PS GmbH
Melisau 1255
6863 Egg / Austria

Table of Contents Version 06.07.2026

- Configuration Options

6. Device Discovery

- Automatic Lock Discovery
- Detecting New Devices
- Reading Device UID
- Discovery Events

7. Device Commissioning

- Assigning Device IDs
- Commissioning Workflow
- Configuration Storage
- Verification

8. Lock Control

- Opening a Lock
- Closing a Lock
- Permanent Unlock
- Lock Status
- Device Information

9. Whitelist Management

- Adding RFID Cards
- Removing RFID Cards
- Reading the Whitelist
- Synchronization

10. Event Handling

- Event System
- Lock Events
- RFID Events
- System Events
- Callback Functions

11. Health Monitoring

- Heartbeat Monitoring
- Device Status
- Online / Offline Detection
- Diagnostics

12. API Reference

- Controller Class
- Lock Class

- Event Classes
- Enumerations
- Utility Functions

13. Code Examples

- Quick Start
- Lock Discovery
- Commissioning Example
- Lock Control Example
- Event Handling Example

14. CAN Protocol Reference

- Message Definitions
- CAN Frames
- Command Reference
- Response Reference

15. Appendix

- Error Codes
- Troubleshooting
- Glossary
- Version History

PS GmbH
Melisau 1255
6863 Egg / Austria

0. Purpose and Scope

Purpose

The PS Locks OIP SDK provides a standardized software interface for integrating PS Locks electronic locking systems into third-party applications running on a Raspberry Pi.
The SDK abstracts the low-level CAN communication and offers a stable, object-oriented programming interface for commissioning, monitoring and controlling one or more PS Locks devices.

Scope

This document describes the public SDK API.
It includes

- installation
- controller configuration
- lock discovery
- automatic device detection
- lock commissioning
- lock control
- whitelist management
- health monitoring
- event handling
- CAN protocol reference

Target Audience

This document is intended for

- software developers
- system integrators
- OEM partners
- automation engineers

who want to integrate PS Locks into their own applications.

Supported Platform

- Raspberry Pi 4
- Raspberry Pi 5
- Linux
- SocketCAN
- Python 3.11+

Communication

The SDK communicates with PS Locks controllers via CAN 2.0A using standard 11-bit identifiers.
All low-level CAN communication is encapsulated by the SDK.
Applications interact exclusively with the public SDK API.

New Features in Version 2.0

Compared to previous SDK versions, Version 2.0 introduces:

- Automatic lock discovery
- Detection of uncommissioned locks
- Lock identification by UID
- Automatic and manual lock ID assignment
- Improved heartbeat handling
- Simplified Raspberry Pi integration
- Updated Python SDK
- Revised API documentation

PS GmbH
Melisau 1255
6863 Egg / Austria

Out of Scope

The following topics are intentionally excluded from this document:

- internal firmware implementation
- experimental protocol extensions
- deprecated commands
- legacy protocol revisions
- manufacturing procedures

PS GmbH
Melisau 1255
6863 Egg / Austria

1. Physical Layer

1.1 Overview

The PS Locks OIP SDK communicates with PS Locks controllers using a standard CAN 2.0A bus with 11-bit identifiers.
All CAN communication is handled internally by the SDK. Customer applications communicate exclusively through the public SDK API.

1.2 Supported Hardware

Component	Description
Host System	Raspberry Pi 4 / Raspberry Pi 5
Operating System	Raspberry Pi OS (64-bit recommended)
CAN Interface	SocketCAN compatible adapter
Bus Type	CAN 2.0A (11-bit Identifier)
Physical Layer	ISO 11898

1.3 Supported Bit Rates

The default communication speed for new devices is **50 kbit/s**.

1.4 CAN Bus Requirements

The CAN bus shall comply with ISO 11898.
Requirements:

- 120 Ω termination resistor at both bus ends
- Twisted pair wiring
- Common ground recommended
- Maximum cable length depends on bus speed

1.5 SocketCAN

The Raspberry Pi SDK uses the Linux SocketCAN subsystem.
Typical interface:
`can0`
Example:

```
sudo ip link set can0 up type can bitrate 50000
```

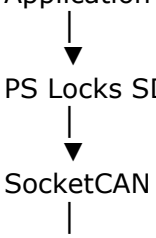
1.6 SDK Abstraction

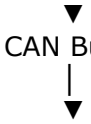
The SDK completely abstracts the CAN interface.
Applications do **not** need to:

- create CAN frames
- calculate identifiers
- parse payloads
- handle retries
- manage acknowledgements

These functions are implemented internally by the SDK.

1.7 Communication Model





1.8 Notes

The SDK automatically validates communication errors, manages retries where applicable, and provides a consistent API regardless of the underlying CAN hardware.

PS GmbH
Melisau 1255
6863 Egg / Austria

2. CAN Identifier Layout

2.1 Overview

The PS Locks OIP protocol uses **11-bit standard CAN identifiers (CAN 2.0A)**.
Each CAN identifier consists of two logical parts:

- **Message Type**
- **Device ID**

The combination of these fields uniquely identifies the destination or source of every message transmitted on the CAN bus.

2.2 CAN Identifier Format

Bits	Description
10..8	Message Type
7..0	Device ID

The CAN identifier is generated using the following formula:
`CAN Identifier = Base Identifier | Device ID`

2.3 Base Identifiers

Message Type	Base Identifier	Direction
COMMAND	0x100	Controller → Lock
STATUS	0x200	Lock → Controller
HEALTH	0x300	Lock → Controller
MASTER RESPONSE	0x400	Controller → Lock
ID REQUEST	0x500	Lock → Controller
UID PART 1	0x600	Lock → Controller
UID PART 2	0x700	Lock → Controller

2.4 Device ID

Each lock is assigned a unique **Device ID**.
Valid IDs:

Value	Description
0x01 – 0x7E	Assigned device IDs
0x7F	Uncommissioned device (Default ID)

A lock operating with the default ID (**0x7F**) has not yet been commissioned and cannot be addressed individually.

