X-Copter Studio

User documentation

2014-09-03

Table of Contents

Introduction....4
- Terms and abbreviations....4
- Typography....4

Installation....5
- Hardware requirements....5
- Recommended third party software....5
- Windows....5
- Linux....6

Launching....6
- Server....6
  - Windows....6
  - Linux....6
- Onboard....6
  - Windows....6
  - Linux....6
- Client....6

Architecture....7
- Dataflow graph....7

Graphical user interface....8
- Main navbar....8
- Onboard navbar....9
  - Dataflow graph....9
  - Scripting console....10
  - X-copter control....11
- Widgets....12
  - Raw data....12
  - Gauge data....12
  - Progress data....13
  - Chart data....13
  - Video....13
- Single-user application....14
- Safety....14

Tutorials....16
- Play and display logged data....16
- Manual flying with x-copter....18

Nodes....19
- CheckpointMovement....19
  - Inputs....19
  - Outputs....19
  - Parameters....19
- Control....20
  - Inputs....20
  - Outputs....20
  - Parameters....20

Datalogger....20
Inputs....20
Outputs....20
Parameters....21
Dataplayer....21
Outputs....21
Parameters....21
Executor....21
Inputs....22
Outputs....22
Parameters....22
FlyControl....22
Inputs....22
Outputs....23
Parameters....23
FlyControlDual....23
Inputs....23
Outputs....23
Parameters....23
FlyControlTriple....23
Inputs....23
Outputs....24
Parameters....24
FlyControlAggregator....24
Inputs....24
Outputs....25
Parameters....25
Localization....25
Inputs....25
Outputs....26
Parameters....26
PID....26
Inputs....26
Outputs....27
Parameters....27
RedDot....27
Inputs....27
Outputs....27
Parameters....27
SemanticReceiver....27
Outputs....28
Parameters....28
XciDodo....28
Inputs....28
Outputs....29
Configuration parameters....29

XciParrot....29
Inputs....29
Outputs....30
XciVrep....31
Inputs....31
Outputs....31
Gui (special)....31
Inputs....32
Parameters....32
Configuration....33
Onboard....33
Environment....33
Settings files....33
Syntax....33
Server....33
Extending X-Copter Studio....35
New nodes for DFG....35
Writing own nodes in C++....35
Writing own nodes in urbiscript....37
Adding own nodes to X-Copter Studio....38
Nodes with generic inputs....38
General notes for node implementers....39
Implementing XCI....39
Sensors....39
Commands....39
Configuration....40
Skeleton of XCI implementation....40
Appendix....41
Syntactic and semantic types....41
Syntactic types....41
Semantic types....43
Directory structure....45
Onboard....45

Introduction

This is user documentation for X-Copter Studio. Its target readers are:

- users who interact almost exclusively with graphical user interface,
- advanced users who modifies behavior via configuration
- and user-programmers who extend X-Copter Studio through provided API.

Terms and abbreviations

Client	with capital letter denotes one of the system component
DFG	dataflow graph
FOF	frame of reference
(G)UI	(graphical) user interface
Linux	operating system with GNU/Linux kernel, based on Debian distribution
Onboard	with capital letter denotes one of the system component
Server	with capital letter denotes one of the system component
V-REP	virtual robot experimentation platform
XCI	XCS interface (unified interface for x-copters)
XCS	X-Copter Studio
x-copter	pretentious name of quadrocopters et al.

Typography

some label	Represents labels (e.g. buttons) in GUI
`string values`	Represents particular string value (e.g. paths, filenames)

Installation

Hardware requirements[1]

Server

- Optimal
  - 1 GHz processor
  - 512 MB RAM
  - 100 MB available hard disk space
  - Internet connection during installation

Onboard

- Minimum without video processing
  - 1 GHz processor
  - 512 MB RAM
  - 420 MB available hard disk space
- Optimal
  - 2.5 GHz dual core processor
  - 4 GB RAM
  - 420 MB available hard disk space

Client[2]

  - Intel Pentium 4 or later
  - 512 MB RAM
  - 350 MB available hard disk space

Recommended third party software

- Google Chrome browser
- V-REP robotic simulator

Windows

- Download from http://drones.ms.mff.cuni.cz/xcs/wiki/doku.php/download:start latest version of X-Copter Studio onboard and X-Copter Studio server.
- Go to the directory where you downloaded setup files and run both.
- During X-Copter Studio onboard installation you can choose whether developer files and documentation will be installed and during X-Copter Studio server installation you may disable Node.js installation if you have it already installed.

1 Hard disk space requirements are valid for Windows version only. Real space consumption on Linux depends on previously installed dependencies for XCS.

2 Google Chrome browser requirements.

Linux[3]

Add following lines to your `/etc/apt/sources.list`

```
deb http://drones.ms.mff.cuni.cz/xcs/packages trusty unknown
deb-src http://drones.ms.mff.cuni.cz/xcs/packages trusty unknown
```

Then run following command to install XCS with all supplied components

```
sudo apt-get update
sudo apt-get install xcs-server xcs-onboard xcs-nodes-all\
     xcs-drivers-all
```

Packages are not digitally signed so when you will be warned that packages cannot be authenticated, install them anyway.

Launching

Server

Windows

Click on "X-Copter Studio server" icon on the desktop or run bat script in location `C:\Program Files\X-Copter Studio server\run.bat`

Linux

Just run `xcs-server` command.

Onboard[4]

Windows

Click on "X-Copter Studio onboard" icon on the desktop or run bat script in location `C:\Program Files\X-Copter Studio onboard\onboard.bat`

Linux

Run `xcs-onboard` command (be sure you have the Server running)

Client

Since Client was developed as web application, launching Client is as simple as entering Servers' address with a right port into the browser (e.g. http://192.168.1.10:3000). In case of default localhost installation it is http://localhost:3000. In order for the whole XCS features to work properly it is strongly recommended to use Google Chrome browser (Chromium should also work).

3 We currently support only Ubuntu distribution, version 14.04.

4 Be sure you have the Server running. It's important for Onboard to initiate connection with a living Server.

Architecture

X-Copter Studio is not a typical desktop application neither client-server one. It comprises of three components: Onboard, Server and Client, each of which run their own process, i.e. they can be deployed on different machines.

Dataflow graph

The basic idea is the user defines behavior for the x-copter and it's executed remotely at Onboard. The behavior is expressed via dataflow graph (DFG) and custom scripts that interacts with the DFG. The cornerstone of the DFG are nodes, they are units of execution that produces new data or react to data changes (hence the dataflow name).

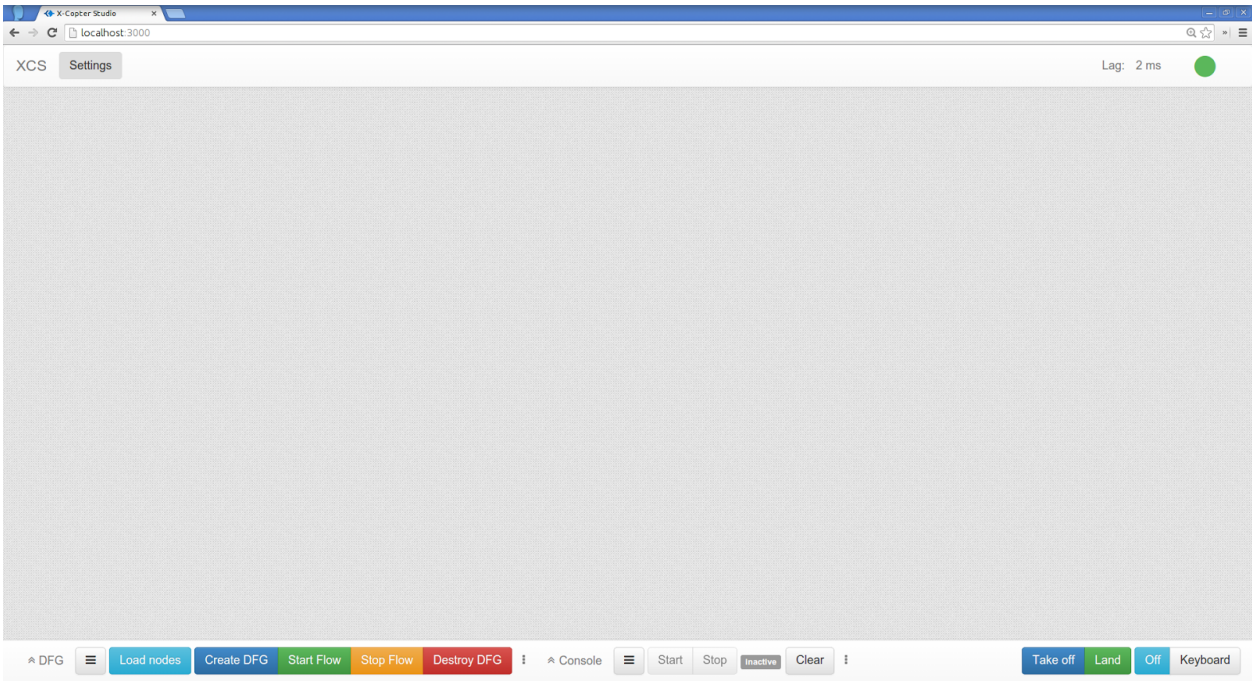
Every node can have multiple outputs as well as inputs and it's also part of the DFG how they are interconnected. Every output and input has semantic and syntactic type and output can be connected with compatible input only (typically semantic as well as syntactic type names must match).

