OPEN COLLISION AVOIDANCE PROTOCOL

SOFTWARE USER MANUAL

Document	OCAP_Software_User_Manual
Revision	1.0
Date	20.06.2025

Document: OCAP_User_Manual
Revision: 1.0
Date: 20.06.2025

TABLE OF CONTENTS

CHANGE LOG

Date	Author	Version	Change
19.06.2024	A. Schweizer		Document created
26.05.2025	A. Schweizer	0.9	Internal draft release
20.06.2025	A. Schweizer	1.0	Public release

1 INTRODUCTION

1.1 Overview

The OCAP project took place from 2023 to 2025 with the goal to create an inter-operable, open, transparent and royalty-free software to provide collision warnings to aircraft pilots.

The output of the project is a software package which is available online on Github at the following URL:

https://github.com/aschweiz/project-ocap

During the development of the software, the project team has performed, in collaboration with external providers, a large number of test flights with different aircraft. These test flights have helped to develop, parameterize, validate and improve the implementation. The data from several test flights is available in the source code repository. Together with a complete 3D simulation environment, this data can be used for manual testing and automated regression testing of future software changes and improvements.

This document provides detailed information on the structure and implementation of the software package. It explains how to integrate it into vendor systems and how to use the simulation environment, test data and supplementary tools to analyze and improve the OCAP software.

The illustration below gives a simple example of how the system works:

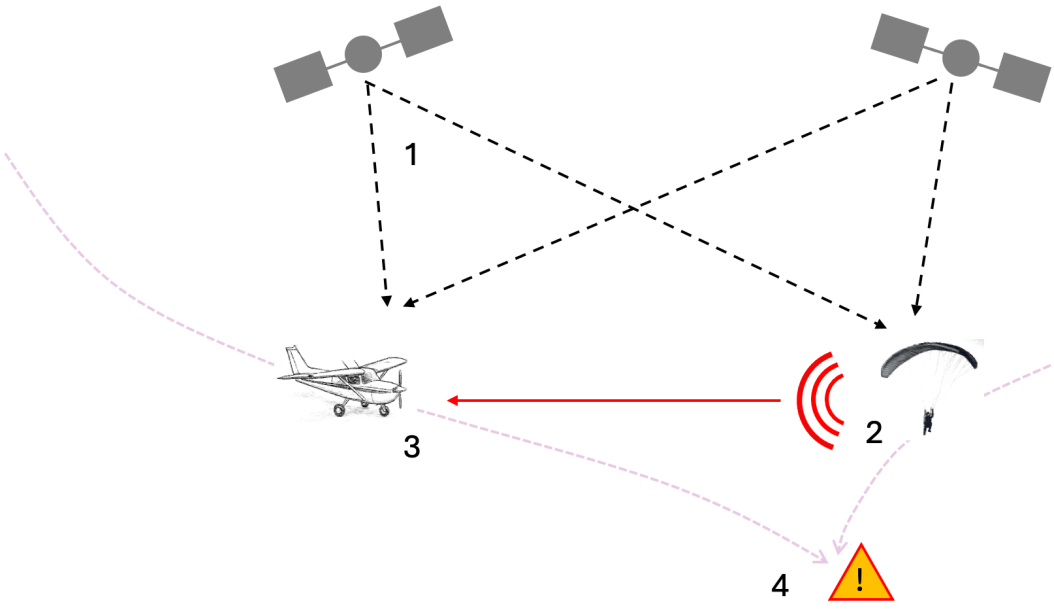


Figure 1: System level example

1. All participating aircraft contain a GNSS receiver capable of receiving the current time as well as their own position and movement information from one or multiple GNSS constellations such as GPS, Galileo and Glonass.
2. All participating aircraft broadcast their identification, position, movement information and ideally path curvature in defined time intervals, typically once per second.
3. Aircraft or ground stations that want to detect potential collisions receive the transmitted data packets.
4. Aircraft or ground stations that want to detect potential collisions feed the received data to the OCAP algorithm and process the generated collision warnings, typically by alerting the pilot.

The OCAP software supports steps 2, 3 and 4. Step 1 depends on the hardware and capabilities of the GNSS receiver and needs to be implemented by the vendor in the device firmware. In step 4, the OCAP software generates a data structure with information on the alert, but the presentation of that information, e.g. creating a sound or displaying a message on a screen, needs to be implemented in the device firmware.

1.2 Compatiblity with existing radio protocols

The OCAP software needs information about nearby aircraft to calculate collision warnings. The source for this information is typically a radio receiver that receives and decodes data packets transmitted by these aircraft. Each data packet needs to contain at least an identification of the transmitting aircraft, its position, speed and direction. Ideally, for improved performance and safety, the data packet also contains precise curvature information so that the curvature of the flight path is available after receiving a single packet.

In 2023, EASA has started the development of the ADS-L radio protocol, with the goal to make aircraft visible to other aircraft and to ground stations (iConspicuity). The focus of the initial versions 1 and 2 of ADS-L is on air-to-ground visibility, with air-to-air extensions planned for the upcoming version 3.

https://www.easa.europa.eu/en/domains/general-aviation/%21conspicuity

In addition, in 2024, the Swiss FOCA had started their FASST-CH initiative as part of their broader AVISTRAT strategy, with the goal to enable e-conspicuity in Switzerland. FOCA recommends ADS-B out on 1090 MHz and ADS-L in the 860 MHz band (see their technical recommendations).

https://www.bazl.admin.ch/bazl/en/home/themen/aviation-policies/fasst-ch/fasst_project/fasst-analyse.html

https://www.bazl.admin.ch/bazl/en/home/themen/aviation-policies/fasst-ch.html

The OCAP software itself doesn't depend on how the information about surrounding aircraft arrives at the receiver. Technically, this could be any means, for example the cellular network, dedicated (reserved) frequencies, shared ISM band communication or even via the Internet (for example, Open Glider Network), as long as the above-mentioned information is made available. However, given the strong support for ADS-L at the time of the development of the OCAP software, it was decided to include an implementation for ADS-L out and ADS-L in as part of the OCAP reference implementation and to use ADS-L to verify the performance of the OCAP software.

1.3 libocap and libadsl

The OCAP software is split in two separate software libraries, **libocap** and **libadsl**, to give device vendors a choice of the part that they want to integrate. The collision warning algorithm is implemented in libocap, while libadsl provides an implementation of ADS-L in and out.

If a device can receive other protocols such as FANET, FLARM or cellular, the device vendor needs to implement software to decode the packets so that the device firmware can forward the data to libocap for collision warning calculations.

Complementary, if a device vendor would like to generate ADS-L out packets to make an aircraft visible but doesn't need to process incoming packets and generate collision warnings, for example in a beacon device for hang gliders, the device vendor can use **libadsl** to create the packets and can ignore libocap.

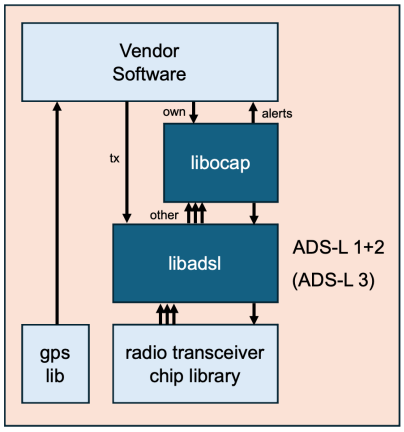


Figure 2: Software structure

The programming language for libocap and libadsl is C. Aside from the C standard library, no additional libraries are needed. This makes it easy to integrate the OCAP software into embedded system firmware.

See chapter 2 for information on how to integrate libadsl and libocap into vendor firmware.

1.4 Prediction algorithm

To predict potential collisions, the OCAP software extrapolates flight paths of the own aircraft and of received aircraft in discrete time steps of 1 second. The extrapolation depth can be configured in the software; the default is 30 seconds. At each time step, the algorithm calculates the distance between the aircraft and compares it to three threshold values, corresponding to three different alarm levels. Distance thresholds are determined dynamically and depend mainly on the velocity of the two aircraft and the current depth of extrapolation.

