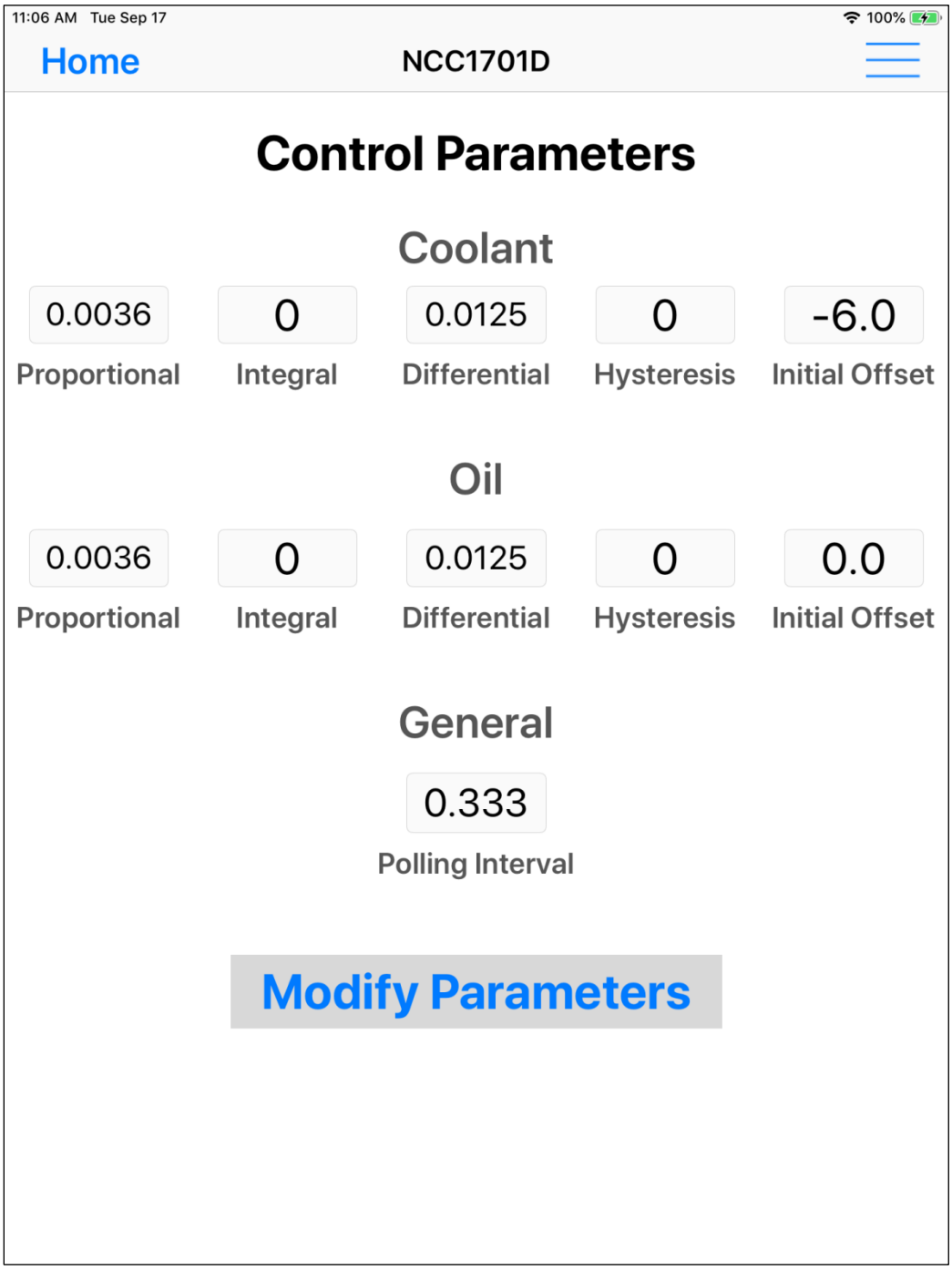

Figure 16 - Control Parameters Screen

6 Developer Guide

6.1 Changing the included Controller Firmware Version

The firmware for the controller is packaged in the iOS app as a DataSet Asset. The app always contains two firmware versions; these should be the same firmware version, they just get installed at different locations in the controller's memory and reference different memory addresses.