Lifecycle of the DFG (or its nodes more precisely) is described in the section about DFG editor and further details are in the section about scripting.

Graphical user interface

Client's UI has been designed with user accessibility in mind. Therefore it was developed as a web application and can be run in browser. It was tested in Google Chrome, so it is strongly recommended to run XCS in this browser (Chromium browser should also work).

User interface consists of top main navbar, bottom Onboard navbar and operating viewport in the middle.



Main navbar

Right section of main navbar displays connection information. LED indicates following three states.

RED blinking	Server is disconnected. Possible reason for this can be that Server is down or the connection was lost due to various network failures.
RED shining	Server connected, Onboard disconnected.
GREEN shining	Onboard connected – all is good.

When GREEN led is shining, Lag indicator shows. It signals latency between Client's browser and Onboard component.

User can also be notified when latency exceeds preset latency threshold. This value can be set under Settings button on the left.

Onboard navbar

This navbar serves to operate directly on Onboard. It consists of dataflow graph panel, console panel and x-copter control section.

Dataflow graph

By default only DFG control buttons are visible and DFG editor can be shown on demand (DFG button).

DFG toolbox

In the DFG you can use nodes with predefined configurations (see Configuration), the variety of them is displayed in the toolbox. You add new instances of nodes to DFG by drag-and-drop.

The toolbox is filled upon pressing Load nodes (Reload nodes) with nodes that are available on the connected Onboard. Default DFG (nothing more than DFG named "default") is loaded together with the available nodes.

DFG lifecycle

Suppose you have designed a DFG, this DFG exists only in the Client in the editor.

After you press Create DFG the designed nodes and connections are created on onboard. It only means that nodes do exist, however they don't produce any data or react to them – this makes it easier to manipulate the graph.

Start Flow button enables nodes' operation. The order in which individual nodes are started is not defined.

Consequently, Stop Flow disables nodes' operation. Neglecting internal state of the nodes (which of course is important), DFG is similar to the situation after creation and can be started again. Stopping order is not defined neither.

When you finished your task or need to clear the onboard, use Destroy DFG. It stops the dataflow and destroys the nodes at Onboard. Onboard is then empty and DFG exists only in the editor.

Other action are collapsed into options button (it has three horizontal lines). When you want to start from scratch, use Reset DFG. It does the same like Destroy DFG. Moreover, it loads default DFG.

The lifecycle as was described above actually applies to each node separately (state of a node is indicated by color), thus you can control DFG lifecycle more finely via the context menu of a node (right click). In this menu there's also possibility to Delete node both at Onboard and in the editor.

DFG editor

Basically you design the DFG in the editor. You can connect nodes' outputs to inputs (with respect to semantic and syntactic compatibility). Particular details about connections are described for concerned nodes.

Furthermore, you can load stored DFGs (Manage DFG) and save them under custom names (Save and Save as). This concerns only design of the DFG in the editor, actual nodes are created via standard lifecycle.

Node instances naming

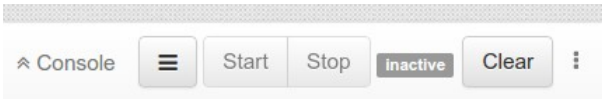
Node instances' names are assigned automatically when added to the DFG and are displayed on the node's icon. Typically name is concatenation of node configuration name (first letter is lowercase) and a counter. The counter is unique for each node in DFG toolbox and is increasing during whole session. Brief example – if you create first instance of FlyControl node, it will be named `flyControl`. If you create instance of FlyControl again, then it will be named `flyControl2` and so on until you refresh page in your browser.

Scripting console

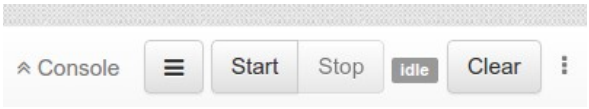
Console allows you to send urbiscript snippets to Onboard and execute them. urbiscript is a prototype based language related to JavaScript and IO language. For further details about its syntax and concepts see official Urbi SDK documentation[5].

The script is executed in its own scope but thanks to urbiscript resolution mechanism (see official documentation) nodes of the DFG are accessible as local variables.

The console cooperates with the Executor node that ensures execution within the proper environment. This means that it's necessary to have the Executor node instance in the DFG and it must be connected with Client (all this is already implemented in the default DFG). Unless this is satisfied, the console indicate this with inactive state.



Ready console is in the idle state.



Any code being executed (e.g. infinite loops, sleeps) can be paused or (forcibly) terminated by appropriate buttons – that's a feature of urbiscript. The execution is also terminated when the Executor node is stopped.

You can print to output with urbiscript `echo` function – alas, redirecting output from Onboard to Client is possible only for calls of `echo` on the top level. It means that `echo` function will not work, when you change context – e.g. context is changed inside a

5 We bundled a PDF with documentation into `doc` directory, see directory structure.

pseudoclasses. But it should not be any problem – see two following examples. First example shows change of context and second shows workaround.

```
// this will print start
echo("start");

class FooClass{
  function foo() {
    echo("foo");
  };
};

var fooClass = FooClass.new();

// this will not print anything
fooClass.foo();

// this will print stop
echo("stop");
```

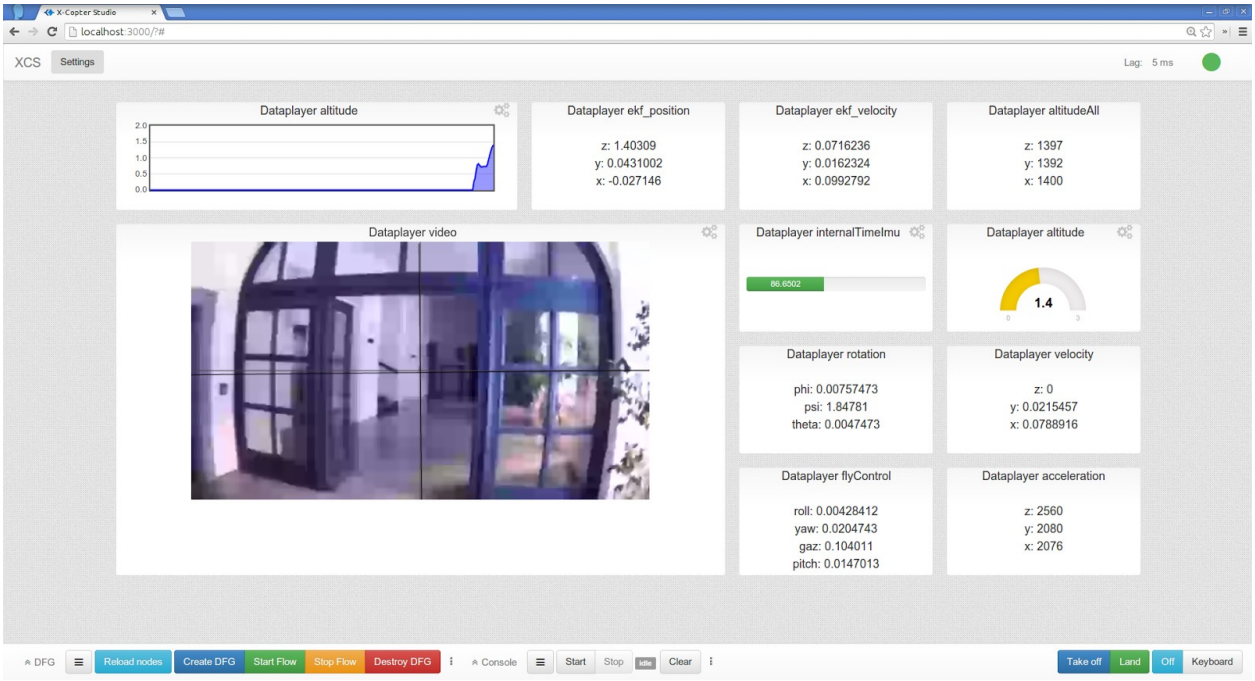
```
var globalEcho = echo;

// this will print start
globalEcho("start");

class FooClass{
  function foo() {
    globalEcho("foo");
  };
};

var fooClass = FooClass.new();

// this will print foo
fooClass.foo();

// this will print stop
globalEcho("stop");
```

You can load and store your scripts analogously to DFGs.

X-copter control

Control section contains buttons for taking x-copter off and landing as well as switch for enabling desired controller. By default, controller is turned off. In this version XCS supports only keyboard controller. Other controllers may be added in the future.
Note: Beware of typing while keyboard controller is enabled (for example when in console).

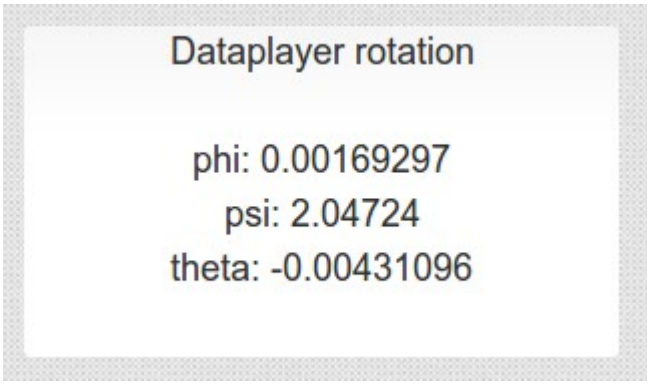
Widgets



Widgets are showed automatically according to DFG composition (see Gui node). They are presented in viewport grid and can be reordered by simple drag-and-drop operation. In top right corner widget settings can be found for some widgets. All types of widgets are described below.

