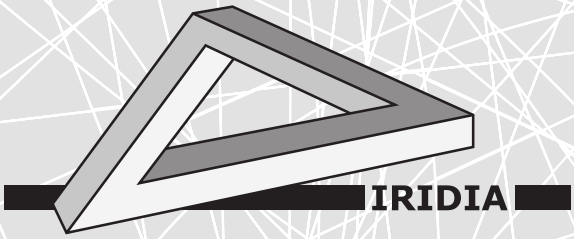


**Université Libre de Bruxelles**

*Institut de Recherches Interdisciplinaires*
*et de Développements en Intelligence Artificielle*

# Software support for the IRIDIA drone

M. Allwright

**IRIDIA – Technical Report Series**

Technical Report No.
TR/IRIDIA/2022-002

March 2022
Last revision: October 2022

**IRIDIA – Technical Report Series**
ISSN 1781-3794

Published by:

IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2022-002

Revision history:

| | |
|---|---|
| TR/IRIDIA/2022-002.001 | March 2022 |
| TR/IRIDIA/2022-002.002 | October 2022 |
| TR/IRIDIA/2022-002.003 | October 2022 |

# Table of contents

# Introduction

This technical report covers the software support for the IRIDIA drone. This report has been generated in part from the documentation online in the following repositories. If you find any errors in this report, please open an issue in the appropriate repository so that this report can be kept up to date.

1. ARGoS Plugin for the IRIDIA drone
2. The meta-drone layer for the Yocto build system

# ARGoS Plug-in for the IRIDIA drone

The IRIDIA drone is supported in the ARGoS simulator. The plugin in the ARGoS repository defines the control interface and provides an implementation of the drone in simulation. This implementation provides a visualization for the `qt-opengl` plugin and a model for the `pointmass3d` physics engine plugin.

The control interface defined in the ARGoS repository is also used on the real robot. The plugin for the real robot is available in the argos3-drone repository. This plugin can compiled as is, however, it is advisable to instead follow the instructions in the meta-drone repository, which provides a step-by-step guide on how to set up a Docker image that builds a complete, bootable, USB stick for the drone that includes support for ARGoS.

## Usage

A drone can be added to the ARGoS simulator by adding a `drone` tag under the `arena` section of the experiment configuration file as follows:

```
<drone id="drone">
  <body position="0,0,0" orientation="0,0,0"/>
  <controller config="my_controller"/>
</drone>
```

### Controllers

In the example above, we referenced a controller called `my_controller`, which must be declared in the experiment configuration file under the `controllers` section, for example:

```
<lua_controller id="my_controller">
  <actuators/>
  <sensors />
  <params />
</lua_controller>
```

This controller defines a controller written in Lua, whose script can be passed using the `script` attribute in the `params` tag. The controller cannot do much at the moment since it does not declare any sensors or actuators.

### Physics Engines

To instantiate a drone in ARGoS, an instance of the Point-mass 3D physics engine must be declared. It should be noted that this engine does not support collisions with other objects in the simulation.

### Media

Media in ARGoS are the indices in which different entities can be looked up based on their current location. These indices predominately enable the sensors of a robot to look up entities of interest. For example, by placing tags or LEDs in their respective indices, they can be detected by a camera on another robot.

The drone's simulation model in ARGoS contains LEDs and a crude approximation of wifi whereas messages are simply broadcasted to all nearby robots. To enable the drone's LEDs to be seen by another robot, the attribute `led_medium` must be set to a `directional_led` medium declared in the `media` section of the experiment configuration file. Likewise, to enable the sending and receiving of the messages, the attribute `wifi_medium` must be set to a `simple_radio` medium. The following XML provides an example of the required configuration:

```
<arena size="1, 1, 1" positional_index="grid" positional_grid_size="1,1,1">
  <drone id="drone" led_medium="leds" wifi_medium="wifi">
    <body position="0,0,0" orientation="0,0,0"/>
    <controller config="my_controller"/>
  </drone>
</arena>

<media>
  <directional_led id="leds" index="grid" grid_size="5,5,5" />
  <simple_radio id="wifi" index="grid" grid_size="5,5,5" />
</media>
```

Note that the LEDs can still appear in the visualization and be controlled without `led_medium` being specified. By contrast, not specifying `wifi_medium` will result in the simple radio (used for approximating wifi) not being added to the robot.

