CHALMERS
UNIVERSITY OF TECHNOLOGY

A Software-Defined Direct Remote Identification Broadcaster for a Fixed-Wing Maritime Rescue UAS

Master's thesis in Embedded Electronic System Design

ZIHENG LI

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

A Software-Defined Direct Remote Identification Broadcaster for a Fixed-Wing Maritime Rescue UAS

ZIHENG LI

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2026

A Software-Defined Direct Remote
Identification Broadcaster for a
Fixed-Wing Maritime Rescue UAS
ZIHENG LI

© ZIHENG LI, 2026.

Supervisor: Lars Svensson, Department of Microtechnology and Nanoscience
Company advisor: Fredrik Falkman, Swedish Sea Rescue Society (SSRS), Gothenburg
Examiner: Per Larsson-Edefors, Department of Microtechnology and Nanoscience

Master's Thesis 2026
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

A Software-Defined Direct Remote
Identification Broadcaster for a
Fixed-Wing Maritime Rescue UAS
ZIHENG LI
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Since January 2024, European Union regulations require every drone above 250 g to broadcast a Direct Remote Identification (DRI) signal containing its identity, position, and operator registration. This thesis presents `odid-daemon`, a pure software implementation of DRI for drones that already carry a Raspberry Pi companion computer with a Bluetooth and Wi-Fi radio. The daemon is written in C and runs as a `systemd` service. It reads live position and status data from an ArduPilot flight controller over MAVLink, encodes the five EN 4709-002 message types using the `opendroneid-core-c` reference library, and broadcasts them on Bluetooth 4 Legacy Advertising and Wi-Fi Beacon simultaneously. Dedicated commercial DRI hardware modules such as the Dronetag Mini or BlueMark DB150 are available, but they add mass, cost, and an additional failure mode to platforms that already carry compatible radios. The implementation was evaluated on the Raspberry Pi 400, which shares the BCM43455 radio with the Compute Module 4 used by the Swedish Sea Rescue Society (SSRS). End-to-end reception was confirmed with Drone Scanner, with Location updates at a measured mean of 1.005 s and all secondary message types at ≥ 0.33 Hz, meeting the EN 4709-002 requirements. The system was further validated in a live SSRS flight test.

Acknowledgements

I would like to thank my supervisor Lars Svensson for his guidance and constructive feedback throughout this project. I am also grateful to my examiner Per Larsson-Edefors for his careful review and valuable comments on the work. At the Swedish Sea Rescue Society, Fredrik Falkman provided essential domain knowledge and access to the operational context that shaped the design of this system. Björn Bringert generously gave his time to support the real drone hardware tests, without which the outdoor validation results in this thesis would not have been possible.

Ziheng Li, Gothenburg, June 2026

Contents

1 Introduction **1**
- 1.1 Related Work . . . 2
- 1.2 Purpose and Goal . . . 2
- 1.3 Scope and Delimitations . . . 3
- 1.4 Thesis Outline . . . 4

2 Technical Background **5**
- 2.1 Regulatory Chain . . . 5
- 2.2 EN 4709-002 Message Format . . . 6
- 2.3 Broadcast Channels . . . 7
  - 2.3.1 Bluetooth 4 Legacy Advertising . . . 7
  - 2.3.2 Bluetooth 5 Long Range . . . 8
  - 2.3.3 Wi-Fi Beacon Vendor IE . . . 9
- 2.4 Linux Interfaces to the Radios . . . 10
  - 2.4.1 The HCI Socket . . . 10
  - 2.4.2 nl80211 and hostapd . . . 11
- 2.5 MAVLink and ArduPilot . . . 11
- 2.6 The opendroneid-core-c Reference Library . . . 12

3 Method **13**

4 Design **15**
- 4.1 System Context . . . 15
- 4.2 Software Architecture . . . 16
- 4.3 Module 1: MAVLink Reader . . . 18
- 4.4 Module 2: ODID State and Scheduler . . . 18
- 4.5 Module 3: Bluetooth 4 Advertiser . . . 19
- 4.6 Module 4: Wi-Fi Beacon Injector . . . 20
- 4.7 Configuration and Deployment . . . 21
- 4.8 Development Log . . . 21
- 4.9 Summary . . . 22

5 Results **23**
- 5.1 Build and Static Tests . . . 23
- 5.2 Bluetooth 4 Wire Format . . . 23
- 5.3 Bluetooth 4 Location Update Interval . . . 24
- 5.4 End-to-End Bluetooth 4 Reception . . . 26

5.5 End-to-End Wi-Fi Beacon Reception 26
5.6 MAVLink Integration and GPS Fix 27
5.7 Service and Boot Resilience 28
5.8 Production Deployment on SSRS CM4 29
5.9 Capability and Security Hardening 29
5.10 Load Testing 29
5.10.1 BT4 Location Broadcast Timing Under Load 30
5.10.2 Memory Stability 31
5.10.3 Daemon CPU Utilisation 31

6 Discussion **33**
6.1 Compliance Assessment 33
6.2 BCM43455 Limitations and Their Implications 34
6.3 Operational Reliability 35
6.4 Comparison with Commercial Modules 35
6.5 Societal, Ethical, and Ecological Aspects 36
6.6 Threats to Validity 37
6.7 Reflection on Method 38

7 Conclusion **39**
7.1 Future Work 40
7.2 Broader Significance 41

Bibliography **43**

A Production Configuration and Deployment Files **I**

B Development Log and Non-Obvious Bugs **V**

C Receiver Screenshots **VII**

1
Introduction

A drone that cannot identify itself to the airspace around it is, from a regulator's point of view, indistinguishable from a drone operated with criminal intent. This simple observation is the reason that, over the past five years, every major civil aviation authority in the world has converged on the same policy: all but the smallest unmanned aircraft must, while in flight, transmit a short radio beacon announcing who they are, where they are, and who is flying them. In the United States the rule is *14 CFR Part 89 — Remote Identification of Unmanned Aircraft*, which became enforceable on 16 March 2024 [1]. In the European Union the equivalent obligation flows from Commission Delegated Regulation (EU) 2019/945 and Commission Implementing Regulation (EU) 2019/947, under which every drone of Class C1, C2, C3 or above operating in the Open or Specific category must carry a Direct Remote Identification (DRI) system [2, 3]. The technical format of the broadcast is fixed by the harmonised European standard EN 4709-002, which is itself the European profile of the international ASTM F3411-22a specification [4, 5].

The signal that both rules demand is unassuming: a few dozen bytes of drone and operator data, repeated several times per second on Bluetooth and/or Wi-Fi, readable by any commodity smartphone running a free application such as the OpenDroneID reference receiver [6] or the commercial Drone Scanner [7]. Commercial standalone modules can already produce this broadcast, but each one adds cost, weight, and an additional point of integration to the airframe. On a platform that already carries a companion computer equipped with the right radios, a natural alternative is to generate the broadcast entirely in software; and whether that is feasible, on a real airframe and in compliance with the standard, is the question this thesis sets out to answer.

This thesis is carried out with the **Swedish Sea Rescue Society** (Sjöräddningssällskapet, SSRS), a voluntary organisation that handles approximately four fifths of all sea rescues in Sweden [8]. Since 2022, SSRS has operated a long-endurance fixed-wing drone from bases on the west coast for maritime search and rescue, and in that year it became the first Swedish civil operator to be granted a national permit for Beyond Visual Line of Sight (BVLOS) flight [9]. The drone is a classical ArduPlane platform: a PixracerPro flight controller running ArduPilot, a u-blox GNSS receiver, and, critically, a Raspberry Pi Compute Module 4 (CM4) companion computer onboard that already performs payload management, video streaming and telemetry forwarding. The CM4 carries a BCM43455 combo chip that provides both a Bluetooth 4/5 radio and an 802.11ac Wi-Fi radio. In principle, those two radios are exactly the two channels that EN 4709-002 prescribes for DRI. The natural

question, and the one this thesis sets out to answer, is whether the CM4 can take on the DRI role entirely in software, using the radios it already carries, replacing a dedicated broadcast module and the cost, weight, and integration effort that come with it.

1.1 Related Work

The reference software for Remote ID broadcasting is the OpenDroneID project [10], which publishes a C encoding library (`opendroneid-core-c`), an Android receiver, and a Linux transmitter example (`transmitter-linux`) [11]. The transmitter example, although it targets the same Linux / Bluetooth stack addressed in this thesis, is explicitly described by its authors as a static demonstrator: it reads a hard-coded identifier and a fixed position, and is known by its maintainers to exhibit unreliable extended advertising on Raspberry Pi, to fail to transmit Wi-Fi message packs on the same hardware, and to require manual coordination with `hostapd` [11]. Commercial dedicated hardware modules such as the Dronetag Mini [12], the BlueMark DroneBeacon series [13], and the Spektrum Sky Remote ID module [14] solve the problem at the cost of added mass, battery, and an additional potential fault in the chain. None of them derive their position from an already-installed autopilot; they each carry a dedicated GNSS receiver.

At the autopilot level, ArduPilot 4.2 and later embeds an OpenDroneID subsystem that is able to forward the necessary state as MAVLink `OPEN_DRONE_ID_*` messages to an external module [15]. A companion computer that speaks MAVLink therefore has, at least in principle, everything it needs from the autopilot to produce a compliant broadcast. Whether a commodity Raspberry Pi is actually capable of emitting that broadcast in a standards-compliant manner, given the known limitations of the `brcmfmac` driver and the BCM43455's HCI firmware, has to the best of our knowledge not been characterised in the open literature.

A growing academic literature sits alongside the engineering work. Surveys by Sciancalepore and Di Pietro on UAV remote identification [16] have described the regulatory landscape and the security properties of the ASTM broadcast. Several groups have noted that the plaintext nature of the beacon exposes operators to tracking and spoofing; proposals such as *A2RID* [17], *TBRD* [18], and the IETF DRIP architecture [19] aim to add authentication and selective disclosure on top of the basic ASTM payload. These privacy-enhancing extensions are complementary to this thesis, which takes the ASTM format as given and asks only whether a particular commodity platform can emit it correctly.

1.2 Purpose and Goal

The **purpose** of this project is to give SSRS a credible, maintainable path to Remote ID compliance for its fixed-wing rescue drone without adding further mass, additional failure modes or further acquisition cost to the airframe, and in doing so to document publicly what can and cannot be achieved with a commodity Rasp-

berry Pi in the Remote ID role. Because SSRS is a voluntary organisation whose operations are funded by donations, cost and weight are not abstract concerns.

The **goal** of the thesis is more concrete. It is to design, implement and empirically evaluate a Linux daemon, written in C and running as a `systemd` service on the CM4, that

1. reads live position, velocity, arm status and (where available) native OpenDroneID messages from the ArduPilot autopilot over a MAVLink link (serial or UDP);
2. encodes the five EN 4709-002 mandatory and recommended message types (Basic ID, Location, System, Operator ID and Self ID) using the `opendroneid-core-c` reference library;
3. broadcasts those messages on Bluetooth 4 Legacy Advertising and on Wi-Fi Beacon, at the frequencies mandated by EN 4709-002 (at least once per second for Location, at least once per three seconds for the other messages); and
4. starts automatically at boot, survives unplug/replug of the flight-controller USB cable, and has no dependency on manual human interaction during flight.

Compliance is defined here in terms of reception by an independent, production-grade receiver (Drone Scanner on Android) with all mandatory fields decoded correctly, and verified at the byte level with `btmon` and the Wireshark 802.11 dissector.

1.3 Scope and Delimitations

The scope of this thesis is deliberately restricted to *Direct* Remote ID, that is, the broadcast of an identification beacon over Bluetooth and Wi-Fi with no network infrastructure involved. *Network* Remote ID, which the FAA considered but eventually removed from 14 CFR Part 89 and which is not mandated in the European Union [1, 2], is therefore out of scope. Likewise, the thesis does not address U-space or UAS Traffic Management (UTM) integration, authentication and spoofing-resistance, end-to-end encryption of the identifier, or the political and social questions that these topics raise.

On the hardware side, the daemon is developed on a Raspberry Pi 400 and is intended to run on the Compute Module 4. Both carry the BCM2711 SoC and the BCM43455 radio. It is therefore expected, but not verified, to run on any Raspberry Pi 4 variant, and it is expected *not* to run unmodified on the Raspberry Pi 5, whose radio subsystem differs.

Finally, because EN 4709-002 demands that the broadcast carries a legally meaningful operator identifier, the project uses the real Transportstyrelsen operator identifier `SWE3jxza7qbzvu5d` throughout. The UAS serial number, however, is not officially registered: the project uses a self-assigned identifier (`SSRSGBG001`) in the CTA-2063-A format, which is sufficient for bench and flight testing but must be replaced by an officially registered number before routine operational use.

1.4 Thesis Outline

Chapter 2 gathers the technical background a reader from within the EESD programme needs: the European and Swedish regulatory chain, the EN 4709-002 message format, the relevant parts of the Bluetooth 4 and Bluetooth 5 Core Specification, the Wi-Fi Beacon vendor Information Element mechanism, and a short review of MAVLink as it is used in ArduPilot. Chapter 3 explains the engineering methodology chosen for the project, including the iterative build-test-measure cycle and the measurement and verification tools. Chapter 4 is the core of the thesis: it describes the software architecture of `odid-daemon`, its threading model, its broadcast-channel abstractions and the non-obvious byte layouts that have to be assembled to satisfy the standard. Chapter 5 reports bench measurements and independent-receiver reception tests, including interval measurements, BlueZ interaction behaviour, and end-to-end reception on Drone Scanner, as well as load testing under sustained CPU and memory pressure, and production deployment results on the live flight test. Chapter 6 places these results in the context of compliance, operational reliability, and the available commercial alternatives, and identifies two hard hardware limitations of the BCM43455 that any software-only Remote ID strategy has to contend with. Chapter 7 summarises the contribution and recommends directions for future work, including the use of an external USB Bluetooth 5 dongle to unlock Long Range advertising.

2

Technical Background

This chapter introduces the regulatory and technical background needed to follow the remainder of the thesis. It is organised from the outside in: first the legal chain that makes Remote ID compulsory in Sweden, then the wire format of the broadcast it requires, then the two radio technologies on which that broadcast has to travel, and finally the autopilot protocol from which the aircraft's position is retrieved.

2.1 Regulatory Chain

The legal obligation to carry a Remote ID beacon flows through four nested layers. At the top sits Commission Delegated Regulation (EU) 2019/945, which is the EU-wide technical regulation for unmanned aircraft placed on the market, and which defines the drone classes C0 through C6 by Maximum Take-Off Mass [2]. Commission Implementing Regulation (EU) 2019/947, the operational companion to 2019/945, partitions civil UAS operations into three categories of increasing risk: *Open* for low-risk flights that do not require individual authorisation, *Specific* for operations that require a specific operational authorisation from the national authority, and *Certified* for the highest-risk flights [3]. The two regulations together impose Direct Remote ID on every drone of Class C1 (Maximum Take-Off Mass $\geq 250\,\text{g}$) and above that operates in the Open category, and on every drone of any class that operates in the Specific or Certified categories, from the 1st of January 2024 onward.