The firmware binaries are first base64-encoded and then are added to a JSON file. The JSON file is structured as followed:

```
{
  "version":"1.0.0",
  "crc_A":"abcd1234",
  "crc_B":"ef098765",
  "fileData_A":"a1a1a1…",
  "fileData_B":"b2b2b2…"
}
```

crc_A is the CRC value for the fileData_A binary. crc_B is the CRC value for the fileData_B binary. The version applies to both of the binaries.

There is a bash script which will base64 encode the two image binary files and then create the JSON file.

1. Check out both the wng01_ios and the wng01_fw git repos to the same folder
2. Copy the two binary files into the wng01_ios/scripts/ folder. The files should be named as follows:
    a. `actuator_controller_img_a_fw_app.bin`
    b. `actuator_controller_img_b_fw_app.bin`
3. Run the build_firmware_json.sh file, specifying the firmware version number as the only parameter
4. If successful, this will generate a firmwareFile.json file in the same directory
5. Copy firmwareFile.json to wng01_ios/Wings/Wings/Assets.xcassets/Firmware.dataset/
6. Build and run the app on a local device. Switch to the Settings view and make sure that Included Firmware Version now shows the version that you specified
7. Attempt to upgrade the firmware on a test controller locally and verify that it's working

6.2 Code Structure

Swift files don't correspond 1:1 with classes; a single file can have any number of classes, structs, and protocols. The codebase is mostly broken up so that each file only contains a single class, however sometimes a file will also have a protocol or another class if it makes sense to keep them together.

The app uses the Delegate Pattern; delegate classes implement the protocol, which allows the delegate manager to call functions in the delegate classes and pass data to them.

Here is a summary of the different files in the project:

- **Wings-bridging-header.h**: The graphing framework package is implemented in objective-c, not swift. The bridging header file ties in the objective-c header files and allow the graphing functions to be used in swift
- **AppDelegate:** This is the main entry-point for the app. There are a number of delegate functions that can be implemented, which will be called when the app is started, backgrounded, requested to close, etc. This file sets up the main data store and the notification center, which handles system push notifications. It's also got a custom error message function that can be called to pop up an error message in the app.
- **BLEManager:** The app's largest class, it sets up the app as the BLE Central and the controller as the BLE Peripheral. It watches for changes to the GATT database on the controller and sends the data to the other classes in the project. All communication between the controller and the app is handled through this class and so it's pretty big.

  The class is created as a singleton that is instantiated at app startup.

  The BLEManagerDelegate protocol at the top of the file defines the delegate functions that other classes can implement. All of the functions in this protocol are optional, so delegate classes don't have to implement all of them. When a delegate class attempts to instantiate the BLEManager class, it receives the singleton instance.

  The top section of the class implements the CoreBluetooth delegate functions for handling connection/disconnection events with the peripheral and the functions for scanning for peripherals.

  The next section implements the delegate functions for discovering the services and characteristics of the peripheral. When a new service is discovered after connecting, it's scanned for characteristics. Read characteristics are subscribed to so that any changes to them are known. Write characteristics are registered so that they can be written to later.

The next section builds up the functionality for handling updates to the various GATT characteristics. When a characteristic's value is changed in the GATT database on the controller, one of these functions will be called.

The next section contains the functions for writing values to the write characteristics on the controller.

This class maintains some state, such as the current BLE connection status and the state of the engine.

- **MultiDelegate**: This class is used to create a collection for adding multiple delegates to the BLEManager. Generally, a class can only have one delegate, so this allows multiple classes to implement the delegate protocol of the BLEManager.
- **Constants**: All constants, such as the UUIDs of the GATT profile, general system constants, and important temperature constants are stored here.
- **DataConverter**: A utility class that converts data received over BLE from raw data to the required data type, and vice-versa. It also contains some more specific converters for some of the characteristics.
- **LogFileStorageManager**: Manages the connection to the CoreData storage and the CRUD functions for the Log File
- **LogFileManager**: Middleman between BLEManager and LogFile that takes care of incoming log data
- **LogFile:** Reassembles the log data sent over the BLE connection, stores it in the local datastore, and writes it to a CSV file.