## Sensors

The following section lists the sensors that are specific to the drone. There are also a few generic sensors (e.g., `simple_radios` for wifi) that can be added. These sensors are in part discoverable by running `argos3 -q sensors` in a terminal. The examples below make use of the Lua controller in ARGoS which allows writing control software in the form of a Lua script. The advantage of writing controllers in Lua is that there is no need for (cross-)compilation or linking. That being said, all sensors and actuators do provide a C++ interface for which the code itself is the documentation.

### drone_cameras_system

This sensor will detect tags underneath the drone's four cameras. The following configuration adds this sensor to the drone.

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <drone_camera_system implementation="default"
      tag_medium="tags"
      show_tag_rays="false"
      show_frustum="false" />
  </sensors>
  <params />
</lua_controller>
```

The attributes `show_tag_rays` and `show_frustum` default to false, however, the attribute `tag_medium` must be supplied and be an index in which the camera should try and find tags. By default the cameras are disabled and need to be enabled as per the following Lua code:

```
function init()
  for id, camera in pairs(robot.cameras_system) do
    camera.enable()
  end
end
```

The detected tags can be iterated over and printed to the ARGoS log as follows:

```
function step()
  for i, camera in pairs(robot.cameras_system) do
    log('camera ' .. camera.id)
    for j, tag in ipairs(robot.front_camera.tags) do
      log('  tag ' .. tag.id)
      log('    position = ' .. tag.position)
      log('    orientation = ' .. tag.orientation)
      log('    corners')
      for k, corner in ipairs(tag.corners) do
        log('      ' .. corner)
      end
    end
  end
end
```

The type of `tag.position` is a `vector3` and its units are in meters. The type of `tag.orientation` is a `quaternion`. The type of the tag corners is `vector2` and the units are in pixels.

### drone_flight_system

This sensor represents the input from a Pixhawk device. The sensor can be added to the `sensors` section of a controller as follows:

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <drone_flight_system implementation="default" />
  </sensors>
  <params />
</lua_controller>
```

This configuration shows the sensor being added to a Lua controller. Several parameters can be read back from the Pixhawk device (either simulated or real) as shown in the following Lua code:

```
function step()
  log('position = ' .. robot.flight_system.position)
  log('orientation = ' .. robot.flight_system.orientation)
  log('target_orientation = ' .. robot.flight_system.target_orientation)
  log('velocity = ' .. robot.flight_system.velocity)
  log('angular_velocity = ' .. robot.flight_system.angular_velocity)
  log('target_position = ' .. robot.flight_system.target_position)
  log('height = ' .. robot.flight_system.height)
  log('battery_voltage = ' .. robot.flight_system.battery_voltage)
  log('temperature = ' .. robot.flight_system.temperature)
end
```

### drone_rangefinders

The rangefinders sensor is used to detect nearby obstacles around the drone. The following configuration demonstrates how to use this sensor in a controller.

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <drone_rangefinders implementation="default" />
  </sensors>
  <params />
</lua_controller>
```

The following code demonstrates how to use the readings from the sensor in Lua by printing the current readings to the ARGoS log. The following code also shows how to get the position and orientation of each sensor relative to an anchor (a local coordinate system) on the robot.