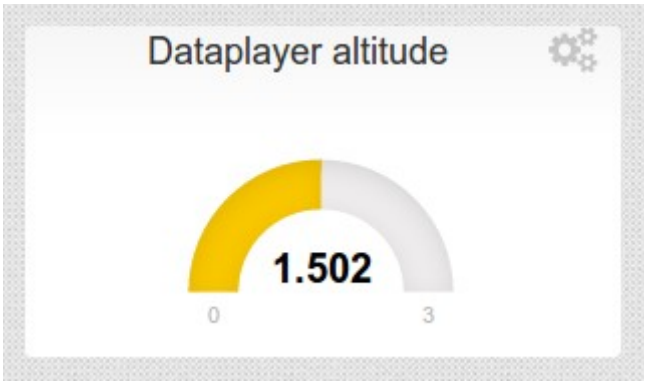
Raw data

It shows text data in widget as it arrived from Onboard. Various data like tuples, vectors, etc. can be shown in this widget.



Gauge data

Gauge widget shows rounded scale for chosen data. Minimum and maximum value for the scale can be set in widget settings. Data like velocity or float data in general can be displayed by this widget.

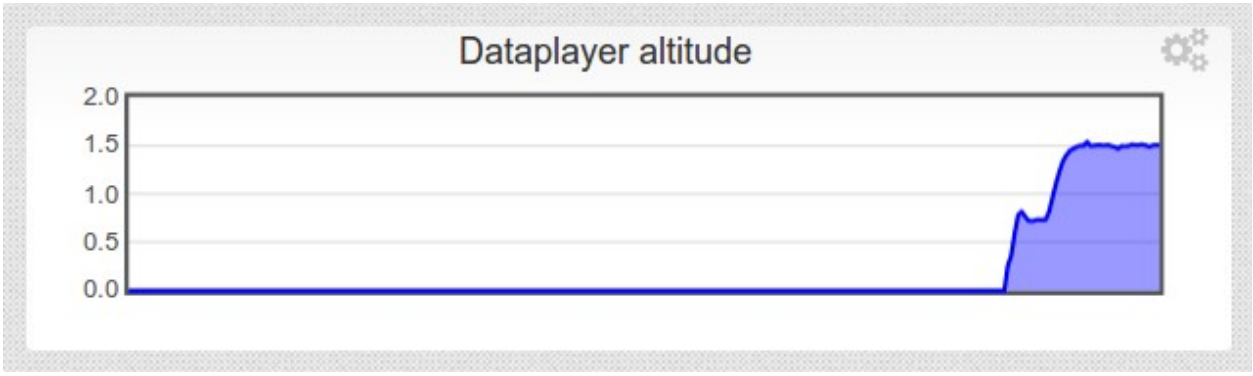


Progress data

This widget is intended to display progress data in range of 0 to maximum value preset in widget settings. Data like battery or velocity can be showed by this widget.

Chart data

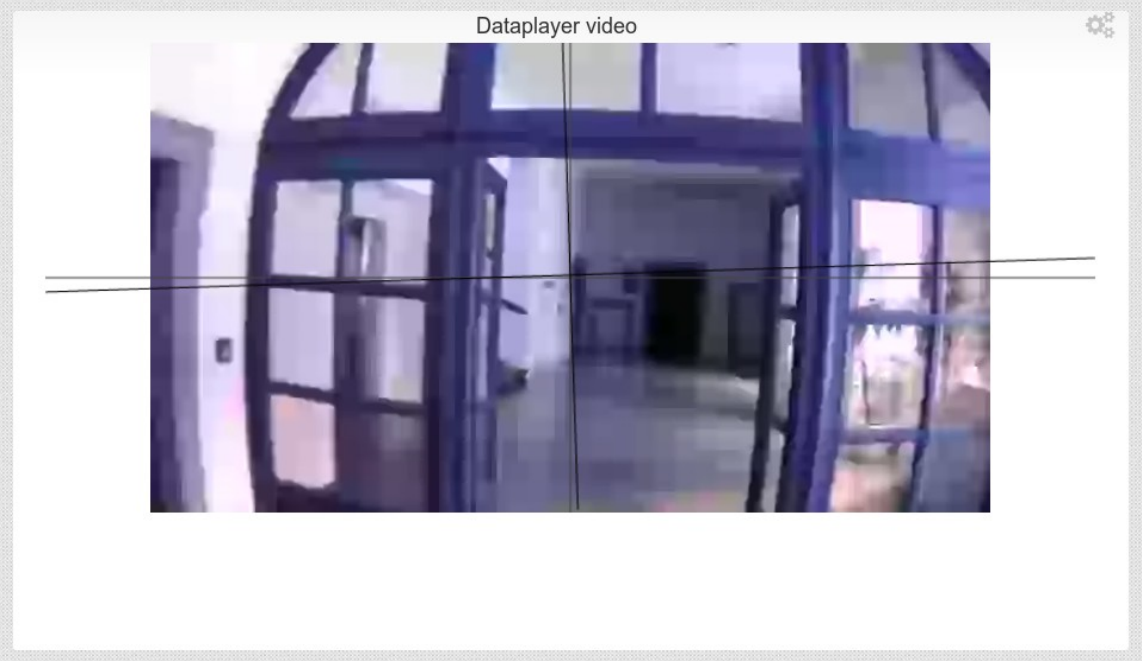
This widget displays continually updated chart according to incoming data. Chart scale can be set in widget settings. Data like altitude, battery or float data in general can be showed by this widget.



Video

This widget displays video received from Onboard. When used to display video from a front-aiming camera, very simple attitude indicator can be displayed thanks to data from inertial sensors. For this data of semantic type ROTATION must be sent to Client (for example to default input port).

In widget settings, attitude indication can be turned off as well as set pitch indication factor (coefficient between pitch angle and vertical shift of the indicator).



Single-user application

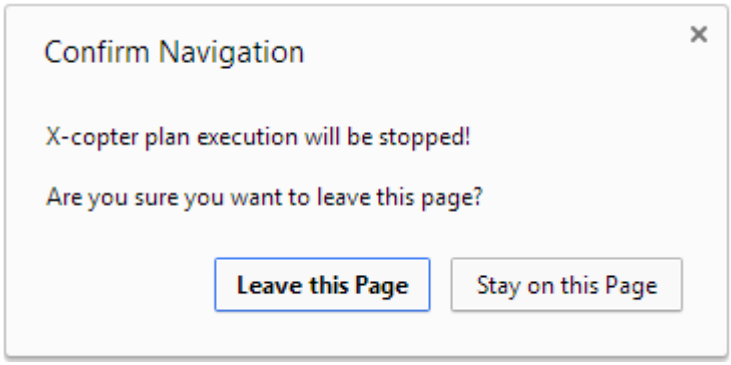
XCS is prepared just for a single user. If there are more users connecting to same server at once, access will be granted exclusively to the first of them. Others will be informed they cannot use XCS at the moment.

Safety

You can control a real x-copter (not just a virtual x-copter) with XCS, therefore XCS monitors connection latency between Client's browser and Onboard component. If there are some problems such as network failure, whether it is on Client-Server side or

Onboard-Server side, XCS will destroy DFG (execution of the user urbiscript will be stopped too).

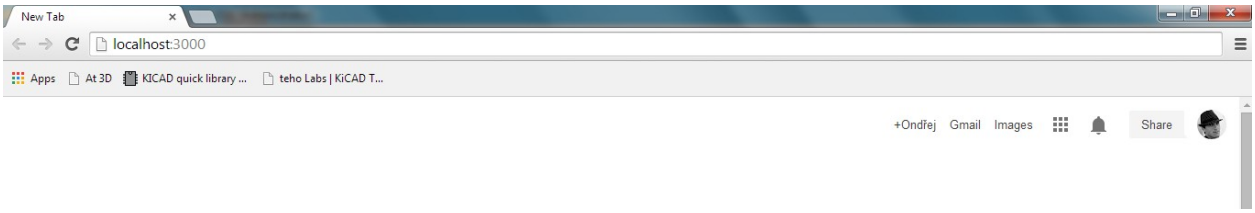
When you try to leave Client's browser site, you have to confirm this. It prevents unintentional quitting. The question text is dependent on a browser.