Three different scenarioes are considered for the flight path extrapolation. In the most generic scenario, the flight path extrapolation is performed under the assumption of zero acceleration along the axis of movement (i.e. the aircraft is neither increasing nor decreasing its velocity) and constant acceleration in radial direction. I.e., the prediction considers curved flight paths at constant velocity in all 3 dimensions. Simplified extrapolation methods are applied for very short and very large radii. For very short radii, the algorithm assumes that the object is quasi static in a sphere. For very large radii, the algorithm assumes that the aircraft is flying along a straight line.

To calculate the curvature of the flight path, the algorithm needs at least two data points with position and movement information. The OCAP software includes calculation of the center point (Z vector) of the arc for the generic scenario, both for the own aircraft and also for surrounding aircraft. If the radio protocol permits it, the Z vector should be included in the transmitted packet so that the receiver can determine the curvature of the flight path based on a single, received packet. In addition, the sender can calculate the Z vector with higher precision than the receiver because it has more detailed information available than the receiver, and the receiver can save computing time.

The following illustration shows how the algorithm can predict the potential collision of aircraft A and B by considering the curved flight path of aircraft B. Considering a straight flight path for aircraft B (along the dashed line) would be insufficient in this case and the alert would only happen later, very close to the collision point:

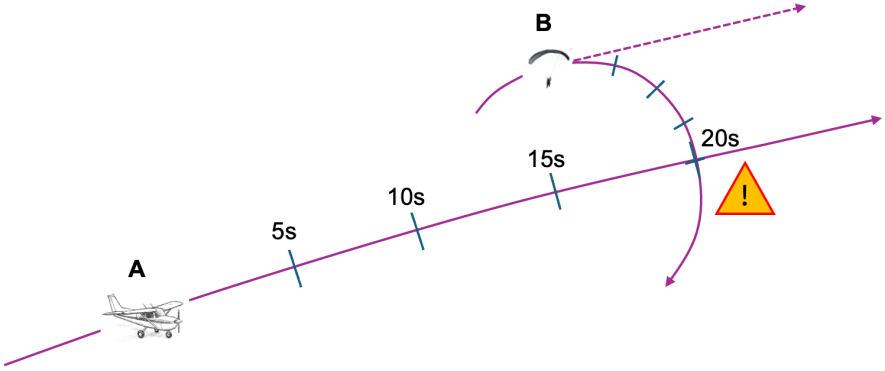


Figure 3: Curved flight path

For detailed information about the OCAP algorithms, see chapter 3.

1.5 Simulation environment, test data

Several test flights with different aircraft have been performed and recorded as part of the development of the OCAP software. During these test flights, the pilots have executed specific, pre-defined flight patterns. These test flights have helped to verify and optimize the software. Recorded data from the flights is available for regression testing, i.e. to verify the software after future extensions and improvements.

Part of the OCAP software is an OpenGL based 3D simulation environment. In this environment, recorded flights can be re-played, either automatically as part of an automated test setup or manually under the control of the developer. The simulation environment integrates libocap and libadsl and simulates the generation and reception of ADS-L data packets as well as the calculation of collision warnings.

Flight path information for the simulation environment is stored in ".flp" files. These files contain position and motion information at equally spaced points in time. Test case files, ".tst", combine multiple ".flp" files into a test case. The source code repository includes an importer tool to derive ".flp" files from IGC files. See section 5.1 for more information.

In addition to ".flp" data, the simulator can also process recorded ADS-L packets. This allows, for example, to analyze in detail with a debugger the code execution in libadsl and libocap.

The simulation environment consists of three windows, as shown in the following screen shot:

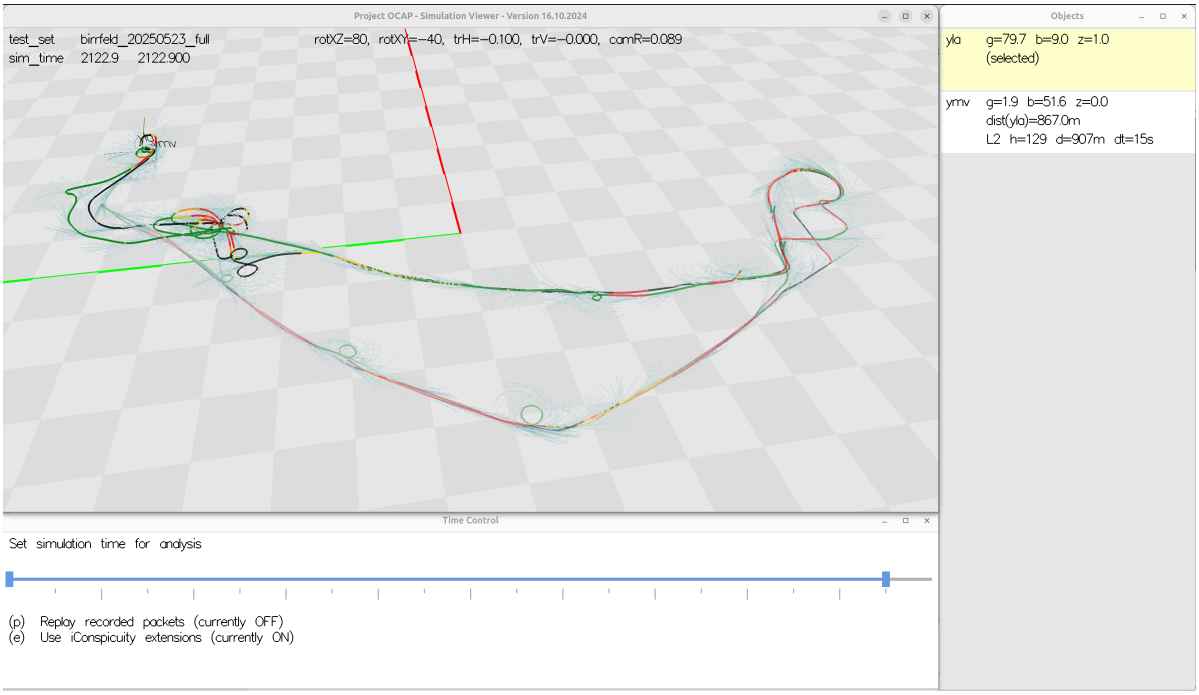


Figure 4: OCAP simulation environment

In interactive mode, the user can trigger and analyze individual calculation steps. Collision warnings are shown in the console and graphically as a color change on the aircraft's flight path in the main window at the top left.

In the automatic mode, the simulator is started from the command line and automatically simulates a flight or part of a flight. Calculation results are stored in an output test file, making it easy to detect changes with respect to earlier runs.

See chapter 4 for details about the simulator and test data.

2 INTEGRATION INTO VENDOR FIRMWARE

2.1 Options for integration

All source code for libadsl, libocap, the simulation environment and support tools is available on github:

https://github.com/aschweiz/project-ocap

Libadsl and libocap are written in the C programming language and depend only on the C standard library.

There exist two different options to integrate the libraries into vendor firmware:

- The first option is to copy the source and header files into the firmware project and compile it together with the rest of the firmware.
- The second option is to create separate build targets for libadsl and libocap and compile them into static libraries. The integration into the firmware then happens by linking the firmware with the static libraries and by including the header files of the libraries.

Using separate build targets for libadsl and libocap is the recommended option, as it helps to keep the code separate. This approach may also improve compilation speed.

2.2 Integration overview

A complete integration of libocap and libadsl consists of 3 parts:

- Transmitting packets about the aircraft's position and movement,
- receiving packets from surrounding aircraft about their position and movements,
- and processing information to generate collision warnings.

To support the integration process, libocap includes a logging facility. It defines a logging interface and calls logging functions to display relevant calculation results. Vendor firmware should provide an implementation of the logging interface and make the log output available for example via console, UART or SD card, depending on the capabilities of the vendor device.

The following subsections provide a high-level overview for each of these steps, and later sections describe the integration in more detail.

Note: Part of the software package is a simulation environment (see chapter 4). The simulator fully integrates libadsl and libocap. It may help to have a look at the simulator code for an integration example or single-stepping through one of the test cases to see the input and output of each individual step.