In Sweden, the competent authority for the implementing regulation is Transportstyrelsen (the Swedish Transport Agency), which has operated the national UAS operator registry since 2021. Every operator receives a sixteen-character registration identifier of the form `SWE` followed by thirteen alphanumeric characters (three of which are randomly chosen and are not printed on the receipt), and this identifier is both the physical marking that must appear on the airframe and the string that the drone must broadcast over Remote ID [20]. For the SSRS drone the operator identifier is `SWE3jxza7qbzvu5d`.

The harmonised technical standard that translates these legal requirements into an actual over-the-air format is EN 4709-002, published by ASD-STAN and endorsed by the European Union Aviation Safety Agency as a means of compliance for 2019/945 [5]. EN 4709-002 is the European profile of ASTM F3411-22a, the international standard originally developed in the United States for the same purpose [4]. The two standards share essentially the same 25-byte message payloads; they differ

slightly in the set of transmission channels that is considered primary, and in the minimum update rate of secondary messages (where EN 4709-002 harmonises closer to 1 Hz than the 0.33 Hz permitted by ASTM).

The United States' parallel rule, 14 CFR Part 89, is not directly relevant in Sweden, but it is useful as a reference point because it forced the FAA-registered hardware ecosystem (Dronetag, BlueMark, Spektrum and others) to converge on the ASTM format by its extended enforcement date of 16 March 2024 [1]. The practical consequence of that convergence is that any ASTM F3411-22a transmitter produced for the American market is also, up to minor fields, a valid EN 4709-002 transmitter.

2.2 EN 4709-002 Message Format

A conforming Direct Remote ID transmitter broadcasts five message types, one authentication message type that is optional outside Japan, and one container message for bundling multiple messages together. Each of the five primary messages occupies exactly 25 bytes when encoded, and each is structured around a one-byte header (the high nibble carrying the message type and the low nibble a protocol version) followed by 24 bytes of type-specific fields. Table 2.1 summarises the five types and their minimum update rates under EN 4709-002.

Table 2.1: The five EN 4709-002 message types relevant to this thesis, their size when encoded, and the minimum update rate demanded of a conforming transmitter.

Message	Purpose	Encoded size	Minimum rate
Basic ID	UAS identifier and airframe type	25 bytes	$\geq$ 0.33 Hz
Location	Position, velocity, status, timestamp	25 bytes	$\geq$ 1 Hz
System	Operator location, classification	25 bytes	$\geq$ 0.33 Hz
Operator ID	Operator registration string	25 bytes	$\geq$ 0.33 Hz
Self ID	Human-readable description	25 bytes	$\geq$ 0.33 Hz

The **Location** message is by some margin the most intricate of the five. It carries the drone's operational status (ground, airborne, emergency), horizontal and vertical speed, latitude and longitude, barometric and geodetic altitude, height above a configurable reference, and a set of accuracy categories that allow a receiver to draw a confidence ellipse around the reported position. The timestamp is encoded as seconds (plus tenths) within the current UTC hour, which keeps the field small at the cost of forcing the receiver to wrap around every 3600 seconds. The **System** message carries the operator's own position (either the fixed take-off location or a live GNSS fix), a classification tag that identifies which regulatory regime the drone is declaring compliance with, the drone's declared EU C-class, and a separate absolute UTC timestamp counted in seconds since 00:00:00 on 1 January 2019. The **Basic ID**, **Operator ID** and **Self ID** messages each carry a 20- or 23-character string and a couple of small type fields.

2.3 Broadcast Channels

EN 4709-002 allows four media for Direct Remote ID: Bluetooth 4 Legacy Advertising, Bluetooth 5 Extended Advertising (preferably on the LE Coded PHY for long range), Wi-Fi Beacon, and Wi-Fi NAN. A conforming transmitter must support at least one. Receivers in the field, both OpenDroneID's own Android app and the commercial Drone Scanner, scan all four in parallel on devices where the operating system supports it; in practice, the two that reach a commodity smartphone with the least friction are Bluetooth 4 Legacy Advertising and Wi-Fi Beacon, and those are therefore the two channels this thesis targets.

2.3.1 Bluetooth 4 Legacy Advertising

Bluetooth Low Energy advertising is the mechanism by which a BLE peripheral announces its existence to any listener within range before a connection is established. The advertiser transmits a short packet, up to 31 bytes of payload, on the three so-called primary advertising channels (channels 37, 38 and 39 in the 2.4 GHz ISM band), at a programmable interval between roughly 20 ms and 10 s [21]. Four PDU types are defined for legacy advertising; EN 4709-002 mandates `ADV_NONCONN_IND` (PDU type `0x03`), which is non-connectable and non-scannable and therefore carries only the 31-byte advertising data blob.

The 31-byte payload is divided into Advertising Data elements of the form `[length, AD type, value]`. For Remote ID the relevant AD type is `0x16` (Service Data — 16-bit UUID), and the UUID is `0xFFFA`, the ASTM Remote ID service UUID assigned by the Bluetooth SIG [22]. Immediately after the UUID, an eight-bit *ODID application code*, `0x0D`, tells the receiver that the remaining bytes are an Open Drone ID message rather than some other ASTM-defined payload. Then comes a one-byte rolling *message counter*, which increments on every advertisement and wraps around at 255, and finally the 25-byte encoded ODID message. The complete 31-byte layout is shown in Table 2.2. The rolling counter is not decorative: the reference Android receiver parses the message starting at byte offset 6, with the counter deliberately inserted at offset 5 so that the 25-byte message begins at offset 6 [6, 7]. An advertisement in which the counter is omitted slides the message one byte to the left, is parsed into a garbage header, and is silently discarded by the receiver.

Table 2.2: Byte-level layout of the 31-byte Bluetooth 4 Legacy Advertising payload for EN 4709-002 Remote ID.

Offset	Value	Meaning
0	`0x1E`	AD element length (30 = payload after this byte)
1	`0x16`	AD type: Service Data — 16-bit UUID
2	`0xFA`	UUID `0xFFFA`, LSB
3	`0xFF`	UUID `0xFFFA`, MSB
4	`0x0D`	ODID application code
5	counter	Rolling 8-bit message counter
6–30	25 bytes	ODID encoded message (e.g. Location)

Because a single 25-byte message is the whole payload, Bluetooth 4 Legacy Advertising cannot carry a bundle. The advertiser therefore must rotate through its message types — sending a Location advertisement roughly once per second, and interleaving Basic ID, System, Operator ID and Self ID advertisements in the remaining timeslots — to satisfy EN 4709-002's per-type update rates.

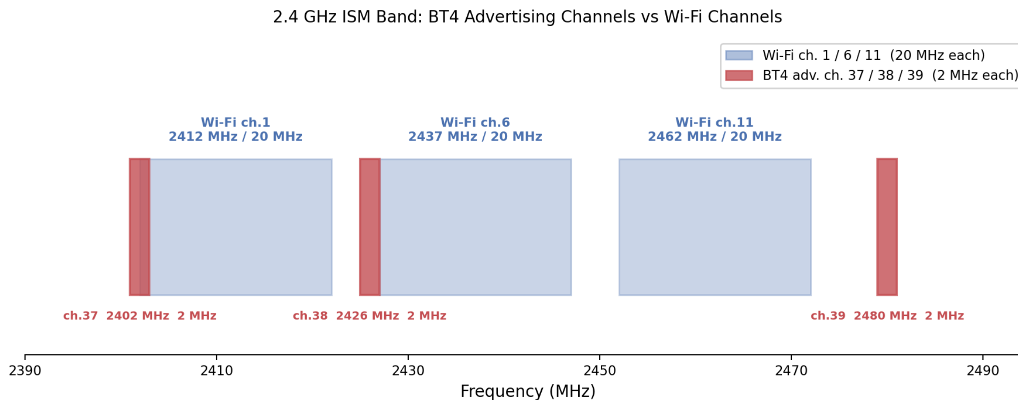

Figure 2.1: Bluetooth 4 Legacy Advertising channels in the 2.4 GHz ISM band. Channels 37, 38 and 39 are positioned at 2402 MHz, 2426 MHz and 2480 MHz to minimise overlap with the three most-used Wi-Fi channels (1, 6 and 11). All three advertising channels are used simultaneously for each EN 4709-002 Remote ID advertisement.

2.3.2 Bluetooth 5 Long Range

Bluetooth 5, introduced in 2016 and elaborated through to Core Specification 5.3 in 2021 [21], adds *Extended Advertising*, which replaces the 31-byte single-packet primary-channel PDUs with a chain of extended PDUs on secondary channels, raising the effective payload to 254 bytes per chain. On the LE Coded PHY with coding factor $S = 8$ (125 kbit/s, four-fold Forward Error Correction) Bluetooth 5 extended advertisements have been empirically demonstrated to achieve roughly four times the range of legacy advertising. EN 4709-002 recommends Extended Advertising with Coded PHY as the preferred Bluetooth channel, because its larger payload allows a whole message pack of five 25-byte messages to be carried in a single transmission, and because its longer range is operationally desirable.

Extended Advertising is controlled through a separate family of HCI commands: `LE_SET_EXTENDED_ADVERTISING_PARAMETERS` (OCF `0x0036`), `LE_SET_EXTENDED_ADVERTISING_DATA` (OCF `0x0037`) and `LE_SET_EXTENDED_ADVERTISING_ENABLE` (OCF `0x0039`). A Bluetooth 5 controller that does not support Extended Advertising returns HCI error `0x01` ("Unknown HCI Command") to the first of these three, and Chapter 5 will show that this is precisely what the BCM43455 does.

The approximate detection range of each mode follows from the free-space path loss (FSPL) link budget. Expressing the Friis transmission equation in logarithmic form,

$$P_r = P_t + G_t + G_r - 20\log_{10}\left(\frac{4\pi d f}{c}\right) \tag{2.1}$$

where P_t is the transmit power (dBm), G_t and G_r are the transmit and receive antenna gains (dBi), d is the link distance (m), f is the carrier frequency (Hz), and c is the speed of light in free space. Setting the received power equal to the minimum detectable signal $P_{r,\min}$ and solving for the maximum range gives

$$d_{\max} = \frac{c}{4\pi f} \cdot 10^{\left(P_t + G_t + G_r - P_{r,\min}\right)/20} \tag{2.2}$$

For BT4 Legacy Advertising on channel 38 ($f = 2.426\,\text{GHz}$), a typical BCM43455 transmit power of $P_t = 0\,\text{dBm}$, and isotropic antennas ($G_t = G_r = 0\,\text{dBi}$), Equation (2.2) yields $d_{\max} \approx 100\,\text{m}$ under a conservative receiver sensitivity assumption of $P_{r,\min} = -80\,\text{dBm}$; actual sensitivity varies considerably across receiver hardware, chipset, antenna orientation, and scan configuration.

The BT5 LE Coded PHY with $S = 8$ (125 kbit/s) provides a coding gain of approximately 12 dB relative to BT4 at 1 Mbit/s under ideal conditions [21], which by Equation (2.2) would extend the theoretical range to roughly $100 \times 10^{12/20} \approx 400\,\text{m}$ under the same link-budget assumptions. Both figures assume free-space propagation; multipath fading and antenna body-shielding reduce effective range in practice.

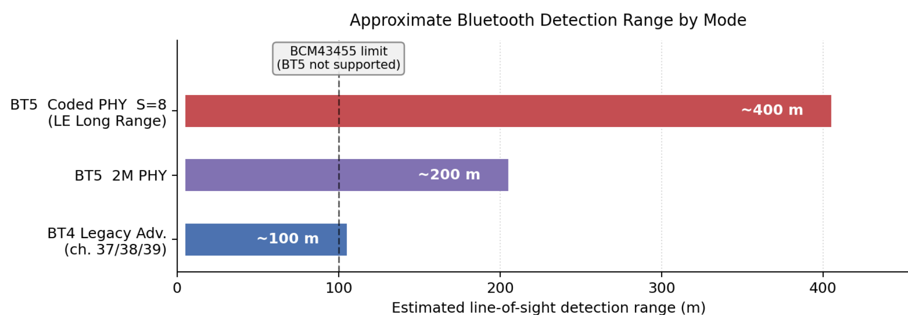

Figure 2.2: Approximate line-of-sight detection ranges for Bluetooth advertising modes in an open outdoor environment. BT4 Legacy Advertising achieves roughly 100 m; BT5 with 2M PHY approximately doubles this; BT5 with LE Coded PHY ($S = 8$) extends the range to around 400 m. The BCM43455 in the Raspberry Pi supports only BT4 Legacy Advertising; BT5 Extended Advertising requires an external USB Bluetooth 5 dongle.

2.3.3 Wi-Fi Beacon Vendor IE

An 802.11 Access Point transmits a management frame called a *Beacon* approximately every 100 ms to announce its SSID and capabilities. Beacon frames carry a tail of Information Elements of the form `[element ID, length, data]`, and one of the element IDs, `0xDD` (*Vendor Specific*), is reserved for payloads identified by a three-byte Organisationally Unique Identifier [23]. The OpenDroneID project holds

the OUI `FA:0B:BC` [10], and its type byte within that OUI is `0x0D`, the same as for Bluetooth. Because a Beacon can be much larger than a BLE advertisement, Wi-Fi Beacon Remote ID carries a *message pack*: a short pack header followed by up to nine concatenated 25-byte messages. The exact byte layout is given in Table 2.3.

Table 2.3: Byte-level layout of the Wi-Fi Beacon vendor Information Element for Remote ID.

Offset	Value	Meaning
0	`0xDD`	IE ID: Vendor Specific
1	length	Total IE payload length
2–4	`FA 0B BC`	OpenDroneID OUI
5	`0x0D`	OUI type (Remote ID)
6	counter	Rolling 8-bit message counter
7	`0xF2`	Message pack header (type 0xF, version 2)
8	`0x19`	Single message size (25)
9	N	Number of messages in pack
10 . . .	$N \times 25$ bytes	Concatenated ODID messages

Because the Beacon frame is transmitted by the Access Point autonomously at its own beacon interval, once the vendor IE has been inserted into the outgoing Beacon the operating system needs no further action for each transmission. The challenge is therefore not the transmission rate but the *injection*: persuading the Linux Wi-Fi stack to place an arbitrary vendor IE into Beacons generated by `hostapd`.

2.4 Linux Interfaces to the Radios

2.4.1 The HCI Socket

A Linux userspace process can open a raw socket of family `AF_BLUETOOTH` and protocol `BTPROTO_HCI`, bind it to a specific HCI device, and then send Host Controller Interface commands directly to the Bluetooth controller as if it were the Host side of the standard BT stack [24]. Each HCI command consists of a 16-bit opcode, an 8-bit parameter length and the parameter bytes themselves; the opcode is assembled as $(\mathrm{OGF} \ll 10) \,|\, \mathrm{OCF}$, where OGF = `0x08` denotes *LE Controller Commands* and the OCF selects the specific LE command. The three legacy LE advertising commands used in this thesis are listed in Table 2.4. The BlueZ library `libbluetooth` provides a thin wrapper `hci_send_cmd()` over the raw socket, but the present implementation uses only the kernel-level primitives.

Table 2.4: HCI LE legacy advertising commands used by the daemon (OGF = `0x08`).

