
RTCBot Documentation

Release 0.2.4

Daniel Kumor

Jul 24, 2023

CONTENTS

1	Documentation	3
1.1	Installing RTCBot	3
1.2	Tutorials & Examples	5
1.3	API Documentation	51
2	Indices and tables	141
	Python Module Index	143
	Index	145

RTCBot's purpose is to provide a set of tutorials and simple modules that help in developing remote-controlled robots in Python, with a focus on the Raspberry Pi.

The tutorials start from a basic connection between a Raspberry Pi and Browser, and encompass creating a video-streaming robot controlled entirely over a 4G mobile connection, all the way to a powerful system that offloads complex computation to a desktop PC in real-time.

All communication happens through [WebRTC](#), using Python 3's `asyncio` and the wonderful [aiortc](#) library, meaning that your robot can be controlled both from the browser and through Python, even when it is not connected to your local network.

DOCUMENTATION

1.1 Installing RTCBot

RTCBot uses some very powerful libraries that have not yet made it to the standard repositories. This can make it a bit difficult to install on some systems.

It is recommended that you first install it and try the examples on a Raspberry Pi or Ubuntu machine, since there might still be some bugs on other operating systems.

1.1.1 Raspbian

RTCBot requires several dependencies which are best installed using apt-get:

```
sudo apt-get install python3-numpy python3-cffi python3-aiohttp \  
    libavformat-dev libavcodec-dev libavdevice-dev libavutil-dev \  
    libswscale-dev libswresample-dev libavfilter-dev libopus-dev \  
    libvpx-dev pkg-config libsrtplib2-dev python3-opencv pulseaudio
```

Then, you can complete the installation with pip:

```
sudo pip3 install rtcbot
```

Warning: You might need to reboot your Pi for RTCBot to work! If RTCBot freezes when starting microphone or speaker, it means that you need to start PulseAudio.

Note: It is recommended that you use the Pi 4 with RTCBot. While it was tested to work down to the Raspberry Pi 3B, it was observed to have extra latency, since the CPU had difficulty keeping up with encoding the video stream while processing controller input. This is because RTCBot currently cannot take advantage of the Pi's hardware acceleration, meaning that all video encoding is done in software.

Note: These instructions were made with reference to Raspbian Buster. While the library *does* work on Raspbian Stretch, you'll need to install aiohttp through pip, and avoid installing opencv.

1.1.2 Ubuntu

RTCBot requires several dependencies which are best installed using apt-get:

```
sudo apt-get install python3-numpy python3-cffi python3-aiohttp \
    libavformat-dev libavcodec-dev libavdevice-dev libavutil-dev \
    libswscale-dev libswresample-dev libavfilter-dev libopus-dev \
    libvpx-dev pkg-config libsrt2-dev python3-opencv pulseaudio
```

Then, you can complete the installation with pip:

```
sudo pip3 install rtcbot
```

Warning: You might need to reboot, or manually start PulseAudio if it was not previously installed. If RTCBot freezes when starting microphone or speaker, it means that PulseAudio is not running.

1.1.3 Mac

To install on Mac, you will want a modern python 3 (either through [MiniConda](#) or Homebrew), and have Xcode's development tools installed. Then, you can run:

```
brew install ffmpeg opus libvpx pkg-config
conda install opencv
pip install rtcbot
```

Note: If you have trouble installing OpenCV, you can skip it, or create a new conda environment.

1.1.4 Windows

Installing on Windows is pretty involved, since you need to manually compile one of the required Python libraries. Nevertheless, if you enjoy a challenge, you can start with setting up [Miniconda](#), after which you can install the basic requirements:

```
conda install aiohttp cffi numpy
conda install -c conda-forge av opencv
```

Note: If you have trouble installing OpenCV, you can skip it, or create a new conda environment.

The library that enables RTCBot to use WebRTC, aiortc, must be compiled from scratch, since no builds are available. To do so, you'll need the [Visual Studio C++ Build Tools](#), and follow these steps:

1. Download and extract the latest [aiortc source code](#)
2. Download and extract the [msvc15 build of libopus](#) into the aiortc folder, so that its lib and include directories are right by setup.py
3. Download and extract the [msvc15 build of libvpx](#) same as libopus (the include and lib folders should merge while extracting)
4. Build the extension:


```
python setup.py build_ext --include-dirs=./include --library-dirs=./lib/x64
```

5. Install aiortc:

```
python setup.py install
```

6. Go to the bin/x64 folder, take the vpx and opus dll files, and copy them to C:\Users\Username\Anaconda3\Library\bin

Finally, run:

```
pip install rtcbot
```

1.2 Tutorials & Examples

The tutorials will lead you through creating remote control software for your own robot.

Each tutorial builds upon the previous one's code, so it is important to go in order.

The full code for each of the tutorials can be seen in the [examples directory](#).

1.2.1 RTCBot Basics

This tutorial will teach you the fundamentals of using RTCBot for your projects. RTCBot is a Python 3 [asyncio](#) library, meaning that it is meant to run in an event loop.

Asyncio Basics

The most basic asyncio program is the following:

```
import asyncio

# Run the event loop
asyncio.get_event_loop().run_forever()
```

You can exit the program with CTRL+C. Right now, the program does nothing, just runs in a loop. Let's fix that:

```
import asyncio

async def myfunction():
    while True:
        await asyncio.sleep(1)
        print("1 second passed")

asyncio.ensure_future(myfunction())
asyncio.get_event_loop().run_forever()
```

This will print "1 second passed" each second.

Notice that myfunction is run in an infinite loop. The utility of an event loop is that you can run many functions *concurrently*, which behaves as if your program was running with many threads at once:

```
import asyncio

async def myfunction1():
    while True:
        await asyncio.sleep(1)
        print("1 second passed")

async def myfunction2():
    while True:
        await asyncio.sleep(2)
        print("2 seconds passed")

asyncio.ensure_future(myfunction1())
asyncio.ensure_future(myfunction2())
asyncio.get_event_loop().run_forever()
```

The key here is the `await` keyword, used in an `async` function (called a coroutine). The `await asyncio.sleep(1)` command pauses execution of the function until one second has passed, allowing the event loop to spend time running the other function.

This means that the event loop is a good way to program where multiple things need to happen in response to events, such as incoming data, or timers, which is precisely the situation in a robot.

RTCBot is a set of tools allowing you to easily use an `asyncio` event loop to pass information between parts of your robot.

To learn more about `asyncio`, it is recommended that you look at a more in-depth tutorial [here](#).

View a Video Feed

To introduce you to the basic concepts of RTCBot, we will start with the simplest task, viewing a webcam video feed:

```
import asyncio
from rtcbot import CVCamera, CVDisplay

camera = CVCamera()
display = CVDisplay()

@camera.subscribe
def onFrame(frame):
    print("got video frame")
    display.put_nowait(frame)

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

The camera might take several seconds to initialize, but after it finishes, a window with a live feed of your webcam will pop up.

The `CVCamera` and `CVDisplay` objects use OpenCV in the background to process frames. The `camera.subscribe` function allows you to subscribe to video frames incoming from the webcam, firing the `onFrame` function 30 times a

second with `numpy` arrays containing BGR images captured by the camera. The `put_nowait` function is then used to send the frame to the window where the image is displayed.

These two functions are part of RTCBot's core abilities. Every producer of data (like `CVCamera`) has a `subscribe()` method, and every consumer of data (like `CVDisplay`) has a `put_nowait` method to insert data.

Note: If you are using the official Raspberry Pi camera, you should replace `CVCamera` with `PiCamera`.

Warning: `CVDisplay` does not work on Mac due to issues with threading in the display toolkit - if using a Mac, you'll have to wait for the video streaming tutorial to view the video feed!

Subscriptions

Using a callback function with the `subscribe` method is not the only way to get data out of a data-producing object. The `subscribe` method is also able to create what is called a subscription.

To understand subscriptions, let's take a quick detour to python Queues:

```
import asyncio

# An asyncio Queue has put_nowait and get coroutine
q = asyncio.Queue()

# Sends data each second
async def sender():
    while True:
        await asyncio.sleep(1)
        q.put_nowait("hi!")

# Receives the data
async def receiver():
    while True:
        data = await q.get()
        print("Received:", data)

asyncio.ensure_future(sender())
asyncio.ensure_future(receiver())
asyncio.get_event_loop().run_forever()
```

Here, the `sender` function sends data, and the `receiver` awaits for incoming data, and prints it. Notice how the queue had a `get` coroutine from which data could be awaited.

We can use the `subscribe` method in a similar way to the above code snippet. When run without an argument, `subscribe` actually returns a subscription, which `CVCamera` automatically keeps updated with new video frames as they come in:

```
import asyncio
from rtcbot import CVCamera, CVDisplay

camera = CVCamera()
display = CVDisplay()
```

(continues on next page)

(continued from previous page)

```
frameSubscription = camera.subscribe()

async def receiver():
    while True:
        frame = await frameSubscription.get()
        display.put_nowait(frame)

asyncio.ensure_future(receiver())

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

This program displays a live video feed, just like the previous version.