  The top section of this class has a collection of handler functions that process the various log data value types.
- **Various View Controllers:** These manage the controls on the various view controllers, handle navigation between the views, and respond to user input.
- **Main.Storyboard:** This is where the layouts for the various view are created and can be modified. Some of the controls on the view are created programmatically in the view controllers but most of the layout is created here.

6.3 Connection Process

Triggered when the app is started or the BLE antenna is turned on

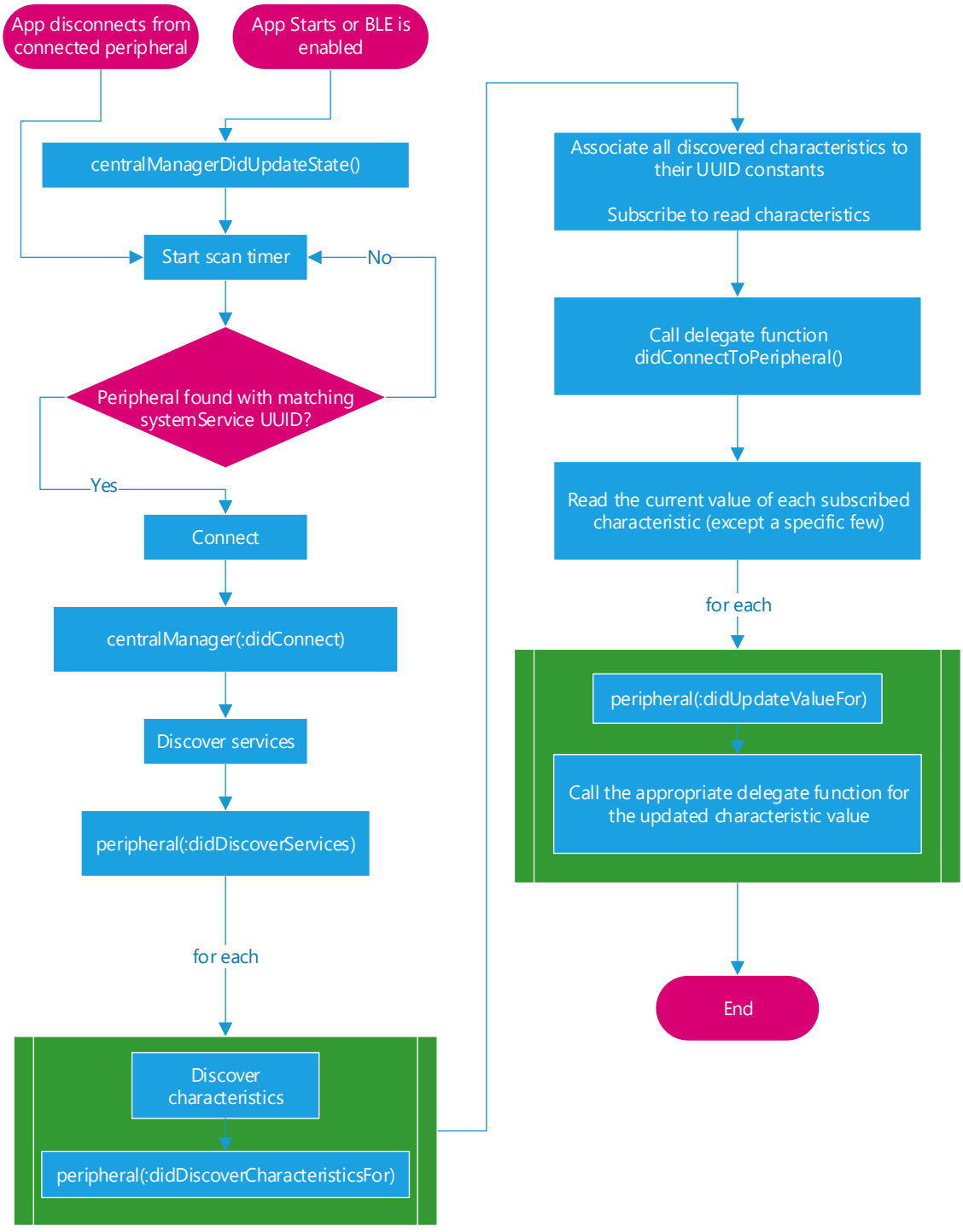

Figure 17 - BLE Connection Process Flowchart

Appendixes

Appendix A – GATT Protocol

*All UUIDs begin with **c064816f-3b70-470c-a18d-a352fbfd**, only the last two bytes differ from one characteristic to another. All UUIDs mentioned in this document should be assumed to include that prefix.*