2.2.1 Transmitting packets

An aircraft periodically transmits information about its identity, position and movement. It typically gets position and movement information from a GNSS receiver (e.g. GPS, Galileo) and static information such as the ID and type of aircraft from configuration memory.

For ADS-L, it is recommended in version 1 of the standard to transmit a packet once per second. Version 2 which is still in the draft phase will likely relax this requirement so that the device software may decide to reduce the transmission rate, for example if the aircraft is not yet airborne or if the frequency is overloaded with too many nearby aircraft.

The following diagram shows how the different parts of the system interact to transmit a packet:

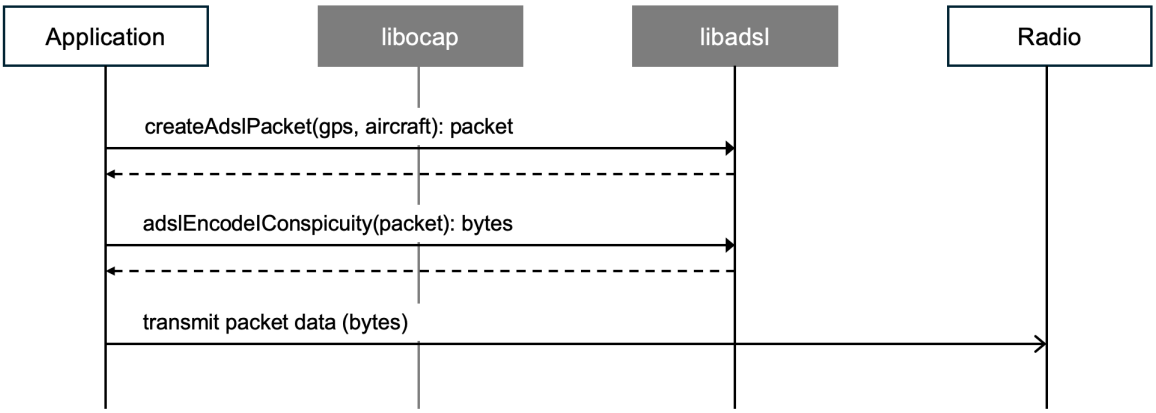


Figure 5: Packet transmission overview

In the first step, when it's time to transmit a packet, the application software passes GNSS and configuration data to libadsl. The library collects the necessary information and constructs an ADS-L iConspicuity data packet which it returns back to the application.

In the second step, the application passes the ADS-L iConspicuity data packet to libadsl to convert it to a sequence of bytes for transmission. Not shown in the diagram are the calculation of the checksum (could be provided by the radio chipset) and the scrambling (optional starting in ADS-L version 2). Libadsl includes helper functions for these two processing steps which can be used if necessary. For ADS-L version 2, which is currently in a draft state, it is planned to make the XXTEA scrambling optional.

Step 3, the last step, consists of passing the packet bytes to the radio chip. Neither libadsl nor libocap contain any code that interacts directly with the radio chip. Instead, the application code needs to implement this functionality.

2.2.2 Receiving packets

While not transmitting, the aircraft should configure the radio chipset for packet reception so that it can receive packets from surrounding aircraft.

The following diagram shows how the different parts of the system interact to receive a packet. The steps shown in the diagram need to be executed once for every received packet:

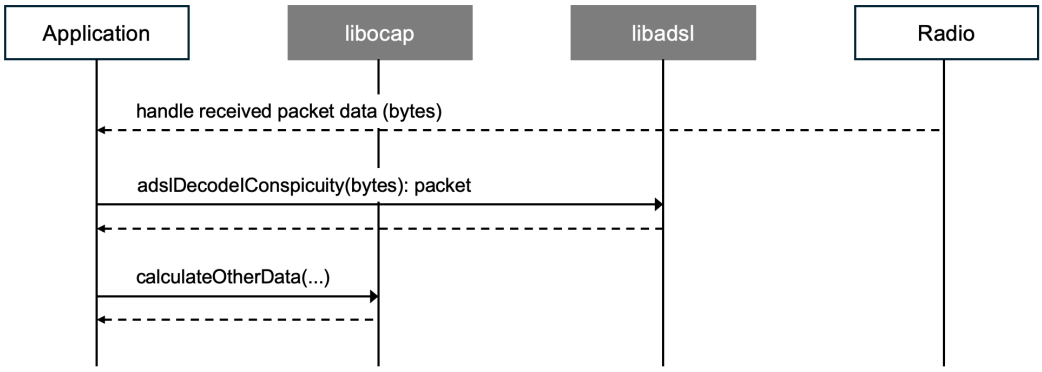


Figure 6: Packet reception overview

In a first step, the application code retrieves a received packet from the radio chipset, typically after receiving a hardware intererrupt. The application should verify the packet checksum if not already done by the radio chipset and should descramble the packet payload if necessary. Libadsl contains helper functions for these two actions.

In a second step, the application provides the received, descrambled raw data bytes to libadsl for deserialization. The output of this step is an iConspicuity data structure which contains information about the identity, position and movement of the aircraft.

In a third step, the application passes the received data to libocap so that it can be considered in the collision warning.

2.2.3 Generating collision warnings

Once per second, at the end of the data reception time window, the application invokes libocap functions to calculate collision warnings. The following diagram shows the interaction:

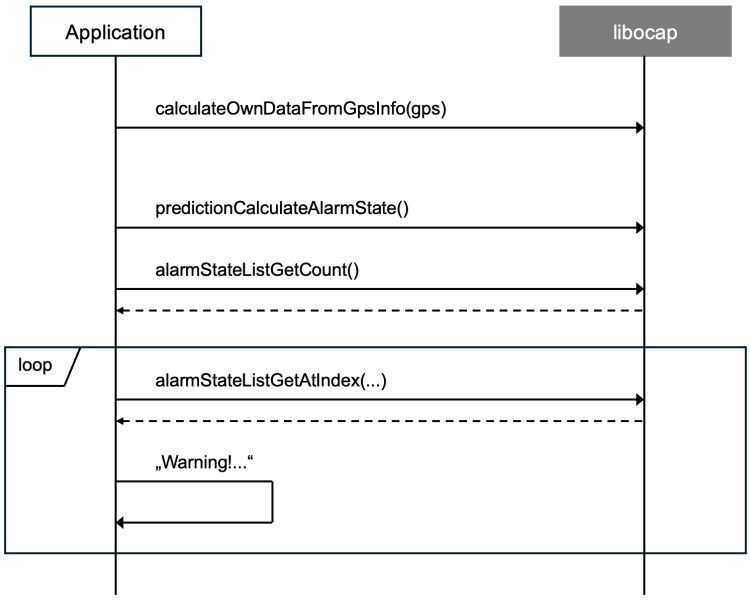


Figure 7: Calculating collision warnings

1. The application needs to provide position and movement information about the own aircraft to libocap. This may happen at any time, for example right after the data becomes available, or right before the subsequent steps.
2. the application invokes a libocap function to predict potential collisions and calculate collision warnings. Libocap stores all collision warnings in a list.
3. The application needs to process the entries of the list. There may be no warnings, exactly one warning or multiple warnings in the list. The list is sorted by severity of the alert, so that the first entry in the list is the most critical alert.

It's up to the application software if it wants to provide only the most critical alert or multiple alerts to the pilot, and how the presentation of the alert should occur, i.e. if it should be a sound, an information on a display or both.

The following sections provide details on the integration of libadsl and libocap.

2.3 libadsl integration

Libadsl contains code to create and decode ADS-L packets. If the vendor firmware already contains this functionality, or if only alternative radio protocols will be used, libadsl doesn't need to be integrated.

As of now, only the iConspicuity packet as defined in ADS-L version 1 is supported. The library is designed in a way that makes it easy to add more packet types once newer versions of the ADS-L standard become available.

The following subsections describe how to create and how to decode ADS-L packets with libadsl.

To support the integration process, a main.c file is provided in the libadsl source directory which implements the described steps. You can compile main.c to a small test program and step with the debugger through the code to see how each step works. You can also use the test data provided in main.c to test the integration into the vendor firmware.