The `receiver` function is just running `put_nowait` on each frame received from the subscription. This can be done automatically using the `putSubscription` method, making this a shorthand for the above program:

```
import asyncio
from rtcbot import CVCamera, CVDisplay

camera = CVCamera()
display = CVDisplay()

frameSubscription = camera.subscribe()
display.putSubscription(frameSubscription)

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

Finally, the `camera` object has a `get` coroutine, meaning that it can be passed into `putSubscription` directly:

```
import asyncio
from rtcbot import CVCamera, CVDisplay

camera = CVCamera()
display = CVDisplay()

display.putSubscription(camera)

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

Generalizing to Audio

The above code examples all created a video stream, and displayed it in a window. RTCBot uses *exactly the same* API for **everything**. This means that we can trivially add audio to the previous example:

```
import asyncio
from rtcbot import CVCamera, CVDisplay, Microphone, Speaker

camera = CVCamera()
display = CVDisplay()
microphone = Microphone()
speaker = Speaker()

display.putSubscription(camera)
speaker.putSubscription(microphone)

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
    microphone.close()
    speaker.close()
```

Here, a video stream should be displayed in a window, and all microphone input should be playing in your headphones (or speakers).

Summary

This tutorial introduced the basics of RTCBot, with a focus on the fundamentals:

1. Every data producer has the `subscribe` method and `get` coroutine
2. Every data consumer has a `putSubscription` method and a `put_nowait` method
3. `putSubscription` takes any object with a `get` coroutine
4. Subscribe can also be used for direct callbacks, or with custom subscriptions.

Extra Notes

Each producer can have multiple subscriptions active at the same time. This code shows two different windows with the same video feed:

```
import asyncio
from rtcbot import CVCamera, CVDisplay

camera = CVCamera()
display = CVDisplay()
display2 = CVDisplay()

display.putSubscription(camera)
subscription2 = camera.subscribe()
display2.putSubscription(subscription2)
```

(continues on next page)

(continued from previous page)

```
try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
    display2.close()
```

The `get` coroutine of camera behaves as a single default subscription, so it can only be used by one display (it returns each frame once). The `subscribe` function allows creating an arbitrary number of independent subscriptions/callbacks.

1.2.2 WebRTC Basics

This example will show the absolute basics involved with establishing a real-time connection between Python and your browser.

You can run this example entirely on your Raspberry Pi, or use a browser on a desktop or laptop to connect to the Pi. To find your Pi's IP address, run `ip address` in a terminal, which should display connection info, including an ip address, like `192.168.1.24`.

Set up a Basic Server

To start off, we want to create an `aihttp` server, which will host the html and javascript for the browser.

```
# basic.py
from aiohttp import web
routes = web.RouteTableDef()

@routes.get("/")
async def index(request):
    return web.Response(content_type="text/html", text='''
    <html>
    <head>
    <title>RTCBot: Basic</title>
    </head>
    <body style="text-align: center;padding-top: 30px;">
    <h1>Click the Button</h1>
    <button type="button" id="mybutton">Click me!</button>
    <p>
    Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
    var mybutton = document.querySelector("#mybutton");
    mybutton.onclick = function() {
    console.log("I was just clicked!");
    };
    </script>
    </body>
    </html>
    ''')
```

(continues on next page)

(continued from previous page)

```
app = web.Application()
app.add_routes(routes)
web.run_app(app)
```

You can now run `python3 basic.py`, and navigate your browser to `http://localhost:8080` or `http://<pi ip>:8080`. Try clicking on the button to make sure that a message shows up in the browser console, and make sure that no errors show up.



Click the Button

Click me!

Open the browser's developer tools to see console messages (CTRL+SHIFT+C)

Screenshot

of the webpage generated by html code above.

Talking to Python from the Browser

With a basic server set up, we now add the RTCBot library both to our python and to our javascript code, and establish a WebRTC data connection between the two

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCCConnection, getRTCBotJS

# For this example, we use just one global connection
conn = RTCCConnection()

@conn.subscribe
def onMessage(msg): # Called when each message is sent
    print("Got message:", msg)
```

(continues on next page)

(continued from previous page)

```

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text="""
<html>
  <head>
    <title>RTCBot: Data Channel</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <h1>Click the Button</h1>
    <button type="button" id="mybutton">Click me!</button>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();

      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
      }
      connect();

      var mybutton = document.querySelector("#mybutton");

```

(continues on next page)

(continued from previous page)

```

        mybutton.onclick = function() {
            conn.put_nowait("Button Clicked!");
        };
    </script>
</body>
</html>
""")

async def cleanup(app=None):
    await conn.close()

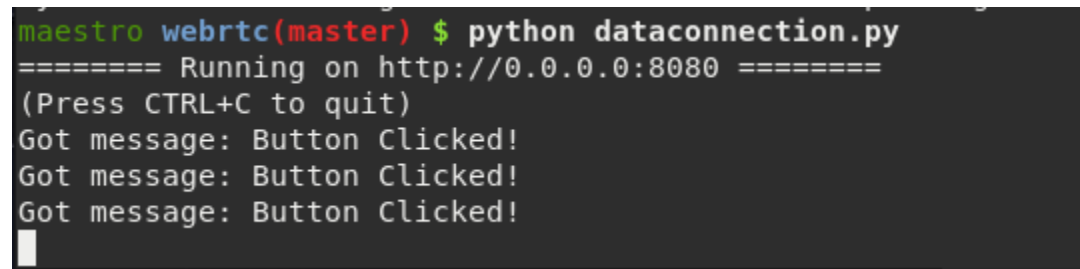
app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)

```

Note: If you use Safari, you might want to add an additional adapter script to the head element to fix connection issues when running locally:

```
<script crossorigin src="https://webrtc.github.io/adapter/adapter-latest.js"></script>
```

The above example establishes a WebRTC connection from the browser to python, and sends a “Button Clicked!” message to python each time the button is clicked in the browser.



```

maestro webrtc(master) $ python dataconnection.py
===== Running on http://0.0.0.0:8080 =====
(Press CTRL+C to quit)
Got message: Button Clicked!
Got message: Button Clicked!
Got message: Button Clicked!

```

Screenshot of the messages received by Python when the button is clicked

The code might be a bit confusing at first, so let’s split it up into its basic components:

```

from rtcbot import RTCTConnection, getRTCBotJS

conn = RTCTConnection() # For this example, we use just one global connection

@conn.subscribe
def onMessage(msg): # Called when messages received from browser
    print("Got message:", msg)

```

This piece of code creates a single global connection for our program. We are assuming only one person is connecting to our robot at a time. The onMessage function is then subscribed to messages coming in through the connection.