System Status Service – UUID: 0000

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
engine_power	0001	read, notify	1	Uint8	0 – engine off 1 – engine on
flight_number	0002	read, notify	2	Uint 16	0 < value < 10000
controller_temp	0003	read, notify	2	Int16	Represents 1/100ths of a degree Celsius (i.e. 10500 = 105 C)
set_timestamp	0004	write	4	timestamp	Unix timestamp
alert_bitmask	0005	read, notify	1	8-bit bitmask	0b00000001 – high controller temp mask 0b00000010 – watchdog tripped mask
registration_number	0006	write, read, notify	8	String	A-Z, a-z, 0-9, -

Table 10 - Chracteristic Details - System

Oil Monitor Service – UUID: 0100

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
system_mode	0101	read, notify	1	Uint8	0 – manual mode 1 – manual open mode 2 – manual close mode 3 – automatic mode
temp	0102	read, notify	2	Int16	Represents 1/100ths of a degree Celsius (i.e. 10500 = 105 C)
door_position	0103	read, notify	1	Uint8	0 <= value <= 100 Represents % open
alert_bitmask	0104	read, notify	1	8-bit bitmask	0b00000001 – high oil temp mask 0b00000010 – oil actuator failure mask 0b00000100 – power good tripped

Table 11 - Chracteristic Details - Oil

Coolant Monitor Service – UUID: 0200

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
system_mode	0201	read, notify	1	Uint8	0 – manual mode 1 – manual open mode 2 – manual close mode 3 – automatic mode
temp	0202	read, notify	2	Int16	Represents 1/100ths of a degree Celsius (i.e. 10500 = 105 C)
door_position	0203	read, notify	1	Uint8	0 <= value <= 100 Represents % open
alert_bitmask	0204	read, notify	1	8-bit bitmask	0b00000001 – high coolant temp mask 0b00000010 – coolant actuator failure mask 0b00000100 – power good tripped

Table 12 - Chracteristic Details - Coolant

Configuration Service – UUID: 0300

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
oil_temp_setpoint	0301	read, write, notify	2	Int16	Represents 1/100ths of a degree Celsius (i.e. 10500 = 105 C)
coolant_temp_setpoint	0302	read, write, notify	2	Int16	Represents 1/100ths of a degree Celsius (i.e. 10500 = 105 C)
clear_logs	0303	write	1	UInt8	1 – clear logs
calibrate_door	0304	write	1	UInt8	1 – calibrate oil door 2 – calibrate coolant door
oil_proportional_gain	0305	read, write, notify	4	Int32	**TBD**
oil_integral_gain	0306	read, write, notify	4	Int32	**TBD**
oil_differential_gain	0307	read, write, notify	4	Int32	**TBD**
oil_hysteresis_threshold	0308	read, write, notify	4	Int32	**TBD**
coolant_proportional_gain	0309	read, write, notify	4	Int32	**TBD**
coolant_integral_gain	0310	read, write, notify	4	Int32	**TBD**
coolant_differential_gain	0311	read, write, notify	4	Int32	**TBD**
coolant_hysteresis_threshold	0312	read, write, notify	4	Int32	**TBD**
temp_adc_filter_window	0313	read, write, notify	4	Uint32	**TBD**
ctrl_loop_interval	0314	read, write, notify	4	Uint32	**TBD**