2.3.1 Creating an iConspicuity ADS-L packet with libadsl

Aircraft transmit, among other data, their identification, position, velocity and direction of flight in an ADS-L iConspicuity packet. In the ADS-L version 1 standard, it is recommended to transmit an iConspicuity packet once every second. This requirement will likely be relaxed in version 2, for example, if the device can detect that the aircraft isn't airborne yet or if the frequency is overloaded.

Libadsl needs as input dynamic GNSS input as well as static information about the aircraft. This information is provided in two different data structures, `SAircraftConfig` and `SGpsData`.

The following example shows how to prepare the `SAircraftConfig`:

```
SAircraftConfig aircraftConfig;
aircraftConfig.addrMapEntry = 9; // Manuf. page 0
aircraftConfig.addr = 0x123456;
aircraftConfig.acftCategory = ADSL_ICONSP_AIRCRAFT_CATEGORY_ROTORCRAFT;
aircraftConfig.flightState = ADSL_ICONSP_FLIGHT_STATE_AIRBORNE;
```

Vendors can request an address map entry from EASA. This entry needs to be provided in the corresponding field of the aircraft configuration structure, together with the address of the aircraft. The aircraft category is typically configurable by the end user of the device.

Dynamic information about the position and movement of the aircraft is provided in the `SGpsData` data structure, as shown in the next example. Replace the static data with information available in the device software:

```
SGpsData gpsData;
gpsData.ts_sec_in_hour = 200;
gpsData.lat_deg_e7 = 475000000; // 47.5N
gpsData.lon_deg_e7 =  85000000; // 8.5E
gpsData.height_m = 583;
gpsData.hacc_cm = 170;
gpsData.vacc_cm = 440;
gpsData.vel_u_cm_s = -80;
gpsData.gspeed_cm_s = 220;
gpsData.heading_deg_e1 = 400;  // 40 deg
gpsData.sacc_cm_s = 200;
```

The data for this data structure comes from the GNSS receiver. If the vendor firmware includes sensor fusion algorithms or filtering, providing the processed data may improve the performance of the flight path prediction.

Once the data structures have been prepared, pass them into the `createAdslPacket` function. This function fills an iConspicuity data structure:

```
// From GPS and aircraft state to ADS-L iConspicuity packet.
SAdslIConspicuity adslPacket;
int z[3] = { 0 };
createAdslPacket(
 &gpsData, &aircraftConfig, &adslPacket,
 ADSL_ICONSP2_PATH_MODEL_LINEAR, z);
```

If you want to use the Z vector extension for OCAP, you can provide the Z vector and path model to the function; otherwise, pass the constant for the linear path model and a 0 vector, as shown in the example.

To transmit the packet over the air, it is necessary to serialize the iConspicuity packet into a stream of bytes, by calling the function `adslEncodeIConspicuity`. This function takes as input an iConspicuity packet and provides the serialized data in an output byte array:

```
// From ADS-L iConspicuity packet to data bytes for transfer.
int encodeResult = adslEncodeIConspicuity(&adslPacket, dataOut22, 22);
```

A return value of 0 indicates successful encoding.

Note that the function neither scrambles the payload data with the XXTEA algorithm nor calculates the checksum.

Scrambling the payload with a public key is required in ADS-L version 1. For version 2, it is planned to make scrambling optional by setting the key index to 3. To scramble the payload, a helper function `adslXxteaEncodeWithPubkey` is provided as part of libadsl.

Calculating the checksum needs to be done after scrambling. Libadsl contains a helper function, `adslCrc`, which can be used if no other option exists such as calculating the checksum on the radio chip or in hardware on the microcontroller. Note that the first byte of the packet is **not** included in the calculation of the checksum; also, the checksum needs to be truncated to 24 bits and added to the packet as 3 bytes in big-endian representation. So, for example, if the function returns the value 0x5678abcd, you need to append bytes 0xcd, 0xab, 0x78 in this order and drop the byte 0x56.

2.3.2 Decoding an iConspicuity ADS-L packet with libadsl

Libadsl provides functions to derive the `SAircraftConfig` and `SGpsData` data structures from a received ADS-L iConspicuity data packet. Decoding the packet involves similar steps as encoding, but in reverse order.

In the first step, the checksum should be verified. You can use the `adslCrc` function from libadsl for this purpose. Apply the function to the received bytes (not including the first byte) and compare it to the received checksum. If the values differ, the packet is invalid and should be dropped.

In the second step, the packet payload needs to be descrambled with the XXTEA algorithm. Libadsl includes the function `adslXxteaDecodeWithPubkey` for this purpose. Fill the packet payload bytes into an array of 5 uint_32 values and call the function on this array. The function descrambles the data in place.

After descrambling, the iConspicuity packet can be passed to libocap, as explained in the next section.

2.4 libocap integration

Four steps need to be implemented to integrate libocap:

1. Provide input data about surrounding aircraft.
2. Provide input data about the own aircraft.
3. Trigger the collision warning algorithm in libocap.
4. Retrieve collision warning alert output from libocap and presenting it to the pilot.

The following subsections contain source code examples for these steps.

In addition, vendor software should provide an implementation of the logging interface. The library calls logging functions at different moments during the processing to provide intermediate calculation results useful for integration and debugging.

To support the integration process, a main.c file is provided in the libocap source directory which implements the described steps. You can compile main.c to a small test program and step with the debugger through the code to see how each step works. You can also use the test data provided in main.c to test the integration into the vendor firmware.

2.4.1 Implementing the logger interface

The logger interface is defined in the header file OcapLog.h. It contains several functions which should be implemented in vendor software, i.e. create a C file that imports OcapLog.h and provide implementations of the logging functions. The OcapLogSim.c file can be used as a starting point for the implementation.

Provide empty stubs to omit logging functions or for performance optimization in production builds.

Generic functions to log text and one or two numbers:

```
void ocapLogStrInt(const char *str, int i);
void ocapLogStrIntInt(const char *str, int i, int j);
```

Function to log the position and velocity of the own aircraft:

```
void ocapLogFlVec(int own, TVector *pos, TVector *vel);
```

Function to log the old (nr=0) and new (nr=1) tangential and normal vectors during flight path extrapolation

```
void ocapLogFlTN(int nr, TVector *vecT, TVector *vecN);
```

Functions to log information about the own aircraft:

```
void ocapLogFlOwn(TFlightObjectOwn *flOwn);
void ocapLogFlOwnPath(TVector *ownFlightPath);
```

Functions to log information on other aircraft.

```
void ocapLogFlOtherPath1(TVector *otherPos, int t);
void ocapLogFlOtherTs(int rxTs, int ts);
```

Functions to log intermediate results of the flight path extrapolation and prediction model calculations:

```
void ocapLogFpe(int own, TFlightPathExtrapolationData *fpe);
void ocapLogRZxyV(int own, TFlightPathExtrapolationData *fpe);
void ocapLogModelZxyzV10(int own, EOcapPathModel pathModel, int zx, int zy, int zz,
int v10);
```

The following is an example implementation for the `ocapLogFlTN` function as provided in the OcapLogSim.c file. Vendor firmware could replace the printf function for example with an UART or SD card write function.

```
void ocapLogFlTN(int nr, TVector *vecT, TVector *vecN) {
      printf("OCAP,FL-T-N,%d,%f,%f,%f,%f,%f,%f\n",
       nr,
       vecT->x, vecT->y, vecT->z,
       vecN->x, vecN->y, vecN->z);
}
```

For more examples and as a starting point for the implementation, please have a look at OcapLogSim.c.

2.4.2 Providing input data about surrounding aircraft

The data about surrounding aircraft should ideally be provided immediately after it has been received and decoded.

Libocap maintains a list of aircraft which should be updated after receiving the data. If the vendor firmware already maintains a database of surrounding aircraft and the aircraft data structure can be extended with the information declared in libocap's `TFlightObjectOther`, that database may be used instead.