```

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")

```

(continues on next page)

(continued from previous page)

```
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())
```

For convenience, the RTCBot javascript library can be accessed directly by calling `getRTCBotJS`. This allows you to directly host a version of the library guaranteed to be compatible with your code. The above code lets you add the RTCBot javascript to your html:

```
<script src="/rtcbot.js"></script>
```

The next python-based piece is the code that sets up a WebRTC connection:

```
# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)
```

We will create what is called an “offer” in the browser, and POST it to `/connect`, which will create a response, and send back the information necessary to complete the connection.

Finally, on application exit, we close the connection:

```
async def cleanup(app=None):
    await conn.close()

app.on_shutdown.append(cleanup)
```

Next, let’s look at the javascript:

```
var conn = new rtcbot.RTCConnection();

async function connect() {
    let offer = await conn.getLocalDescription();

    // POST the information to /connect
    let response = await fetch("/connect", {
        method: "POST",
        cache: "no-cache",
        body: JSON.stringify(offer),
    });

    await conn.setRemoteDescription(await response.json());

    console.log("Ready!");
}
connect();
```

This code uses the javascript version of `RTCConnection`, which is used in exactly the same way as its Python counterpart. First, a global connection `conn` is created. When connecting, it generates an offer, which is POSTed to the server code, and the server’s response is used to complete the connection.

Finally, we replace the original `console.log` with a `conn.put_nowait` to send the message to Python instead (remember from the previous tutorial that `put_nowait` is used everywhere in RTCBot to insert/send data):

```
var mybutton = document.querySelector("#mybutton");
mybutton.onclick = function () {
    conn.put_nowait("Button Clicked!");
};
```

And there you go! That is all that's required to create a full WebRTC connection.

A bit about WebRTC

To understand what is happening in the above code, it is important to understand the basics of WebRTC, which is the technology used for connections in rtcbot. If you are familiar with WebRTC, feel free to skip this section.

WebRTC's core goal is fast peer-to-peer communication between clients. An example of this is video chat. Suppose you and your friend both connect to a web server to talk with each other. The video from your friend's webcam first travels to the server, and then is forwarded from the server to you. This is not ideal - if the server is in another country, your video connection would have a lot of latency, since it needs to travel a large distance - even if you and your friend are connected to the same wifi!

WebRTC fixes this by trying to establish a direct connection between you and your friend - with WebRTC, the video signal would never even leave your local network, giving high quality and very low latency communication. The remote server is only used to help create the connection. Furthermore, the protocol includes mechanisms for passing connections through firewalls, and other complex network configurations.

The above technology is unimportant to us at the moment, since we will connect directly to the server anyways (no intermediate hops), but will become relevant once we try controlling the robot over a 4G connection, where the server and peer become decoupled.

Even without using the above benefits, WebRTC is a better fit than something like a websocket for controlling a robot, since it is designed from the ground up for very low latency and high throughput communication. Furthermore, it natively supports video, with video stream quality adjusting for network speed. This results in a robust and fast connection.

Connection Setup

Unfortunately, establishing a WebRTC connection between a local device (such as your browser) and the remote device (robot) can be a bit involved. Three things need to happen:

1. The local device prepares the type of data it needs to be able to send or accept (raw data, video, audio, etc)
2. The local device needs to gather information about how others can connect to it, such that this data can be sent efficiently. For example, things on your local network could possibly talk with each other using local addresses, like 192.168.1.153. Other times, they must go over the internet, where you have a different IP. The device does some setup, and gathers all the ways that the peer could connect to it. These candidate connection methods are called [ICE Candidates](#).
3. The resulting information needs to be sent to the remote device (robot)
4. The remote device (robot) needs to do the same thing, sending back its own information.
5. Finally, the two sides use this information to create a direct connection

Steps 3 and 4 involve a "Signaling Server", which sends this info from one device to the other. Right now, we don't separate out the signaling server from our python code. That will come in a later tutorial.

Sending JSON to Python and Back

In the previous example, we sent a string, one way: from the browser to Python. In the interest of completeness, we can modify the example given above to both send and receive JSON on button press:

The first modification we make is subscribing to incoming messages in javascript,

```
conn.subscribe((m) => console.log("Received from python:", m));
```

...and sending messages as json:

```
var mybutton = document.querySelector("#mybutton");
mybutton.onclick = function () {
  conn.put_nowait({ data: "ping" });
};
```

The next modification is receiving the json in Python, and sending back a message:

```
@conn.subscribe
def onMessage(msg): # Called when messages received from browser
    print("Got message:", msg["data"])
    conn.put_nowait({"data": "pong"})
```

Notice that RTCBot has direct json support, converting it to Python dicts and javascript objects automatically as the messages are received.

The full code is therefore:

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCConnection, getRTCBotJS

conn = RTCConnection() # For this example, we use just one global connection

@conn.subscribe
def onMessage(msg): # Called when messages received from browser
    print("Got message:", msg["data"])
    conn.put_nowait({"data": "pong"})

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
```

(continues on next page)

(continued from previous page)

```

async def index(request):
    return web.Response(
        content_type="text/html",
        text="""
<html>
  <head>
    <title>RTCBot: Data Channel</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <h1>Click the Button</h1>
    <button type="button" id="mybutton">Click me!</button>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();

      conn.subscribe(m => console.log("Received from python:", m));

      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
      }
      connect();

      var mybutton = document.querySelector("#mybutton");
      mybutton.onclick = function() {
        conn.put_nowait({ data: "ping" });
      };
    </script>
  </body>
</html>
""")

async def cleanup(app=None):
    await conn.close()

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)

```

(continues on next page)

(continued from previous page)

```
web.run_app(app)
```

Summary

This tutorial introduced the `RTCTConnection` object, which can be used both from Python and from javascript to create a WebRTC connection.

1. You create an offer with `conn.getLocalDescription()`
2. You create a response by passing a previously created offer to `conn2.getLocalDescription(offer)`
3. You set the response from the second connection with `conn.setRemoteDescription(response)`

With that, a connection between `conn` and `conn2` is established. Both in javascript and in Python you can use `put_nowait` and `subscribe` to send and receive messages, respectively.

Extra Notes

The javascript version of `RTCTConnection` tries to be as similar as possible to the Python version. However, it is not as powerful as the Python version, allowing only callback subscriptions to receive messages, and only allowing `put_nowait` to send them, rather than allowing one to `putSubscription`, as can be done in the Python version.

Also, we only created the `RTCTConnection` globally for simplicity in the tutorial. In real apps, you will want to create connections inside the `/connect` handler to be able to handle multiple clients, or even a single client connecting multiple times.

1.2.3 Streaming Video

In the previous tutorial, a data connection was created between your python program and a browser, allowing to send messages back and forth. This tutorial will build upon the previous one's code, culminating in a 2-way video and audio connection, where the Python code displays the video stream it gets from your browser, and the browser displays the video stream from the server.

You should use a browser on your laptop or desktop for this one, and put the server on a Raspberry Pi if you want to try streaming from the PiCamera.

Skeleton Code

If you have not done so yet, you should look at the previous tutorial, where the basics of an `RTCTConnection` are explained. For the skeleton of this part, the button from the previous tutorial was removed, and replaced with a video element. Also removed was all code involving messages, to keep this tutorial focused entirely on video.

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCTConnection, getRTCBotJS

# For this example, we use just one global connection
conn = RTCTConnection()

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
```

(continues on next page)

(continued from previous page)

```

async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text=r"""
<html>
  <head>
    <title>RTCBot: Skeleton</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center; padding-top: 30px;">
    <video autoplay playsinline muted controls></video>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();

      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
      }
      connect();
    </script>
  </body>
</html>
      """)

async def cleanup(app=None):
    await conn.close()

```

(continues on next page)

(continued from previous page)

```
app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)
```

This code establishes a WebRTC connection, and nothing else. It can be seen as a minimal example for RTCBot.

Streaming Video from Python

The first thing we'll do is send a video stream from a webcam to the browser. If on a desktop or laptop, you should use `CVCamera`, and if on a Raspberry Pi with the camera module, use `PiCamera` instead - they get their video differently, but behave identically.

All you need is to add a couple lines of code to the skeleton to get a fully-functional video stream:

```
from aiohttp import web
routes = web.RouteTableDef()

-from rtcbot import RTCConnection, getRTCBotJS
+from rtcbot import RTCConnection, getRTCBotJS, CVCamera

+# Initialize the camera
+camera = CVCamera()

# For this example, we use just one global connection
conn = RTCConnection()

+# Send images from the camera through the connection
+conn.video.putSubscription(camera)

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text=r"""
<html>
  <head>
    <title>RTCBot: Skeleton</title>
```

(continues on next page)

(continued from previous page)

```

    <script src="/rtcbot.js"></script>
</head>
<body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline muted controls></video>
    <p>
    Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
        var conn = new rtcbot.RTCConnection();

+         // When the video stream comes in, display it in the video element
+         conn.video.subscribe(function(stream) {
+             document.querySelector("video").srcObject = stream;
+         });

        async function connect() {
            let offer = await conn.getLocalDescription();

            // POST the information to /connect
            let response = await fetch("/connect", {
                method: "POST",
                cache: "no-cache",
                body: JSON.stringify(offer)
            });

            await conn.setRemoteDescription(await response.json());

            console.log("Ready!");
        }
        connect();

    </script>
</body>
</html>
""")

async def cleanup(app=None):
    await conn.close()
+     camera.close() # Singletons like a camera are not awaited on close