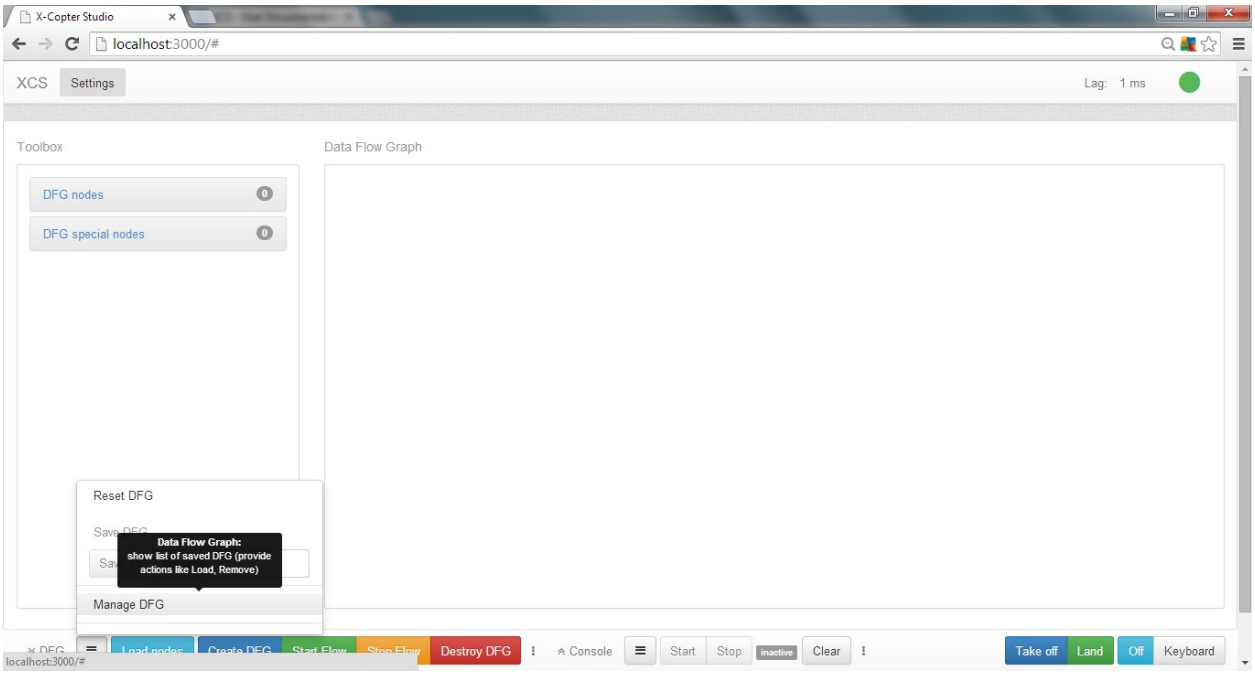
Tutorials

Play and display logged data

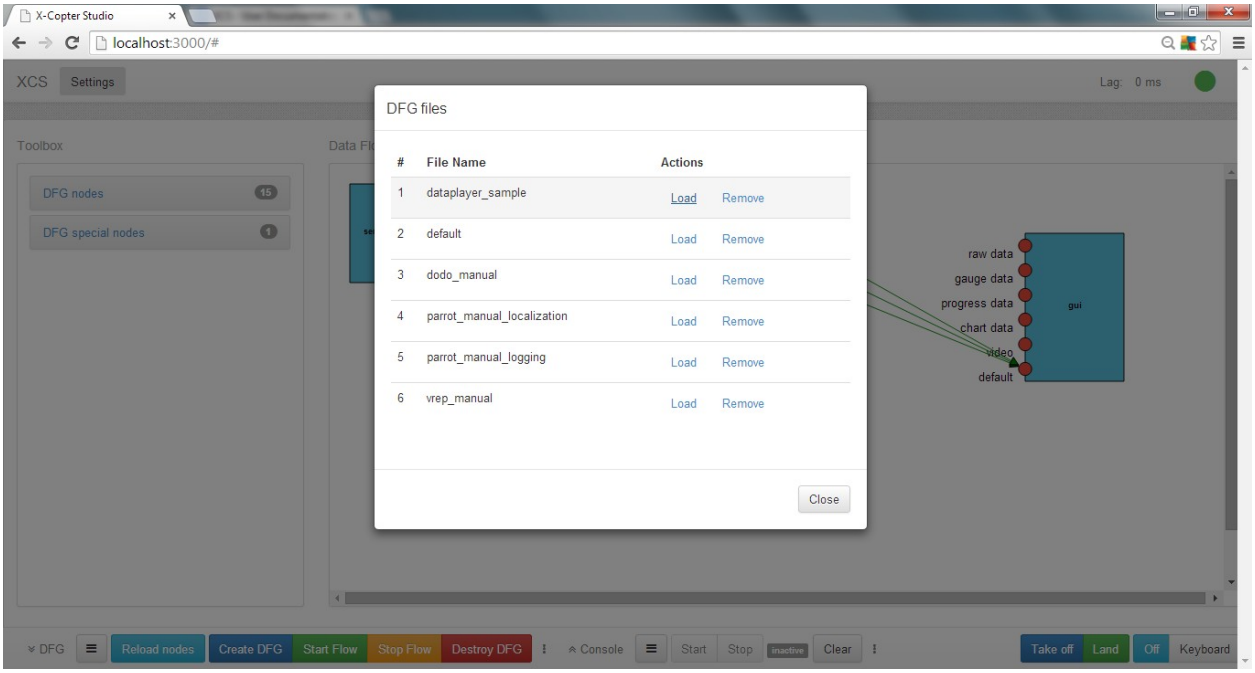
1. Launch Server and Onboard.
2. Launch Chrome browser and enter X-Copter Studio Server address.



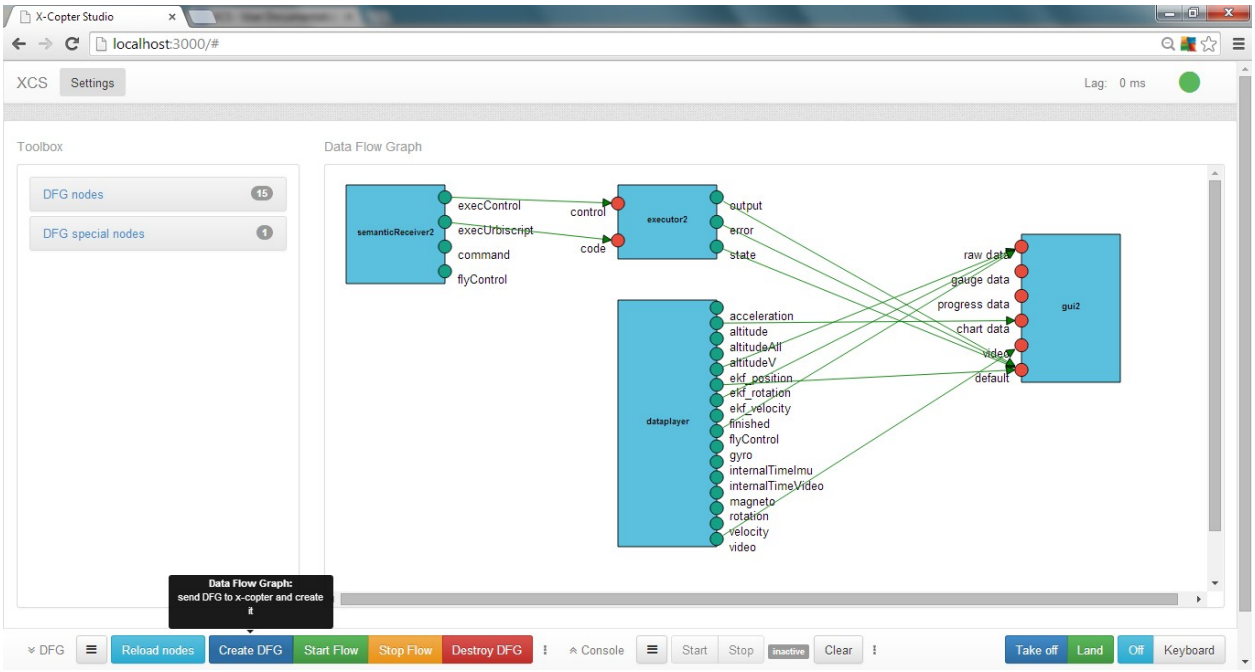
3. Load nodes from the server by clicking Load nodes button in bottom section on the page.
4. Open dataflow graph file manager by clicking Manage DFG button.



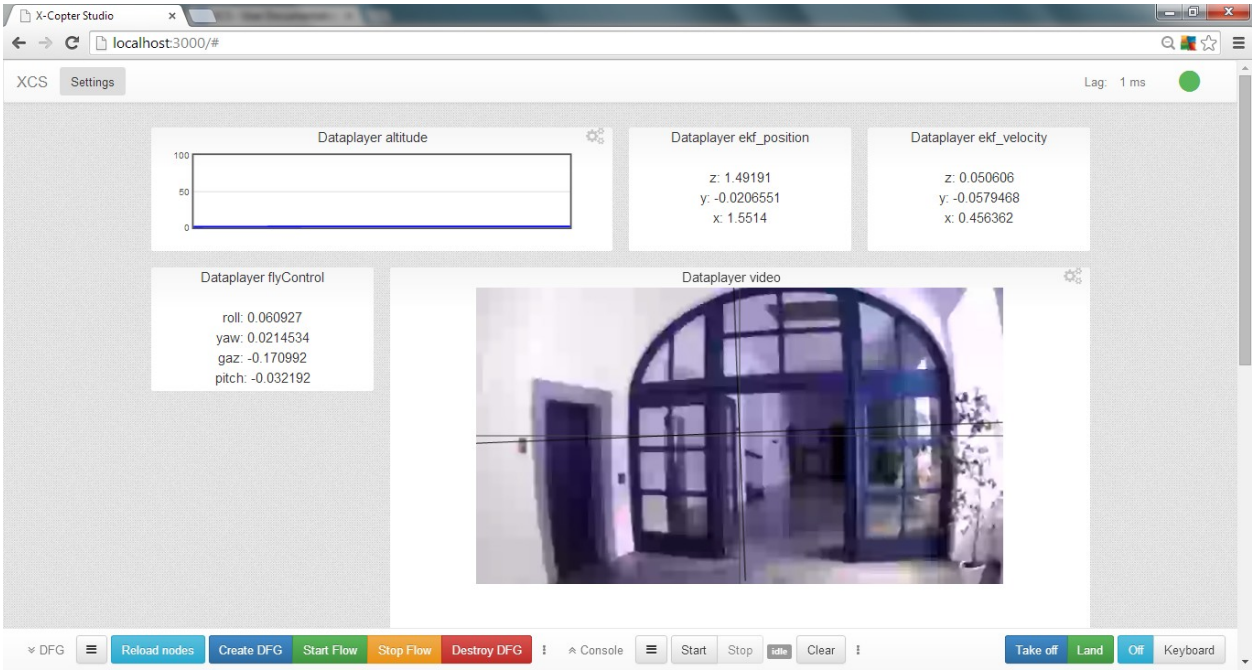
5. Load specific dataflow graph (`dataplayer_sample`) .