The following code example assumes that information for an aircraft with ID `otherIdNr` has been received:

```
    // If we already know this object, we take it from the list of known objects.
    for (int i = 0; i < flightObjectListGetOtherCount(); i++) {
        TFlightObjectOther *fCur = flightObjectListGetOtherAtIndex(i);
        if (fCur->id == otherIdNr) {
            fOther = fCur;
            break;
        }
```

```
}
// If we don't know the object yet, we add it to the list.
if (!fOther) {
    fOther = flightObjectListAddOther(otherIdNr);
}
// We update the data for this object.
calculateOtherDataFromInfo(
    fOther,
    msg.txStartTimeMs / 1000,
    &posMtr,
    &velMtrSec,
    hasZ ? &zMtr : nullptr, pathModel);
```

The `posMtr` and `velMtrSec` vectors come from the decoded radio packet. In addition, if the OCAP Z vector extension is used and Z vector information together with a path model is available in the received packet, the Z vector and path model should also be provided to the `calculateOtherDataFromInfo` function.

The `calculateOtherDataFromInfo` function prepares OCAP data structures for path extrapolation in step 2.

2.4.3 Providing input data about the own aircraft and running the collision warning algorithm

The steps to provide input data about the own aircraft and about running the collision warning algorithm can be combined in one place. The following code example shows how to perform these steps. The `ts` variable contains the current UNIX time in seconds.

```
// Radio messages of surrounding acft have already been received and applied.
// Apply our own data, then run the collision warning algorithm.

calculateOwnDataFromGpsInfo(ts, &posNewMtr, &velNewMtrSec);

// Perform collision prediction.

predictionCalculateAlarmStates(ts);

// Show the result (most critical alarm at the head of the list).

int nofAlarmStates = alarmStateListGetCount();

TAlarmState *a = NULL;
if (nofAlarmStates > 0) {
    a = alarmStateListGetAtIndex(0);
    // ...
}
```

The call to `predictionCalculateAlarmStates` invokes the flight path extrapolation and collision prediction algorithms. Details are provided in chapter 3.

The last if statement in the example above is executed if at least one collision warning has been generated. The application code should present the alarm to the pilot.

It may happen that running the OCAP algorithm generates more than one collision warning. In this case, the alarm state list contains more than one entry, and the code can for example loop through these entries and show them on a display.

In a scenario where only a single alarm should be presented to the pilot, the application firmware can use the `AlarmService` module to filter the alarm states. The filter works as a high-pass filter for new alarms and a low-pass filter for existing alarms. For example, if a new potential collision is detected, it immediately outputs the alarm at the given alarm level. If later the alarm disappears, for example due to dropped radio packets or jitter in the calculations, it slows down the decay of the alarm level to smoothen it. Use the functions `alarmServiceUpdateMostCritical` and `alarmServiceGetMostCritical` of the alarm service module. See the simulator for an example.

3 LIBOCAP IMPLEMENTATION DETAILS

3.1 Software Structure

Libocap receives information about the own aircraft (typically from a GPS receiver) and from other aircraft (typically from received data packets, for example from libadsl).

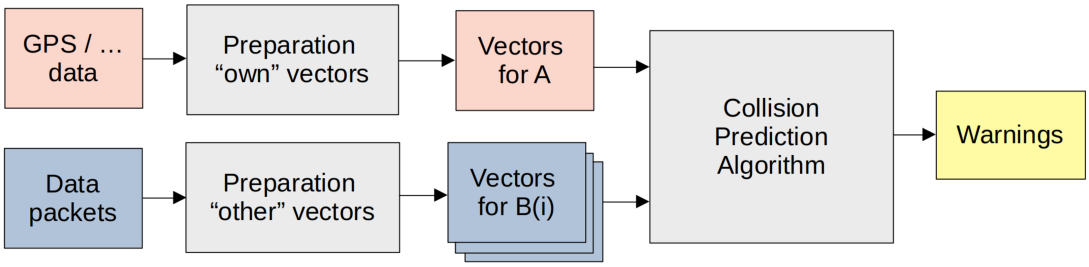


Figure 8: Data processing overview

From this information, it extrapolates the flight path of the own aircraft as well as the flight paths of surrounding aircraft. Section 3.2 explains how the flight path extrapolation is implemented.

To generate collision warnings, the algorithm then compares the distance between the own and the surrounding aircraft at discrete times in the future, along the extrapolated flight paths (collision prediction algorithm). Section 3.3 contains more information about how the warnings are generated.

The output of libocap is a list of alarms, with information about the aircraft involved in the possible collision, the time until a potential collision and the severity, expressed as a level between 1 (low) and 3 (high).

It is up to the device vendor to process the alarms, i.e. displaying them to the pilot or creating a sound.

3.2 Flight path extrapolation

At the core of libocap is the extrapolation of flight paths, implemented in the FlightPathExtrapolation.c source code file.

This section describes the general case of a **curved** 3D flight path. In some cases, the algorithm switches to a spherical or linear path model. For large radii r0-Z, a straight line is a good approximation for the flight path. Similarly, for small radii, modelling the aircraft as "static" will give sufficient accuracy.

The `EOcapPathModel` enum defines the 3 possible models:

```
typedef enum {
      OCAP_PATH_MODEL_LINEAR = 0,
      OCAP_PATH_MODEL_SPHERIC = 1,
      OCAP_PATH_MODEL_ARC = 2,
      // Reserved 3
} EOcapPathModel;
```

The OCAP algorithm assumes a spherical model (static aircraft) for small radii below 15*v, the standard curved (arc) model described in the previous section for the general case and a linear model for large radii above 1920*v. These limits make it possible to store the path model and Z vector information in 32 bits, 2 bits for the path model and 3x10 bits for the Z vector components, with each component split in 1 sign bit, 3 bits for the exponent and 6 bits for the mantissa, with a resulting worst-case error for a 30 second extrapolation of about one third of the distance that the aircraft flyes in one second.

The error is covered by the alarm criterion.

Detailed information on how the values have been determined is available on the project blog:

https://www.project-ocap.net/post/proposed-ads-l-extension

For performance reasons, vector geometry was used instead of trigonometric functions in the position extrapolation algorithm. Vector operations, such as addition, scaling, and dot products, enable more efficient computation of positions and directions, requiring fewer computationally intensive steps

The following graphic helps to understand how the extrapolation is implemented:

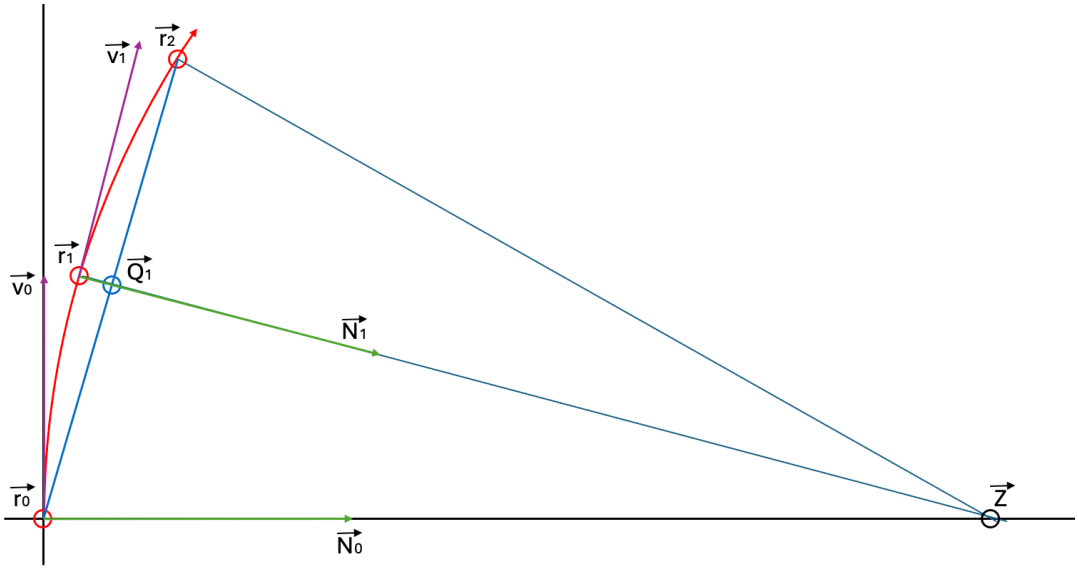


Figure 9: Extrapolation step

The OCAP algorithm supports two cases:

- In the ideal case, the Z vector is available in the data packet transmitted by the aircraft. In this case, the vectors r1 / v1 together with Z are enough to extrapolate r2 and v2. Function `flightPathExtrapolationPrepareRvz` handles this case.
- If Z is not available, the algorithm first calculates Z based on r0 / v0 and r1 / v1. The logic is implemented in function `flightPathExtrapolationPrepare2rv`. The function first calculates tangential vectors T0 and T1 in r0 and r1, then perpendicular normal vectors N0 and N1 (pointing towards Z). The normal vectors are used to calculate Z.