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)

```

One major difference between javascript and Python, is that the audio/video `subscribe` in javascript is only called once, and returns a video stream object. In Python, the same function would get called on each video frame.

Also, remember to subscribe/put all subscriptions into `conn` *before* initializing the connection with `getLocalDescription`. This is because `getLocalDescription` uses knowledge of which types of streams you want to send and receive to construct its offer and response.

Note: In some cases you will need to click play in the browser before the video starts.

Adding Audio

Warning: Be aware that a Pi 3 with USB microphone might struggle a bit sending both audio and video at the same time. Try the code on your desktop/laptop or a Pi 4 first to make sure it works before attempting use with the Pi 3.

Based on what you know of RTCBot so far, and knowing that you can use a microphone with the `Microphone` class, do you think you can figure out audio just looking at the video code above?

The modifications to add audio use exactly the same ideas:

```
from rtcbot import RTCTConnection, getRTCBotJS, CVCamera, Microphone

camera = CVCamera()
mic = Microphone()

conn = RTCTConnection()
conn.video.putSubscription(camera)
conn.audio.putSubscription(mic)
```

Also, don't forget to close the microphone at the end with `mic.close()`!

On the browser side, we add an `<audio autoplay></audio>` element right after the `<video>` element, and update the javascript:

```
var conn = new RTCTConnection();

conn.video.subscribe(function (stream) {
  document.querySelector("video").srcObject = stream;
});
conn.audio.subscribe(function (stream) {
  document.querySelector("audio").srcObject = stream;
});
```

Browser to Python

Thus far, we used Python to stream video and audio to the browser, which is the main use case in a robot. However, RTCBot can handle streaming both ways. Since it is assumed that you are at a single computer, we can't stream from Python and the browser at the same time (both will try to use the same webcam). We will switch the stream directions instead.

This bears repeating, so let's reiterate a bit of the basics of RTCBot's python API:

- Anything that outputs data has a `subscribe` method
- Anything that takes in data has a `putSubscription` method, which takes in a subscription: `putSubscription(x.subscribe())`

- An RTCCConnection `conn` has *both* outputs and inputs for messages sent through the connection. Furthermore, it also has video and audio streams `conn.video` and `conn.audio`, which *also* can be used as both inputs and outputs.

With this in mind, reversing the stream direction is a simple matter:

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCCConnection, getRTCBotJS, CVDisplay, Speaker

display = CVDisplay()
speaker = Speaker()

# For this example, we use just one global connection
conn = RTCCConnection()
display.putSubscription(conn.video.subscribe())
speaker.putSubscription(conn.audio.subscribe())

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text=r"""
<html>
  <head>
    <title>RTCBot: Skeleton</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline controls></video> <audio autoplay></audio>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCCConnection();

      async function connect() {

          let streams = await navigator.mediaDevices.getUserMedia({audio: true,
↪ video: true});
          conn.video.putSubscription(streams.getVideoTracks()[0]);
```

(continues on next page)

(continued from previous page)

```
        conn.audio.putSubscription(streams.getAudioTracks()[0]);

        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
            method: "POST",
            cache: "no-cache",
            body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
    }
    connect();

</script>
</body>
</html>
""")

async def cleanup(app=None):
    await conn.close()
    display.close()
    speaker.close()

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)
```

In the above code, instead of `CVCamera` and `Microphone`, `CVDisplay` and `Speaker` are used. In the javascript, we moved the subscribing code to the `connect` function, because `getUserMedia` is an asynchronous function, and cannot be awaited outside an async function (like `connect`).

Summary

This tutorial introduced video and audio streaming over WebRTC. Everything here relied on the `RTCCConnection` object `conn`, which can be initialized both from browser and Python.

1. `conn.video` is both a data producer and a consumer, allowing both to subscribe to remote video and send video streams
2. `conn.audio` behaves in exactly the same way as `conn.video`

Put together with messages that can be sent directly using `conn` (see previous tutorial), this allows you to send data back and forth however you like.

Extra Notes

While the `RTCConnection` was created globally here, but should generally be created for each connection, the camera/microphone/speaker/display objects should be used as singletons, initialized once at the beginning of the program, and closed when the program is exiting.

1.2.4 Keyboard & Xbox Controller

It is time to move towards the “bot” portion of RTCBot: robot control. With the previous tutorials, a robot’s webcam can be streamed to the browser. Now, it is time to send commands from the keyboard or Xbox controller to your robot. This will allow true remote control, with you watching a video feed on your computer, and controlling the robot with your keyboard, while the robot roams around your house.

Make sure to go through the previous tutorial before starting this one.

Skeleton Code

Just like the previous tutorials, we start with the basic skeleton that just establishes a WebRTC connection between Python and the browser.

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCConnection, getRTCBotJS

conn = RTCConnection()

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text=r"""
<html>
<head>
    <title>RTCBot: Skeleton</title>
    <script src="/rtcbot.js"></script>
</head>
<body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline controls></video> <audio autoplay></audio>
    <p>

```

(continues on next page)

(continued from previous page)

```

    Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
  </p>
  <script>
    var conn = new rtcbot.RTCConnection();

    async function connect() {
      let offer = await conn.getLocalDescription();

      // POST the information to /connect
      let response = await fetch("/connect", {
        method: "POST",
        cache: "no-cache",
        body: JSON.stringify(offer)
      });

      await conn.setRemoteDescription(await response.json());

      console.log("Ready!");
    }
    connect();

  </script>
</body>
</html>
""")

async def cleanup(app=None):
    await conn.close()

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)

```

Keyboard

We now add keyboard support. This is done with the `rtcbot.Keyboard` javascript class

```

from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCConnection, getRTCBotJS

conn = RTCConnection()

+@conn.subscribe
+def onMessage(m):
+    print("key press", m)

# Serve the RTCBot javascript library at /rtcbot.js

```

(continues on next page)

(continued from previous page)

```

@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text=r"""
<html>
  <head>
    <title>RTCBot: Skeleton</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline controls></video> <audio autoplay></audio>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();
      var kb = new rtcbot.Keyboard();

      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        kb.subscribe(conn.put_nowait);

        console.log("Ready!");
      }
      connect();
    </script>
  </body>

```

(continues on next page)

(continued from previous page)

```
</html>
""",
)

async def cleanup(app=None):
    await conn.close()

app = web.Application()
web.run_app(app)
```

This javascript code creates a Keyboard object in the browser, which internally uses the `onkeydown` and `onkeyup` events to gather keyboard data. It then subscribes the `put_nowait` function of the connection to key events once the connection is set up.

Running the above code gives the following output in the Python console:

```
===== Running on http://0.0.0.0:8080 =====
(Press CTRL+C to quit)
key press {'type': 'keydown', 'altKey': False, 'shiftKey': True, 'keyCode': 16, 'key':
↳ 'Shift'}
key press {'type': 'keydown', 'altKey': False, 'shiftKey': True, 'keyCode': 72, 'key': 'H
↳ '}
key press {'type': 'keyup', 'altKey': False, 'shiftKey': True, 'keyCode': 72, 'key': 'H'}
key press {'type': 'keyup', 'altKey': False, 'shiftKey': False, 'keyCode': 16, 'key':
↳ 'Shift'}
key press {'type': 'keydown', 'altKey': False, 'shiftKey': False, 'keyCode': 69, 'key':
↳ 'e'}
key press {'type': 'keyup', 'altKey': False, 'shiftKey': False, 'keyCode': 69, 'key': 'e
↳ '}
key press {'type': 'keydown', 'altKey': False, 'shiftKey': False, 'keyCode': 76, 'key':
↳ 'l'}
key press {'type': 'keyup', 'altKey': False, 'shiftKey': False, 'keyCode': 76, 'key': 'l
↳ '}
key press {'type': 'keydown', 'altKey': False, 'shiftKey': False, 'keyCode': 76, 'key':
↳ 'l'}
key press {'type': 'keyup', 'altKey': False, 'shiftKey': False, 'keyCode': 76, 'key': 'l
↳ '}
key press {'type': 'keydown', 'altKey': False, 'shiftKey': False, 'keyCode': 79, 'key':
↳ 'o'}
key press {'type': 'keyup', 'altKey': False, 'shiftKey': False, 'keyCode': 79, 'key': 'o
↳ '}
```


Xbox Controller

The keyboard would work for controlling your robot in a pinch, but an xbox controller is more useful, since it has analog sticks and triggers. Those will allow you fine-grained control of your robot's speed and movement.

Thankfully, RTCBot has you covered - you only need to replace a single line in the above Keyboard code to switch to an xbox controller:

```
-var kb = new rtcbot.Keyboard();
+var kb = new rtcbot.Gamepad();
```

Running this code gives the following Python output:

```
key press {'value': -0.2838831841945648, 'type': 'axis1'}
key press {'value': -0.009033478796482086, 'type': 'axis0'}
key press {'value': 0, 'type': 'axis1'}
key press {'value': 0.004119998775422573, 'type': 'axis3'}
key press {'value': 0.006103701889514923, 'type': 'axis3'}
key press {'value': -0.008697775192558765, 'type': 'axis0'}
key press {'value': 0, 'type': 'axis3'}
key press {'value': -0.009338663890957832, 'type': 'axis0'}
key press {'value': 0, 'type': 'axis0'}
key press {'value': True, 'type': 'btn0'}
key press {'value': False, 'type': 'btn0'}
key press {'value': 0.27201148867607117, 'type': 'axis2'}
key press {'value': 1, 'type': 'axis2'}
key press {'value': -1, 'type': 'axis2'}
key press {'value': -0.8708761930465698, 'type': 'axis2'}
key press {'value': -0.7945494055747986, 'type': 'axis2'}
```

The controller's buttons give boolean values, and the joysticks give float values between -1 and 1. By default, the controller is polled at 10Hz as not to overwhelm a Pi 3 with tons of data each time a joystick is moved.

Remote Control

And now, we put everything together, with a video stream sent from Python, and controls sent back from the browser. This code directly allows you to sit at your computer and remotely control a Pi placed in a different room. We combine the keyboard example above, with the video streaming example from the previous tutorial.

We use the WASD keys for movement, decoding the current controls in Python's onMessage:

```
keystates = {"w": False, "a": False, "s": False, "d": False}

@conn.subscribe
def onMessage(m):
    global keystates
    if m["keyCode"] == 87: # W
        keystates["w"] = m["type"] == "keydown"
    elif m["keyCode"] == 83: # S
        keystates["s"] = m["type"] == "keydown"
    elif m["keyCode"] == 65: # A
        keystates["a"] = m["type"] == "keydown"
    elif m["keyCode"] == 68: # D
        keystates["d"] = m["type"] == "keydown"
```

(continues on next page)

(continued from previous page)

```
print({
    "forward": keystates["w"] * 1 - keystates["s"] * 1,
    "leftright": keystates["d"] * 1 - keystates["a"] * 1,
})
```

This code keeps track of which keys are currently pressed, and prints out the robot controls. Leftright is -1 on left, and 1 on right. Similarly, forward is 1 when w is pressed, and -1 when s is pressed:

```
{'forward': 1, 'leftright': -1}
{'forward': 0, 'leftright': -1}
{'forward': 0, 'leftright': 0}
{'forward': -1, 'leftright': 0}
{'forward': -1, 'leftright': 1}
{'forward': 0, 'leftright': 1}
{'forward': 0, 'leftright': 0}
{'forward': 1, 'leftright': 0}
{'forward': 1, 'leftright': -1}
{'forward': 0, 'leftright': -1}
{'forward': 0, 'leftright': 0}
```

With this, we have a fully functional remote control system! All that's left is connecting your robot's motors.

```
# Full code for the above example

from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCTConnection, CVCamera, getRTCBotJS
cam = CVCamera()

# For this example, we use just one global connection
conn = RTCTConnection()
conn.video.putSubscription(cam)

keystates = {"w": False, "a": False, "s": False, "d": False}

@conn.subscribe
def onMessage(m):
    global keystates
    if m["keyCode"] == 87: # W
        keystates["w"] = m["type"] == "keydown"
    elif m["keyCode"] == 83: # S
        keystates["s"] = m["type"] == "keydown"
    elif m["keyCode"] == 65: # A
        keystates["a"] = m["type"] == "keydown"
    elif m["keyCode"] == 68: # D
        keystates["d"] = m["type"] == "keydown"
    print({
        "forward": keystates["w"] * 1 - keystates["s"] * 1,
        "leftright": keystates["d"] * 1 - keystates["a"] * 1,
    })
```

(continues on next page)

(continued from previous page)

```

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text=r"""
<html>
  <head>
    <title>RTCBot: Remote Control</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline controls></video> <audio autoplay></audio>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();
      conn.video.subscribe(function(stream) {
        document.querySelector("video").srcObject = stream;
      });

      var kb = new rtcbot.Keyboard();

      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        kb.subscribe(conn.put_nowait);

        console.log("Ready!");
      }
      connect();
    </script>
  </body>
</html>
"""
    )

```

(continues on next page)

(continued from previous page)

```
        </script>
    </body>
</html>
""")

async def cleanup(app=None):
    await conn.close()

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)
```

Summary

In this section, keyboard and gamepad control was introduced, culminating in a fully remote-controlled system where commands were sent from browser, and a live video stream was sent to the browser.

This example can be directly extended to link to a robot's controls - you can link the control output with your robot's actuators and motors in just a few lines of code!

Extra Notes

In the above examples, we used a simple event-based control scheme. For robustness, it is better to send the full state from the browser rather than the individual events. That is, rather than sending *just* the keydown event, it is generally better to process controls in javascript, and send the full state (ie: {"forward":1, "leftright":0}).

1.2.5 Connecting over 4G

Thus far, the tutorials have all had you connect directly to the robot, which meant that it had to be on your local wifi network. In this tutorial, we will finally decouple the server and the robot.

Rather than connecting to the robot, we will have two separate Python programs. The first is a server, which will be served at a known IP address. The second will be the robot, which connects to the server with a websocket, and waits for the information necessary to initialize a WebRTC connection directly to your browser.

Note: The server must be accessible from the internet. Running your own server might involve a bit of configuration in your router settings or setup of a cloud server, such as a virtual machine on DigitalOcean. You can also use the provided server at <https://rtcbot.dev> to help establish connections (see below).

In a previous tutorial, we developed a connection that streamed video to the browser. This tutorial will implement exactly the same functionality, but with the robot on a remote connection.

The browser-side code will remain unchanged - all of the work here will be in Python.

Server Code

Most of the server code is unchanged. The only difference is that we set up a listener at /ws, which will establish a websocket connection with the robot:

```
ws = None # Websocket connection to the robot
@routes.get("/ws")
async def websocket(request):
    global ws
    ws = Websocket(request)
    print("Robot Connected")
    await ws # Wait until the websocket closes
    print("Robot disconnected")
    return ws.ws
```

The above code sets up a global ws variable which will hold the active connection. We then use this websocket in the /connect handler. Instead of establishing a WebRTC connection ourselves, the server forwards the information directly to the robot using the websocket:

```
# Called by the browser to set up a connection
@routes.post("/connect")
async def connect(request):
    global ws
    if ws is None:
        raise web.HTTPInternalServerError("There is no robot connected")
    clientOffer = await request.json()
    # Send the offer to the robot, and receive its response
    ws.put_nowait(clientOffer)
    robotResponse = await ws.get()
    return web.json_response(robotResponse)
```

This is all that is needed from the server - its function is simply to route the information necessary to establish the connection directly between robot and browser. The full server code is here:

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import Websocket, getRTCBotJS

ws = None # Websocket connection to the robot
@routes.get("/ws")
async def websocket(request):
    global ws
    ws = Websocket(request)
    print("Robot Connected")
    await ws # Wait until the websocket closes
    print("Robot disconnected")
    return ws.ws

# Called by the browser to set up a connection
@routes.post("/connect")
async def connect(request):
    global ws
    if ws is None:
```

(continues on next page)

(continued from previous page)

```

    raise web.HTTPInternalServerError("There is no robot connected")
    clientOffer = await request.json()
    # Send the offer to the robot, and receive its response
    ws.put_nowait(clientOffer)
    robotResponse = await ws.get()
    return web.json_response(robotResponse)

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text="""
<html>
  <head>
    <title>RTCBot: Remote Video</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline muted controls></video>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();

      conn.video.subscribe(function(stream) {
        document.querySelector("video").srcObject = stream;
      });

      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
      }
      connect();

    </script>
  </body>

```

(continues on next page)

(continued from previous page)

```

</html>
""")

async def cleanup(app=None):
    global ws
    if ws is not None:
        c = ws.close()
        if c is not None:
            await c

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)

```

Remote Code

For simplicity, we will just run both server and robot on the local machine. The robot connects to the server with a websocket, and waits for the message that will allow it to initialize its WebRTC connection.

```

import asyncio
from rtcbot import Websocket, RTCCConnection, CVCamera

cam = CVCamera()
conn = RTCCConnection()
conn.video.putSubscription(cam)

# Connect establishes a websocket connection to the server,
# and uses it to send and receive info to establish webRTC connection.
async def connect():
    ws = Websocket("http://localhost:8080/ws")
    remoteDescription = await ws.get()
    robotDescription = await conn.getLocalDescription(remoteDescription)
    ws.put_nowait(robotDescription)
    print("Started WebRTC")
    await ws.close()

asyncio.ensure_future(connect())
try:
    asyncio.get_event_loop().run_forever()
finally:
    cam.close()
    conn.close()

```

With these two pieces of code, you first start the server, then start the robot, and finally open `http://localhost:8080` in the browser to view a video stream coming directly from the robot, even if the robot has an unknown IP.

rtcbot.dev

The above example requires you to have your own internet-accessible server at a known IP address to set up the connection, if your remote code is not on your local network. The server's only real purpose is to help *establish* a connection - once the connection is established, it does not do anything.