2.5 Uncommissioned Devices

New locks always start with the **Default Device ID (0x7F)**.
During commissioning, the controller:

1. Detects the unknown lock.
2. Reads the device UID.
3. Assigns a unique Device ID.
4. Stores the configuration permanently.
5. Activates the lock for normal operation.

2.6 Example

A status message from **Lock 5** is transmitted using:

```
Base Identifier : 0x200
Device ID       : 0x05
```

```
------------------------------
CAN Identifier  : 0x205
```

2.7 Identifier Ranges

Range	Usage
0x100 – 0x17F	COMMAND
0x200 – 0x27F	STATUS
0x300 – 0x37F	HEALTH
0x400 – 0x47F	MASTER RESPONSE
0x500 – 0x57F	ID REQUEST
0x600 – 0x67F	UID PART 1
0x700 – 0x77F	UID PART 2

2.8 Notes

- The SDK generates all CAN identifiers automatically.
- Applications never need to calculate CAN identifiers manually.
- Device IDs remain persistent after successful commissioning.
- The SDK validates all received CAN identifiers before processing incoming messages.

PS GmbH
Melisau 1255
6863 Egg / Austria

3. General Conventions

3.1 Overview

The PS Locks OIP SDK uses a consistent communication model for all CAN messages.
Each command, response and event follows a standardized frame structure to simplify integration and ensure predictable behavior.

3.2 Device ID

Every lock connected to the CAN bus is identified by its **Device ID**.
The Device ID is used as part of the CAN identifier and uniquely addresses a single lock.
Device IDs are assigned during the commissioning process and stored permanently inside the lock.

3.3 Lock ID

Some controllers manage multiple physical locks.
Within a device, each lock is identified by its **Lock ID**.

Lock ID	Description
1	First lock
2	Second lock
...	Additional locks

The combination of **Device ID** and **Lock ID** uniquely identifies a physical lock.

3.4 Correlation ID

The SDK uses a **Correlation ID (CorrID)** to associate responses with previously transmitted commands.
Each command generated by the controller receives a unique Correlation ID.
The responding device returns the same value in its acknowledgement.
Example:

```
Controller                Lock

OPEN (CorrID = 42)
------------------------>

                ACK (CorrID = 42)
<------------------------
```

Applications normally do not need to manage Correlation IDs manually.

3.5 Byte Order

Multi-byte values are transmitted using **Big Endian (Network Byte Order)**.
Example

```
Value : 0x1234

Byte 0 = 0x12
Byte 1 = 0x34
```

3.6 Message Direction

Communication always follows one of the following directions.

Direction	Description
Controller → Lock	Commands
Lock → Controller	Status information
Lock → Controller	Health information
Lock → Controller	Events

Direction	Description
Controller → Lock	Device configuration

3.7 Message Acknowledgement

Every command sent by the controller is acknowledged by the addressed device.
The acknowledgement confirms that the command has been received and processed.
If no acknowledgement is received within the configured timeout, the SDK automatically reports a communication timeout.

3.8 Timeouts

The SDK manages communication timeouts internally.
Typical default values:

Function	Timeout
Command Acknowledgement	300 ms
Discovery	Configurable
Health Request	Configurable
Heartbeat Monitoring	Configurable

Applications may override these values if required.

3.9 Error Handling

Communication errors are reported as SDK exceptions or events.
Typical errors include:

- Communication timeout
- Unknown device
- Invalid device ID
- Invalid command
- CAN bus unavailable
- Device offline

3.10 SDK Responsibilities

The SDK automatically handles:

- CAN frame creation
- CAN identifier generation
- Message parsing
- Correlation IDs
- Timeout monitoring
- Retry handling (where applicable)
- Device state management

Applications communicate exclusively with the public SDK API and are not required to process raw CAN frames.

PS GmbH
Melisau 1255
6863 Egg / Austria

4. COMMAND Messages

4.1 Overview

COMMAND messages are transmitted from the controller to one or more locks.
They are used to control lock operation, retrieve information, configure devices and manage the commissioning process.
The SDK automatically creates and transmits all COMMAND messages.
Applications call high-level SDK methods and never generate CAN frames directly.

4.2 Message Format

All COMMAND messages use the following payload structure.

Byte	Description
0	Lock ID
1	Command
2	Correlation ID
3..7	Command specific data

The payload length depends on the selected command.

4.3 Standard Commands

OPEN_LOCK
Unlocks the selected lock.

Command	Value
OPEN_LOCK	0x01

Example
`lock.open()`
Expected Response

- COMMAND_ACK
- STATUS update

CLOSE_LOCK
Locks the selected lock.

Command	Value
CLOSE_LOCK	0x02

Example
`lock.close()`
Expected Response

- COMMAND_ACK
- STATUS update

REQUEST_STATUS
Requests the current lock status.

Command	Value
REQUEST_STATUS	0x03

Example
`status = lock.status()`
Expected Response

- STATUS

OPEN_HOLD
Keeps the lock permanently unlocked until released.

Command	Value
OPEN_HOLD	0x04

Example
`lock.hold_open()`

OPEN_RESET
Cancels permanent unlock mode.

Command	Value
OPEN_RESET	0x05

Example
`lock.release()`

4.4 Device Management Commands

REQUEST_VERSION
Reads firmware information.
`version = lock.version()`

REQUEST_HEALTH
Requests health information.
`health = lock.health()`

DEVICE_RESET
Performs a software reset.
`lock.reset()`

FACTORY_RESET
Restores factory settings.
`lock.factory_reset()`
Warning
A factory reset removes the assigned Device ID and returns the device to commissioning mode.