Once Z is available, either from the received data packet or from `flightPathExtrapolationPrepare2rv`, the algorithm can calculate vectors r2 and v2, implemented in `flightPathExtrapolationExecute`.

The above functions store input, temporary and output data in the `TFlightPathExtrapolationData` data structure declared in FlightPathExtrapolation.h.

3.3 Generating collision warnings

Generating the collision warnings is implemented in the Prediction.c source code file. The main entry point is the function `predictionCalculateAlarmStates`. This function performs the following two steps:

1. Extrapolate the flight path of the own aircraft, in `predictionExtrapolateOwnFlightPath`.
2. Loop through the list of surrounding aircraft. For each aircraft, it extrapolates in discrete time steps of 1 second and with a depth of 30 seconds (configurable in constant `T_MAX_SEC`) the position and velocity of that aircraft. It invokes the function `predictionCalculateAlarmStateForFlightObject` which calculates the distance between the own and the other aircraft. It applies the alarm criterion to check if a collision warning should be generated and what alarm level should be assigned to the warning.

The following graphic shows the distance threshold as a function of extrapolation depth in seconds (horizontal axis) and velocities vA and vB of the two aircraft:

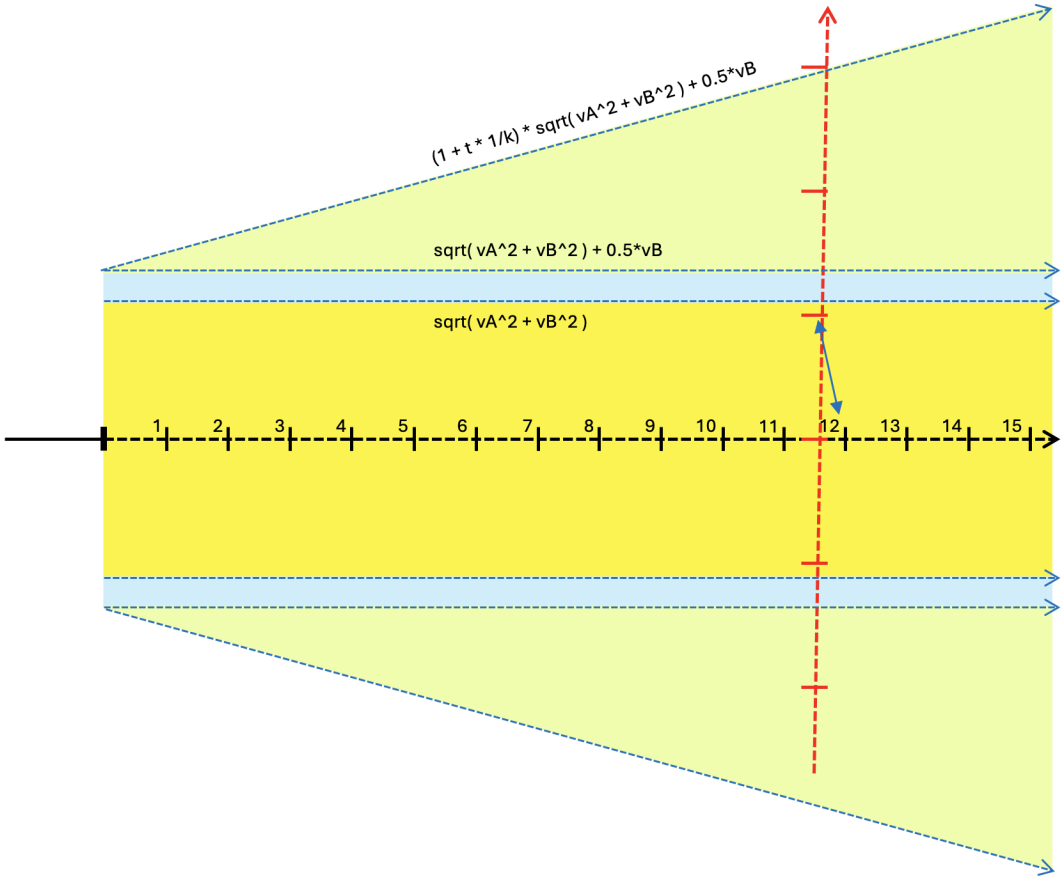


Figure 10: Alarm distance threshold as a function of extrapolation depth and velocities.

All alarms are collected in an alarm state list, which is ordered by severity of the alarm. The data type `TAlarmState`, declared in AlarmState.h, describes the properties of an alarm, such as level (1 = low, 3 = high), time to encounter in seconds, involved aircraft and distance to the aircraft.

It's up to the device manufacturer if only a single alarm should be presented to the user (e.g. with a sound signal), or if multiple alarms should be presented (e.g. on a screen).

4 SIMULATION ENVIRONMENT AND TEST DATA

The OCAP code repository contains a simulation environment (in short, the "simulator") and recorded test data which can be used to test and analyze the implementation. The simulator was developed and tested in Linux Ubuntu 24.04. It should also work on other Linux variants and on macOS if the developer tools and necessary dependencies are installed.

It is possible to run the simulator in two different modes. **Interactive mode** is used to analyze a specific test flight or calculation. **Automatic mode** is used to execute test cases with no user interaction, typically as part of regression testing the libraries after changing the implementation.

4.1 Simulation environment

4.1.1 Compiling the simulation environment

The simulation environment is written in C and C++.

Invoke the Makefile in the src/sim subdirectory to build the **sim** executable.

The simulator has dependencies on libadsl, libocap and on OpenGL. Make sure to compile libadsl and libocap first by invoking the Makefile in their respective directories and to install OpenGL in your system.

4.1.2 Running the simulator in interactive mode

To start the simulator in interactive mode, you need to invoke the **sim** binary with a test case file (.tst) as command line parameter. For example, if the current working directory is the root directory of the source code package, you can invoke the simulator like this:

$ ./src/sim/sim testflights/20250408_grenchen/test_grenchen1.tst

A large number of test case files is available in the testflights subdirectory of the source code package. Of course, you can also create new test cases manually or by importing test flight paths with the importer tool (see section 5.1).