Table 13 - Chracteristic Details - Configuration

Logs Service – UUID: 0400

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
control	0401	write	3	Uint8 + UInt32	First byte specifies the command type. Last two bytes represent the value. 0 – request flight number (DEPRECATED) ➢ value is always 0 1 – set chunk size – number of log records between CRCs ➢ 1 <= value <= 100 2 – request flight log ➢ value = latest rowID stored in app database 3 – CRC Success ACK ➢ optional chunk size [1 <= value <= 50] 4 – CRC Failure ACK ➢ optional chunk size [1 <= value <= 50] 5 – Stop ➢ value is always 0 6 – Request most recent rowId ➢ Latest rowID stored on controller
data	0402	read, notify	20	data	Flight log data, padded with 0s if necessary to reach 20 bytes
response	0403	read, notify	5	UInt8 + Uint32	First byte specifies the response type. Last four bytes represent the value. 0 – flight number ➢ 0 < value < 10000 1 – CRC ➢ 4-byte CRC 2 – EOF ➢ value is always 0 3 – Return most recent rowID Latest rowID stored on controller

Table 14 - Chracteristic Details - Logs

Firmware Service – UUID: 0500

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
firmware_version	0501	read, notify	4	firmware version	First 8 bits – major version Second 8 bits – minor version Last 16 bits – patch version
firmware_control	0502	write	5	UInt8 + UInt32	First byte specifies control code. Last four bytes represent the value, if any. 1 – start firmware transfer ➢ firmware version 2 – end of firmware transfer ➢ CRC32 3 – abort firmware transfer ➢ value is always 0
firmware_data	0503	write	20	data	Firmware binary data
firmware_response	0504	read, notify	2	UInt8 + UInt8	First byte represents response code. Second byte represents value, if any 1 – Ready to receive data ➢ 0 – Request image A ➢ 1 – Request image B 2 – Data Received ➢ value is always 0 3 – Firmware upgrade aborted ➢ value is always 0 4 – Error ➢ 0 – write error ➢ 1 – CRC error ➢ 2 – Not Allowed

Table 15 - Chracteristic Details - Firmware

Data Service – UUID: 0600

Name	UUID	Properties	Length (Bytes)	Value Type	Valid Values
send_data	0601	write	20	UInt16 + 18 Bytes	Varies First two bytes are the type of data being sent. Remaining 18 bytes are the data, padded with 0's
receive_data	0602	read, notify	20	UInt16 + 18 Bytes	Varies First two bytes are the type of data being sent. Remaining 18 bytes are the data, padded with 0's

Table 16 - Chracteristic Details - Data

Appendix B – Log File Transfer Protocol

Log Service

Characteristics:

Name	Value	Properties	Type
control	see control table	Write	-
data	bytes	Read	Subscribe
response	see response table	Read	Subscribe

Table 17 - Logs Characteristics Properties

Control Characteristic Values

Control characteristics are written from the central to the peripheral for the purpose of controlling the log data flow. They consist of a code, which corresponds to a command, and sometimes a value. **Log control packets will always be 5 bytes, padded with 0s if necessary**

Purpose	Code	Value	Comment
Request the flight number of the most recent flight logged	00	none	DEPRECATED
Set chunk size	01	Number of log records *[UInt16]*	Number of log records to send before sending a CRC
Request flight log	02	Latest rowID in app database *[UInt32]*	The Controller should begin sending log data, starting from the next rowID after the latest rowID in the app database
CRC Success Acknoledgement	03	Number of packets *[UInt16]*	Acknowledges that a CRC comparison succeeded Optionally, a value may be passed to change the chunk size
CRC Failure Acknoloedgement	04	Number of packets *[UInt16]*	Acknowledges that a CRC comparison failed Optionally, a value may be passed to change the chunk size
Stop	05	none	Tells the controller it should stop sending more data
Request most recent rowId	06	none	Tells the controller to send the most recent rowId that it has logged up to this point

Table 18 - Log Transfers - Control Characteristic Values

Response Characteristic Values

Response characteristics are sent from the peripheral for responding to controls and sending statuses. They consist of a code, which corresponds to a response, and sometimes a value. **Log response must always be 5 bytes, padded with 0s if necessary**

Purpose	Code	Value	Comment
Return most recent flight number	00	Flight number *UInt16*	DEPRECATED
Return CRC	01	CRC *UInt32*	The CRC for the most recent chunk of data that was returned
End of File	02	none	EOF is returned to indicate that there is no more log data available for the requested flight number
Return most recent rowID	03	UInt32	Returns the rowID of the most recent log record stored on the controller