```
function step()
  for i, sensor in ipairs(robot.rangefinders)
    log('rangefinder ' .. i .. ':')
    log('  proximity = ' .. sensor.proximity)
    log('  illuminance = ' .. sensor.illuminance)
    log('  transform:')
    log('    anchor = ' .. sensor.transform.anchor)
    log('    position = ' .. sensor.transform.position)
    log('    orientation = ' .. sensor.transform.orientation)
  end
end
```

### drone_system

The system sensor is used to report the current time and temperature of the CPU. It can be added to a controller as follows:

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <drone_system implementation="default" />
  </sensors>
  <params />
</lua_controller>
```

These values are available from a Lua controller and can be printed to the ARGoS log.

```
function step()
  log('time = ' .. robot.system.time)
  log('temperature = ' .. robot.system.temperature)
end
```

## Actuators

The following section lists the actuators that are specific to the drone. There are also a few generic actuators (e.g., `simple_radios` for wifi) that can be added. These actuators are in part discoverable by running `argos3 -q actuators` in a terminal.

### drone_flight_system

The drone flight system enables configuring/writing data to a Pixhawk device:

```
<lua_controller id="my_controller">
  <actuators>
    <drone_flight_system implementation="default" />
```

```
  </actuators>
  <sensors />
  <params />
</lua_controller>
```

This actuator exports the following functions to Lua:

| Function | Description |
|---|---|
| `ready` | Returns a boolean when the Pixhawk is active |
| `set_armed` | Takes a boolean to configure whether or not the drone is armed |
| `set_offboard_mode` | Takes a boolean to configure whether or not the drone is in off-board mode |
| `set_target_pose` | Takes a vector3 and a number that describes the position and orientation to which the drone should fly to with respect to where it took off from |

```
function step()
  -- take off and hover at one meter above the ground
  if robot.flight_system.ready() then
    robot.flight_system.set_offboard_mode(true)
    robot.flight_system.set_armed(true)
    robot.flight_system.set_target_pose(vector3(0,0,1)), 0)
  end
end
```

### drone_leds

The LED actuator sets the colors of LEDs on the drone. It can be added to a controller as follows:

```
<lua_controller id="my_controller">
  <actuators>
    <drone_leds implementation="default" />
  </actuators>
  <sensors />
  <params />
</lua_controller>
```

In the Lua, the following functions are exported by the actuator:

| Function | Arguments |
|---|---|
| `set_led_index` | The index (integer between 1..4) and the color (either a string or three RGB colors) |
| `set_leds` | the color (either a string or three RGB colors) |

For example, the following code turns off all LEDs on the drone except for LED group with index 1 which is set to red.

```
function step()
  robot.leds.set_leds('black')
  robot.leds.set_led_index(1, 'red')
end
```

# The meta-drone layer for the Yocto build system

## Description

The meta-drone repository contains a layer for the Yocto build system, which generates a complete, bootable Linux OS ready to be run on the IRIDIA drone. This system comes preinstalled with:

- ARGoS3 and a plugin for the drone
- Python3

## Quick start

To ensure reproducible builds on systems with varying configurations, the following steps will explain how to create an image using Docker. Note that you will probably need to use sudo or to switch to a root user to install Docker or to create and run its containers.

To get started, you first need to decide where you want to build the system for the drone. Keep in mind that while the resulting image will be approximately 1 GB in size, the build system itself will require around **50 GB** of free disk space. The following steps will set up the build system. These steps assume that the build location is `/home/$(id -un)/yocto-drone` where `$(id -un)` will be replaced with the current username.

```
# Create a directory for the build system
mkdir /home/$(id -un)/yocto-drone
```

We now need to clone the layers for the build system as follows:

```
# Switch to the build location
cd /home/$(id -un)/yocto-drone
# Clone the Yocto repository
git clone git://git.yoctoproject.org/poky \
 --branch zeus --single-branch
# Clone additional layers inside the Yocto repository
cd poky
git clone git://git.openembedded.org/meta-openembedded \
 --branch zeus --single-branch
```

```
git clone git://git.yoctoproject.org/meta-intel.git \
 --branch zeus --single-branch
git clone https://github.com/iridia-ulb/meta-drone.git
```