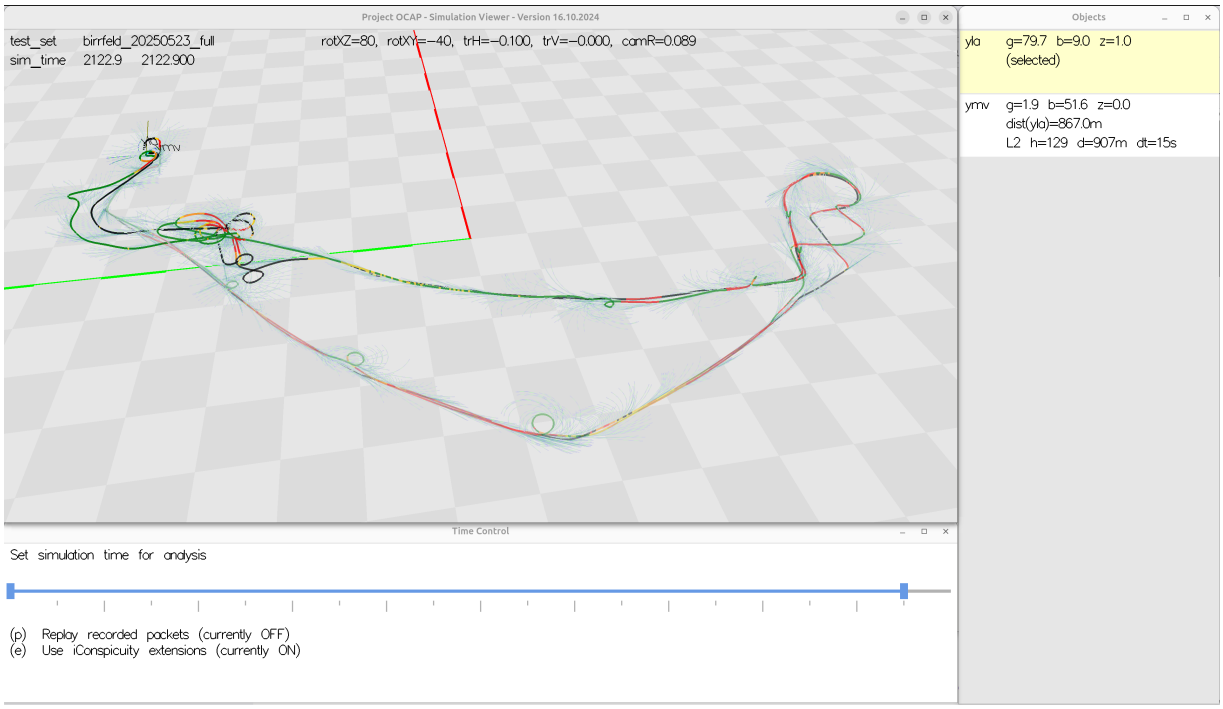


Figure 11: OCAP simulation environment

The GUI of the simulation environment consists of 3 windows:

The **main window** is a 3D representation of the simulated world. At the top left, you see the loaded test case and the time offset into the simulation, in seconds. All loaded aircraft appear in the view, annotated with their name as defined in the test case file. If the window has the focus, you can use the cursor keys to rotate (or move, if the shift key is pressed) the view, and the + and - keys to zoom in or out.

The **simulation time window** below the main window lets you set the simulation time with the slider control. Drag the slider buttons with the mouse to set the start and end of the displayed flight path for each aircraft in the main window. After selecting a "main" aircraft in the object window, you can press the "space" key to start and stop the collision detection calculation. Use the right arrow cursor to advance the time by 0.1 (or 1.0 with shift pressed) seconds With the "e" key, you can enable or disable the use of OCAP Z vector extensions. With the "p" key, you can enable and disable replaying recorded ADS-L packets instead of generating packets in the simulator. For this to work, the test case directory needs to contain log files with recorded packets.

The **object window** at the right side of the screen contains a list of loaded aircraft. Each list entry contains the identifier of the aircraft as well as the ground speed, bearing and vertical speed, on the first line. If the collision prediction is running, the second line contains the distance to the main aircraft, in meters, at the current point in time, and the third line contains information about a potential collision warning alarm message caused by the corresponding aircraft. Alarm messages range from level 1 (lowest) to 3 (most critical).

4.2 Test data

This section describes the two custom file formats used by the simulation environment and the included test cases.

4.2.1 Flight path files

Flight path files (.flp) define for a single aircraft the longitude and latitude in degrees as well as the altitude in meters at the start of every second. They are similar to IGC files, but unlike IGC files, no data points are omitted. Lines starting with # are comment lines.

The following is an example for the content of a flight path file:

```
#Title     : 1CB
#Source    : 1CB.flp
#Aircraft  :
#ResSec    : 1.0
#OriginSec : 29971.000000
#OriginLoc : 7.412250;47.182182;429.000000
7.4122486;47.1821823;428.8
7.4122486;47.1821823;428.7
7.4122491;47.1821823;428.5
7.4122496;47.1821823;428.3
7.4122500;47.1821823;428.1
```

A simple importer tool is provided in the source code directory to convert IGC files to flp format, see section 5.1. The importer applies a spline function to the IGC data to derive a smooth flight path.

4.2.2 Test case files

Test case files (.tst) group a set of flight path files together. They define a time range (start and end time) for automated testing and contain various metadata to configure the viewer and information about the test itself.

If a set of flight paths contain more than one test case, it's possible to reference the flight path files from more than one test case file with different start and end times for the automated test run.

It's also possible to combine otherwise unrelated flight path files in a test case. In this case, it may make sense to add shifting information in time (dt) and location (dx, dy, dz) on the line where the flight path is imported to construct the desired situation.

The following is an example for the content of a test case file:

```
#
# Project OCAP
# Simulation Test Case
#

# File format version
2

# Test case name
grenchen_20250514_1

# Flight path directory
testflights

# Start and end time of the data set (seconds)
0; 6130

# Viewer configuration
rotXZ=160
rotXY=-50
trH=0.2
trV=-0.0
camR=0.074

# Start and end time for automated test runs (seconds)
1184; 1381

# Objects
# name; dt; dx; dy; dz; file

C1F;       0;   0; 0; 0; 20250514_grenchen/C1F.flp
68D;       0;   0; 0; 0; 20250514_grenchen/68D.flp
```

Note that the object name defined in the test case file for each flight path is used to annotate the corresponding aircraft in the simulation environment.

4.2.3 Included test cases

Several real-world flights have been designed and executed during the project by professional pilots to test the performance of the OCAP algorithm. Test cases derived from these flights are available in the repository, in the "testflights" directory.

For regression testing, it is recommended to execute all recorded test sets:

20250110_birrfeld includes 8 test cases from two test flights with motorized airplanes.

20250222_birrfeld includes 6 test cases from two test flights with motorized airplanes.

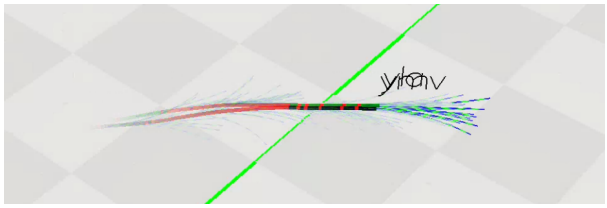
20250408_grenchen includes 16 different test cases.

20250514_grenchen includes 3 test cases with frontal approaches and 6 test cases with approaches from behind.

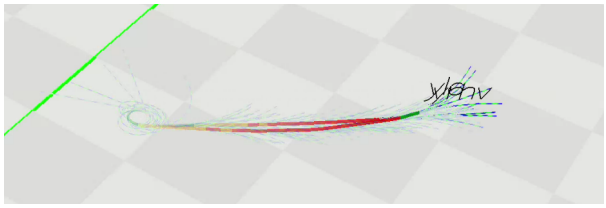
20250523_birrfeld includes 4 test cases with rear-end approaches, 2 test cases with right-angle approaches, 2 test cases with loopings and approaches on diving down, one lateral offset flight, one pursuit without approaching and 4 test cases to test the arc prediction model.

The test sets include a single set of flight path (flp) files and, for each test case, one test case (tst) file. The test case file defines the start and end time of each test case within the flight path files and optimizes the viewer configuration.

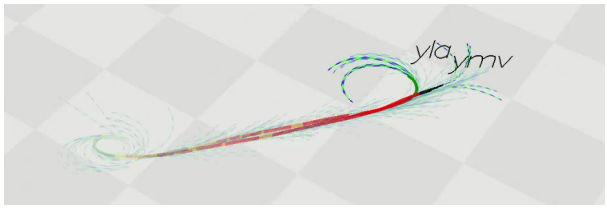
For example, the 20250523_birrfeld test set includes these test cases.