Table 19 - Log Transfers - Response Characteristic Values

Sequence Flow

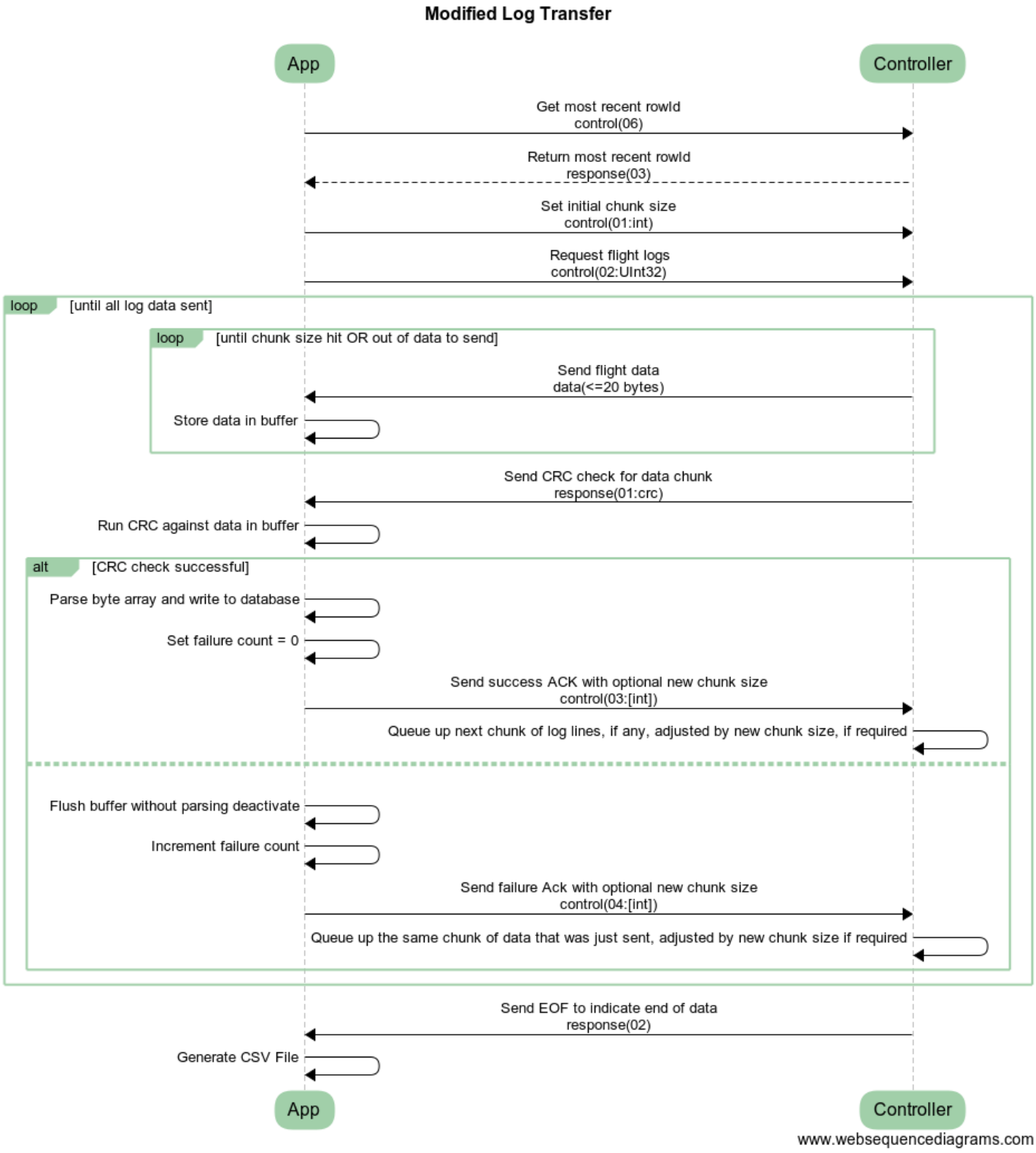

Figure 18 - Log Transfer Sequence Diagram

Process

1. App requests the rowID of the most recent log record stored on the controller
    a. App checks its local database for the most recent rowID that it already has
2. App sets the initial chunk size for the data transfer (i.e. 50)
3. App requests flight logs from the controller, sending the rowID of the most recent log record that it already has in its database
4. Beginning with the next rowID following the rowID sent by the App, the Controller prepares a number of log records equal to the chunk size (i.e. 100 log records)
    a. Controller calculates a CRC for this chunk of data
5. Controller starts sending this chunk of log data, in order, 20 bytes at a time (the last packet is padded with 0's if necessary)
    a. App stores this in a buffer
6. Once the whole data chunk has been sent, Controller sends the CRC
    a. App checks the data in the buffer against the CRC
        i. If the CRC is good, App parses the data and writes it to the database, then sends a success ACK. Optionally, the App may increase the chunk size for the following chunks by sending a new chunk size with the ACK.
        i. If the CRC is bad, App flushes the buffer, increments the failure count and returns a failure ACK. Optionally, the App may decrease the chunk size for the following chunks by sending a new chunk size with the ACK.
            1. If the failure counter reaches the maximum allowed attempts for a single chunk, App sends a stop command and discards the incomplete log data for the flight number that was in the process of being transferred. It does not request more flight logs.
    b. If the Controller received a success ACK, it prepares another chunk of data and a CRC.
    c. If the Controller received a failure ACK, it resends the last chunk and CRC, unless a new chunk size was requested, in which case it creates a smaller chunk from the same data and calculates a new CRC.
    d. If the Controller received a stop command, it does not send any more data.
7. Once the Controller has sent the final chunk of data, and the App has successfully acknowledged the CRC, the Controller sends an EOF response.
8. Upon receiving an EOF, the App parses the last remaining chunk of data to the database, then generates a CSV file from all of the log data in its database