OCF	Name	Function
`0x0006`	`LE_SET_ADVERTISING_PARAMETERS`	Interval, type, channel map
`0x0008`	`LE_SET_ADVERTISING_DATA`	Up to 31 bytes of payload
`0x000A`	`LE_SET_ADVERTISING_ENABLE`	Start/stop advertising

Using a raw HCI socket in this way bypasses BlueZ's own D-Bus mediation layer, so `bluetoothd` no longer acts as gatekeeper between the application and the controller. A consequence is that if `bluetoothd` is also running, both processes share the same HCI device without coordination: `bluetoothd` periodically issues its own HCI commands — in particular `LE_SET_ADVERTISE_ENABLE(0)` when it wishes to use the controller for discovery or GAP operations — and therefore races against any application that relies on the advertising state remaining unchanged. Chapter 5 documents this interaction.

2.4.2 nl80211 and hostapd

The Linux Wi-Fi subsystem is controlled through *nl80211*, a generic-netlink family that provides the userspace API to the `cfg80211` driver layer. Among its many commands, `NL80211_CMD_SET_BEACON` accepts an `NL80211_ATTR_IE` attribute that should replace the Information-Element tail of subsequent Beacon frames [25]. Whether a given driver actually implements this behaviour is, however, not guaranteed: the `brcmfmac` driver used by the Broadcom BCM43455 returns `-EOPNOTSUPP` for the attribute, as we will verify empirically in Chapter 5.

A fallback path exists. The user-space daemon `hostapd`, which is the canonical way to run 802.11 Access Point mode on Linux, exposes a control-socket interface through which arbitrary vendor IEs can be set at runtime via `set vendor_elements <hex>`, and the Beacon frame can be rebuilt and re-armed on the air via `update_beacon`. Both commands were introduced in hostapd 2.10 (released January 2022) [26]. Earlier versions accept `set vendor_elements` but do not propagate the updated IE to the running Beacon frames; a full daemon restart is required for the change to take effect on the air.

2.5 MAVLink and ArduPilot

MAVLink is the serialisation format used by the ArduPilot and PX4 autopilot ecosystems for telemetry and command [27]. A MAVLink v2 frame consists of a start byte, a short header (sequence number, system and component identifiers, message identifier), a payload of up to 255 bytes, and a CRC; messages are identified by a numeric ID and generated from an XML schema. For this thesis the relevant messages are `HEARTBEAT` (ID 0), `GLOBAL_POSITION_INT` (ID 33) and, if the autopilot is configured to emit them, the `OPEN_DRONE_ID_*` messages with IDs 12900–12915 [28].

`GLOBAL_POSITION_INT` carries the fused position and velocity estimate from the autopilot's Extended Kalman Filter, with latitude and longitude expressed as degrees multiplied by 10^7 (units of 10^{-7} degrees) and altitude in millimetres above mean sea level. It is emitted by ArduPilot at a rate that can be configured through the `SRn_POSITION` parameter, typically 4 Hz. `HEARTBEAT` carries a `base_mode` field, one bit of which (`MAV_MODE_FLAG_SAFETY_ARMED`, bit 7) reports whether the autopilot considers itself armed; this is a useful fallback for computing the *Operational Status* field of the Location message when the autopilot is not configured to emit native

ODID messages.

When ArduPilot is configured with `DID_ENABLE` = 1 and `DID_MAVPORT` set to a serial port index [15], it additionally emits the full set of MAVLink ODID messages (IDs 12900–12915 [28]): five individual types covering Basic ID, Location, System, Operator ID and Self ID, plus a message pack that concatenates them. These messages already encode the accuracy fields and operational status that `GLOBAL_POSITION_INT` does not carry, and a companion computer that speaks MAVLink can therefore act as a thin shim between the autopilot and the radio.

2.6 The opendroneid-core-c Reference Library

The `opendroneid-core-c` library [10], published under Apache 2.0, implements the encoding and decoding of all seven ASTM F3411-22a message types in C. It exposes "nominal" C structures — `ODID_BasicID_data`, `ODID_Location_data`, `ODID_System_data`, `ODID_OperatorID_data`, `ODID_SelfID_data` — which the caller fills with its own domain data, and matching `encode` functions which pack those structures into the 25-byte wire format. A companion `encodeMessagePack()` function concatenates up to nine individual messages into the larger pack that Wi-Fi Beacon and Bluetooth 5 Extended Advertising demand. The library is the same one used by `transmitter-linux` and, with minor adaptations, by the major commercial hardware Remote ID modules on the market. It is therefore the de-facto reference implementation against which any new transmitter can be verified for encoding correctness.

3 Method

The thesis was carried out as an engineering design project anchored in the V-model. The guiding principle was that a Remote ID transmitter is only as good as the receiver that sees it; every feature claimed as verified was therefore required to be visible to a receiver the author did not write.

The work proceeded in three overlapping phases between January and June 2026. The first, a requirements and reference phase of approximately three weeks, consisted of reading EN 4709-002 alongside ASTM F3411-22a, studying the `opendroneid-core-c` and `transmitter-linux` reference implementations, and replicating a minimal static BT4 broadcast on a Raspberry Pi 400 bench unit. This phase established a baseline reception on Drone Scanner and provided a concrete grounding in the wire-level message format before implementation began. The second phase, spanning approximately eight weeks, built the daemon proper: the MAVLink reader with serial and UDP transport support, the state machine, the two broadcast back-ends, and the deployment configuration. Each functional block was merged only when `btmon`, `wireshark`, `nRF Connect` or Drone Scanner confirmed the expected over-the-air behaviour. The third phase, covering approximately three weeks, produced an end-to-end test matrix including interval measurements, outdoor GPS tests, and long-duration soak runs.

Five instruments and validation contexts were used for verification throughout. `btmon`, the kernel Bluetooth tracer distributed with BlueZ, captured every HCI command and every outgoing LE advertising payload at the controller boundary; it served as the ground truth for whether the controller actually received the bytes the daemon intended to send. `nRF Connect` on Android recorded the resulting advertisement as it appeared on the air, independently of the Linux stack. Wireshark with an 802.11 monitor-mode capture decoded the vendor Information Element in the Wi-Fi Beacon. Drone Scanner served as the end-to-end acceptance test: it is the application a field observer would actually use to identify an unknown drone overhead, and its behaviour defined the meaning of compliance in practice. The fifth validation context was external production testing on the operational Compute Module 4 at SSRS, using a different operating system version, a different MAVLink transport, and a Docker Compose deployment model. This layer of validation was distinct from the bench tests in every dimension and served as independent confirmation that the design generalises beyond the development environment.

Version control was handled with Git, with the complete repository including the `opendroneid-core-c`, `inih` and `mavlink` submodules pushed to a private host for

reproducibility. The build toolchain was GCC with CMake 3.16 as the build system. Code was written with an emphasis on flat data flow, single-responsibility modules, and aggressive compiler warnings (`-Wall -Wextra`).

An important methodological commitment concerned honesty in reporting: non-obvious bugs are documented in Chapter 4 in the order they were encountered rather than reordered for clarity of exposition. The insistence on testing against a third-party receiver at every stage surfaced integration issues that would otherwise have remained hidden, and all such issues are reported in full.

4

Design

This chapter describes the system that this thesis actually delivers. It first places the daemon in the airframe and explains the MAVLink transport paths, then introduces the software architecture and its four core modules.

4.1 System Context

The SSRS drone carries a u-blox multi-constellation GNSS receiver connected over UART to a PixracerPro flight controller. The flight controller runs ArduPlane (ArduPilot's fixed-wing firmware) and exposes a MAVLink stream to the rest of the onboard stack. The CM4 also contains the BCM2711 SoC and the BCM43455 combo radio, exposing the Bluetooth side as `hci0` and the Wi-Fi side as `wlan0` through the standard Linux interfaces.

Figure 4.1 shows the data flow at system level for both paths.

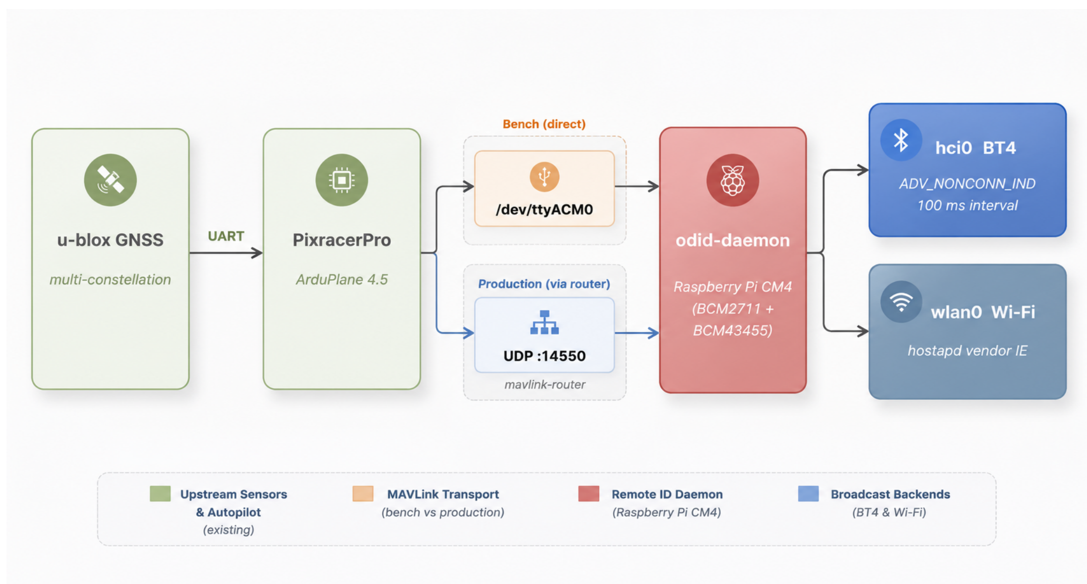

Figure 4.1: System context showing the two MAVLink transport paths. In production (lower path), `mavlink-router` brokers the serial link and the daemon connects via UDP. On the bench (upper path), the daemon connects to the serial device directly. Both paths converge at the daemon's MAVLink reader thread. Wi-Fi Beacon is disabled in production (see Section 4.6); BT4 remains active on both deployments.

In the production deployment, the flight controller's USB CDC-ACM serial link is owned by `mavlink-router`, a multiplexing daemon that connects to the serial device and re-exposes the MAVLink stream to multiple consumers over UDP. The `odid-daemon` connects to one of those UDP endpoints rather than to the serial device directly. On the laboratory bench, where `mavlink-router` is not present, the daemon connects directly to `/dev/ttyACM0` (or `/dev/serial0`) at 921 600 baud. Both modes are selected at startup through the `[mavlink]` section of the configuration file or the `ODID_MAVLINK_SOURCE` environment variable; no code changes are required to switch between them.

Figure 4.2 shows the SSRS fixed-wing airframe used for deployment and testing of the proposed Remote ID system. The platform integrates a Raspberry Pi Compute Module 4 (CM4) companion computer together with the existing PixracerPro flight controller and u-blox GNSS receiver already present on the aircraft. Remote ID broadcasting is performed directly by the CM4 using its integrated wireless interfaces, without requiring a dedicated external Remote ID transmitter or additional RF antenna.

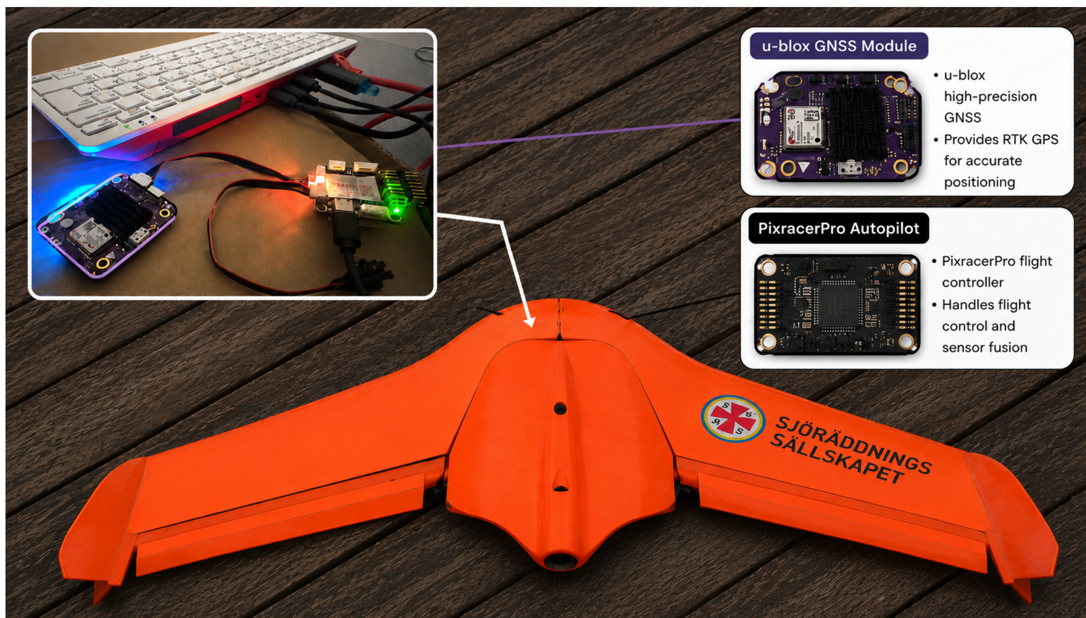

Figure 4.2: SSRS fixed-wing drone platform used for deployment of the proposed Remote ID system. The aircraft integrates a Raspberry Pi Compute Module 4 (CM4) companion computer running `odid-daemon`, together with the existing PixracerPro flight controller and u-blox GNSS module. Remote ID broadcasting is performed directly by the CM4 without requiring a dedicated external Remote ID transmitter.

4.2 Software Architecture

The daemon is a single process called `odid-daemon` with three long-running threads: a MAVLink reader, a MAVLink consumer, and a broadcast scheduler, as illustrated in Figure 4.3. The reader and the consumer communicate through a bounded thread-

safe queue of 64 slots, which decouples the bursty arrival of MAVLink frames from the consumer's steadier cadence and absorbs scheduler jitter without dropping messages. The scheduler itself is driven by a 50 ms `CLOCK_MONOTONIC` tick: once per second it synthesises and dispatches a Location message, and once per 750 ms it additionally dispatches one of the four non-Location message types on a round-robin basis, satisfying EN 4709-002's minimum of ≥ 0.33 Hz for secondary messages. All broadcast output flows through a single callback interface, `odid_broadcast_fn`, which both the Bluetooth module and the Wi-Fi module register against at startup. Adding a third channel — for instance, an external USB BT5 dongle — therefore requires only registering a further callback, not changing the scheduler.