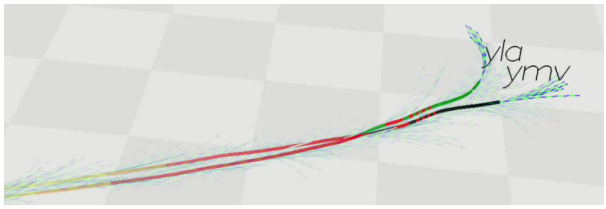
Test case 1: Rear-end collision (1 of 4)



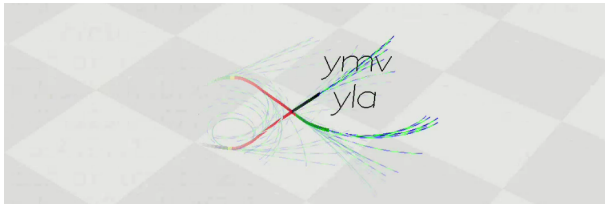
Test case 2: Rear-end collision (2 of 4)



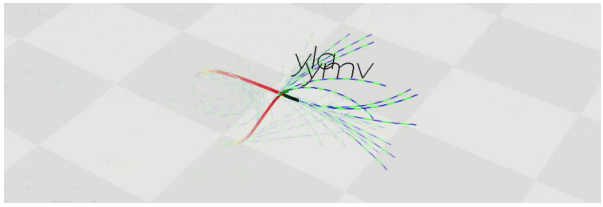
Test case 3: Rear-end collision (3 of 4)



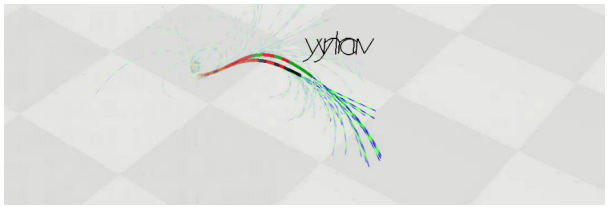
Test case 4: Rear-end collision (4 of 4)



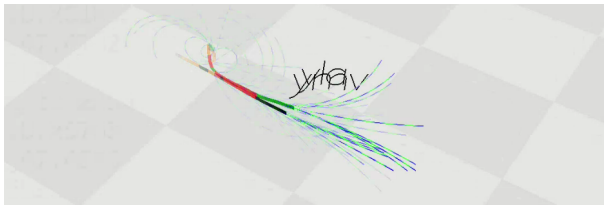
Test case 5: Right-angle approach (1 of 2)



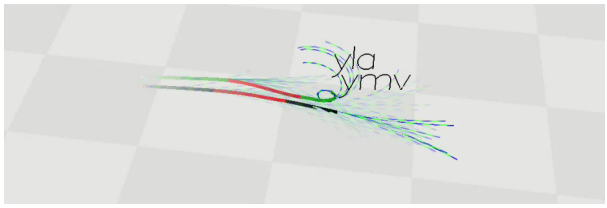
Test case 6: Right-angle approach (2 of 2)



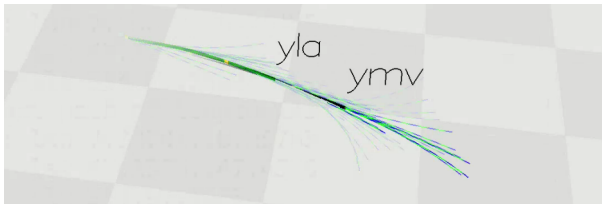
Test case 7: Looping, approach when coming down (1 of 2)



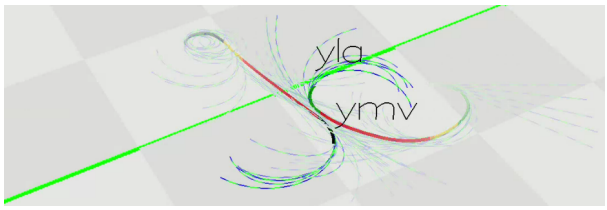
Test case 8: Looping, approach when coming down (2 of 2)



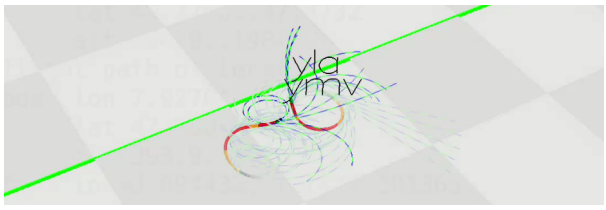
Test case 9: Laterally offset flight



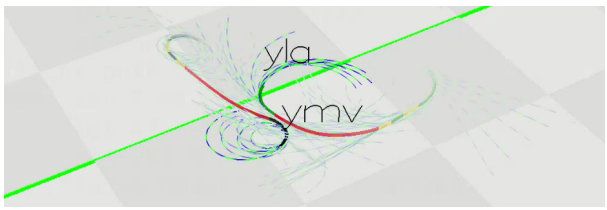
Test case 10: Pursuit without approaching



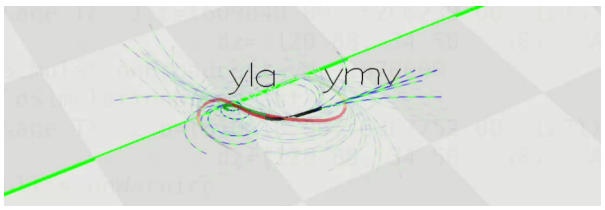
Test case 11: Arc with collision course (1 of 4)



Test case 12: Arc with collision course (2 of 4)



Test case 13: Arc with collision course (3 of 4)



Test case 14: Arc with collision course (4 of 4)

To automatically execute a test set, for example 20250523_birrfeld, you can invoke the corresponding Make target in the root directory, as follows:

$ make birrfeld_20250523

In this case, the make tool will invoke each of the 14 test cases of the birrfeld_20250523 test set in sequence. For each test case, the simulator will open, load and execute the test case, save the result and quit.

To automatically execute all test sets, invoke make without target:

$ make

To automatically execute only a single test case, for example test_birrfeld_1.tst, you can invoke the simulator from the root directory with the following command:

$ ./src/sim/sim --autorun testflights/20250523_birrfeld/test_birrfeld_1.tst output/result_20250523_birrfeld_1.txt

The output of an automated test run is a sequence of collision warnings with detailed information about the simulation time, alarm level, distance, time to collision and relative orientation of the aircraft.

A software developer can check the effect of changes on the library code by running the automated test cases and verifying the differences in the output files.

5 SUPPORT TOOLS

5.1 Import utility

For each aircraft that should be simulated, the simulator needs an ".flp" flight path input file. It can't directly process the classic IGC files which are normally used to record flights. The import utility converts from IGC to ".flp" by using spline interpolation to recover a smooth path. The spline function is then sampled at constant time intervals.

The following illustration shows the input from a typical IGC file at the left side and the derived spline function at the right side:

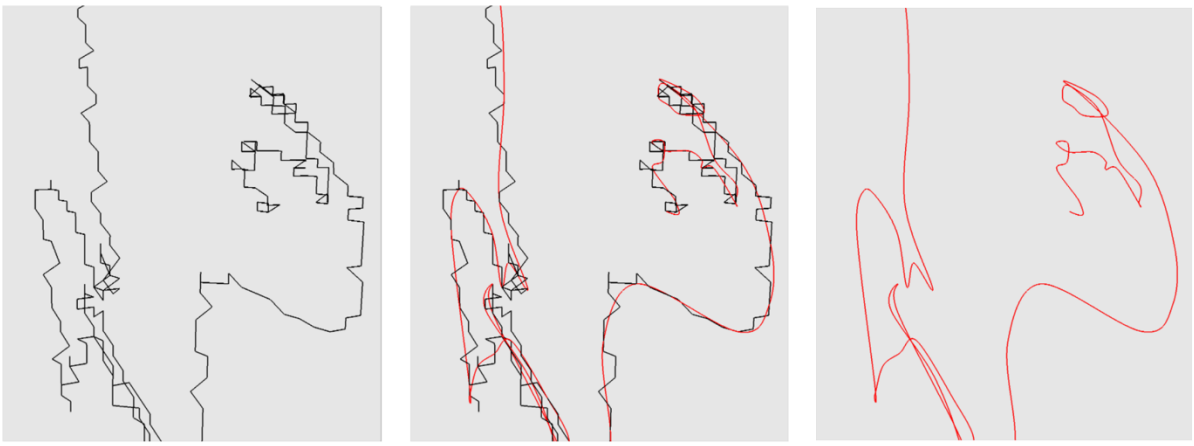


Figure 12: The importer utility uses splines to recover a smooth flight path from information in IGC files.

The source code of the importer utility is provided in the src/tools/importer directory. Invoking the make file generates the importer binary which can then be used like this:

$ ./importer <IGC-File> <aircraft-name> <flp-file>

The tool stores flight path information into the specified <flp-file>.