Appendix C – Firmware Upgrade Protocol

Firmware Service

Characteristics:

Name	Value	Properties	Type
control	see control table	Write	Subscribe
data	bytes	Write	(Possibly with response)
response	see response table	Read	-

Table 20 - Firmware Characteristic Properties

Control Characteristic Values

Control characteristics are written from the central to the peripheral for the purpose of controlling the firmware data flow. They consist of a code, which corresponds to a command, and sometimes a value. **Firmware upgrade control packets will always be 5 bytes, padded with 0s if necessary**

Purpose	Code	Value	Comment
Start fw transfer	01	Fw version *Uint32*	The firmware version as a uint32. Expected as MAJOR.MINOR.PATCH where MAJOR is the least significant byte. MAJOR & MINOR = 1 byte PATCH = 2 byte
End of fw transfer	02	Crc32 *UInt32*	The crc32 value of the whole firmware file.
Abort fw transfer	03	none	

Table 21 - Firmware Upgrade - Control Characteristic Values

Data Characteristic Values

Data characteristic contains the fw payload data with a 1 byte overhead per packet to indicate the size of the data transferred. The data packet looks like the following:

Type	Data Length	Data
Byte	0	1-19

Table 22 - Firmware Upgrade - Description of Data Packet

Response Characteristic Values

Response characteristics are sent from the peripheral for responding to controls and sending statuses. They consist of a code, which corresponds to a response, and sometimes a value.

Firmware upgrade response must always be 2 bytes, padded with 0s if necessary

Purpose	Code	Value	Comment
Ready to receive data	01	Firmware file to receive	0 = Firmware image A 1 = Firmware image B
Data received	02	none	
Firmware upgrade aborted	03	none	
Error	04	Type of error	0 = Write error 1 = Crc Error 2 = Not allowed

Table 23 - Firmware Upgrade - Response Characteristic Values

Control Sequence

The sequence of a firmware upgrade is as follows:

iOS	*Controller*
• **Start fw transfer**	
	• Clear internal flash space to prepare for fw transfer. • **Send ready to receive data**
• Send data over data characteristic • Send next packet once write response is received	
	• Receive packet and write to flash
• **Send end of fw transfer with firmware crc**	
	• Check crc and compare with calculated crc. If correct, initiate an upgrade process and restart. **Otherwise send a crc error response**

Table 24 - Firmware Upgrade Sequence