## Create the Docker image

The following command will execute the Dockerfile in the meta-drone repository and create a Docker image based on Ubuntu 18.04 LTS. The image will contain a user and a group, which match the identifiers of current user and group. Setting the user and group in this way enables trivial access to the build system from the host.

```
sudo docker build -t yocto-drone:latest \
 https://github.com/iridia-ulb/meta-drone.git#:docker \
 --build-arg host_user_id=$(id -u) \
 --build-arg host_group_id=$(id -g)
```

## Create the Docker container

Once the above command has completed successfully, you can run the following command to create a container from the image. Note the two paths given after the `-v` option. The format of this argument is `path/on/host:path/in/container` where `path/on/host` is a directory on your host system and `path/in/container` is a directory inside the Docker container. This command will map the home directory inside the container to a directory called `yocto-drone` under the current user's home directory on the host.

```
sudo docker create --tty --interactive \
 --volume /home/$(id -un)/yocto-drone:/home/developer \
 --name yocto-drone --hostname yocto-drone yocto-drone:latest
```

After executing this command, you should have a new container with the build environment. The following commands will start and attach to that container.

```
sudo docker start yocto-drone
sudo docker attach yocto-drone
```

## Start the build

After following the steps above, you should have a terminal that is attached to the docker container and be inside a directory called `build`. To build the entire image for the drone, just run the following command:

```
bitbake upboard-image-base
```

Occasionally, the build can fail due to internet connectivity issues or due to an oversight in the dependency tree. These issues are normally resolved by just re-executing the command above.

## Copying the image

The most straightforward way to copy a bootable image to the USB stick is to use the `dd` utility available on most Linux systems. Before using `dd` to transfer the image, check the kernel messages (via, e.g., `sudo dmesg`) after attaching the USB stick and carefully note the device name of the USB stick. It usually takes the form of `/dev/sdX`. To write the image to the device, execute the following commands replacing `sdX` with the device you noted while inspecting the kernel messages:

```
# unmount the device and/or its partitions
umount /dev/sdX*
# write the image to the device
dd if=PATH/TO/upboard-image-base-up-core.hddimg of=/dev/sdX
# run sync to ensure that all data has been copied
sync
# unmount the device and/or its partitions again (in case they were mounted)
umount /dev/sdX*
```

## Booting the image and accessing the console

The easiest and most reliable way to get access to the drone console is by using a USB keyboard and HDMI monitor. These peripherals can be directly connected to the Up Core board.

*WARNING: Do not use the DC power jack while the Up Core is connected to the drone PCB.*

## Installation

After testing and debugging is complete, it is recommended to install the operating system onto the internal flash storage. This can be achieved by booting from the USB stick, selecting the install option from the boot menu, and accepting all the default options/prompts. If the drone is not booting from the USB stick and is instead booting an old installation, try to change the boot order in the BIOS. By installing the OS and booting from the internal storage, an additional USB port is made available which, for example, could be used to save footage from the drone camera system.

## BIOS configuration

This layer has been tested with the following BIOS version: `UP_CHR1 R1.8 (UCR1BM18) (04/12/2019)`. If this is not the current BIOS version, it is recommended that you downgrade/upgrade your BIOS following the instructions on the Up Community Wiki.

## Wifi configuration

The wireless connection is controlled using the `iwctl` command. This interactive command makes the process of connecting to a wireless network relatively painless. Once you are connected, the wireless network is saved on the drone under `/var/lib/iwd/SSID.KEY_TYPE`. By

default, the drone will connect to a network with the name `MergeableNervousSystem` using PSK authentication with the password `uprising`. The network should automatically connect on boot and fetch an IP address using DHCP.

# Acknowledgements

This work was supported by the Program of Concerted Research Actions (ARC) of the Université libre de Bruxelles and by the Office of Naval Research Global (Award N62909-19-1-2024).