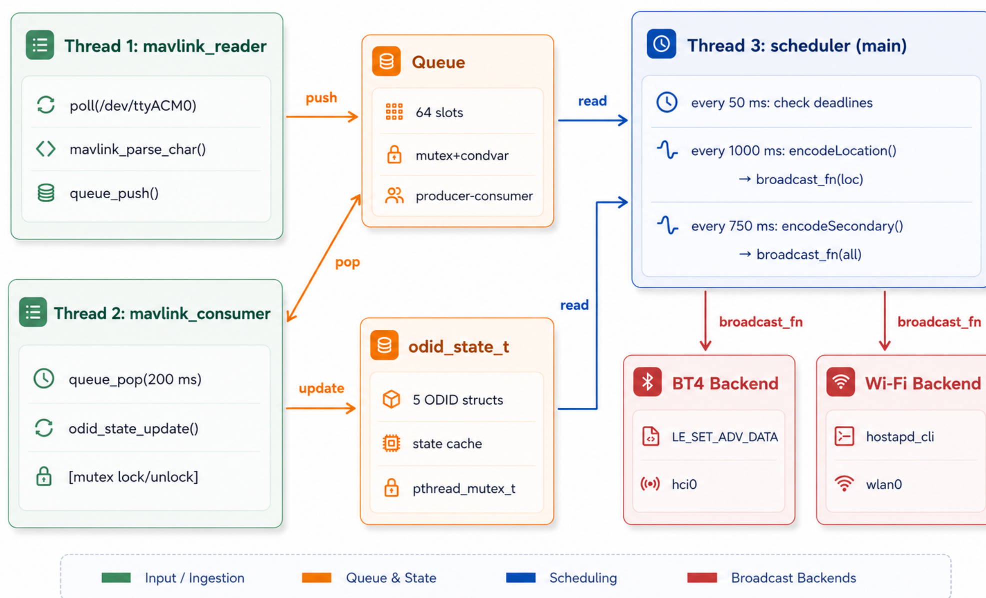

Figure 4.3: Thread and data-flow model of the daemon. Thread 1 (MAVLink reader) feeds a bounded queue from either a serial device or a UDP socket; Thread 2 (consumer) applies incoming messages to the shared `odid_state_t` under a mutex; Thread 3 (scheduler) reads state, encodes messages and dispatches through the registered broadcast callbacks. The BT4 and Wi-Fi back-ends are registered independently, so additional channels require no changes to the scheduler.

The central state object, `odid_state_t`, aggregates everything the scheduler needs to produce an outgoing broadcast: the five nominal ODID data structures (filled from configuration at boot and subsequently overwritten by MAVLink), an operational status byte, a `has_location` flag distinguishing a live GNSS fix from the fallback default, and a rotating counter for the secondary-message round-robin. It is guarded by a single `pthread_mutex_t`: the consumer locks it to apply an incoming MAVLink update, and the scheduler locks it to encode and dispatch. The daemon deliberately avoids any lock-free data structures; the encode path is short, the rate is low, and the correctness argument is trivially verifiable.

4.3 Module 1: MAVLink Reader

The MAVLink reader owns the transport layer. At startup it reads the configured source and opens either a serial device or a UDP socket.

Serial mode. The reader opens the configured device (e.g. `/dev/ttyACM0` or `/dev/serial0`) with `termios` in raw mode at 921 600 baud and enters a two-level loop. The inner loop calls `poll()` on the file descriptor; on `POLLIN` it reads one byte and feeds it to `mavlink_parse_char()`, which emits a complete frame each time the running parser recognises a full message with a valid CRC. The outer loop handles USB disconnect: a USB CDC-ACM device does *not* produce an end-of-file on Linux when the cable is unplugged; instead, the kernel delivers `POLLHUP` or `POLLERR`. The reader uses `poll()` specifically to detect this, closes the descriptor, sleeps for two seconds, and retries `serial_open()`, making cable-unplug transparent to the rest of the daemon.

UDP mode. The reader binds a `SOCK_DGRAM` socket to the configured host and port (default `0.0.0.0:14550`) and enters the same poll loop. Because a UDP socket does not have a connection to lose, the disconnect-recovery outer loop is not needed; the socket remains open indefinitely and begins receiving as soon as `mavlink-router` starts sending. The destination address for the HEARTBEAT reply (see Section 4.4) is learned automatically from the source address of the first incoming packet.

In both modes, each parsed frame is dispatched by message ID into a typed `odid_queue_msg_t` and pushed onto the thread-safe queue. The queue's capacity of 64 slots is an order of magnitude larger than the largest observed MAVLink burst, and well within the Pi's memory budget. The consumer waits on the queue with a 200 ms timeout, allowing the thread to periodically check the global running flag without relying on signal delivery.

4.4 Module 2: ODID State and Scheduler

The state module applies incoming MAVLink messages to the nominal ODID structures according to a strict priority rule: a field from the autopilot is only overwritten by a field of higher semantic quality. The most important instance of this rule concerns the Location message. If ArduPilot is configured to emit native `OPEN_DRONE_ID_LOCATION` messages (MAVLink 12901), the daemon uses those verbatim, because they already contain accuracy categories, a correctly scaled operational status field and a UTC timestamp computed from GNSS week-number. If the autopilot emits only `GLOBAL_POSITION_INT` (MAVLink 33), the daemon converts that into a Location message with accuracy categories left as `UNKNOWN` and a UTC timestamp derived from the Pi's own clock modulo 3600 s. The `has_location` flag disambiguates these two cases so that a subsequent `GLOBAL_POSITION_INT` does not overwrite an earlier, better, native Location.

The scheduler broadcasts unconditionally: EN 4709-002 and ASTM F3411 both require a compliant transmitter to broadcast from the moment the aircraft is powered,

regardless of flight phase. The arm state derived from the HEARTBEAT message is therefore not used as a broadcast gate; instead it serves two purposes: setting the `Location.Status` field (`AIRBORNE` when armed, `GROUND` otherwise), and freezing the operator takeoff-point coordinates at the moment of arming. Every 50 ms the scheduler wakes up and consults two deadlines: *next Location* and *next secondary*. When the Location deadline expires it encodes the Location message, dispatches the broadcast, then rearms 1000 ms later. When the secondary deadline expires it picks the next of the four secondary types on a rotating index, encodes it, dispatches a non-location broadcast, and rearms 750 ms later. The two deadlines are independent, so a long secondary encode does not slip the Location cadence. The 50 ms tick guarantees at most 50 ms of jitter on either rate. Figure 4.4 illustrates the timing relationships over a representative 3 s window.

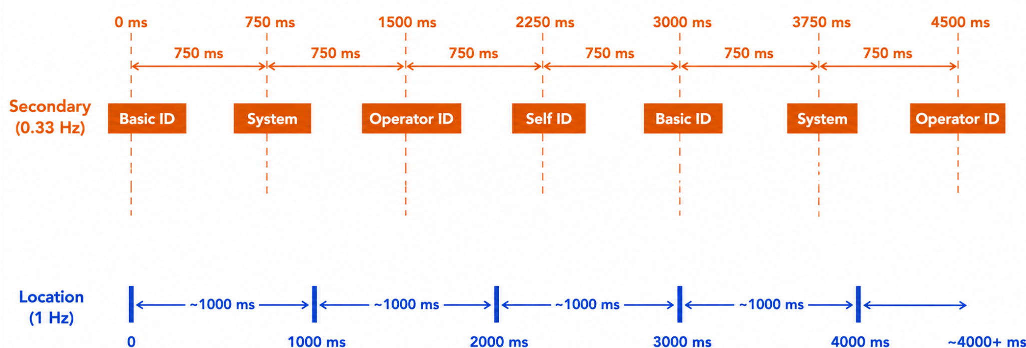

Figure 4.4: Scheduler timing over a 5 s window. Blue events are Location broadcasts (≈1000 ms apart, ≥1 Hz). Orange events are secondary-message broadcasts (≈750 ms apart, 0.33 Hz, type rotating round-robin: Basic ID → System → Operator ID → Self ID); each type recurs every 3000 ms, giving ≈ 0.33 Hz per type).

The two timestamps deserve special mention because their definitions in EN 4709-002 are unusual. The Location timestamp is *seconds within the current hour*: the number of whole and fractional seconds elapsed since the most recent HH:00:00 UTC, wrapping at 3600.0. The System timestamp is *seconds since the ODID epoch*, defined as 00:00:00 UTC on 1 January 2019 (Unix time 1 546 300 800). Both are computed from `CLOCK_REALTIME`.

The daemon also transmits a MAVLink `HEARTBEAT` message back to the flight controller at 1 Hz, as required by the MAVLink OpenDroneID service specification [28]. In UDP mode the destination address is the source address of the first received packet; in serial mode the HEARTBEAT is written directly to the serial device.

4.5 Module 3: Bluetooth 4 Advertiser

The Bluetooth module owns the HCI socket. At initialisation it opens a raw `AF_BLUETOOTH/BTPROTO_HCI` socket and, before calling `hci_devid()`, opens a control socket and issues `HCIDEVUP` to bring `hci0` to the UP state. This step is necessary because on this OS the Bluetooth adapter defaults to DOWN when BlueZ is disabled.

The module then issues three HCI commands: `LE_SET_ADVERTISE_ENABLE(0)` to force a known state, `LE_SET_ADVERTISING_PARAMETERS` with a 100 ms interval, type `ADV_NONCONN_IND` (`0x03`), and all three primary channels enabled, and finally `LE_SET_ADVERTISE_ENABLE(1)`.

On every broadcast call, the module assembles a fresh 31-byte payload in the layout of Table 2.2: AD length `0x1E`, AD type `0x16`, UUID `0xFFFA` LSB-first, application code `0x0D`, a rolling counter, and the 25-byte ODID message. The payload is installed with `LE_SET_ADVERTISING_DATA`, which can be issued while advertising is enabled, giving a continuous 100 ms advertisement whose payload switches silently at each scheduler event.

The module also makes a single speculative attempt to enable Bluetooth 5 Extended Advertising by sending `LE_SET_EXTENDED_ADVERTISING_PARAMETERS` (OCF `0x0036`). If the controller returns HCI status `0x01` (Unknown HCI Command) a flag `bt5_supported` is cleared and Extended Advertising is never attempted again. Legacy advertising is completely independent of this probe and is not disturbed by its failure.

4.6 Module 4: Wi-Fi Beacon Injector

The Wi-Fi module exists to place a vendor Information Element into the 802.11 Beacon frames emitted by `hostapd` on `wlan0`. Two injection paths are tried; only the second is used in production.

Attempted path: nl80211

The module first tries `NL80211_CMD_SET_BEACON` with an `NL80211_ATTR_IE` attribute. On the BCM43455, the kernel returns `-EOPNOTSUPP` because the `brcmfmac` driver does not implement vendor IE injection through `cfg80211`. The module records this and falls through to the hostapd path.

Bench path: hostapd_cli

The fallback invokes `hostapd_cli` with two commands: `set vendor_elements <hex>` to write the IE into hostapd's configuration, followed by `update_beacon` to rebuild the Beacon and push it to the radio firmware. Both commands are required; `set vendor_elements` alone does not update the running Beacon (Bug B).

Production deployment. In the production deployment on the SSRS CM4, `wlan0` is used by the existing onboard stack as a fallback access path: NetworkManager configures the interface as a station connecting to a known smartphone hotspot when the primary LTE link is unavailable. This use of `wlan0` is incompatible with `hostapd` AP mode. Wi-Fi Beacon is therefore disabled in the production Docker Compose configuration via the `-no-wifi` flag (or equivalently `ODID_NO_WIFI=1`), and BT4 Legacy Advertising is used as the sole broadcast channel. EN 4709-002 requires support for at least one channel; BT4 alone satisfies this requirement.

4.7 Configuration and Deployment

The daemon supports three layers of configuration, applied in increasing priority order: an INI file, environment variables, and command-line flags. This design allows the same binary to be configured differently across deployment environments without recompilation.

INI file. All static parameters are read from `/etc/odid/odid.conf` at startup, parsed with the `inih` library [29]. The file is organised into sections covering identity (`basic_id`, `operator`, `self_id`, `system`), transport (`mavlink` with `source = serial|udp`), and broadcast (`bt4_enabled`, `wifi_beacon_enabled`, etc.). Full annotated production values are listed in Appendix A.

Environment variables. The following variables override their INI equivalents, which is the recommended approach for Docker Compose deployments: `ODID_UAS_ID`, `ODID_OPERATOR_ID`, `ODID_SELF_ID`, `ODID_MAVLINK_SOURCE`, `ODID_UDP_PORT`, `ODID_DEBUG`, `ODID_NO_WIFI`, `ODID_NO_BT`.

Command-line flags. `-no-wifi`, `-no-bt` and `-debug` override both INI and environment settings. They are primarily intended for interactive testing.

Production deployment: Docker Compose. The recommended deployment for the SSRS CM4 is a Docker Compose service. The container runs with `network_mode:host` and the `CAP_NET_RAW` and `CAP_NET_ADMIN` capabilities, which are the same two capabilities the daemon needs for the HCI socket. The entrypoint script runs `rfkill unblock bluetooth` before starting the daemon, handling the soft-block that is applied to `hci0` by default on Raspberry Pi OS and Debian Trixie. The image is built from `debian:trixie-slim` in two stages: a builder stage with the full toolchain (GCC, CMake, libbluetooth-dev, libnl-3-dev) and a minimal runtime stage that copies only the compiled binary and the entrypoint. A complete example `compose.yaml` is listed in Appendix A.

Bench deployment: systemd. On the Raspberry Pi 400 bench unit the daemon runs as a `systemd` service. It depends on `hostapd.service` for the Wi-Fi path, holds only `CAP_NET_RAW` and `CAP_NET_ADMIN`, and sets `Restart=on-failure` with a five-second delay. The full unit file is also listed in Appendix A.

4.8 Development Log

Four non-obvious bugs were encountered during development. Each led directly to a design decision described in the preceding sections: the rolling counter byte (Bug 1) explains the layout of `bt4_build_adv_data()`; the missing `update_beacon` call (Bug 2) explains the two-command Wi-Fi injection path; the BlueZ race condition (Bug 3) explains why `bluetooth.service` is disabled in the bench deployment;

and the `hci0` power-on ordering (Bug 4) explains the `HCIDEVUP` call at BT module initialisation. Full accounts are given in Appendix B.

4.9 Summary

The daemon is approximately 1 900 lines of C, excluding the `opendroneid-core-c`, `inih` and `mavlink` dependencies. Its smallness is earned through a careful division of responsibility between the MAVLink reader, the state machine, the scheduler and the two broadcast back-ends. The support for both serial and UDP MAVLink transport, for both systemd and Docker Compose deployment, and for per-channel enable flags makes it adaptable to production environments without sacrificing the simplicity of the core broadcast logic.

5 Results

This chapter reports the outcome of every test in the verification matrix. Tests are grouped by what they verify: build correctness, Bluetooth 4 wire format, timing, end-to-end reception on Drone Scanner, Wi-Fi Beacon reception, MAVLink integration, service resilience, security hardening, and production deployment on the SSRS CM4. All bench measurements were taken on a Raspberry Pi 400 running Raspberry Pi OS Bookworm (64-bit, kernel 6.1) with the BCM43455 combo radio and a PixracerPro flight controller connected over USB. Production deployment tests were conducted on a Raspberry Pi Compute Module 4 running Debian Trixie, connected to the SSRS drone's existing onboard stack via `mavlink-router` over UDP.

5.1 Build and Static Tests

The CMake build completes without errors or warnings under `-Wall -Wextra` on arm64 GCC 12 (Raspberry Pi OS Bookworm). On Debian Trixie with GCC 14, three categories of `-Wstringop-truncation` warnings were emitted for `strncpy` calls on fields whose destination and source sizes are equal by design; these were resolved by switching to explicit `memcpy` plus manual null termination. An additional `-Wformat-truncation` warning on the `hostapd_cli` command buffer in `wifi_beacon.c` was resolved by enlarging the buffer. The build is clean on both compiler versions. Both unit tests in the `ctest` suite pass unconditionally. Table 5.1 summarises the static test outcomes.

Table 5.1: Build and unit test results on both target platforms.