4.5 Discovery Commands

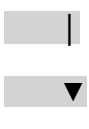
The SDK automatically performs all commissioning commands.
Applications normally do not send these commands manually.
Supported operations include:

- Detect unknown devices
- Read device UID
- Assign Device ID
- Verify commissioning
- Activate device

Example
`controller.discover()`
or
`controller.assign_id(device)`

4.6 Command Execution

Every command follows the same execution sequence.



SDK

|

COMMAND

|

▼

Lock

|

COMMAND_ACK

|

▼

STATUS

|

▼

Application

4.7 Timeouts

If no response is received within the configured timeout, the SDK reports a communication error.
The SDK may automatically retry selected commands where appropriate.
Applications are informed through exceptions or event callbacks.

4.8 Best Practice

Applications should always use the SDK methods instead of transmitting raw COMMAND messages.
Example
✓ Recommended
lock.open()
Not recommended
Build raw CAN frame manually
Transmit frame
Wait for ACK
Parse STATUS
The SDK performs these operations automatically.

PS GmbH
Melisau 1255
6863 Egg / Austria

5. STATUS Messages

5.1 Overview

STATUS messages are transmitted from a lock to the controller.
They provide real-time information about the current operating state of a lock and confirm the successful execution of controller commands.
STATUS messages are generated automatically by the lock and processed internally by the SDK.
Applications receive the decoded information through the public SDK API.

5.2 Message Types

Status Message	Description
BASIC	Current lock status
COMMAND_ACK	Command acknowledgement
RFID_EVENT	RFID access event
WHITELIST	Whitelist information
DEVICE_INFO	Device information
ERROR	Error condition

5.3 BASIC Status

The BASIC status reports the current operating state of a lock.

Byte	Description
0	Status Type
1	Lock ID
2	Lock State
3	Error Code

Example

```
status = lock.status()

print(status.state)
print(status.error)
```

5.4 Lock States

Value	Description
1	Locked
2	Unlocked
3	Door Open
4	Door Closed
5	Opening
6	Hold Open

The SDK converts these numeric values into readable status objects.

5.5 COMMAND_ACK

Every successfully received command generates a COMMAND_ACK message.
Applications normally do not process acknowledgements directly.
The SDK automatically validates acknowledgements and matches them with the corresponding command.
Example

```
Controller
     |
```

```
OPEN
  |
  ▼
Lock
  |
COMMAND_ACK
  |
STATUS
  ▼
SDK
```

5.6 RFID Events

Whenever an RFID card is presented, the lock generates an RFID event.
Possible events include:

- Card detected
- Local access granted
- Local access denied
- Remote authorization required
- Access completed

Applications may subscribe to these events.
Example

```
controller.on_rfid(callback)
```

5.7 Whitelist Status

Whitelist operations generate status messages indicating whether an operation has completed successfully.
Typical events include:

- Entry added
- Entry removed
- Whitelist cleared
- Whitelist synchronized

Example

```
lock.whitelist.add(uid)
```

The SDK automatically waits for the corresponding acknowledgement.

5.8 Device Information

The lock reports device information on request.
Available information includes:

- Firmware version
- Hardware revision
- Manufacturer ID
- Device type
- Supported capabilities

Example

```
info = lock.device_info()
```

5.9 Error Messages

When a command cannot be executed, the lock returns an ERROR status.
Typical errors include:

Error	Description
Invalid Command	Command not supported
Invalid Parameter	Parameter out of range

Error	Description
Communication Error	CAN communication failure
Device Busy	Device currently unavailable
Timeout	Operation timed out

The SDK converts protocol errors into descriptive exceptions.
Example

```
try:
    lock.open()
except LockTimeoutError:
    print("Communication timeout")
```

5.10 Automatic Status Updates
The SDK continuously monitors all connected locks.
Whenever the state of a lock changes, the corresponding Lock object is updated automatically.
Applications always access the latest known state.
Example

```
if lock.is_open:
    print("Lock is open")
```

5.11 Event Notifications
Applications can subscribe to status changes.
Typical events include:

```
controller.on_lock_changed(callback)

controller.on_lock_online(callback)

controller.on_lock_offline(callback)

controller.on_error(callback)
```

The SDK dispatches these events asynchronously whenever the corresponding STATUS message is received.

5.12 Best Practice
Applications should use the SDK status objects instead of decoding STATUS messages manually.
✓ Recommended

```
status = lock.status()

if status.locked:
    ...
```

✗ Not Recommended

```
Receive raw CAN frame

Decode payload

Interpret lock state

Handle acknowledgements manually
```

The SDK performs these operations automatically.

PS GmbH
Melisau 1255
6863 Egg / Austria

6. HEALTH Messages

6.1 Overview

HEALTH messages provide diagnostic information about the operational condition of a device. Unlike STATUS messages, which report the current state of a lock, HEALTH messages provide information about the device itself, including firmware, hardware, operating conditions and diagnostic data.
Applications typically request health information during commissioning, maintenance or troubleshooting.

6.2 Available Health Information

The SDK provides access to the following diagnostic information.