For this reason, I am hosting a free testing server online at <https://rtcbot.dev> that performs the equivalent of the following operation from the above server code:

```
@routes.get("/ws")
async def websocket(request):
    global ws
    ws = Websocket(request)
    print("Robot Connected")
    await ws # Wait until the websocket closes
    print("Robot disconnected")
    return ws.ws

# Called by the browser to set up a connection
@routes.post("/connect")
async def connect(request):
    global ws
    if ws is None:
        raise web.HTTPInternalServerError("There is no robot connected")
    clientOffer = await request.json()
    # Send the offer to the robot, and receive its response
    ws.put_nowait(clientOffer)
    robotResponse = await ws.get()
    return web.json_response(robotResponse)
```

Since the server at rtcbot.dev is open to anyone, instead of `/ws` and `/connect`, you need to choose some random sequence of letters and numbers that will identify your connection, for example `myRandomSequence11`.

Once you have chosen your sequence, you can both connect your websocket and POST to <https://rtcbot.dev/myRandomSequence11>:

Note: If you open <https://rtcbot.dev/myRandomSequence11> in your browser, you can see if your remote code is connected with a websocket, and optionally open a video connection.

When using rtcbot.dev, the remote connection code becomes:

```
async def connect():
    ws = Websocket("https://rtcbot.dev/myRandomSequence11")
    remoteDescription = await ws.get()
    robotDescription = await conn.getLocalDescription(remoteDescription)
    ws.put_nowait(robotDescription)
    print("Started WebRTC")
    await ws.close()
```

and the local browser's connection code becomes:

```
let response = await fetch("https://rtcbot.dev/myRandomSequence11", {
  method: "POST",
  cache: "no-cache",
```

(continues on next page)

(continued from previous page)

```
body: JSON.stringify(offer),
});
```

With `rtcbot.dev`, you no longer need your local server code to run websockets or a connection service. Its only purpose is to give the browser the html and javascript necessary to establish a connection. We will get rid of the browser entirely in the next tutorial.

If it doesn't work over 4G

The above example should work for most people. However, some mobile network operators perform routing that disallows creating a direct WebRTC connection to a mobile device over 4G. If this is your situation, you need to use what is called a TURN server, which will forward data between the browser and robot.

Note: You can check if your mobile operator allows such connections by using your phone to create a wifi hotspot, to which you can connect your robot. If video streaming works with the code above, you can ignore this section!

Warning: Because a TURN server essentially serves as a proxy through which an entire WebRTC connection is routed, it can send and receive quite a bit of data - make sure that you don't exceed your download and upload limits!

There are two options through which to setup a TURN server: `coTURN` and `Pion`. Pion is meant to be a more simple and temporary solution that's easy to setup while `coTURN` is recommended for more permanent setups.

Setup with Pion

The Pion server is easy to set up on Windows, Mac and Linux - all you need to do is [download the executable](#), and run it from the command line as shown.

Linux/Mac:

```
chmod +x ./simple-turn-linux-amd64 # allow executing the downloaded file
export USERS='myusername=mypassword'
export REALM=my.server.ip
export UDP_PORT=3478
./simple-turn-linux-amd64 # simple-turn-darwin-amd64 if on Mac
```

Windows: You can run the following from powershell:

```
$env:USERS = "myusername=mypassword"
$env:REALM = "my.server.ip"
$env:UDP_PORT = 3478
./simple-turn-windows-amd64.exe
```

With the Pion server running, you will need to let both Python and Javascript know about it when creating your `RTCConnection`:

```
from aiortc import RTCConfiguration, RTCIceServer

myConnection = RTCConnection(rtcConfiguration=RTCConfiguration([
```

(continues on next page)

(continued from previous page)

```
RTCIceServer(urls="stun:stun.l.google.com:19302"),
RTCIceServer(urls="turn:my.server.ip:3478",
              username="myusername",credential="mypassword")
)))
```

```
var conn = new rtcbot.RTCConnection(true, {
  iceServers:[
    { urls: ["stun:stun.l.google.com:19302"] },
    { urls: "turn:my.server.ip:3478?transport=udp",
      username: "myusername", credential: "mypassword", },
  ]});
```

Setup with coTURN

Setting up a coTURN server takes a bit more work and is only supported on Linux and Mac. The following steps will assume a Linux system running Ubuntu.

Install coTURN and stop the coTURN service to modify config files with

```
sudo apt install coturn
sudo systemctl stop coturn
```

Edit the file `/etc/default/coturn` by uncommenting the line `TURN_SERVER_ENABLED=1`. This will allow coTURN to start in daemon mode on boot.

Edit another file `/etc/turnserver.conf` and add the following lines. Be sure to put your system's public facing IP address in place of `<PUBLIC_NETWORK_IP>`, your domain name in place of `<DOMAIN>`, and your own credentials in place of `<USERNAME>` and `<PASSWORD>`.

```
listening-port=3478
tls-listening-port=5349
listening-ip=<PUBLIC_NETWORK_IP>
relay-ip=<PUBLIC_NETWORK_IP>
external-ip=<PUBLIC_NETWORK_IP>
realm=<DOMAIN>
server-name=<DOMAIN>

user=<USERNAME>:<PASSWORD>
lt-cred-mech
```

Note: If you are running coTURN within a local network, `<DOMAIN>` can be whatever you want.

Restart the coTURN service, check that it's running, and reboot.

```
sudo systemctl start coturn
sudo systemctl status coturn
sudo reboot
```

With the coTURN server running, you will need to let both Python and Javascript know about it when creating your `RTCConnection`:

```
from aiortc import RTCTransport, RTCEngine

myConnection = RTCTransport(rtcConfiguration=RTCTransportConfiguration([
    RTCEngine(urls="stun:stun.l.google.com:19302"),
    RTCEngine(urls="turn:<PUBLIC_NETWORK_IP>:3478",
               username="myusername", credential="mypassword")
]))
```

```
var conn = new rtcbot.RTCConnection(true, {
  iceServers: [
    { urls: ["stun:stun.l.google.com:19302"] },
    { urls: "turn:<PUBLIC_NETWORK_IP>:3478?transport=udp",
      username: "myusername", credential: "mypassword", },
  ];
});
```

Note: If you are running coTURN on a local network, replace <PUBLIC_NETWORK_IP> with the public facing IP of the system running coTURN. If coTURN is running on a server with a domain, replace <PUBLIC_NETWORK_IP> with the domain/realm set in /etc/turnserver.conf.

With either of the options above, you should be able to stream video to your browser using 4G, even if your mobile operator disallows direct connections.

Summary

This tutorial split up the server and robot code into distinct pieces. Also introduced was rtcbot's websocket wrapper, allowing you to easily establish a data-only connection. Finally, TURN servers were introduced, and instructions were given on how to set one up if direct connections fail.

Extra Notes

Be aware that throughout these tutorials, all error handling and robustness was left out in the interest of clarity in the fundamental program flow. In reality, you will probably want to make sure that the connection did not have an error, and add the ability to connect and disconnect multiple times.

1.2.6 Offloading Computation

Most hobbyists can't afford to do complex computations on their robot, because the little single-board computers (SBCs) available for a reasonable price do not have sufficient processing power for advanced functionality. While this is slowly changing with things like [Nvidia's Jetson Nano](#), there is still a large gap in power between SBCs and an average desktop.

The ideal situation would be if you could strap an entire desktop to your robot. With RTCBot, we can do the next best thing: we can stream the robot's inputs to a desktop, which can then perform computation, and send back commands.