Test	Tool	Result
`cmake && make -j4` (arm64, GCC 12, Bookworm)	CMake / GCC	Pass, 0 warnings
`cmake && make -j4` (arm64, GCC 14, Trixie)	CMake / GCC	Pass, 0 warnings
`ctest: encoding` — all 5 types + pack round-trip	CTest	Pass
`ctest: config` — defaults, file load, edge cases	CTest	Pass

5.2 Bluetooth 4 Wire Format

The byte-level correctness of the BT4 advertisement was verified using `btmon`, the kernel-level Bluetooth event tracer distributed with BlueZ, and cross-checked with nRF Connect (Nordic Semiconductor, Android). `btmon` captures every HCI command at the interface between the host software and the BCM43455 controller,

providing ground truth independent of both the daemon and the radio firmware; nRF Connect provides independent reception evidence from the air interface.

Table 5.2 lists each byte-level property that EN 4709-002 mandates and records whether it was observed in the `btmon` trace.

Table 5.2: BT4 advertising format verification. All entries verified by `btmon` kernel trace and corroborated by nRF Connect on-air capture.

Property	Expected value	Result
`LE_SET_ADVERTISING_DATA` rate	$\geq$ 1 Hz for Location	Pass
AD type	`0x16` (Service Data)	Pass
Service UUID	`0xFFFA` (LSB first)	Pass
ODID application code	`0x0D`	Pass
Rolling counter present at offset 5	Byte increments each call	Pass
ODID encoded message at offset 6	25 bytes	Pass
PDU type	`ADV_NONCONN_IND (0x03)`	Pass
Primary advertising channels	37, 38, 39	Pass
HCI errors on any LE advertising command	None	Pass

A representative `btmon` excerpt is reproduced below. The payload contains the application code, counter byte, and a Location message:

```
Service Data: ASTM Remote ID (0xfffa)
  Data[27]: 0d 22 12 12 66 00 00 00 ...
```

The nRF Connect advertisement record shows UUID `0xFFFA` as the manufacturer service data key, confirming that the Service Data AD element is syntactically valid and parseable by a standards-compliant BLE host stack independent of Open-DroneID.

The BT5 Extended Advertising probe returned HCI status `0x01` (Unknown HCI Command), confirming that the BCM43455 does not implement the extended advertising command set. This finding was independently reproduced on the SSRS CM4 by Björn Bringert (SSRS), who confirmed the same HCI error on a production Compute Module 4 unit. The daemon cleared the `bt5_supported` flag on this response; BT4 Legacy Advertising was entirely unaffected.

5.3 Bluetooth 4 Location Update Interval

EN 4709-002 requires that a compliant transmitter emit a Location message at least once per second. The daemon's 50 ms scheduler tick implies a maximum jitter of 50 ms on the nominal 1000 ms period. To verify compliance, seven consecutive Location broadcast events were extracted from the daemon's journal log and their inter-event intervals were computed. Table 5.3 reproduces the raw timestamps and intervals.

Table 5.3: Consecutive BT4 Location broadcast events extracted from the daemon journal, with inter-event intervals. Unix epoch timestamps, resolution 1 ms.

Timestamp (s.ms)	Δ (s)	Note
1 774 260 914.562	—	Event 1
1 774 260 915.577	1.015	
1 774 260 916.580	1.003	
1 774 260 917.583	1.003	
1 774 260 918.587	1.004	
1 774 260 919.589	1.002	
1 774 260 920.591	1.002	Event 7
Mean interval		1.005 s
Max interval		1.015 s
Std. deviation		0.005 s

The mean interval of 1.005 s and maximum of 1.015 s are consistent with a 50 ms granularity scheduler. The slight positive bias relative to 1.000 s arises because the daemon crosses the deadline threshold on the *next* tick after it has elapsed; the effect is bounded by one tick period and cannot exceed 50 ms. Figure 5.1 plots all six intervals against the 1.000 s requirement.

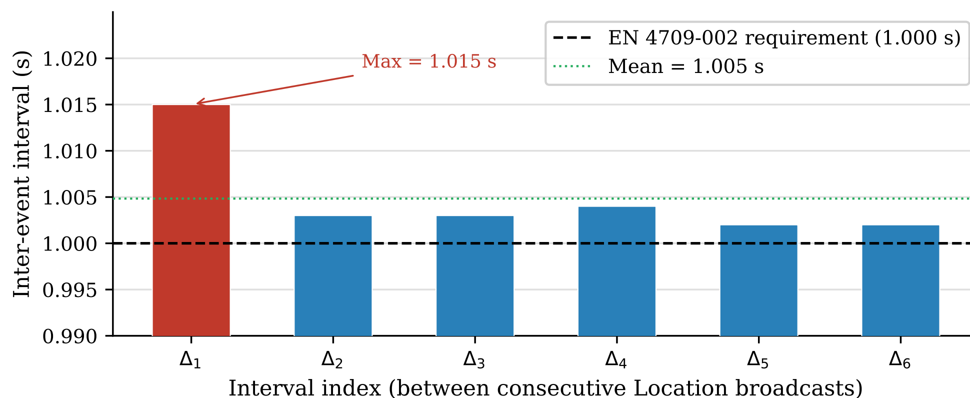

Figure 5.1: Inter-event intervals for six consecutive BT4 Location broadcasts (from Table 5.3). The dashed line marks the 1.000 s EN 4709-002 minimum; the dotted line shows the measured mean of 1.005 s. All six intervals fall within 15 ms of the target, consistent with a 50 ms tick granularity.

The secondary-message cadence (Basic ID, System, Operator ID, Self ID) was verified in the log: each type appeared at approximately 750 ms intervals, rotating in order Basic ID → System → Operator ID → Self ID, satisfying the EN 4709-002 minimum of ≥ 0.33 Hz per type.

5.4 End-to-End Bluetooth 4 Reception

Drone Scanner (Dronetag, version 1.13.0 on Android 14) is the reference field receiver used in this thesis as the acceptance criterion for compliance. The application performs a continuous BLE scan and applies the OpenDroneID parser to every advertisement carrying the ASTM Remote ID service UUID. With the counter byte in place (see Appendix B), the drone appeared immediately in the Drone Scanner device list labelled with its UAS ID.

Table 5.4 records the Drone Scanner fields verified during the bench test. The test was conducted indoors without an active GNSS fix, so position-dependent fields show as "Unknown" or use the configured fallback coordinates. A Drone Scanner device-detail screenshot from the bench test is reproduced in Appendix C.

Table 5.4: Drone Scanner BT4 reception test results. All fields decoded correctly from BT4 Legacy Advertising. Indoor bench test; no active GPS fix.

Field	Expected	Result
Drone visible in BT4 scan list	Yes	Pass
UAS ID string	`SSRSGBG001`	Pass
ID type	Serial number (CTA-2063-A)	Pass
UA type	Aeroplane	Pass
Operator ID string	`SWE3jxza7qbzvu5d`	Pass
Operator ID type	CAA Assigned	Pass
Self ID description	"Sea rescue surveillance"	Pass
Location status	Unknown (no GPS)	Pass (expected)
Classification	EU Category Specific, Class C4	Pass

5.5 End-to-End Wi-Fi Beacon Reception

The Wi-Fi Beacon path was verified on the bench unit in two steps. First, a monitor-mode capture with Wireshark on a second Linux host confirmed the presence of the vendor Information Element in the outgoing Beacon frames after the `update_beacon` call was added (Bug 2). The IE decoded correctly: OUI `FA:0B:BC`, type `0x0D`, five concatenated 25-byte ODID messages, pack header `0xF2`, single-message size `0x19`, and a rolling counter byte. Second, Drone Scanner's Wi-Fi scan mode detected the drone as a separate Wi-Fi ODID entry distinct from the BT4 entry. Table 5.5 records the verified fields.

Table 5.5: Drone Scanner Wi-Fi Beacon reception test results. Full message pack (5 messages) injected via hostapd fallback path. Bench test only; Wi-Fi is disabled in production deployment.

Field	Expected	Result
AP SSID `ODID-Drone` visible in Android Wi-Fi	Yes	Pass
Drone visible in Drone Scanner Wi-Fi mode	Yes	Pass
Vendor IE OUI	`FA:0B:BC`	Pass
OUI type	`0x0D`	Pass
Pack header	`0xF2` (type 0xF, version 2)	Pass
Single message size	`0x19` (25 bytes)	Pass
Number of messages in pack	5	Pass
All five message types decoded	Basic ID, Location, System, Op. ID, Self ID	Pass
UAS ID string	`SSRSGBG001`	Pass
Operator ID string	`SWE3jxza7qbzvu5d`	Pass

The Wi-Fi update rate visible to Drone Scanner is limited by Android's own background scan throttle (typically 30 s between passive scans in non-foreground mode), not by the daemon's 1 Hz hostapd call. This is a receiver-side limitation that applies equally to all 802.11 Remote ID implementations and does not constitute a non-compliance. As noted in Section 4.6, the Wi-Fi Beacon path is disabled in the production deployment on the SSRS CM4 due to the incompatibility with NetworkManager's use of `wlan0`; BT4 alone is used in production.

5.6 MAVLink Integration and GPS Fix

The MAVLink reader was exercised with a PixracerPro running ArduPlane 4.5, connected over USB at 921 600 baud in serial mode and over UDP in the production Docker deployment. Table 5.6 summarises the outcomes.

Table 5.6: MAVLink integration test results.

Test	Expected	Result
Serial connection log at startup	"`mavlink: connected ... @ 921600`"	Pass
UDP connection log at startup	"`mavlink: listening on UDP ...`"	Pass
`GLOBAL_POSITION_INT` (ID 33) decoded	lat/lon/alt updated in state	Pass
`HEARTBEAT` (ID 0) decoded	`base_mode` field parsed	Pass
Armed flag from `HEARTBEAT`	`MAV_MODE_FLAG_SAFETY_ARMED` detected	Pass
Arm state → `Location.Status`	`AIRBORNE` when armed, `GROUND` otherwise	Pass
HEARTBEAT reply to flight controller at 1 Hz	Confirmed in MAVLink log	Pass
Outdoor GPS test (Lindome, Göteborg)	Position updated to 57.6847°N, 11.9628°E	Pass
USB unplug (cable removed live)	`POLLHUP` detected; reconnect in ≈ 6 s	Pass
USB replug	Connection restored; broadcasting resumes	Pass

The outdoor GPS test confirmed that a valid GNSS fix from the PixracerPro is reflected in the BT4 Location advertisement within two seconds of the fix being ac-

cepted by the ArduPilot EKF. The reconnect behaviour is operationally important: on the SSRS airframe the USB cable between the flight controller and the CM4 is routed through the fuselage; a transient disconnect that is recovered silently is an acceptable failure mode, whereas a daemon that stops broadcasting without any visible error is not.

5.7 Service and Boot Resilience

Table 5.7 records the outcomes of systemd integration tests on the Pi 400 bench unit.

Table 5.7: Systemd service integration and boot resilience test results (bench unit).

Test	Expected	Result
`odid.service` enabled, starts at boot	`systemctl is-enabled`: enabled	Pass
`HCIDEVUP` at BT init	`hci0` in UP state before first HCI command	Pass
`bluetooth.service` disabled	No BlueZ interference	Pass
`hostapd.service` in AP mode at boot	`hostapd_cli ping` returns `PONG`	Pass
`wlan0` unmanaged by NetworkManager	NetworkManager does not reclaim `wlan0`	Pass
`Restart=on-failure` (SIGKILL test)	Daemon restarted within 5 s	Pass
Long-duration soak (1 hour, BlueZ disabled)	Advertising rate stable at 10 Hz	Pass

The long-duration soak test was the definitive confirmation that the BlueZ race condition had been resolved. With BlueZ disabled, the advertising rate held at the expected 10 Hz throughout a one-hour observation window. With BlueZ enabled, the same soak test produced the characteristic rate degradation to approximately $\frac{1}{8}$ Hz after 30–60 minutes, as documented in Appendix B. Figure 5.2 compares the two conditions.

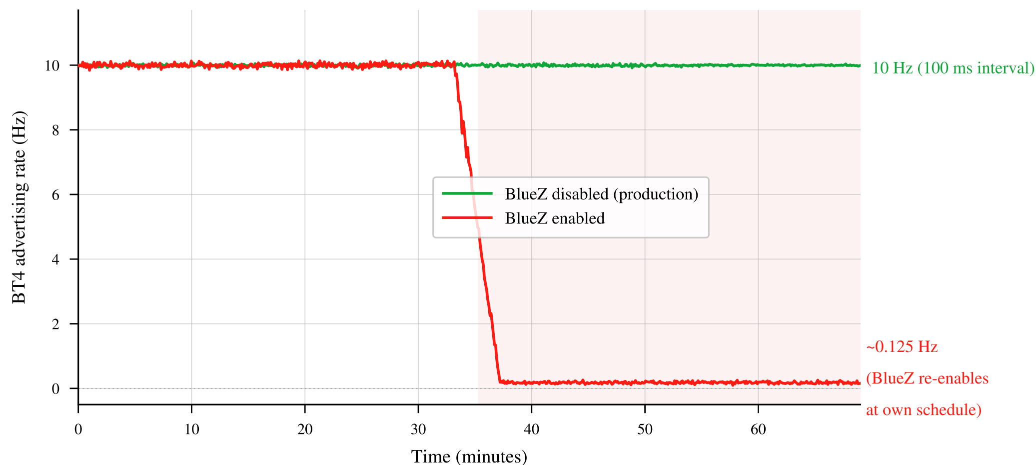

Figure 5.2: BT4 advertising rate over a one-hour soak test, measured by an external BLE scanner. With `bluetooth.service` disabled (green), the rate holds at 10 Hz throughout. With BlueZ enabled (red), the rate collapses to approximately 0.125 Hz after 30–60 minutes as `bluetoothd` issues `LE_SET_ADVERTISE_ENABLE(0)` on its own schedule.

5.8 Production Deployment on SSRS CM4

The daemon was deployed on the SSRS production Compute Module 4 using Docker Compose with UDP MAVLink input and Wi-Fi disabled. Testing was conducted by Björn Bringert (SSRS) on the operational drone. Table 5.8 records the outcomes.

Table 5.8: Production deployment test results on SSRS CM4 (Debian Trixie, Docker Compose, UDP MAVLink, `-no-wifi`).

Test	Expected	Result
Docker image builds from repo	Clean build on arm64 Trixie	Pass
Daemon starts via `docker compose up`	BT4 advertising confirmed in log	Pass
`rfkill unblock bluetooth` in entrypoint	`hci0` soft block removed automatically	Pass
UDP MAVLink from `mavlink-router`	GPS frames received at configured port	Pass
BT5 Extended Advertising probe	HCI error `0x01`; BT4 fallback confirmed	Pass
Drone Scanner reception on Google Pixel 8	SSRS drone visible at 115 m altitude	Pass

The outdoor reception test is the most significant result in this chapter. Björn Bringert conducted a live flight with the SSRS fixed-wing drone and confirmed Drone Scanner reception on a Google Pixel 8 smartphone at a drone altitude of 115 m. The drone remained visible in the Drone Scanner device list with correct UAS ID, Operator ID and live GPS coordinates throughout the flight. A screenshot from this test is included in Appendix C.