Information	Description
Firmware Version	Installed firmware version
Hardware Revision	Hardware revision
Device Type	Device model
Manufacturer ID	Manufacturer identifier
Supply Voltage	Device supply voltage
Temperature	Internal device temperature
Uptime	Device runtime
Free Memory	Available RAM
Diagnostics	Internal diagnostic information
Capabilities	Supported device features

6.3 Reading Health Information

Applications can request the current health information of any connected device.
Example

```
health = lock.health()
```

6.4 Firmware Information

The SDK returns firmware information as part of the health object.
Example

```
print(health.firmware.major)
print(health.firmware.minor)
print(health.firmware.patch)
```

Example Output

```
Firmware Version : 2.1.5
```

6.5 Hardware Information

Hardware information identifies the physical device.
Example

```
print(health.hardware_revision)
print(health.device_type)
print(health.manufacturer)
```

6.6 Electrical Information

The SDK provides access to measured operating values.
Example

```
print(health.voltage)
print(health.temperature)
```

Typical Output

```
Voltage      : 3.1 V
Temperature  : 31 °C
```

6.7 Device Runtime

The runtime indicates how long the device has been operating since the last restart.
Example

```
print(health.uptime)
```

Example Output

```
4 days 13 hours
```

6.8 Device Capabilities

Each device reports the features supported by its firmware.
Example

```
caps = lock.capabilities()

if caps.rfid:
    print("RFID supported")

if caps.whitelist:
    print("Whitelist supported")

if caps.door_sensor:
    print("Door sensor available")
```

Typical capabilities include:

- RFID Reader
- Whitelist Management
- Door Sensor
- Lock Position Sensor
- LED Indicator
- Buzzer
- Health Monitoring
- Firmware Version Reporting

6.9 Diagnostics

Diagnostic information is primarily intended for maintenance and troubleshooting.
Example

```
diag = lock.diagnostics()

print(diag.reset_reason)
print(diag.can_tx_errors)
print(diag.can_rx_errors)
```

6.10 Automatic Health Monitoring

The SDK continuously monitors connected devices.
Health information is updated automatically whenever a new HEALTH message is received.
Applications always access the latest available information.
Example

```
if health.voltage < 2.8:
    print("Warning: Low supply voltage")
```

6.11 Health Events

Applications may subscribe to health-related events.
Example

```
controller.on_health_changed(callback)

controller.on_device_warning(callback)

controller.on_device_error(callback)
```

These events are generated automatically whenever significant diagnostic changes occur.

6.12 Best Practice

Health information should be used for diagnostics and preventive maintenance rather than normal lock operation.

✓ Recommended

```
health = lock.health()

if health.temperature > 70:
    print("Device temperature warning")
```

✗ Not Recommended

```
Request health information continuously
before every lock operation.
```

Normal lock operation should rely on STATUS messages. HEALTH information should be requested periodically or when diagnostic information is required.

PS GmbH
Melisau 1255
6863 Egg / Austria

7. Device Discovery & Commissioning

7.1 Overview

The PS Locks OIP SDK automatically detects locks connected to the CAN bus.
During the discovery process, the SDK identifies both commissioned and uncommissioned devices, retrieves the required identification information and prepares new devices for operation.
The complete commissioning process is managed by the SDK and normally requires only a few API calls.

7.2 Discovery Process

The discovery process consists of the following steps:

1. Scan the CAN bus.
2. Detect commissioned devices.
3. Detect uncommissioned devices.
4. Read the device UID.
5. Assign a new Device ID (if required).
6. Verify communication.
7. Register the device.
8. Make the lock available to the application.

7.3 Discovering Locks

Applications start the discovery process by calling:
`controller.discover()`
The SDK returns a collection of all detected locks.
Example

```
controller.start()

locks = controller.discover()

print(f"{len(locks)} lock(s) found")
```

7.4 Detecting New Devices

New locks are automatically recognized by their default Device ID.
These devices are reported as **uncommissioned** until a permanent Device ID has been assigned.
Example

```
unknown = controller.find_uncommissioned()

for device in unknown:
    print(device.uid)
```

7.5 Reading the Device UID

Every lock contains a unique hardware identifier (UID).
The SDK automatically retrieves this identifier during commissioning.
Example

```
uid = device.uid

print(uid)
```

Example Output
`9F-3A-74-91-11-58-AC-04-21-6B-7E-10`

7.6 Assigning a Device ID

After successful detection, the application assigns a permanent Device ID.
Example

```
controller.assign_id(
    device=device,
    device_id=5
)
```
After assignment, the SDK verifies that the device responds using its new Device ID.

7.7 Automatic ID Assignment
The SDK also supports automatic commissioning.
Example
```
controller.assign_next_id(device)
```
The next available Device ID is assigned automatically.
This function is recommended during installation and production testing.

7.8 Discovery Result
Each discovered lock is represented by a **Lock** object.
Example
```
lock = controller.get_lock(5)

print(lock.device_id)
print(lock.online)
```

7.9 Device States
A device may be in one of the following states.

State	Description
Uncommissioned	Default Device ID
Commissioning	ID assignment in progress
Online	Device ready for operation
Offline	Device not responding
Error	Communication error detected

Applications can query the current state at any time.

7.10 Rediscovery
The SDK supports rescanning the CAN bus without restarting the controller.
Example
```
controller.rediscover()
```
Existing locks remain available while new devices are detected automatically.

7.11 Device Replacement
When replacing a physical lock, the SDK supports assigning the previous Device ID to the new hardware.
Typical procedure:
1. Install replacement lock.
2. Detect the new device.
3. Read its UID.
4. Assign the original Device ID.
5. Verify communication.
6. Resume normal operation.

7.12 Discovery Events
Applications may subscribe to discovery events.
Example

```
controller.on_lock_found(callback)

controller.on_lock_removed(callback)

controller.on_new_device(callback)

controller.on_commissioned(callback)
```
These events are generated automatically during device discovery.

7.13 Best Practice

Perform device discovery during application startup and whenever new hardware is installed.

✓ Recommended

```
controller.start()

controller.discover()
```
Avoid assigning Device IDs manually unless a fixed addressing scheme is required.
The automatic commissioning functions simplify installation and reduce the risk of duplicate Device IDs.

PS GmbH
Melisau 1255
6863 Egg / Austria

8. Lock Control API

8.1 Overview

The Lock API provides all functions required to monitor and control a physical lock.
Each detected lock is represented by a **Lock** object created by the Controller during the discovery process.
Applications should always interact with locks through the Lock object instead of sending CAN commands directly.