In this tutorial, we will go back to a single file for both server and robot for simplicity. We set up a connection to the robot from Python, allowing you to control the robot with an xbox controller without a browser.

Note: While with a Raspberry Pi there might be a non-negligible delay between sending a video frame and getting back a command, this is not a limitation of the approach, since it is possible to stream [video games with barely-noticeable](#)

lag. In particular, rtcbot currently cannot take advantage of the Pi's hardware acceleration, meaning that all video encoding is done in software, which ends up adding to video delay.

Python to Python Streaming

To start offloading, we get rid of the browser - we will create a connection from Python on your desktop to Python on your robot, stream video from the robot, and stream controls from the desktop.

The robot code is identical to the code we have seen in previous tutorials. All we did was remove the browser code, since it is not needed.

```
# robot.py
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCTConnection, CVCamera
cam = CVCamera()
conn = RTCTConnection()
conn.video.putSubscription(cam)

@conn.subscribe
def controls(msg):
    print("Control message:", msg)

@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

async def cleanup(app):
    await conn.close()
    cam.close()

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)
```

Then, on the desktop, we run the following:

```
# desktop.py

import asyncio
import aiohttp
import cv2
import json
from rtcbot import RTCTConnection, Gamepad, CVDisplay

disp = CVDisplay()
g = Gamepad()
conn = RTCTConnection()
```

(continues on next page)

(continued from previous page)

```

@conn.video.subscribe
def onFrame(frame):
    # Show a 4x larger image so that it is easy to see
    resized = cv2.resize(frame, (frame.shape[1] * 4, frame.shape[0] * 4))
    disp.put_nowait(resized)

async def connect():
    localDescription = await conn.getLocalDescription()
    async with aiohttp.ClientSession() as session:
        async with session.post(
            "http://localhost:8080/connect", data=json.dumps(localDescription)
        ) as resp:
            response = await resp.json()
            await conn.setRemoteDescription(response)
    # Start sending gamepad controls
    g.subscribe(conn)

asyncio.ensure_future(connect())
try:
    asyncio.get_event_loop().run_forever()
finally:
    conn.close()
    disp.close()
    g.close()

```

This code manually sends the connect request, and establishes a webRTC connection with the response. Also introduced was the Python version of Gamepad. The browser version was used in a previous tutorial.

The robot code's output is now:

```

===== Running on http://0.0.0.0:8080 =====
(Press CTRL+C to quit)
Control message: {'timestamp': 1553379212.684861, 'code': 'BTN_SOUTH', 'state': 1, 'event': 'Key'}
Control message: {'timestamp': 1553379212.684861, 'code': 'ABS_Y', 'state': -1, 'event': 'Absolute'}
Control message: {'timestamp': 1553379213.192862, 'code': 'BTN_SOUTH', 'state': 0, 'event': 'Key'}
Control message: {'timestamp': 1553379214.14487, 'code': 'BTN_SOUTH', 'state': 1, 'event': 'Key'}
Control message: {'timestamp': 1553379214.964878, 'code': 'BTN_SOUTH', 'state': 0, 'event': 'Key'}
Control message: {'timestamp': 1553379216.172882, 'code': 'BTN_SOUTH', 'state': 1, 'event': 'Key'}
Control message: {'timestamp': 1553379216.48489, 'code': 'BTN_SOUTH', 'state': 0, 'event': 'Key'}
Control message: {'timestamp': 1553379216.872889, 'code': 'ABS_X', 'state': -11, 'event': 'Absolute'}
Control message: {'timestamp': 1553379216.884891, 'code': 'ABS_X', 'state': -64, 'event': 'Absolute'}
Control message: {'timestamp': 1553379216.892888, 'code': 'ABS_X', 'state': -95, 'event': 'Absolute'}

```

(continues on next page)

(continued from previous page)

```
Control message: {'timestamp': 1553379216.904886, 'code': 'ABS_X', 'state': -158, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.912884, 'code': 'ABS_X', 'state': -599, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.924894, 'code': 'ABS_X', 'state': -1240, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.932888, 'code': 'ABS_X', 'state': -1586, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.944887, 'code': 'ABS_X', 'state': -2080, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.952887, 'code': 'ABS_X', 'state': -2689, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.964892, 'code': 'ABS_X', 'state': -3833, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.972892, 'code': 'ABS_X', 'state': -4957, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.972892, 'code': 'ABS_Y', 'state': -53, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.984889, 'code': 'ABS_X', 'state': -7944, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.984889, 'code': 'ABS_Y', 'state': -106, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379216.992891, 'code': 'ABS_X', 'state': -10170,
↳: 'event': 'Absolute'}
Control message: {'timestamp': 1553379216.992891, 'code': 'ABS_Y', 'state': -137, 'event'
↳: 'Absolute'}
Control message: {'timestamp': 1553379217.004892, 'code': 'ABS_X', 'state': -12567,
↳: 'event': 'Absolute'}
```

Warning: The output for the *Gamepad* object is currently different in Javascript and in Python. Make sure you don't mix them up!

1.2.7 Multiple Connections & Reconnecting

Thus far, all of the tutorials included a single `RTCTConnection` object for simplicity. While it makes the code easy to understand, it also means that refreshing the browser page, or connecting from another tab at the same time will not work, since each `RTCTConnection` object can only be used once.

This tutorial will show how to set up your server to handle multiple connections.

Video Streaming Template

We will build upon the video-streaming tutorial. The basic template code is copied over to this tutorial, with all references to the global `RTCCConnection` removed:

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCCConnection, getRTCBotJS, CVCamera
camera = CVCamera()

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()

    ## WHAT GOES HERE? ##

    return web.json_response(serverResponse)

async def cleanup(app=None):
    camera.close()

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# Serve the webpage!
@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text="""
<html>
  <head>
    <title>RTCBot: Video</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline controls muted></video>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCCConnection();

      conn.video.subscribe(function(stream) {
        document.querySelector("video").srcObject = stream;
      });

      async function connect() {
        let offer = await conn.getLocalDescription();
```

(continues on next page)

(continued from previous page)

```

        // POST the information to /connect
        let response = await fetch("/connect", {
            method: "POST",
            cache: "no-cache",
            body: JSON.stringify(offer)
        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
    }
    connect();

</script>
</body>
</html>
""",
)

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)

```

The Connection Handler

Most of the template above is code to display the video box in a browser. We therefore focus only on the parts relevant to this tutorial:

```

camera = CVCamera()

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()

    ## WHAT GOES HERE? ##

    return web.json_response(serverResponse)

async def cleanup(app=None):
    camera.close()

```

Remember that thus far, all tutorials used a single global connection:

```

camera = CVCamera()

# For this example, we use just one global connection
conn = RTCConnection()
# Subscribe to the video feed

```

(continues on next page)

(continued from previous page)

```

conn.video.putSubscription(camera)

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    # Set up the connection
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

async def cleanup(app=None):
    await conn.close()
    camera.close()

```

This global connection can only be initialized once, so once the `connect` function is run, it will not work again. The solution is to create a new `RTCConnection` each time `connect` is called. We also need to keep track of the active connections, and subscriptions to video frames, since it is now possible for there to be multiple streams at once!

The most robust way to achieve this is to create a class wrapping the connection object, which handles preparation of the connection as well as cleanup. The code we will use is shown here:

```

camera = CVCamera()

class ConnectionHandler:
    active_connections = [] # This array keeps track of all current connections

    def __init__(self):
        self.conn = RTCConnection()

        # Subscribe to the video frames - each connection gets its own subscription
        global camera
        self.videoSubscription = camera.subscribe()
        self.conn.video.putSubscription(self.videoSubscription)

        # Perform cleanup when the connection is closed
        self.conn.onClose(self.close)

        # Add this connection to the list of active connections
        ConnectionHandler.active_connections.append(self)

    def close(self):
        # When done, unsubscribe from the video feed
        global camera
        camera.unsubscribe(self.videoSubscription)

        # Remove from list of active connections
        ConnectionHandler.active_connections.remove(self)

    async def getLocalDescription(self, clientOffer):
        # Pass the connection setup result
        return await self.conn.getLocalDescription(clientOffer)

    @staticmethod

```

(continues on next page)

(continued from previous page)

```
async def cleanup():
    # Close all active connections, making sure to use an array copy [:]
    # since closing removes the item from the array!
    for c in ConnectionHandler.active_connections[:]:
        await c.conn.close()