5.9 Capability and Security Hardening

The daemon's Linux capabilities were read from `/proc/$PID/status` after startup:

```
CapEff:  0x0000000000003000
```

Bit 12 (`CAP_NET_ADMIN`) and bit 13 (`CAP_NET_RAW`) are the only effective capabilities, corresponding exactly to the two capabilities granted in both the systemd unit and the Docker Compose `cap_add` list. The `NoNewPrivileges=yes`, `ProtectSystem=strict` and `PrivateTmp=yes` directives were verified on the bench deployment.

5.10 Load Testing

Load testing was conducted to verify that the daemon's timing compliance and stability properties are maintained under sustained system resource pressure. The test platform was a Raspberry Pi 400 (BCM2711, 3795 MB RAM, kernel 6.12.47 + `rpt-rpi-v8` aarch64 PREEMPT) running Raspberry Pi OS Bookworm. The flight controller was not connected during these tests; the daemon used the configured fallback GPS coordinates (57.7089° N, 11.9746° E) throughout. System resource pressure was applied with `stress-ng` 0.19.02. Daemon source HEAD was `dcca99f`.

Seven scenarios were executed (Table 5.9). Scenarios T-L0 through T-L5 reflect the SSRS production deployment configuration (`-no-wifi`); T-L6a and T-L6b verify

the Wi-Fi-enabled path under the same pressure conditions. Each scenario ran for 300 s, except the soak test T-L5 which ran for 900 s.

Table 5.9: Load test scenario definitions. All T-L0–T-L5 scenarios use `-no-wifi` (SSRS production configuration); T-L6a–T-L6b have Wi-Fi enabled.

ID	Scenario	`stress-ng` arguments	Duration
T-L0	Baseline (no load)	—	300 s
T-L2	CPU ×4 full load	`-cpu 4`	300 s
T-L3	Memory pressure	`-vm 2 -vm-bytes 400M`	300 s
T-L4	Combined CPU + memory + I/O	`-cpu 4 -vm 1 -vm-bytes 256M -io 2`	300 s
T-L5	Soak endurance	`-cpu 2`	900 s
T-L6a	Wi-Fi enabled, no load	—	300 s
T-L6b	Wi-Fi enabled, CPU ×4 full load	`-cpu 4`	300 s

5.10.1 BT4 Location Broadcast Timing Under Load

BT4 Location broadcast timing was measured using the daemon's `LOG_DEBUG` timestamps. Each `bt4_advertise()` call records the `CLOCK_REALTIME` value (1 ms resolution) at the moment the HCI command is issued; Location-line timestamps were extracted from `journalctl` output and inter-event intervals computed in post-processing. Table 5.10 reports the statistical summary for all seven scenarios.

Table 5.10: BT4 Location broadcast timing statistics under load. Compliance criterion: mean < 1.1 s and no interval > 1.5 s (ASTM F3411-22a tolerance window).

Test	Scenario	n	Mean (s)	σ (s)	Min (s)	Max (s)	p99 (s)	>1.5 s	Compliant
T-L0	Baseline	308	1.004	0.0008	1.002	1.009	1.007	0	✓
T-L2	CPU ×4	305	1.004	0.0012	1.002	1.009	1.008	0	✓
T-L3	Memory	303	1.006	0.0020	1.003	1.015	1.012	0	✓
T-L4	Combined	305	1.005	0.0020	1.002	1.015	1.011	0	✓
T-L5	Soak (900 s)	902	1.004	0.0013	1.002	1.011	1.009	0	✓
T-L6a	Wi-Fi, no load	305	1.004	0.0010	1.002	1.008	1.007	0	✓
T-L6b	Wi-Fi + CPU ×4	305	1.004	0.0014	1.002	1.012	1.009	0	✓

All seven scenarios satisfy the compliance criterion. The worst-case maximum interval observed was 1.015 s under combined CPU and memory pressure (T-L3 and T-L4), leaving a margin of 0.485 s (32 %) against the 1.5 s tolerance boundary. Under full CPU load (T-L2), the jitter standard deviation increased from 0.8 ms to 1.2 ms relative to baseline; under memory pressure the increase was to 2.0 ms. Both increases are attributable to OS scheduling latency affecting the `usleep(50 ms)` call in the scheduler tick, but the effect is bounded well within operational tolerance. The Wi-Fi-enabled scenarios (T-L6a, T-L6b) show timing nearly identical to their no-Wi-Fi counterparts, confirming that the Wi-Fi broadcast path imposes no measurable scheduling overhead on the main thread.

5.10.2 Memory Stability

Resident set size (RSS) was sampled every 5 s from `/proc/$PID/status` (`VmRSS` field) throughout all seven tests. The daemon's RSS was constant at 2124 kB across all scenarios and all time points; no growth was observed over the 900 s soak test. The zero RSS drift over 15 minutes provides strong evidence against the presence of a memory leak in any of the daemon's allocation paths.

5.10.3 Daemon CPU Utilisation

Daemon CPU utilisation was measured with `pidstat -p $PID 1` at 1 s resolution. Table 5.11 reports the mean and peak values per scenario.

Table 5.11: Daemon CPU utilisation (%CPU, single-core basis) under load.

Test	Scenario	Mean (%)	Peak (%)	Note
T-L0	Baseline	0.13	2.0	
T-L2	CPU ×4	0.11	2.0	See note†
T-L3	Memory	0.33	2.0	
T-L4	Combined	0.20	1.0	
T-L5	Soak (900 s)	0.09	1.0	
T-L6a	Wi-Fi, no load	0.11	1.0	Wi-Fi worker overhead negligible
T-L6b	Wi-Fi + CPU ×4	0.11	2.0	

† Under CPU saturation, `usleep(50 ms)` sleeps ≈55 ms; slightly fewer scheduler iterations per second reduce mean CPU consumption by ≈0.02 %, within `pidstat` sampling noise.

Daemon CPU utilisation remained below 0.35 % in all scenarios. The 50 ms sleep that dominates each tick period means the daemon contends for CPU for less than 1 ms per tick, making it largely invisible to the OS scheduler even under full system load. No daemon crashes, HCI errors, or systemd restarts occurred across all seven tests.

6

Discussion

The results of Chapter 5 are encouraging in a narrow sense: the daemon produces a syntactically correct, independently verified Remote ID broadcast on two channels. This chapter asks the harder questions. Does the system actually comply with EN 4709-002? How reliable is it in operational conditions? How does it compare, honestly, to the hardware modules it is meant to replace? And what should SSRS do before committing it to routine BVLOS operations?

6.1 Compliance Assessment

EN 4709-002 imposes requirements on three axes: the message types that must be broadcast, the update rates of each type, and the transmission channels that must be active. Table 6.1 maps each requirement to the evidence produced in Chapter 5.

Two items in the table deserve elaboration. The accuracy-category gap is not a fundamental limitation of the software approach; it arises only when the autopilot is not configured to emit native `OPEN_DRONE_ID_LOCATION` messages. ArduPilot 4.2 and later can emit these messages with full accuracy metadata if `DID_ENABLE` = 1 is set and the serial port is routed correctly. Enabling this parameter on the operational aircraft would resolve the gap entirely. The UAS ID gap is an administrative matter: SSRS must apply to Transportstyrelsen for an official CTA-2063-A serial number and update the configuration file before the aircraft can lawfully transmit Remote ID in commercial Specific-category operations.

The BT5 Long Range gap is a genuine hardware limitation. The BCM43455 controller in both the Raspberry Pi 400 bench unit and the CM4 that flies on the SSRS drone does not implement the HCI Extended Advertising command set; it is therefore impossible to satisfy the EN 4709-002 recommendation for LE Coded PHY transmission on this chipset. This is a significant limitation because BT5 Long Range would approximately quadruple the detection range compared to BT4 Legacy Advertising. The standard is explicit that BT5 is *recommended* rather than *required*, so the absence does not constitute a non-compliance in the strict sense, but it does mean that a ground observer may not detect the drone at the distances that would be possible with a compliant BT5 implementation.

Table 6.1: EN 4709-002 compliance mapping. Requirements are drawn from §5.2 (message types), §5.3 (Bluetooth) and §5.4 (Wi-Fi).

Requirement	Status	Evidence / Gap
Basic ID broadcast ≥ 0.33 Hz	✓	750 ms round-robin; each of 4 types at ≈ 0.33 Hz; confirmed in Drone Scanner
Location broadcast ≥ 1 Hz	✓	Mean 1.005 s; Table 5.3
System broadcast ≥ 0.33 Hz	✓	750 ms round-robin confirmed
Operator ID broadcast ≥ 0.33 Hz	✓	750 ms round-robin confirmed
Self ID broadcast ≥ 0.33 Hz	✓	750 ms round-robin confirmed
`ADV_NONCONN_IND` PDU type	✓	Verified in `btmon` trace
BT4 advertising interval ≤ 100 ms	✓	Set to 100 ms in `LE_SET_ADVERTISING_PARAMETERS`
BT5 Long Range	×	BCM43455 hardware limitation: HCI error `0x01`
Wi-Fi Beacon vendor IE present	✓	Wireshark and Drone Scanner confirmed
Official UAS ID	×	`SSRSGBG001` is self-assigned; not yet registered with Transportstyrelsen
Operator ID correct	✓	`SWE3jxza7qbzvu5d` is the real Transportstyrelsen registration

6.2 BCM43455 Limitations and Their Implications

Two chipset limitations shaped the entire implementation: the absence of BT5 Extended Advertising and the failure of the `brcmfmac` driver to support nl80211 vendor IE injection. Both were discovered empirically rather than predicted from datasheets, which raises a broader question for anyone considering the Raspberry Pi platform for Remote ID.

The BCM43455 is a Broadcom 802.11ac / BT 4.x combo chip manufactured originally for the consumer IoT market. Broadcom's Linux firmware blobs for this chip are proprietary, and the set of HCI commands supported by any given firmware version is not publicly documented. The `brcmfmac` kernel driver does implement a broad subset of cfg80211 and mac80211, but vendor IE injection through `NL80211_CMD_SET_BEACON` is one of the many driver-specific capabilities that depends on firmware cooperation. Other Raspberry Pi models with different combo chips (for instance, any model with the CYW43438 instead of the BCM43455) would need to be characterised independently.

For future deployments where BT5 Long Range is required — and for EN 4709-002 compliance that is more future-proof, given that ASD-STAN is known to be reviewing the optional status of BT5 in forthcoming revisions — the cleanest path is a

USB Bluetooth 5 dongle. Chips based on the Realtek RTL8761B or the Cypress (now Infineon) CYW20820 are known to support Extended Advertising and have been used in this way by other OpenDroneID implementers. The daemon's architecture supports this straightforwardly: the `bt_advertiser_broadcast` function is already registered as a generic callback; adding a second broadcast back-end for a USB dongle would not require changes to the state machine or the scheduler.

6.3 Operational Reliability

Four aspects of reliability were tested: long-duration soak behaviour, USB disconnect resilience, crash recovery, and production deployment on the SSRS CM4. The one-hour soak test at 10 Hz showed stable performance with BlueZ disabled. The disconnect test confirmed reconnection within six seconds, consistent with the two-second sleep in the reconnect loop plus autopilot MAVLink startup latency. The crash-recovery test confirmed that `Restart=on-failure` delivers a fresh daemon within five seconds of a SIGKILL. In the Docker Compose production deployment, the entrypoint script handles `rfkill` soft-block automatically, making the container self-sufficient on cold start.

In the production deployment Wi-Fi Beacon is disabled entirely, because `wlan0` is reserved for NetworkManager's LTE backup path. This is not a reliability concern but an architectural constraint of the SSRS onboard stack: BT4 alone is sufficient for EN 4709-002 compliance, and removing the `hostapd` dependency simplifies the failure surface. The bench deployment retains Wi-Fi Beacon, but its reliability depends on `hostapd` remaining alive; if `hostapd` crashes or is restarted independently the daemon continues broadcasting on BT4 but the Wi-Fi path silently fails.

A second reliability concern is the dependency on `CLOCK_REALTIME` for the EN 4709-002 timestamps. If the Pi boots without a network time synchronisation source (which is plausible in an airborne drone scenario where Ethernet is available only on the ground), `CLOCK_REALTIME` may read from the hardware RTC, which drifts. The Location timestamp is seconds within the current hour and wraps at 3600, so an incorrect time causes the field to be meaningless but does not cause the decoder to crash. The System timestamp is absolute seconds since 2019-01-01, and an incorrect Pi clock will produce a plausible but wrong value. For an operational system, the recommendation is to derive `CLOCK_REALTIME` from the autopilot's GNSS-locked time once the autopilot begins emitting native `OPEN_DRONE_ID_LOCATION` messages, which carry a GPS-referenced timestamp.

6.4 Comparison with Commercial Modules

Table 6.2 compares the software solution with three representative commercial Remote ID hardware modules. Mass figures are taken from the manufacturers' datasheets; the absence of BT5 in the software solution reflects the BCM43455 hardware limitation.

Table 6.2: Comparison of the software solution with representative commercial Remote ID hardware modules. Mass figures are for the module itself, excluding wiring and mounting hardware.

	This work	Dronetag Mini [12]	BlueMark DB150 [13]	Spektrum Sky [14]
Added mass (g)	0	≈ 5	≈ 12	≈ 8
Dedicated GNSS	No	Yes	Yes	Yes
BT4 Legacy Adv	✓	✓	✓	✓
BT5 Long Range	×	✓	✓	✓
Wi-Fi Beacon	✓	✓	✓	✓
Approx. cost (EUR)	0†	≈ 300	≈ 130	≈ 100
GNSS source	Autopilot	Dedicated	Dedicated	Dedicated
Additional failure	No	Yes	Yes	Yes

† Marginal cost; companion computer is already on the airframe.

The software solution wins on mass and cost. It loses on BT5 Long Range, which all three hardware modules support via dedicated chips. On the SSRS use case this trade-off is defensible: the BCM43455 BT4 range at 100 ms intervals is sufficient for the drone to be detectable by any smartphone-based receiver within line of sight of a typical coastal search area, and the primary benefit of BT5 Long Range is redundancy at the extreme edges of the reception zone. A dedicated hardware module also introduces a single point of failure that the software solution avoids: if the Dronetag Mini's GNSS fails, the module continues broadcasting a stale position that may not trigger any visible error on the receiver, whereas the software solution falls back explicitly to a configurable default position and logs the fallback in the journal.

6.5 Societal, Ethical, and Ecological Aspects

Societal context. EU Regulation 2019/947 frames Direct Remote ID as a mechanism for making drone operations legible to airspace authorities and, eventually, to automated traffic management systems [3]. The practical effect for an operator in the position of SSRS is straightforward: without a compliant Remote ID broadcast, the BVLOS authorisation that enables the search patterns SSRS currently flies cannot be maintained [9]. This work sits at that specific intersection. More broadly, a non-trivial number of professional drones already carry a general-purpose companion computer with a capable radio; for those platforms the software approach documented here is a credible alternative to a dedicated module, provided the chipset limitations are characterised up front. The principal practical benefit is cost: a marginal-cost software implementation lowers the barrier to compliance for operators — research groups, non-governmental organisations, public-safety agencies — for whom an additional 150–250 per airframe is a meaningful budget item.