8.2 Obtaining a Lock Object

A lock can be accessed by its Device ID.

```
lock = controller.get_lock(5)
```

or

```
for lock in controller.locks:
    print(lock.device_id)
```

8.3 Opening a Lock

Unlock the selected lock.

```
lock.open()
```

The SDK performs the following operations automatically:

- Build COMMAND message
- Send CAN frame
- Wait for acknowledgement
- Validate response
- Update lock status

Returns

```
True
```

or raises an exception.

8.4 Closing a Lock

Lock the selected lock.

```
lock.close()
```

The SDK automatically waits until the operation has completed successfully.

8.5 Hold Open

Keeps the lock permanently unlocked.

```
lock.hold_open()
```

The lock remains unlocked until

```
lock.release()
```

is executed.

8.6 Reading Lock Status

Returns the current lock status.

```
status = lock.status()
```

Example

```
print(status.state)
print(status.door_open)
print(status.error)
```

8.7 Door Status

If supported by the hardware, the SDK provides the current door position.

```
if lock.door_open:
    print("Door is open")
```

Possible values

Value	Description
True	Door open
False	Door closed

8.8 Lock State

The SDK converts protocol values into readable properties.

```
if lock.locked:
    ...
if lock.unlocked:
    ...
if lock.hold_open:
    ...
```

8.9 Device Information

Read information stored in the lock.

```
info = lock.device_info()
```

Example

```
print(info.device_id)
print(info.uid)
print(info.firmware)
print(info.hardware)
```

8.10 Health Information

Read diagnostic information.

```
health = lock.health()
```

Available information

- Supply voltage
- Temperature
- Runtime
- Firmware
- Hardware revision
- Diagnostics

8.11 Identify Device

For commissioning purposes, a lock can be identified physically.

```
lock.identify()
```

Depending on the hardware configuration this may

- flash LEDs
- activate the buzzer
- perform an identification sequence

8.12 Reset Device

Perform a software reset.

```
lock.reset()
```

The device automatically reconnects after restarting.

8.13 Factory Reset

Restore factory settings.

```
lock.factory_reset()
```

After a factory reset

- Device ID is removed
- Device returns to Default ID

- Device enters commissioning mode

8.14 Online Status
The SDK continuously monitors communication.

```
if lock.online:
    print("Device online")
```

or

```
if not lock.online:
    print("Device offline")
```

8.15 Last Communication
Applications may determine when the device was last seen.

```
print(lock.last_seen)
```

Example

```
2026-07-06 14:23:15
```

8.16 Event Handling
Applications can subscribe to lock events.

```
controller.on_lock_open(callback)

controller.on_lock_closed(callback)

controller.on_lock_online(callback)

controller.on_lock_offline(callback)

controller.on_lock_error(callback)
```

8.17 Exceptions
Typical exceptions generated by the SDK

Exception	Description
LockTimeoutError	Communication timeout
LockOfflineError	Device not reachable
InvalidDeviceError	Unknown Device ID
CommandRejectedError	Command rejected
CANCommunicationError	CAN bus failure

8.18 Best Practice
✓ Recommended

```
lock = controller.get_lock(5)

lock.open()

status = lock.status()

if status.locked:
    print("Lock is locked")
```

✗ Not Recommended

```
Build raw CAN frames

Send CAN commands manually
```

PS GmbH
Melisau 1255
6863 Egg / Austria

Decode STATUS frames

Maintain device state inside the application
The SDK manages communication, state synchronization and error handling automatically, allowing the application to work exclusively with high-level objects.

PS GmbH
Melisau 1255
6863 Egg / Austria

9. Whitelist Management

9.1 Overview

The whitelist allows RFID credentials to be stored directly on the lock. Authorized credentials can operate the lock locally without requiring communication with the controller.
Each whitelist is managed independently for every lock.
The SDK provides a complete API for creating, updating, deleting and synchronizing whitelist entries.

9.2 Whitelist Types

Two types of whitelist entries are supported.

Type	Description
Persistent	Stored permanently in the device memory
Ephemeral	Temporary entry automatically removed according to the configured policy

9.3 Adding an RFID Credential

Store a new RFID credential.

```
lock.whitelist.add(uid)
```

Example

```
uid = "04A1B2C3D4E5"

lock.whitelist.add(uid)
```

9.4 Removing an RFID Credential

Delete an RFID credential from the whitelist.

```
lock.whitelist.remove(uid)
```

9.5 Reading the Whitelist

Read all stored RFID credentials.

```
cards = lock.whitelist.list()
```

Example

```
for card in cards:
    print(card.uid)
```

9.6 Clearing the Whitelist

Delete all whitelist entries.

```
lock.whitelist.clear()
```

9.7 Number of Entries

Determine how many RFID credentials are currently stored.

```
count = lock.whitelist.count()
```

9.8 Synchronization

Synchronize the whitelist with the device.

```
lock.whitelist.sync()
```

The SDK automatically transfers only modified entries whenever possible to reduce CAN bus traffic.

9.9 Importing a Whitelist

Load whitelist entries from an external source.

```
lock.whitelist.import_file("members.csv")
```

9.10 Exporting a Whitelist

Save the current whitelist.
```
lock.whitelist.export_file("backup.csv")
```

9.11 Whitelist Entry

Each whitelist entry contains the following information.

Property	Description
UID	RFID card identifier
Type	Persistent or Ephemeral
Created	Date of creation
Last Used	Last successful access
Active	Entry enabled

9.12 Automatic Updates

The SDK automatically updates the internal whitelist after:

- Card added
- Card removed
- Whitelist synchronized
- Device restart
- Factory reset

Applications always work with the current whitelist state.

9.13 Events

Applications may subscribe to whitelist events.

```
controller.on_card_added(callback)

controller.on_card_removed(callback)

controller.on_whitelist_changed(callback)
```

9.14 Best Practice

✓ Keep the whitelist synchronized after configuration changes.
✓ Use persistent entries for regular users.
✓ Use ephemeral entries for temporary access.
✓ Create regular whitelist backups.