6. Create dataflow graph by clicking Create DFG button and then start dataflow by clicking Start Flow button.



7. Watch and enjoy replaying data ;)



Manual flying with x-copter

1. Follow instructions 1.–4. from tutorial Play and display logged data.
2. Choose one of the following DFGs: `dodo_manual`, `parrot_manual_localization`, `vrep_manual`.
3. If you chose `vrep_manual`, start V-REP simulator and prepare the scene by putting Quadricopter model into it (that's all, default name "Quadricopter" is fine).
4. Create and start DFG by clicking Create DFG and Start Flow buttons (simulation in V-REP should start).
5. Switch to Keyboard in x-copter control section.

6. Use Take off in case of `parrot_manual_localization`.
7. You can control the x-copter by cursor keys and A, S, D, W (explanation in Keyboard tooltip) and either see real x-copter moving or at least kinematics data changing(case of `dodo_manual`).

Nodes

DFG toolbox contains by default few nodes, which are described in this section. Furthermore, DFG toolbox can contain nodes from other programmers.

For general information about nodes see Dataflow graph section.

Onboard maintainer can change *Parameters* values in X-Copter Studio onboard settings directory in `xobjects.xs` file. There is no way how to change nodes parameters for common user (it means user which has no access to Onboard) – it is potentially a dangerous action.

CheckpointMovement

Generates velocity control for x-copter that depends on inserted checkpoints so that x-copter may reached all checkpoints from internal queue in the same order in which they were placed. CheckpointMovement evaluates checkpoint as reached when x-copter actual position will be in 10 cm sphere around the checkpoint.

Inputs

checkpoint (CHECKPOINT)
checkpoint structure on the input will be inserted at the end of the checkpoints' queue

command (COMMAND)
`ClearCheckpoint` – cancel actual checkpoint flight and remove all checkpoints from queue

dronePosition (POSITION_ABS)
actual x-copter position

droneRotation (ROTATION)
actual x-copter rotation

Outputs

reachedCheckpoint (EVENT)
publish true whenever actual checkpoint was reached

velocityControl (VELOCITY_CONTROL_ABS)
velocity control for x-copter in world frame

Parameters

none

Control

Receives x-copter desired velocity in all axes and headings. Depending on actual x-copter velocities and rotation compute and publish xcs::FlyControl command for the x-copter.

Control use separate PID regulators for reaching desired velocities and heading. User can change PID regulators values in Control configuration file xcontrol.xs located in xcs setting directory.

Inputs

desireVelocity (VELOCITY_CONTROL_ABS)
x-copter desire velocity

rotation (ROTATION)
x-copter actual rotation

velocity (VELOCITY_ABS)
x-copter actual velocity

Outputs

flyControl (FLY_CONTROL)
x-copter flight control

Parameters

file
Path to file with settings for xcontrol.

Datalogger

Write all registered inputs' data to a file in human readable format. Logged file has two parts – the first is a header with declaration of registered inputs (name, semantic and syntactic types) and the second contains data from the inputs with timestamp when they were logged (timestamps are relative to node's initialization instant).

Inputs

registerXVar(any semantic type)
register input for logging

Outputs

None

Parameters

file
Path to file, which will be used for logging. It is recommended to set the same file path as for Dataplayer.

Beware of write permissions in default installation, logs cannot be stored in read only destinations. In order to work properly you have to use another location, e.g. under your home directory and tell Onboard where it is (see Configuration).

Dataplayer

Playback logged data from file. It plays data with same speed as they were logged. Dataplayer generates dynamically output ports according to the played file.

If you want to replay logged data, you must "reset" your graph. It means to use Stop Flow, Destroy DFG and **Create DFG** again.

Beware when you change played file, it may change output ports therefore always reload nodes when you do this! In fact reloading nodes will not change your Dataplayer nodes in DFG editor (it reloads nodes in DFG toolbox only – see DFG toolbox), so you must take care of it yourself.

Outputs

Outputs are dynamically generated according to the played file.

finished (EVENT)
signals end of played file

Parameters

file
Path to file, which will be used for replay. It is recommended to set the same file path as for Datalogger.

Executor

Executes urbiscript which was published on input `code`. Scripting console uses this node for execution.

Execution is controlled by control input – you can start, stop, pause and resume execution this way. If you want to control execution, you must start flow first (see DFG lifecycle section). If you stop flow, currently executed urbiscript will be stopped as well.

You can send text from urbiscript to output called output with urbiscript `echo` function – alas, redirecting output is possible only for calls of `echo` on the top level (see more in Scripting console).

Inputs

control (CONTROL)
commands controlling urbiscript execution – available commands are in following table

`run`	start urbiscript execution
`stop`	stop urbiscript execution
`freeze`	pause urbiscript execution
`unfreeze`	resume urbiscript execution

code (URBISCRIPT)
urbiscript which will be executed

Outputs

output (EXECUTION_OUTPUT)
text from `echo` function (see description for this node)

error (EXECUTION_ERROR)
contains error message, which may occur during execution

state (EXECUTION_STATE)
state of executed urbiscript (see semantic type EXECUTION_STATE):

Parameters

lobby
urbiscript will be executed in this lobby (for more information about lobby see official urbiscript documentation). It is necessary to set to DFG lobby (i.e. `this`) for scripting console functioning.

FlyControl

Switches between inputs and passes data from selected input to the single `output`. After creation, it is set on `input1`. When data on higher input arrives node automatically switches on this input and ignore all data from lower inputs.

Inputs

input1 (FLY_CONTROL)
input data

input2 (FLY_CONTROL)
input data

Outputs

output (*)
data from selected input

Parameters

`unsigned int` – number of inputs
`bool` – if data from higher input will cause switch to turn on this input
`string` – semantic type of the inputs

FlyControlDual

Switch between inputs and pass data only from one input to the `output`. After creation, it is set on `input1`. User can choose which input will be active by sending input port number in `setInput`.

Inputs

input1 (FLY_CONTROL)
input data

input2 (FLY_CONTROL)
input data

chooseInput (CHANNEL)
choose which input data will be passed to the `output`

Outputs

output (*)
data from selected input

Parameters

unsigned int – number of inputs
bool – if data from higher input will cause switch to turn on this input
string – semantic type of the inputs

FlyControlTriple

Switch between inputs and pass data only from one input to the `output`. After creation, it is set on `input1`. User can choose which input will be active by sending input port number in `setInput`.

Inputs

input1 (FLY_CONTROL)
input data

input2 (FLY_CONTROL)
input data

input3 (FLY_CONTROL)
input data

chooseInput (CHANNEL)
choose which input data will be passed in to the `output`

Outputs

output (*)
data from selected input

Parameters
unsigned int – number of inputs
bool – if data from higher input will cause switch to turn on this input
string – semantic type of the inputs

FlyControlAggregator
FlyControlAggregator is aggregation node (hence the aggregator name). It converts separate data for controlling x-copter movement (gaz, pitch, roll and yaw) in one structure called `xcs::FlyControl`.

Inputs

gaz (*)
x-copter up down velocity in range <-1,1>
-1 maximal descendant velocity
 1 maximal rise up velocity of a x-copter)

pitch (*)
x-copter nose tilt in range <-1,1>
-1 maximal nose down tilt
 1 maximal nose up tilt

roll (*)
x-copter left right tilt in range <-1,1>
-1 maximal left tilt
 1 maximal right tilt

yaw (*)
x-copter rotation speed around z axis in range <-1,1>
-1 maximal rotation speed to the left
 1 maximal rotation speed to the right

Outputs

flyControl (FLY_CONTROL)
aggregated flight controls

Parameters
none

Localization
Provide x-copter localization in 3D space based on IMU data and video stream. It can use only IMU data without video for basic localization but this setting is less accurate. Basic localization needs x-copter velocity, rotation and altitude data from sensors. It can be extended with video stream from camera on the x-copter.

For working video based localization “init” command has to be send twice. First after the x-copter takes off and stabilize in the air and the other after the x-copter moves in one axis (the best is up-down movement) approximately 10 centimeters.