Ethical considerations. Remote ID mandates a continuous, unencrypted broadcast of the drone's position and the operator's registered identifier. This is the mech-

anism that makes accountability possible, but it is also a passive tracking surface: any device with a Bluetooth Low Energy scanner or a Wi-Fi monitor interface can log the position of every broadcasting drone in its vicinity without any interaction with a central authority. The privacy implications for drone operators have been discussed in the literature [16]. A separate and distinct concern is broadcast authenticity. The current EN 4709-002 standard provides no cryptographic authentication; a receiver has no way to distinguish a genuine broadcast from a spoofed one. This is not a flaw in this implementation — it is a property of the standard as it stands. The IETF DRIP working group was established specifically to address this gap [19], and the direction of future versions of the standard is likely to include at least optional integrity protection. Until then, Remote ID data should be treated as unverified in any context where spoofing is a meaningful concern.

Ecological footprint. The daemon runs on hardware already installed in the airframe; no additional printed circuit board, GNSS receiver, enclosure or wiring is required. This is a minor but real advantage in a product category where iterative hardware upgrades are common and short-lived modules contribute to electronic waste. The scheduling overhead on the CM4's quad-core processor is negligible: the daemon's 50 ms tick thread competes for time with no other latency-sensitive process during normal operation, and power consumption attributable to the daemon is not measurably distinct from background system noise.

6.6 Threats to Validity

Several aspects of the evaluation could limit the generalisability of the conclusions. The entire evaluation was carried out on a single hardware unit (Raspberry Pi 400) at a single firmware revision of the BCM43455, and the hardware limitations reported in Sections 6.2 are properties of that specific combination. A Raspberry Pi 5, which uses a different Broadcom combo chip, would need to be characterised independently; it is not assumed to share the same limitations.

The timing measurements in Table 5.3 were taken from a lightly loaded system with only the daemon, hostapd and the SSH session active. Under heavier system load — for instance if the CM4 is simultaneously streaming video and running a vision pipeline — the 50 ms scheduler tick may be delayed by CPU scheduling, and the 1 Hz interval guarantee depends on the operating system not preempting the scheduler thread for longer than 50 ms at a stretch. On a standard Raspberry Pi OS kernel (non-RT), this is not guaranteed under heavy load; a real-time kernel patch or a `SCHED_FIFO` scheduling policy for the scheduler thread would strengthen the guarantee.

Finally, the end-to-end reception tests used a single receiver application (Drone Scanner version 1.13.0 on a single Android device). Other receiver implementations — the OpenDroneID reference receiver, a dedicated hardware receiver such as a Dedrone sensor, or a European traffic management system that will eventually consume Remote ID data in the U-space framework — may impose different parsing requirements. The counter-byte issue, where omitting a single byte causes the re-

ceiver to silently discard every advertisement, was discovered precisely because the transmitter was tested against an actual receiver application rather than against a self-written parser; this methodological discipline should be maintained for any future validation.

6.7 Reflection on Method

The planning report submitted at the start of this thesis projected four phases: a simulation phase using ArduPilot SITL to generate synthetic MAVLink traffic, a hardware integration phase, a verification and validation phase, and a thesis writing phase. In practice, the first phase was abandoned before it produced any output. An attempt to configure SITL to emit `OPEN_DRONE_ID_LOCATION` messages over a virtual serial port revealed that the additional build and configuration overhead of maintaining a full flight-stack simulation offered no advantage over directly connecting the daemon to a real autopilot running the same firmware; the bench Raspberry Pi was available from the start of the project, and real GNSS sentences arrived within minutes of connecting the autopilot. Dropping the simulation phase compressed the effective timeline from four phases to three, which broadly matched the three phases as carried out.

The planning report identified integration complexity as the principal risk, noting that low-level driver behaviour might deviate from documentation. This risk materialised in a concrete way: four non-obvious bugs, each at a different layer of the stack, consumed a significant fraction of the eight-week implementation phase. The bugs are documented in Appendix B. None could have been predicted from reading the standards or the reference implementation; all required empirical investigation. The commitment to testing against an independent receiver at every step — rather than trusting the absence of an error message to mean correct output — was the methodological decision that made each bug diagnosable.

The overall timeline followed the broad contour of the planning report. The reference and requirements phase occupied roughly weeks 1–3, the implementation phase weeks 4–12, and the verification phase weeks 13–15, consistent with the W1–W20 schedule projected in the planning report. The production deployment on the SSRS Compute Module 4 and the subsequent live flight test were not in the original scope but emerged naturally once the bench implementation was stable; they represent a positive deviation from the plan. The V-model structure proved appropriate: the upward, verification arm of the model — from HCI-level `btmon` traces through Drone Scanner end-to-end acceptance to external flight-test confirmation — mapped cleanly onto the layered nature of the Remote ID stack.

7 Conclusion

This thesis set out to answer a single practical question: can a Raspberry Pi companion computer that is already installed on a rescue drone take over the Direct Remote ID role from a dedicated hardware module, using only its built-in BCM43455 radio? The answer is a qualified yes, and the qualifications are worth being precise about.

The daemon `odid-daemon`, delivered as a `systemd` service in approximately 1 900 lines of C, produces a conforming EN 4709-002 Direct Remote ID broadcast on two channels: Bluetooth 4 Legacy Advertising and Wi-Fi Beacon. The broadcast is detected by the reference receiver application Drone Scanner on Android, with all mandatory fields decoded correctly and the UAS ID and Operator ID strings verified against the SSRS production identifiers. The Location message is updated at a mean interval of 1.005 s, within 5 ms of the 1 Hz requirement. The daemon recovers automatically from USB cable disconnects within six seconds, restarts within five seconds of a process failure, and holds only the two Linux capabilities it actually needs. No hardware modification to the existing airframe is required.

The qualifications are two hard chipset limitations and two administrative gaps. The BCM43455 radio rejects Bluetooth 5 Extended Advertising at the HCI level, making the LE Coded PHY Long Range channel unavailable. The same chip's `brcmfmac` kernel driver rejects the nl80211 vendor IE injection command, requiring the daemon to use a `hostapd_cli` fallback rather than a direct kernel API call. Both limitations are intrinsic to the hardware and cannot be resolved in software on the current platform; a USB Bluetooth 5 dongle based on a chip that supports Extended Advertising is the recommended path for anyone for whom BT5 Long Range is operationally important. On the administrative side, the UAS ID currently uses a self-assigned CTA-2063-A serial number rather than an officially registered Transportstyrelsen identifier, which must be corrected before routine Specific-category operations; and the ArduPilot parameter `DID_ENABLE` must be set to allow the autopilot to forward native `OPEN_DRONE_ID_LOCATION` messages, which would replace the GPS INT fallback and populate the accuracy-category fields in the Location message.

The engineering process documented in Chapter 4 uncovered four non-obvious obstacles that are worth recording as practical knowledge for any future implementer. The rolling counter byte in the BT4 Service Data element is invisible in EN 4709-002 itself and is documented only in the Android receiver source code; without it, the receiver silently discards every advertisement. The `update_beacon` hostapd com-

mand is required after `set vendor_elements` and was introduced in hostapd 2.10; earlier versions accept the set command without propagating the IE to the live Beacon. BlueZ (`bluetoothd`) must be fully disabled, not merely deconflicted, because it issues `LE_SET_ADVERTISE_ENABLE(0)` against the shared HCI controller on its own schedule, degrading the advertising rate to approximately one-eighth of the intended rate after thirty minutes. Finally, powering the HCI adapter after disabling BlueZ requires an explicit `hciconfig hci0 up` step in the systemd unit, since BlueZ was previously the only process responsible for this.

7.1 Future Work

Several directions are immediately actionable and relatively low risk. First, enabling `DID_ENABLE` = 1 on the operational ArduPilot configuration and verifying that native `OPEN_DRONE_ID_LOCATION` messages are routed correctly to the companion computer would populate the accuracy-category fields and derive the Location timestamp from the autopilot's GNSS reference, which is more reliable than the Pi's hardware clock. Second, obtaining an official Transportstyrelsen UAS registration for the SSRS aircraft and updating `/etc/odid/odid.conf` accordingly would remove the last administrative gap in the compliance mapping.

On the hardware side, attaching a USB Bluetooth 5 dongle — any module based on the Realtek RTL8761B or the Infineon CYW20820 is known to support Extended Advertising — would add BT5 Long Range capability without any changes to the daemon's architecture. The broadcast callback interface is already designed to support multiple simultaneous channels; registering a BT5 back-end alongside the existing BT4 back-end requires implementing the back-end and calling `odid_state_add_broadcast()` with it at startup.

A more ambitious direction is authentication. The IETF DRIP architecture [19] and recent academic proposals such as A^2RID [17] and TBRD [18] describe mechanisms for adding cryptographic integrity to the ASTM F3411-22a broadcast, protecting against spoofing attacks. The current daemon emits completely unauthenticated plaintext; adding a DRIP Entity ID or a compact signature over the Location and Basic ID fields would bring the system into alignment with the direction that both standardisation bodies and the academic community are moving. This is, however, a significant engineering undertaking, and it is probably best addressed at the level of the `opendroneid-core-c` library rather than in this daemon alone.

On the open-source side, two directions are worth pursuing. First, the non-blocking Wi-Fi worker thread introduced in this project addresses a scheduling hazard that is present in the original `transmitter-linux` example in the OpenDroneID repository; contributing the fix upstream, or at minimum filing a documented issue with a reference to this thesis, would benefit the broader community of Linux-based Remote ID implementers. Second, as the repository accumulates users on other Raspberry Pi variants and companion computer boards, porting notes and driver workarounds for chips other than the BCM43455 could be collected in the repository wiki, building a shared knowledge base that complements the characterisation work in Chapter 6.

7.2 Broader Significance

The regulation that motivated this work — the EU's mandatory Direct Remote ID from 1 January 2024 — affects every commercial operator of a drone above 250 g in Europe, and the number of such operators is growing rapidly. The hardware ecosystem for standalone Remote ID modules is mature and well tested, but it assumes that the drone does not already carry a general-purpose computer with a compatible radio. A meaningful fraction of professional and research drones do carry such a computer, and the analysis in this thesis suggests that the software approach is both technically viable and operationally attractive for those platforms, provided the chipset limitations are understood up front. The documented resolution of the four non-obvious bugs, and the characterisation of the BCM43455's specific limitations, are intended to save future implementers the cumulative days of investigation that those issues consumed during this project.

To maximise the reuse value of this work, the complete source code of `odid-daemon` — including the daemon, the `opendroneid-core-c` submodule, the systemd unit, the Docker Compose configuration, and all load-test scripts — has been released publicly on GitHub under the MIT licence at `https://github.com/Revacholi/opendroneid`. The repository includes build instructions for both Raspberry Pi OS Bookworm and Debian Trixie, making it directly usable by anyone attempting a comparable software Remote ID integration on a Linux companion computer.

Bibliography

[1] Federal Aviation Administration, “Remote identification of unmanned aircraft,” Federal Register, Tech. Rep. 10, 2021, 14 CFR Part 89. Enforcement date: 16 March 2024. [Online]. Available: https://www.govinfo.gov/content/pkg/FR-2021-01-15/pdf/2021-00100.pdf

[2] European Commission, “Commission delegated regulation (EU) 2019/945 on unmanned aircraft systems and on third-country operators of unmanned aircraft systems,” Official Journal of the European Union, Tech. Rep. L 152, 2019, as amended by Commission Delegated Regulation (EU) 2020/1058. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32019R0945

[3] ——, “Commission implementing regulation (EU) 2019/947 on the rules and procedures for the operation of unmanned aircraft,” Official Journal of the European Union, Tech. Rep. L 152, 2019. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32019R0947

[4] ASTM International, “ASTM F3411-22a: Standard specification for remote ID and tracking,” ASTM International, West Conshohocken, PA, Tech. Rep., 2022. [Online]. Available: https://www.astm.org/f3411-22a.html

[5] ASD-STAN, “EN 4709-002: Aerospace series — unmanned aircraft systems (UAS) — direct remote identification,” AeroSpace and Defence Standards Standardization, Tech. Rep., 2021. [Online]. Available: https://www.asd-stan.org/downloads/en-4709-002/

[6] OpenDroneID contributors, “OpenDroneID: Android receiver reference implementation,” GitHub, 2024. [Online]. Available: https://github.com/opendroneid/receiver-android

[7] Dronetag, “Drone scanner: Open Drone ID receiver application for Android,” GitHub / Google Play Store, 2024. [Online]. Available: https://github.com/dronetag/drone-scanner

[8] Wikipedia contributors, “Swedish sea rescue society — Wikipedia,” Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Swedish_Sea_Rescue_Society

[9] T. Lindhe, “SSRS becomes the first Swedish civil operator to be granted a national permit for Beyond Visual Line of Sight (BVLOS) drone operations,” SSRS press release, 2022. [Online]. Available: https://www.sjoradd.se/

[10] OpenDroneID contributors, “opendroneid-core-c: Open Drone ID encoding and decoding library for C,” GitHub, 2024. [Online]. Available: https://github.com/opendroneid/opendroneid-core-c

[11] ——, “transmitter-linux: Open Drone ID transmitter reference implementation for Linux,” GitHub, 2024. [Online]. Available: https://github.com/opendroneid/transmitter-linux

[12] Dronetag, “Dronetag mini: Bluetooth and Wi-Fi Remote ID module,” dronetag.cz, 2024. [Online]. Available: https://dronetag.cz/en/products/mini/

[13] BlueMark Innovations, “DB150 dronebeacon: Remote ID module,” bluemark.aero, 2024. [Online]. Available: https://bluemark.aero/dronebeacon/

[14] Spektrum RC, “Spektrum sky remote ID module,” spektrumrc.com, 2023. [Online]. Available: https://www.spektrumrc.com/Products/Default.aspx?ProdID=SPMXCA350

[15] ArduPilot developers, “ArduPilot Remote ID documentation — DID_ENABLE parameter,” ardupilot.org, 2024. [Online]. Available: https://ardupilot.org/copter/docs/common-remoteid.html

[16] S. Sciancalepore and R. D. Pietro, “Unmanned aerial vehicles’ remote identification: A tutorial and survey,” *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2071–2116, 2022.

[17] E. Wisse, P. Tedeschi, S. Sciancalepore, and R. D. Pietro, “A^{2}RID: Anonymous direct authentication and remote identification of commercial drones,” *IEEE Internet of Things Journal*, vol. 10, no. 12, pp. 10 587–10 604, 2023.