10. Event Handling

10.1 Overview

The PS Locks OIP SDK uses an event-driven architecture to notify applications about changes in device status, communication and lock operation.
Applications can subscribe to events instead of continuously polling device information.
This approach reduces CAN bus traffic and simplifies application development.

10.2 Event Categories

The SDK generates events for the following categories.

Category	Description
Controller	Controller state changes
Discovery	Lock detection and commissioning
Lock	Lock state changes
Door	Door sensor events
Whitelist	Whitelist modifications
Health	Device diagnostics
Communication	CAN communication
Errors	Runtime errors

10.3 Controller Events

Controller events indicate changes to the communication system.
Example

```
controller.on_started(callback)

controller.on_stopped(callback)

controller.on_connected(callback)

controller.on_disconnected(callback)
```

Typical events

- Controller started
- Controller stopped
- CAN interface connected
- CAN interface disconnected

10.4 Discovery Events

Applications can monitor device discovery.

```
controller.on_lock_found(callback)

controller.on_lock_removed(callback)

controller.on_unknown_lock(callback)

controller.on_lock_commissioned(callback)
```

Typical events

- Lock detected
- Unknown lock detected
- Device commissioned
- Device removed

10.5 Lock Events

Applications receive notifications whenever the state of a lock changes.
`lock.on_opened(callback)`

`lock.on_closed(callback)`

`lock.on_locked(callback)`

`lock.on_unlocked(callback)`
Typical events

- Lock opened
- Lock closed
- Lock locked
- Lock unlocked

10.6 Door Events

If a door sensor is available, the SDK generates door events.
`lock.on_door_opened(callback)`

`lock.on_door_closed(callback)`

10.7 Whitelist Events

Whitelist modifications generate events automatically.
`lock.whitelist.on_added(callback)`

`lock.whitelist.on_removed(callback)`

`lock.whitelist.on_changed(callback)`
Typical events

- Card added
- Card removed
- Whitelist updated

10.8 Health Events

Health events notify the application about diagnostic changes.
`lock.on_health_changed(callback)`

`lock.on_temperature_warning(callback)`

`lock.on_voltage_warning(callback)`
Typical events

- Voltage warning
- Temperature warning
- Device restarted
- Diagnostic information updated

10.9 Communication Events

Communication events provide information about the connection to a device.
`lock.on_online(callback)`

`lock.on_offline(callback)`

`lock.on_heartbeat(callback)`
Typical events

- Device online
- Device offline
- Heartbeat received
- Communication timeout

10.10 Error Events
Runtime errors are reported through dedicated error events.
`controller.on_error(callback)`
Example

```
def on_error(error):
    print(error.message)

controller.on_error(on_error)
```

10.11 Event Object
Every event contains standardized information.

Property	Description
Timestamp	Time of event
Device ID	Source device
Lock ID	Source lock
Event Type	Event category
Event Data	Event-specific information

10.12 Event Processing
Applications should keep event handlers short and non-blocking.
✓ Recommended

```
def on_lock_opened(lock):
    logger.info(f"Lock {lock.device_id} opened")
```

Avoid performing long-running operations directly inside callbacks.

10.13 Best Practice
The SDK is designed to work in an event-driven manner.
Use events to react to changes instead of polling devices continuously.
✓ Recommended

```
controller.on_lock_found(...)
controller.on_error(...)
lock.on_opened(...)
lock.on_health_changed(...)
```

This results in better performance, lower CAN bus utilization and a more responsive application.

PS GmbH
Melisau 1255
6863 Egg / Austria

11. Error Handling

11.1 Overview

The PS Locks OIP SDK provides a consistent error handling mechanism for all controller and lock operations.
Communication errors, invalid operations and device failures are reported through exceptions and events, allowing applications to respond appropriately.

11.2 Error Categories

Errors are grouped into the following categories.

Category	Description
Communication	CAN bus communication failures
Controller	Controller related errors
Device	Lock or device errors
Configuration	Invalid configuration parameters
Whitelist	Whitelist operation failures
Discovery	Commissioning and discovery errors
Internal	SDK internal errors

11.3 Communication Errors

Communication errors occur when a device does not respond or the CAN interface is unavailable.
Typical causes

- CAN cable disconnected
- Device powered off
- CAN interface not initialized
- Communication timeout

Example

```
try:
    lock.open()

except CommunicationTimeoutError:
    print("The device did not respond.")
```

11.4 Device Errors

Device errors indicate that the command reached the device but could not be executed.
Examples include

- Invalid command
- Invalid parameter
- Device busy
- Device locked by another operation

Example

```
try:
    lock.factory_reset()

except DeviceBusyError:
    print("Device is currently busy.")
```

11.5 Discovery Errors

Discovery may fail for several reasons.
Possible causes

- Duplicate Device ID
- Invalid UID

- Assignment failed
- Device disappeared during commissioning

Example

```
try:
    controller.assign_id(device, 5)

except DiscoveryError:
    print("Commissioning failed.")
```

11.6 Whitelist Errors

Whitelist operations may generate the following errors.

Error	Description
UID already exists	RFID credential already stored
UID not found	Credential does not exist
Whitelist full	No free storage available
Invalid UID	Unsupported UID format
Synchronization failed	Device update failed

11.7 Configuration Errors

Configuration errors are detected before commands are transmitted.