Inputs

control (CONTROL)
user can send one of the following commands for change in localization behavior

`init`	take initialization keyframe
`keyframe`	take keyframe
`resetPtam`	reset PTAM (will be uninitialized after this)
`resetEkf`	reset localization and set position to (0, 0, current altitude)

flyControl (FLY_CONTROL)
actual x-copter flight control

flyControlSendTime (TIME)
how long does it take between sending x-copter flight control and executing it in x-copter

measuredAltitude (ALTITUDE)
actual x-copter altitude from sensors

measuredRotation (ROTATION)
actual x-copter rotation from gyroscope sensor

measuredVelocity (VELOCITY_LOC)
actual x-copter velocity from accelerometer

ptamEnabled (ENABLED)
disable or enable PTAM video sensor in localization

setPosition (POSITION_ABS)
set localization position

setRotation (ROTATION)
set localization rotation

timeImu (TIME_LOC)
time when actual IMU measurements were taken

video (CAMERA)
video stream from x-copter camera

videoTime (TIME_LOC)
time when actual frame from the video stream was taken

Outputs

position (POSITION_ABS)
computed actual x-copter position

ptamStatus (PTAM_STATUS)
PTAM video localization status (explained in semantic types)

rotation (ROTATION)
computed actual x-copter rotation

velocity (VELOCITY_ABS)
computed actual x-copter velocity

velocityPsi (ROTATION_VELOCITY_ABS)
computed actual x-copter rotation velocity around z axis

Parameters

`string` – Path to file with settings for xlocalization

PID

PID controller.

Inputs

P (PID_PARAM)
set proportional parameter

I (PID_PARAM)
set integral parameter

D (PID_PARAM)
set derivative parameter

actualValue (*)
actual process measured value

desireValue (*)
desire process value

Outputs

control (PID_CONTROL)
computed control value

Parameters
none

RedDot
Finds red circle in input images and provides its position in image from center.

Inputs

video (CAMERA)
image where RedDot is finding red circle

Outputs

enhencedVideo (CAMERA)
input image with highlighted found circle

errorX (PID_ERROR)
x position of a red circle in the image from center

errorY (PID_ERROR)
y position of a red circle in the image from center

found (DECISION)
if it was detected red circle in actual image

Parameters
none

SemanticReceiver
SemanticReceiver is a node that receives data entered by the user in Client component and propagates them further into the DFG. The data (channel names) that it should react to are set in advance and unknown data are not sent into the dataflow. In default DFG it contains channels sufficient for manual control and scripting.

Outputs

execControl (CONTROL)
commands from scripting console

execurbiscript (URBISCRIPT)
urbiscript from script console

command (COMMAND)
commands from fly control panel (e.g. `TakeOff`)

flyControl (FLY_CONTROL)
fly control values from active controller (when keyboard controller is not active, no data are sent)

Parameters

outputs
specifies what data are transferred, it has same format like list of outputs in node written in urbiscript

XciDodo

XXci object with `xci_dodo` driver.

Self-contained XCI implementation useful when physical x-copter neither fine simulator is available. It has very simple (unrealistic) motion model and can replay a video from file in a loop.

Inputs

command (COMMAND)
There are command to control playback of the video.

`Load`	opens video file and prepares for playback
`Play`	start playing video
`Pause`	pause playing video
`Stop`	pause playing video and reset to beginning

flyControl (FLY_CONTROL)
fly controls applied to the motion model

setFlyControlPersistence (FLY_CONTROL_PERSISTENCE)
set period how often XCI will be repeating last FlyControl command

Outputs

alive (ALIVE)
every ~1 second set to true

altitude (ALTITUDE)
altitude from the motion model

flyControlPersistence (FLY_CONTROL_PERSISTENCE)
period how often XCI will be repeating last FlyControl command

rotation (ROTATION)
tilt angles and orientation from the motion model

velocity (VELOCITY_LOC)
velocity from the motion model

video (CAMERA)
last decoded frame from the video

Configuration parameters

Configuration can be changed by calling setConfiguration(key, value) method of appropriate DFG node (e.g. running the code from scripting console).

`video:filename`	filename with video to be played
`video:fps`	playback speed (default value is read from codec)

XciParrot

XXci object with `xci_parrot` driver.

XCI implementation for Parrot AR.Drone 2.0 with auto reconnection functionalities. User can control AR.Drone 2.0 flight parameters through it and read data from its sensors. It should be connected to AR.Drone 2.0 Wi-Fi network before this node starts otherwise it will try to establish connection in regular intervals.

Inputs

command (COMMAND)
You may send one of below described command for changing behavior of the AR.Drone 2.0.

`TakeOff`	AR.Drone 2.0 going to execute to take off
`Land`	AR.Drone 2.0 going to execute to land
`EmergencyStop`	AR.Drone 2.0 stops all rotors. Use only in very serious situations, it may end with damaged AR.Drone 2.0 (AR.Drone 2.0 simply falls down!)

`Normal`	AR.Drone 2.0 change emergency mode on normal mode. We do not know if AR.Drone 2.0 resumes flight.

flyControl (FLY_CONTROL)
set AR.Drone 2.0 flight parameters (roll,pitch,yaw,gaz)

setFlyControlPersistence (FLY_CONTROL_PERSISTENCE)
set period how often XCI will be repeating last FlyControl command

Outputs

acceleration (ACCELERATION)
output from acceleration sensor

alive (ALIVE)
indicate if AR.Drone 2.0 is ready

altitude (ALTITUDE)
aggregated output from altitude sensors (ultrasound and pressure sensor)

altitudeAll (ALTITUDE_ALL)
separate outputs from altitude sensors

altitudeV (ALTITUDE_V)
we do not know (ask questions on Parrot company)

battery (BATTERY)
energy left in the battery

flyControlPersistence (FLY_CONTROL)
period how often XCI will be repeating last FlyControl command

gyro (GYRO_RAW)

internalTimeImu (TIME_LOC)
time when last sensor measurements have been made
when Parrot runs for more than 34 minutes (2048 seconds), sent time representation overflow

internalTimeVideo (TIME_LOC)
time when last video frame has been captured

magneto (MAGNETO_RAW)
raw output from magnetometer sensor

rotation (ROTATION)
AR.Drone 2.0 rotation in 3D space

velocity (VELOCITY_LOC)
computed AR.Drone 2.0 velocity in 3D space in its frame

video (CAMERA)
last captured frame from camera

wifiQuality (QUALITY)
Wi-Fi connection quality

XciVrep

XXci object with `xci_vrep` driver.

XCI implementation for robotic simulator V-REP[6]. Motion model is realistic, however, control model is emulated via flying onto target.

Alas no video is currently available from V-REP simulator and on Windows fly controls aren't executed at all (i.e. XCS can only read data from V-REP).

Inputs

command (COMMAND)
none

flyControl (FLY_CONTROL)
control target object in simulation on which quadricopter flies

setFlyControlPersistence (FLY_CONTROL_PERSISTENCE)
set period how often XCI will be repeating last FlyControl command

Outputs

flyControlPersistence (FLY_CONTROL_PERSISTENCE)
period how often XCI will be repeating last FlyControl command

position (POSITION_ABS)
quadricopter absolute position in 3D simulated world

rotation (ROTATION)
quadricopter absolute rotation in 3D simulated world

velocity (VELOCITY_ABS)
quadricopter absolute velocity in 3D simulated world

video_bottom (CAMERA)
last video frame from bottom quadricopter camera

video_front (CAMERA)
last video frame from front quadricopter camera

Gui (special)

Gui is a special node that represents data sent to Client's browser. It has inputs for widgets and default input for data that are not needed to be displayed but still needed

6 Successfully tested is version 3.1.1. Later versions didn't cooperate well.

in browser. Such data is e.g. rotation for the video widget, which is needed for the attitude cross rendered on top of video screen. Outputs from executor are other example, they are needed for console functioning.

For each type of widget there is separate input (i.e. raw data, gauge data, progress data, chart data, video). For each output of a node in DFG connected to widget input of a Gui node a widget in grid will be created upon the node creation. For more information on widget types see Widgets.

Inputs

raw data (*)
Input for raw data widget.

gauge data (*)
Input for gauge data widget.

progress data (*)
Input for progress data widget.

chart data (*)
Input for chart data widget.

video (*)
Input for video widget.

default (*)
Default input for data – for more information see description of this node.

Parameters

adapter
Adapter is object, which will be used for sending data to Client's browser. It is recommended not to change this parameter.

json
Json is object, which will be used for converting text to json object and vice versa. It is recommended not to change this parameter.

refreshFrequency
It is frequency of checking whether there are some data for sending to Client's browser.

Configuration

Onboard

Environment

Behavior of Onboard is affected by following environment variables.

Name	Meaning	Default value [7]
`XCS_SETTINGS_PATH`	settings directory	`$PWD/data/settings`
`XCS_DFGS_PATH`	directory with stored DFGs	`$PWD/data/dfgs`
`XCS_USER_SCRIPTS_PATH`	directory with scripts (console)	`$PWD/data/scripts`
`XCS_LOGS_PATH`	directory with logged data	`$PWD/data/logs`

Settings files

Settings files are by default searched in the settings directory mentioned above.

In default installation settings can be found in `C:\Program Files\X-CopterStudio onboard\data\settings` (for Windows) or in `/usr/share/xcs/settings` (for Linux).