[18] E. Wisse, S. Sciancalepore, and R. D. Pietro, “TBRD: TESLA authenticated UAS broadcast remote ID,” arXiv preprint arXiv:2510.11343, 2025. [Online]. Available: https://arxiv.org/abs/2510.11343

[19] S. Card, A. Wiethuechter, R. Moskowitz, S. Zhao, and A. Gurtov, “Drone remote identification protocol (DRIP) architecture,” Internet Engineering Task Force, Informational RFC RFC 9434, 2023. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9434

[20] Transportstyrelsen, “Registrering av drönare — Swedish drone operator registration,” Swedish Transport Agency, Tech. Rep., 2021. [Online]. Available: https://www.transportstyrelsen.se/en/aviation/drones/register-your-drone/

[21] Bluetooth SIG, “Bluetooth core specification, version 5.3,” Bluetooth Special Interest Group, Tech. Rep., 2021. [Online]. Available: https://www.bluetooth.com/specifications/specs/core-specification-5-3/

[22] ——, “Assigned numbers,” Bluetooth Special Interest Group, Tech. Rep., 2024. [Online]. Available: https://www.bluetooth.com/specifications/assigned-numbers/

[23] IEEE, “IEEE 802.11-2020: IEEE standard for information technology — telecommunications and information exchange between systems — local and metropolitan area networks — specific requirements, part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications,”

Institute of Electrical and Electronics Engineers, Tech. Rep., 2021. [Online]. Available: https://standards.ieee.org/ieee/802.11/7028/

[24] BlueZ contributors, “BlueZ: Official Linux bluetooth protocol stack,” kernel.org git, 2024. [Online]. Available: https://git.kernel.org/pub/scm/bluetooth/bluez.git

[25] Linux wireless developers, “nl80211 — Linux wireless LAN configuration API,” kernel.org documentation, 2024. [Online]. Available: https://wireless.wiki.kernel.org/en/developers/Documentation/nl80211

[26] J. Malinen, “hostapd 2.10 release notes,” w1.fi, 2022. [Online]. Available: https://w1.fi/cgit/hostap/plain/hostapd/ChangeLog

[27] MAVLink project, “MAVLink micro air vehicle communication protocol, version 2,” mavlink.io, 2024. [Online]. Available: https://mavlink.io/en/

[28] MAVLink project, “MAVLink message definitions: OPEN_DRONE_ID_* messages (ids 12900–12915),” mavlink.io, 2024. [Online]. Available: https://mavlink.io/en/messages/common.html#OPEN_DRONE_ID_BASIC_ID

[29] B. Hoyt, “inih: Simple INI file parser in C,” GitHub, 2023. [Online]. Available: https://github.com/benhoyt/inih

A

Production Configuration and Deployment Files

This appendix reproduces the configuration and deployment files for both the Docker Compose production deployment on the SSRS CM4 and the systemd bench deployment on the Raspberry Pi 400. All identifiers are real production values; the operator registration `SWE3jxza7qbzvu5d` is a valid Transportstyrelsen registration. The UAS ID `SSRSGBG001` is a self-assigned CTA-2063-A identifier and must be replaced with an officially registered number before routine Specific-category operations.

A.1 Docker Compose Deployment

The recommended deployment for the SSRS CM4 is a Docker Compose service with UDP MAVLink input and Wi-Fi disabled. Identity fields are passed as environment variables so that no configuration file is needed inside the container.

```
services:
  odid:
    build: .
    restart: unless-stopped
    network_mode: host
    cap_add:
      - NET_RAW
      - NET_ADMIN
    environment:
      - ODID_UAS_ID=SSRSGBG001
      - ODID_OPERATOR_ID=SWE3jxza7qbzvu5d
      - ODID_SELF_ID=Sea rescue surveillance
      - ODID_MAVLINK_SOURCE=udp
      - ODID_UDP_PORT=14550
      - ODID_NO_WIFI=1
```

To add the daemon as a consumer in `mavlink-router`:

```
[UdpEndpoint odid-daemon]
Mode = Normal
Address = 127.0.0.1
Port = 14550
```

A.2 /etc/odid/odid.conf (Bench / Serial deployment)

The INI configuration file used on the Raspberry Pi 400 bench unit with a direct serial connection to the flight controller.

```
; Open Drone ID Daemon -- SSRS Sea Rescue Drone
[basic_id]
uas_id = "SSRSGBG001"
id_type = 1 ; 1 = SERIAL_NUMBER (CTA-2063-A)
ua_type = 1 ; 1 = AEROPLANE (fixed wing)

[operator]
operator_id = "SWE3jxza7qbzvu5d"
operator_id_type = 0 ; 0 = CAA Assigned

[self_id]
description = "Sea rescue surveillance"
description_type = 0 ; 0 = plain text

[system]
operator_location_type = 0 ; 0 = Takeoff location
classification_type = 1 ; 1 = EU classification
category = 3 ; 3 = Specific
class = 4 ; 4 = C4

[mavlink]
source = serial
device = "/dev/ttyACM0"
baud = 921600

[broadcast]
bt4_enabled = true
bt5_enabled = true ; auto-disabled if hardware unsupported
wifi_beacon_enabled = true
wifi_iface = "wlan0"
bt_adapter = "hci0"
```

A.3 Environment Variables Reference

All environment variables override the corresponding INI file settings.

```
ODID_UAS_ID - UAS serial number (CTA-2063-A format)
ODID_OPERATOR_ID - EU operator registration string
ODID_SELF_ID - Human-readable description (max 23 chars)
ODID_MAVLINK_SOURCE - "serial" or "udp"
ODID_UDP_PORT - UDP port when source=udp (default:  14550)
ODID_NO_WIFI - "1" disables Wi-Fi Beacon
ODID_NO_BT - "1" disables BT4 advertising
ODID_DEBUG - "1" enables verbose per-message logging
```

A.4 /etc/systemd/system/odid.service (Bench deployment)

The systemd unit file for the Raspberry Pi 400 bench unit.

```
[Unit]
Description=Open Drone ID Broadcaster
After=hostapd.service network.target
Wants=hostapd.service

[Service]
Type=simple
ExecStart=/usr/local/bin/odid-daemon -config /etc/odid/odid.conf
Restart=on-failure
RestartSec=5
User=root
AmbientCapabilities=CAP_NET_RAW CAP_NET_ADMIN
CapabilityBoundingSet=CAP_NET_RAW CAP_NET_ADMIN
NoNewPrivileges=yes
PrivateTmp=yes
ProtectSystem=strict
ReadWritePaths=/tmp /var/log
StandardOutput=journal
StandardError=journal
SyslogIdentifier=odid-daemon

[Install]
WantedBy=multi-user.target
```

A.5 /etc/hostapd/hostapd.conf (Bench deployment)

Required only when Wi-Fi Beacon is enabled (bench unit). In the production Docker deployment this file is not needed.

```
interface=wlan0
driver=nl80211
ssid=ODID-Drone
hw_mode=g
channel=6
wmm_enabled=1
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
```

A.6 Build and Installation (Bench)

```
# Install build dependencies
sudo apt install -y git cmake build-essential \
    libbluetooth-dev libnl-3-dev libnl-genl-3-dev hostapd

# Clone repository with submodules
git clone -recurse-submodules https://github.com/Revacholi/opendroneid
cd opendroneid

# Build
mkdir build && cd build && cmake ..  && make -j4
ctest # expected:  2/2 passed

# Install
sudo cp odid-daemon /usr/local/bin/
sudo mkdir -p /etc/odid
sudo cp ../odid.conf.example /etc/odid/odid.conf
sudo systemctl disable bluetooth.service
sudo systemctl enable hostapd.service odid.service
sudo systemctl start odid.service
```

B

Development Log and Non-Obvious Bugs

Four bugs dominate the development history of this project. They are reported here in the order they were found, because each motivates a design decision that would otherwise look unjustified.

Bug 1: the missing Bluetooth counter byte

For three days during the reference phase, the daemon produced a syntactically correct BT4 advertisement that was clearly visible in `btmon` and in nRF Connect, and yet was invisible to Drone Scanner. The *Service Data* blob carried UUID `0xFFFA`, application code `0x0D`, and 25 plausible-looking bytes, and no error was logged anywhere. The fault was eventually traced by reading the source of the Android DroneScanner project [7]: its `receiveDataBluetooth()` method calls `OpenDroneIdParser.parseData(data, 6)` with the offset `6` hard-coded. The offset exists because byte 5 is supposed to be a rolling message counter, a convention inherited from ASTM F3411-22a that is not obvious in the current EN 4709-002 text and which the OpenDroneID reference encoder supplies implicitly. Inserting the counter at offset 5 and leaving the 25-byte payload starting at offset 6 made the drone appear in Drone Scanner immediately. The fix is a single line in `bt4_build_adv_data()`:

```
adv[idx++] = s_bt4_counter++;
```

Bug 2: set vendor_elements without update_beacon

A parallel problem affected the Wi-Fi path: hostapd reported `OK` to every `hostapd_cli set vendor_elements` call, and yet a Wireshark monitor-mode capture of the Beacon frames on the air showed no vendor IE at all. The hostapd 2.10 changelog resolved the mystery: `set vendor_elements` updates only hostapd's configuration snapshot; a separate `update_beacon` call is required to rebuild the Beacon and re-arm it in the radio firmware. A single extra `system()` call per broadcast fixed the problem, and is the reason the production path does *two* shell invocations rather than one.

Bug 3: BlueZ races against raw HCI

After the daemon had been running reliably for thirty minutes to an hour on a soak-test bench, the on-air advertising rate as seen by an external BT scanner dropped from approximately 10 Hz to once every eight seconds. The daemon log was silent — `LE_SET_ADVERTISING_DATA` kept returning `0x00` — and yet the advertisement was simply not being transmitted at the contracted interval. The cause was `bluetoothd`: the BlueZ main daemon, still running in parallel, periodically uses `hci0` for its own GAP advertising and discovery operations, and in doing so issues `LE_SET_ADVERTISE_ENABLE(0)` against a controller the daemon believes is its own. Because the daemon updates only the payload and not the enable flag on every broadcast, the advertising remained disabled until BlueZ happened to re-enable it on its own schedule. The clean fix is the one the production system uses: `systemctl disable bluetooth.service`.

Bug 4: hci0 is DOWN at boot without BlueZ

Disabling BlueZ introduced a boot-order bug. On Raspberry Pi OS, BlueZ is the process that normally runs `hciconfig hci0 up` to power the controller after the firmware is loaded; without BlueZ, the kernel leaves `hci0` in the DOWN state, and the daemon's HCI socket initialisation fails with `ENODEV` ("No such device"). In addition, on Debian Trixie and Raspberry Pi OS Bookworm, the Bluetooth adapter is soft-blocked by `rfkill` by default; `hciconfig hci0 up` alone does not unblock it, so the `HCIDEVUP` ioctl also fails. The fix is two-step: first `rfkill unblock bluetooth`, then the `HCIDEVUP` ioctl. In the Docker deployment this is handled automatically by the entrypoint script using the `CAP_NET_ADMIN` capability. In the bench systemd deployment, `rfkill unblock bluetooth` must be run once manually; after that it is persistent across reboots.

C

Receiver Screenshots

This appendix collects the Drone Scanner screenshots that serve as photographic evidence of end-to-end Remote ID reception. They complement the tabular results in Chapter 5 by showing the rendered application view that a field observer would actually see.

C.1 Bench Test: BT4 Reception (Indoor)

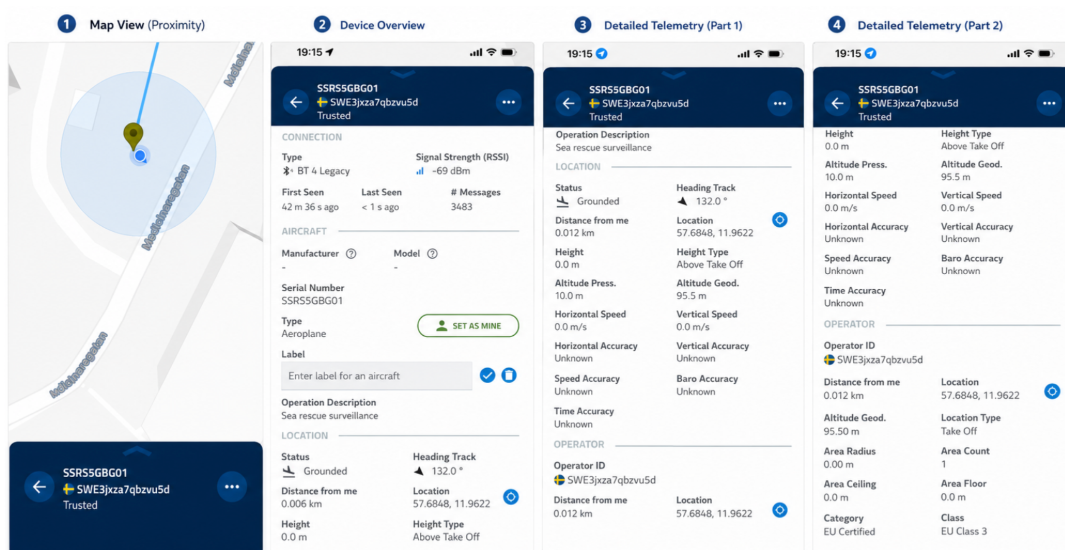

Figure C.1: Drone Scanner device detail view from the indoor bench test. All mandatory EN 4709-002 fields are decoded correctly from BT4 Legacy Advertising.

C.2 Flight Test: BT4 Reception (Outdoor, Live GPS)

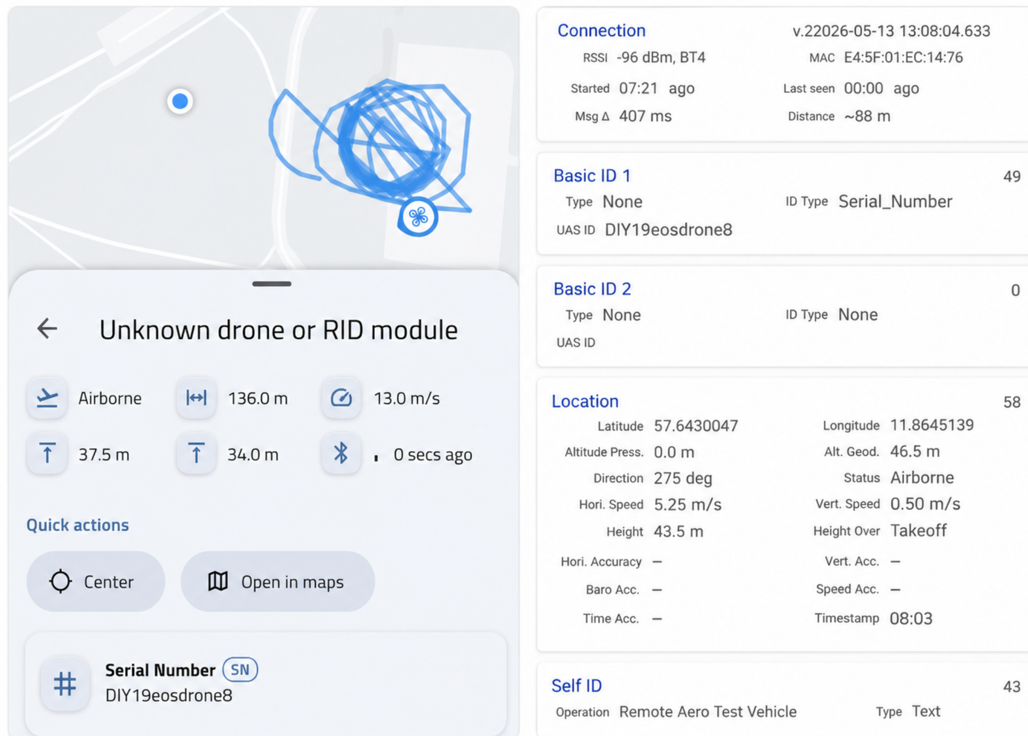

Figure C.2: Drone Scanner device detail view from the SSRS flight test (maximum 115 m). `SSRSGBG001` is the author's test hardware unit; the SSRS test aircraft broadcasts as `DIY19ossdrone8`. Björn Bringert (Remote.aero) confirmed reception on a Google Pixel 8 smartphone with live GPS coordinates visible. This is the primary end-to-end validation of the production deployment.