Examples

```
controller.set_bitrate(12345)
```

Result

```
InvalidConfigurationError
```

11.8 Exception Hierarchy

The SDK uses a structured exception hierarchy.

```
PSLocksError
│
├── CommunicationError
│    ├── CommunicationTimeoutError
│    ├── CANInterfaceError
│    └── DeviceOfflineError
│
├── DeviceError
│    ├── InvalidCommandError
│    ├── DeviceBusyError
│    └── FactoryResetError
│
├── DiscoveryError
│
├── WhitelistError
│
└── ConfigurationError
```

11.9 Error Events

Applications may subscribe to runtime error events.

```
controller.on_error(callback)

lock.on_error(callback)
```

Example

```
def error_handler(error):
    logger.error(error)

controller.on_error(error_handler)
```

11.10 Logging

The SDK provides integrated logging.
Typical log levels

Level	Description
DEBUG	Detailed diagnostic information
INFO	Normal operation
WARNING	Recoverable problems
ERROR	Operation failed
CRITICAL	Fatal errors

Example

```
controller.enable_logging()

controller.set_log_level(LogLevel.INFO)
```

11.11 Recovery

Whenever possible, the SDK automatically attempts to recover from communication failures.
Typical recovery actions include

- Reconnect to CAN interface
- Rediscover devices
- Re-establish communication
- Refresh device status

Applications are notified if automatic recovery fails.

11.12 Best Practice

Applications should always catch SDK exceptions at the highest appropriate level.
✓ Recommended

```
try:

    lock.open()

except PSLocksError as ex:

    logger.error(ex)
```

Avoid handling individual CAN communication errors inside application logic whenever possible.
The SDK is designed to isolate protocol and transport layer errors from the application layer.

PS GmbH
Melisau 1255
6863 Egg / Austria

12. Integration (and Raspberry)

12.1 Overview

The PS Locks OIP API is designed for integration into third-party applications.
Communication with PS Locks devices is performed via the OIP CAN protocol. The API can be integrated into applications running on Linux, Windows, embedded controllers or any other platform providing CAN communication.
The Raspberry Pi is used throughout this document as a reference platform. However, the API is not limited to Raspberry Pi systems.

12.2 System Requirements

Typical integration platforms include:

- Raspberry Pi
- Industrial PCs
- Linux systems
- Windows applications
- Embedded Linux devices
- PLC gateways
- Custom embedded controllers

12.3 Software Installation

The application communicates with the OIP API.
The API is responsible for:

- Device discovery
- Device commissioning
- Lock control
- Whitelist management
- Status monitoring
- Error handling

The underlying CAN interface is implementation-specific and can be adapted to the target platform.

The PS Locks OIP SDK is supplied as part of the PS Locks software package.
Copy the SDK to the Raspberry Pi and change into the project directory.

```
cd PSLOCKS_OIP
```

Create and activate the Python virtual environment.

```
python3 -m venv .venv
source .venv/bin/activate
```

Install the required dependencies.

```
pip install -r requirements.txt
```

Install the PS Locks SDK.

```
pip install pslocks
```

or clone the repository.

```
git clone https://pslocks.com/PSLOCKS_OIP.git
```

12.4 Configure the CAN Interface

Example for a 50 kbit/s CAN bus.

```
sudo ip link set can0 down

sudo ip link set can0 up type can bitrate 50000
```
Verify the interface.
```
ip link show can0
```
Expected output
```
can0: <UP,LOWER_UP>
```

12.5 Starting the SDK

Create a controller instance.
```
from pslocks import Controller

controller = Controller()
```
Start communication.
```
controller.start()
```
The SDK initializes the CAN interface and begins monitoring the CAN bus.

12.6 Discover Connected Locks
```
locks = controller.discover()
```
Example
```
for lock in locks:

    print(lock.device_id)
```

12.7 Working with a Lock

Retrieve a lock object.
```
lock = controller.get_lock(5)
```
Open the lock.
```
lock.open()
```
Read its status.
```
status = lock.status()
```

12.8 Commissioning a New Lock

Detect uncommissioned devices.
```
devices = controller.find_uncommissioned()
```
Assign a Device ID.
```
controller.assign_id(
    devices[0],
    device_id=5
)
```
After commissioning the device becomes available automatically.

12.9 Heartbeat Monitoring

The SDK continuously monitors communication with all connected devices.
Applications may verify communication status.
```
if lock.online:

    print("Device online")
```
or subscribe to heartbeat events.
```
lock.on_heartbeat(callback)
```

12.10 Working with the Whitelist

Add a new RFID credential.

```
lock.whitelist.add(uid)
```
Read the whitelist.
```
cards = lock.whitelist.list()
```
Delete a credential.
```
lock.whitelist.remove(uid)
```

12.11 Multiple Locks

Applications can control multiple locks simultaneously.
```
for lock in controller.locks:

    lock.open()
```
or
```
for lock in controller.locks:

    print(lock.status())
```

12.12 Stopping the SDK

Before shutting down the application, stop the controller.
```
controller.stop()
```
The SDK closes the CAN interface and releases all allocated resources.

12.13 Typical Application Structure

```
Application

    |

Create Controller

    |

Start Controller

    |

Discover Locks

    |

Commission New Locks (optional)

    |

Operate Locks

    |

Monitor Events

    |

Shutdown
```

12.14 Best Practice

PS GmbH
Melisau 1255
6863 Egg / Austria

✓ Initialize the controller only once.
✓ Perform device discovery during application startup.
✓ Use SDK events instead of continuous polling.
✓ Always stop the controller before terminating the application.
✓ Keep all lock operations inside the SDK.

PS GmbH
Melisau 1255
6863 Egg / Austria

13. CAN Protocol Reference

13.1 Overview

The PS Locks OIP SDK communicates with all devices using the PS Locks OIP protocol over a standard CAN 2.0A bus.
Normally, applications do not need to create or decode CAN frames manually. The SDK handles all protocol details internally.
This chapter is intended for developers who require a deeper understanding of the communication protocol for debugging, diagnostics or custom integrations.

13.2 CAN Frame Format

All messages are transmitted using the standard CAN frame.