`onboard.xs`	Onboard core configuration
`xobjects.xs`	configuration of available nodes for toolbox
`xlocalization.xs`	configuration of Localization node
`xcontrol.xs`	configuration of Control node

Syntax

Boost INFO syntax.

Server

Server configuration can be changed in `config.json` file that can be found in default installation in `C:\Program Files\X-Copter Studio server\config.json` (for Windows) or in

7 `$PWD` denotes working directory of Onboard process.

`/opt/xcs/server/config.json` (for Linux).

Extending X-Copter Studio

New nodes for DFG

This section expects understanding of basic concept of Urbi SDK's C++ API. See official documentation, mainly sections: 4 Quick Start, 5.3 Creating new instances, 5.4 Binding functions and 5.5 Notification of a variable change or access.

Writing own nodes in C++

Urbi SDK extension

Pure Urbi SDK has `UObject` class like main unit. See official documentation[8] for more details. We created our custom class `XObject` based on `UObject`. It is enhanced by collecting information about its semantic and syntactic outputs and inputs types. Furthermore, it provides interface for starting and stopping data flow when acting like a DFG node.

Pure Urbi SDK implements dataflow communication with a pair of classes `InputPort` (for inputs) and `UVar` (for outputs or any variable in general). See official documentation[9] for more details. We created our custom classes `XInputPort`, `SimpleXInputPort`, `XVar` and `SimpleXVar` based on `InputPort` and `UVar`. `SimpleXInputPort` and `SimpleXVar` are enhanced with semantic and syntactic types, which you can set through constructor parameters. `XInputPort` and `XVar` are template classes inheriting from `SimpleXInputPort` and `SimpleXVar` – you can set syntactic type through template and semantic type through constructor parameter. `XInputPort` and `XVar` are used in most cases (in comparison to `SimpleXInputPort` and `SimpleXVar`), so when we will write something about `XInputPort` or `XVar`, it will be true for `SimpleXInputPort` and `SimpleXVar` too in most cases.

Pure Urbi SDK has a set of macros which bind C++ code to urbiscript: `UStart`, `UBindVar`, `UNotifyChange`, `UBindFunction`, `URBI_REGISTER_STRUCT`, (...). See official documentation[10] for more details. We created our custom set of macros: `XStart`, `XBindVar`, `XNotifyChange`, `XBindFunction`, (...) which do the same but with `XObject`, `XInputPort` and `XVar` classes. There is a list of our macros in the following table.

Macro	Description

8 Sections 5 The UObject API, page 53.

9 Sections 5.6 Data-flow based programming: exchanging UVars and 5.7 Data-flow based programming: InputPort, pages 58–60.

10 Sections 4.1.2 Wrapping into UObject, pages 44–46, and 5.5 Notification of a variable change or access, page 58, and 5.18.2 Casting simple structures, page 66.

`XStart (class)`	It is an alternative to `UStart` for `XObject`. This macro must be called for each `XObject`, which you want to use in urbiscript or DFG, but use it just once (call it in cpp file in global scope). It binds `XObject` to urbiscript with default name (it is same like `XObject` class name).
`XStartRename (class, name)`	It is an alternative to `UStartRename` for `XObject`. It is same like `XStart`, but it binds `XObject` to urbiscript with custom class name.
`XBindVar (variable)`	It is an alternative to `UBindVar` for `XVar` and `XInputPort`. This macro must be called for each `XVar` and `XInputPort`, but use it only once (call it in C++ constructor or in `init` function (see official documentation[11] for more details about `init` function)). It binds XVar or XInputPort to urbiscript with default name (it is same like XVar or XInputPort variable name).
`XBindVarRename (variable, name)`	It is an alternative to UBindVarRename for XVar and XInputPort. It is same like XBindVar, but binds XVar and XInputPort to urbiscript with custom name.
`XBindPrivateVar (variable)`	It is an alternative to XBindVar. If you want to use some XVar or XInputPort in your C++ code or directly from urbiscript (e.g. from Scripting console) but you don't want to expose XVar or XInputPort like node output or input, then you can use this macro.
`XNotifyChange (variable, function)`	It is an alternative to UNotifyChange for XObject. This macro registers a function that will be called each time when registered XVar or XInputPort is modified.
`XBindFunction (class, function)`	It is an alternative to UBindFunction for XObject. This macro binds your C++ function to urbiscript (so you can call it from Scripting console)
`XBindVarF (variable, function)`	This macro does the same like XBindVar and XNotifyChange. It binds XVar or XInputPort to urbiscript and it registers function, which will be called each time when registered XVar or XInputPort is modified.
`XBindPrivateVarF (variable, function)`	It is an alternative to XBindVarF, but it uses XBindPrivateVar.
`X_REGISTER_STRUCT (struct, ...)`	It is an alternative to `U_REGISTER_STRUCT` for use with `X_GENERATE_STRUCT macro`. When you want to create your own syntactic type or just use some C++ struct in urbiscript, you must call `X_REGISTER_STRUCT` first. This macro declares the struct to the cast system. First argument is struct itself and others are the struct field names which you want to bind. This macro should be called immediately after struct declaration (so if the struct is

11 Sections 5.3 Creating new instances, page 56.

	declared in header file, this macro should be called here too). See Syntactic type section for more info how to create your own structure (syntactic type).
`X_GENERATE_STRUCT (struct, ...)`	If you already call `X_REGISTER_STRUCT` macro, then you will want to call this macro too in most cases. This macro will create urbiscript "structure" (there is nothing like structure in urbiscript, so it is just pseudoclass with correctly set prototypes) that you can use in urbiscript and convert it to or from your C++ struct. You should use this macro only once (call it in a cpp file). See Syntactic type section for more info how to create your own structure (syntactic type).

If you want to create your own node and use its outputs and inputs in DFG, you should use `XObject`, `XInputPort`, `XUvar` classes and X* macros.

Source code

Basically, all you need is to create a class that extends `xcs::nodes::XObject` supplied by XCS. You can find example demonstrating necessary includes in `C:\Program Files\X-Copter Studio onboard\share\xcs\examples\nodes\pid.xob` (on Windows) or in `/usr/share/xcs/examples/pid.xob` (on Linux).

Compilation

You must set correct include paths and link with XCS libraries. If you have default installation[12], you can use provided `CMakeLists.txt` that sets up paths for you. It can be found in `C:\Program Files\X-Copter Studio onboard\share\xcs\examples\nodes\pid.xob` (on Windows) or in `/use/share/xcs/examples/pid.xob/CMakeLists.txt` (on Linux).

Writing own nodes in urbiscript

It is possible to write nodes purely in urbiscript as pseudoclasses extending pseudoclass `XObject`.[13]

DFG interface of the node is specified via variables `inputs_` and `outputs_`. Those are dictionaries with keys being names of inputs or outputs and values are `Pairs`. It has following structure:

- first is urbiscript object representing the syntactic type[14].
- second is a string – name of the semantic type, it can be your own or already existing one.

12 On Windows, you must have `Developer files` component installed. On Linux, be sure you have `xcs-core-dev` package installed.

13 See official Urbi SDK documentation 12.4 Defining Pseudo-Classes (p. 118).

14 `String`, `Float` or fully qualified name of vector type, e.g. `'xcs::FlyControl'` (note the necessary quotation marks as it represents an urbiscript class)

Node's state is set inside `state_` variable. For correct operation, you have to define this variable (slot) locally on node instance. State is an integer and possible values are shown in the table below.

0	freshly created node
1	started node
2	stopped node

It is recommended to use urbiscript `at` construct to handle state changes.

For example of complete urbiscript node see `Executor` node source (it's found among other urbiscript nodes, information about its path).

Adding own nodes to X-Copter Studio

C++ nodes

Resulting libraries of nodes must reside inside directory specified by `URBI_UOBJECT_PATH` environment variable. In default installation it resolves to `C:\Program Files\X-Copter Studio onboard\lib\xcs\xobjects` (for Windows) and `/usr/lib/xcs/xobjects` (for Linux).

urbiscript nodes

urbiscript nodes are searched by default in subdirectory `nodes` within `URBI_PATH`. After installation this variable is set for Onboard to `C:\Program Files\X-Copter Studio\share\xcs\urbiscript\nodes` (for Windows) or `/use/share/xcs/urbiscript/onboard/nodes` (for Linux).

Creating configuration

In order to have new nodes available in the DFG toolbox, you must add them to the configuration file `xobjects.xs` (see Configuration for location of this file).

Nodes with generic inputs

Some nodes cannot have their input interface defined in advance (Datalogger or Gui for instance). Such nodes should provide methods: `registerXVar` that takes arguments describing dataflow metadata (input name, syntactic and semantic types) and an XVar that is intended to be connected to the input of such a generic node and a method `unregisterXVar`. See API documentation of `XDatalogger` for details.

Implementation note

It's not possible to register handler on `UVar` (or `XVar` as well) not in constructor or init method. Furthermore, it's problem to use a functor as a handler too. That is reason for workaround when one has to create auxiliary class (descendant of `UObject`, let's call it

`Catcher`) and instantiate it for each registered `UVar` and only in the `Catcher::init` method register the UVar.