Field	Description
Identifier	11-bit CAN Identifier
DLC	Number of payload bytes
Data	Up to 8 data bytes

13.3 Message Types

Message	Direction	Purpose
COMMAND	Controller → Lock	Execute a command
STATUS	Lock → Controller	Current lock status
HEALTH	Lock → Controller	Device diagnostics
MASTER_RESPONSE	Controller → Lock	Commissioning response
ID_REQUEST	Lock → Controller	Request Device ID
UID_PART_1	Lock → Controller	First part of UID
UID_PART_2	Lock → Controller	Second part of UID

13.4 Typical Communication Sequence

Opening a lock follows the sequence below.

```
Application
     |
     ▼
Controller.open()
     |
     ▼
SDK
     |
     ▼
COMMAND (Open)
     |
     ▼
Lock
     |
COMMAND_ACK
     |
STATUS
     ▼
SDK
     |
     ▼
Application
```

13.5 Discovery Sequence

Commissioning a new device.

Controller

|

Discover

|

▼

Unknown Device

|

Read UID

|

Assign Device ID

|

Verify

|

Online

13.6 Heartbeat Communication

The SDK periodically verifies communication with connected devices.

Controller
|
Heartbeat
|
▼
Device
|
Heartbeat Response
▼
Controller

If no response is received within the configured timeout, the device is marked as **offline**.

13.7 Whitelist Synchronization

Updating the whitelist follows the sequence below.

Application

|

Whitelist Update

|

SDK

|

CAN Messages

|

Device

|

Confirmation

▼

Application
The SDK automatically handles segmentation, acknowledgements and retries.

13.8 Error Recovery
In case of communication errors, the SDK performs the following actions:

1. Detect communication timeout.
2. Retry the operation (where supported).
3. Update the device state.
4. Notify the application.
5. Continue monitoring the CAN bus.

13.9 Developer Notes
For normal application development, direct interaction with CAN messages is **not required**.
Applications should always use the high-level SDK API instead of creating CAN frames manually.
Only diagnostic tools and protocol analyzers should access raw CAN traffic.

13.10 Best Practice
✓ Use the SDK API for all communication.
✓ Use protocol analyzers only for diagnostics.
✓ Do not rely on undocumented CAN messages.
✓ Avoid transmitting custom CAN frames while the SDK is active.

PS GmbH
Melisau 1255
6863 Egg / Austria

14. Examples

14.1 Overview

This chapter contains practical examples demonstrating the most common integration tasks. The examples are intentionally simplified to illustrate the typical application workflow. Complete example projects are provided separately with the SDK.

14.2 Connecting to the Controller

Initialize the controller and establish communication with the CAN bus.

```
from pslocks import Controller

controller = Controller()

controller.start()
```

14.3 Discovering Locks

Search for all connected devices.

```
locks = controller.discover()

for lock in locks:
    print(lock.device_id)
```

14.4 Detecting New Locks

Search for devices that have not yet been commissioned.

```
devices = controller.find_uncommissioned()

for device in devices:
    print(device.uid)
```

14.5 Assigning a Device ID

Commission a new lock.

```
controller.assign_id(
    device=device,
    device_id=5
)
```

14.6 Opening a Lock

```
lock = controller.get_lock(5)

lock.open()
```

14.7 Closing a Lock

```
lock.close()
```

14.8 Reading Lock Status

```
status = lock.status()

print(status.state)
print(status.locked)
print(status.door_open)
```

14.9 Reading Device Information

```
info = lock.device_info()
```

```
print(info.firmware)
print(info.hardware)
print(info.uid)
```

14.10 Reading Health Information

```
health = lock.health()

print(health.voltage)
print(health.temperature)
```

14.11 Working with the Whitelist

Add a credential.

```
lock.whitelist.add(uid)
```

Remove a credential.

```
lock.whitelist.remove(uid)
```

Read all credentials.

```
cards = lock.whitelist.list()
```

14.12 Monitoring Events

```
controller.on_lock_found(callback)

controller.on_error(callback)

lock.on_opened(callback)
```

14.13 Controlling Multiple Locks

```
for lock in controller.locks:

    lock.open()
```

14.14 Graceful Shutdown

```
controller.stop()
```

The SDK closes all communication channels and releases allocated resources.

14.15 Complete Application Example

```
from pslocks import Controller

controller = Controller()

controller.start()

locks = controller.discover()

for lock in locks:

    print(lock.device_id)

    status = lock.status()

    print(status.state)

controller.stop()
```

14.16 Best Practice

✓ Create only one Controller instance.
✓ Perform discovery during application startup.
✓ Use Lock objects instead of raw CAN communication.
✓ Handle exceptions at the application level.
✓ Stop the controller before terminating the application.

14.17 Example Projects

The SDK includes the following example applications.

Example	Description
01_discover.py	Discover all connected locks
02_assign_id.py	Commission a new lock
03_open_lock.py	Open a lock
04_close_lock.py	Close a lock
05_lock_status.py	Read lock status
06_device_info.py	Read device information
07_health.py	Read diagnostic information
08_whitelist_add.py	Add an RFID credential
09_whitelist_remove.py	Remove an RFID credential
10_whitelist_list.py	Read whitelist entries
11_events.py	Subscribe to SDK events
12_multiple_locks.py	Control multiple locks

14.3 SDK Source Code

The complete SDK source code is supplied together with the documentation.
The project structure is organized as follows:
PSLOCKS_OIP/

```
├── api/
├── canbus/
├── examples/
├── docs/
├── tests/
├── requirements.txt
└── README.md
```

14.4 Learning by Example

Developers are encouraged to start with the supplied examples before implementing their own application.
Each example demonstrates one specific feature of the SDK and can be executed independently.

14.5 Best Practice

The supplied examples are part of the SDK and are maintained together with each software release.
Always use the examples matching the SDK version installed on your system.