General notes for node implementers

- When designing the interface (inputs and outputs) of a node, consider whether you cannot use already existing semantic types.
- Possible node states are: created, started, stopped.[15]
- Be sure your node doesn't produce any data when it is in the stopped state.
- In destructor correctly dispose all resources (files, threads, etc.) that node has used.

Implementing XCI

XCI is unified interface to x-copters. When adding support for a new model of x-copter, you have to implement this interface.[16] Technically, it means implementing descendant of provided C++ class `xcs::xci::Xci`.

Individual methods of the interface are described in API documentation., here we describe only implementation concepts.

Sensors

XCI must declare what sensors it makes available. Sensor information consists of:

- sensor name – alphanumerical string unique within the XCI implementation (it propagates as an output name on XCI node, beware of collisions with implicitly existing outputs of XCI node[17]),
- syntactic type – fully qualified name of C++ type that represents values of this sensor (e.g. `std::string`, `xcs::CartesianVector`, see Syntactic types for a reference list),
- semantic type – semantic type of sensor's data, see Semantic types for its meaning.

In order to pass data from sensors into dataflow, one uses `xcs::xci::DataReceiver` class, whose instance is available to XCI constructor. It has method `notify()` that ought to be called upon sensor value updates with data of the type that was declared for the sensor.

Commands

(General) control of x-copter is realized via commands. A command is just a string and XCI implementation should provide a list of existing commands for particular x-copter.

---

15 This is a subset of DFG lifecycle when not-yet-constructed or already-destroyed nodes cannot hold any state.

16 On Windows, you must have `Developer files` component installed. On Linux, be sure you have `xcs-driver-dev` package installed.

17 Currently only forbidden name is `flyControlPersistence`.

Configuration

XCI assumes that x-copter's behavior can be affected by configuration parameters. From XCI's point of view, configuration has simple key-value format, with both keys and values being strings.

It is important that x-copter configuration has (read-only) parameter `XCI_PARAM_FP_PERSISTENCE` that specifies effective duration in milliseconds of fly control command applied via XCI on x-copter. Typical behavior is that for safety reasons x-copter keeps the fly control only for limited period of time if not set again.

Skeleton of XCI implementation

Header file:

```
#include <xcs/xci/xci.hpp>
class XciFoo : public virtual xcs::xci::Xci {
      /* … */
}
```

Source file:

```
using namespace xcs::xci;
      /* sensorList just returns definition of sensors */
     SensorList XciFoo::sensorList() {
           SensorList result;
           result.push_back(Sensor("alive", "ALIVE", "int"));
           result.push_back(Sensor("altitude", "ALTITUDE",
     "double"));
           result.push_back(Sensor("rotation", "ROTATION",
                 "xcs::EulerianVector"));
           return result;
     }
     /* Implement exported constructor wrapper to allow dynamic
     loading. */
     extern "C" {
           Xci* CreateXci(DataReceiver& dataReceiver) {
                 return new XciFoo(dataReceiver);
           }
     }
```

Appendix

Syntactic and semantic types

Each nodes output and input has semantic and syntactic type. Syntactic type is determined by semantic type in many cases, but it is not a rule. There can be two values with a same semantic type but with e.g. different precision (integer vs. decimal number).

Input and output types for connections must be compatible. Typically it means name of syntactic type and semantic type must match. There is also special wildcard character `*` for inputs that accept any type (it can be used both for semantic and syntactic types).

Syntactic types

XCS contains some basic syntactic types derived from C++[18]:

- `std::string` or `const char *` (text),
- `int` (integer),
- `double` (decimal number),
- `bool` (truth value)
- and `std::list` or `std::vector` (list of values).

XCS also contains some structures. The most important of them are in the following table.

Syntactic type	**Structure (C++ syntax)**
`xcs::BitmapType`	const size_t width; const size_t height; uint8_t * const data;
`xcs::CartesianVector`	double x; double y; double z;
`xcs::CartesianVectorChronologic`	double x; double y; double z; Timestamp time; // in milliseconds

`xcs::EulerianVector`	double phi; // x-axis
	double theta; // y-axis
	double psi; // z-axis

`xcs::EulerianVectorChronologic`	double phi; // x-axis
	double theta; // y-axis
	double psi; // z-axis

18 More precisely those are types that are supported by Urbi SDK bindings.

	Timestamp time; // in milliseconds
`xcs::FlyControl`	double roll; double pitch; double yaw; double gaz;
`xcs::Checkpoint`	double x; double y; double z; double xOut; double yOut; double zOut;
`xcs::VelocityControl`	double vx; double vy; double vz; double psi;

If you want to create your own syntactic type, use `X_REGISTER_STRUCT` and `X_GENERATE_STRUCT` macros (for more information about these macros see Writing own nodes in C++ section). This macros will help you create new syntactic type, but node output or input with this type will not be usable for Datalogger or Dataplayer.

See following example for better understanding.

1. Create your structure in C++ (hpp file) and declare it to the cast system.

```
namespace xcs {
struct CartesianVector {
    double x;
    double y;
    double z;
};
}

X_REGISTER_STRUCT(xcs::CartesianVector, x, y, z);
```

2. Create your urbiscript structure (cpp file).

```
#undef X_STRUCT_NAMESPACE
#define X_STRUCT_NAMESPACE "xcs::"

X_GENERATE_STRUCT(CartesianVector, x, y, z);

#undef X_STRUCT_NAMESPACE
```

Define `X_STRUCT_NAMESPACE` is optional. You can use it, if your struct is in a namespace.

Semantic types

XCS contains many semantic types. There is no need to explain all of them. Many of them has self-explaining name (e.g. ALTITUDE is semantic type for altitude).

The most important semantic types are in the following table.

Name	Meaning	Default syntactic type	Note
`POSITION_ABS`	position in the world	`xcs::CartesianVector`	x (right), y (forward), z (up); all in meters
`ROTATION`	pose of the drone	`xcs::EulerianVector`	theta (CW from left view), phi (CW from rear view), psi (CW from top) (applied in this order)
`ROTATION_VELOCITY_ABS`	velocity vector in world FOF	`double`	rad/s
`VELOCITY_ABS`	velocity vector in world FOF	`xcs::CartesianVector`	m/s
`VELOCITY_LOC`	velocity vector in drone's FOF	`xcs::CartesianVector`	m/s
`FLY_CONTROL`	flight control parameters	`xcs::FlyControl`	roll (CW from rear view, tilt), pitch (CW from right view, tilt), yaw (CW top, speed of rotation), gaz (up, speed of rise); relative units
`VELOCITY_CONTROL_ABS`	velocity control parameters	`xcs::VelocityControl`	velocity in all axis x,y,z in m/s and absolute yaw angle in radians all items in world coordinates
`BATTERY`	available battery capacity	`double`	in relative unit [0,1]
`CHECKPOINT`	position in 3D with output vector	`xcs::Checkpoint`	meters
`COMMAND`	special command from predefined subset	`std::string`	
`CAMERA`	images from camera (unspecified)	`xcs::BitmapType`	
`ALIVE`	drone is connected	`int (bool)`	for Parrot implemented as logic

	and able to fly		product of appropriate CTRL_STATE flags (MOTORS_MASK,SOFTWARE_FAULT, VBAT_LOW); updated about once per second is enough
INTEGER	general integer	int	so far used only for Dataplayer seek (I just felt like that)
TIME_LOC	drone's local time	double	timestamp in seconds
PTAM_STATUS	status of PTAM tracking	int	0 – PTAM is idle 3, 4 – PTAM is tracking 7 – PTAM is disabled (other values are internal)
ENABLED	flag of an enabled feature	bool	
CONTROL	general string control	std::string	it's different from COMMAND for safety reasons
URBISCRIPT	urbiscript fragment	std::string	
EXECUTION_STATE	state of executed code	std::string	accessible states are: - idle: execution is off, - running: execution is started, - freezed: execution is paused
EXECUTION_ERROR	last error message, which occur during urbiscript execution	std::string	
EXECUTION_OUTPUT	output of the urbiscript code	std::string	only top-level echo calls are processed within Executor node, others are not possible with current setup
EVENT	node execution event	any numeric type	for more details see corresponding node documentation
CHANNEL	choose channel	unsigned int	choose which channel will be used

Directory structure

Onboard

This describes directory structure under installation prefix (`C:\Program Files\X-Copter Studio onboard` on Windows and `/usr` on Linux).

```
| -- bin
| -- data
     | -- dfgs
     | -- logs
     | -- scripts
     | -- settings
| -- doc
| -- include
| -- lib
| -- share
     | -- xcs
          | -- examples
          | -- urbiscript
```

`bin` – all binaries and dynamic libraries necessary for X-Copter Studio[19]
`data` – X-Copter Studio data configuration files[20]
`dfgs` – dataflow graphs
`logs` – x-copter flight samples
`scripts` – user's urbiscripts
`settings` – configuration files
doc – documentation files[21]
`include` – header files for XCS developers
`lib` – XCS shared libraries
`xcs/xobjects` – DFG nodes
`share/xcs` – X-Copter Studio files
`examples` – X-Copter Studio node example in C++
`urbiscript` – X-Copter Studio startup urbiscripts
onboard/nodes – DFG nodes

19 Valid for Windows version only. Linux version has only executables here.

20 Linux version has `data` inside `share/xcs` directory (part of `xcs-onboard-examples` package).

21 Linux version has `doc` inside `share/xcs` directory (part of `xcs-doc` package).