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    conn = ConnectionHandler() # Our ConnectionHandler class!
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

async def cleanup(app=None):
    await ConnectionHandler.cleanup() # When the app is closed, close all connections
    camera.close()
```

Warning: Each video stream is encoded separately, so a Raspberry Pi might struggle with multiple simultaneous connections.

When building a robot, you might want to allow only a single active connection at a time (which will control the bot!), which can be achieved by checking the number of active connections before creating a new `ConnectionHandler`.

The class-based approach allows easy extension. For example, to receive control messages from the browser, an `onMessage` function can be added:

```
class ConnectionHandler:
    def __init__(self):
        # ...

        self.conn.subscribe(self.onMessage)

    # ---
    def onMessage(self, msg):
        print(msg)
```

Summary

This tutorial showed how to allow multiple connections and reconnecting with RTCBot. If using RTCBot to build a robot, it is recommended that you use a similar approach, rather than a single global connection.

1.2.8 Running Blocking Code

RTCBot uses python's asyncio event loop. This means that Python runs in a loop, handling events as they come in, all in a single thread. Any long-running operation must be specially coded to be async, so that it does not block operation of the event loop.

A Common Issue

Suppose that you have a sensor that you want to use with RTCBot. Your goal is to retrieve values from the sensor, and then send the results to the browser.

We will use the function `get_sensor_data` to represent a sensor which takes half a second to retrieve data:

```
import time
import random

def get_sensor_data():
    time.sleep(0.5) # Represents an operation that takes half a second to complete
    return random.random()
```

We will base this code on the original single-connection video-streaming tutorial for simplicity. We will send the sensor reading once a second:

```
from aiohttp import web

routes = web.RouteTableDef()

from rtcbot import RTCConnection, getRTCBotJS, CVCamera

camera = CVCamera()
# For this example, we use just one global connection
conn = RTCConnection()
conn.video.putSubscription(camera)

+import time
+import random
+import asyncio
+
+
+def get_sensor_data():
+    time.sleep(0.5) # Represents an operation that takes half a second to complete
+    return random.random()
+
+
+async def send_sensor_data():
+    while True:
+        await asyncio.sleep(1)
```

(continues on next page)

(continued from previous page)

```

+         data = get_sensor_data()
+         conn.put_nowait(data) # Send data to browser
+
+
+asyncio.ensure_future(send_sensor_data())

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

# This sets up the connection
@routes.post("/connect")
async def connect(request):
    clientOffer = await request.json()
    serverResponse = await conn.getLocalDescription(clientOffer)
    return web.json_response(serverResponse)

@routes.get("/")
async def index(request):
    return web.Response(
        content_type="text/html",
        text="""
<html>
  <head>
    <title>RTCBot: Video</title>
    <script src="/rtcbot.js"></script>
  </head>
  <body style="text-align: center;padding-top: 30px;">
    <video autoplay playsinline muted controls></video>
    <p>
      Open the browser's developer tools to see console messages (CTRL+SHIFT+C)
    </p>
    <script>
      var conn = new rtcbot.RTCConnection();

      conn.video.subscribe(function(stream) {
        document.querySelector("video").srcObject = stream;
      });

+         conn.subscribe(m => console.log("Received from python:", m));
+
      async function connect() {
        let offer = await conn.getLocalDescription();

        // POST the information to /connect
        let response = await fetch("/connect", {
          method: "POST",
          cache: "no-cache",
          body: JSON.stringify(offer)

```

(continues on next page)

(continued from previous page)

```

        });

        await conn.setRemoteDescription(await response.json());

        console.log("Ready!");
    }
    connect();

</script>
</body>
</html>
""",
)

async def cleanup(app=None):
    await conn.close()
    camera.close()

conn.onClose(cleanup)

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app)

```

If you try this code, the video will freeze for half a second each second, while the sensor is being queried (i.e. while `time.sleep(0.5)` is being run). This is because all of RTCBot's tasks happen in the same thread, and while reading the sensor, RTCBot is not sending video frames!

To fix this issue, the sensor needs to be read in a different thread, so that the event loop is not blocked. The sensor data then needs to be moved to the main thread, where it can be used by rtcbot.

Producing Data in Another Thread

Thankfully, RTCBot has built-in helper classes that set everything up for you here. The `ThreadedSubscriptionProducer` runs in a system thread, allowing arbitrary blocking code, and has built-in mechanisms that let you queue up data for use from the asyncio event loop.

The code that blocks the connection:

```

import time
import random
import asyncio

def get_sensor_data():
    time.sleep(0.5) # Represents an operation that takes half a second to complete
    return random.random()

async def send_sensor_data():
    while True:

```

(continues on next page)

(continued from previous page)

```
        await asyncio.sleep(1)
        data = get_sensor_data()
        conn.put_nowait(data) # Send data to browser

asyncio.ensure_future(send_sensor_data())
```

can be fixed by moving the sensor-querying code into a ThreadedSubscriptionProducer:

```
import time
import random
import asyncio

from rtcbot.base import ThreadedSubscriptionProducer

def get_sensor_data():
    time.sleep(0.5) # Represents an operation that takes half a second to complete
    return random.random()

class MySensor(ThreadedSubscriptionProducer):
    def _producer(self):
        self._setReady(True) # Notify that ready to start gathering data
        while not self._shouldClose: # Keep gathering until close is requested
            time.sleep(1)
            data = get_sensor_data()
            # Send the data to the asyncio thread,
            # so it can be retrieved with await mysensor.get()
            self._put_nowait(data)
        self._setReady(False) # Notify that sensor is no longer operational

mysensor = MySensor()

async def send_sensor_data():
    while True:
        data = await mysensor.get() # we await the output of MySensor in a loop
        conn.put_nowait(data)

asyncio.ensure_future(send_sensor_data())

...

async def cleanup(app=None):
    await conn.close()
    camera.close()
    mysensor.close()
```

Consuming Data in Another Thread

RTCBot has an equivalent mechanism for ingesting data - you can retrieve data, and then use it to control things with blocking code.

```
import time

def set_output_value(value):
    time.sleep(0.5) # Represents an operation that takes half a second to complete
    print(value)

from rtcbot.base import ThreadedSubscriptionConsumer, SubscriptionClosed

class MyOutput(ThreadedSubscriptionConsumer):
    def _consumer(self):
        self._setReady(True)
        while not self._shouldClose:
            try:
                data = self._get()
                set_output_value(data)
            except SubscriptionClosed:
                break

        self._setReady(False)

myoutput = MyOutput()
```

You can now use `myoutput.put_nowait` in `rtcbot` to queue up data, which will be retrieved from the consumer thread.

Summary

This tutorial introduced the `ThreadedSubscriptionProducer` and `ThreadedSubscriptionConsumer` classes, which allow you to use blocking code with the `asyncio` event loop. These functions allow handling the connection in the main thread, and doing all actions that might take a while in separate threads.

1.3 API Documentation

RTCBot includes code that simplifies certain tasks. For example, it handles getting frames from your camera in a way easily compatible with `asyncio`, or sending commands to an Arduino efficiently.

1.3.1 RTC Connection

API

`class rtcbot.connection.ConnectionAudioHandler(rtc)`

Bases: *SubscriptionProducerConsumer*

Allows usage of `RTCConnection` as follows:

```
r = RTCConnection()
audioSubscription = r.audio.subscribe()

r.audio.putSubscription(audioSubscription)
```

It uses the first incoming audio stream for `subscribe()`, and creates a single outgoing audio stream.

Subscribing to the tracks can be done

addTrack(*subscription=None, sampleRate=48000, canSkip=True*)

Allows to send multiple audio tracks in a single connection. Each call to `putTrack` *adds* the track to the connection. For simple usage, where you only have a single audio stream, just use *putSubscription* - it automatically calls `putTrack` for you.

close()

Cleans up and closes the object.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:",myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```


offerToReceive(*num=1*)

Set the number of tracks that you can receive

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

onTrack(*callback=None*)

Callback that gets called each time a audio track is received:

```
@r.audio.onTrack
def onTrack(track):
    print(track)
```

The callback actually works exactly as a `subscribe()`, so you can do:

```
subscription = r.audio.onTrack()
await subscription.get()
```

Note that if you have more than one track, you will need to tell rtcbot how many tracks to prepare to receive:

```
r.audio.offerToReceive(2)
```

`putSubscription(subscription)`

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

`put_nowait(data)`

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s
```

(continues on next page)

(continued from previous page)

```
assert myobj.subscription is None
```

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the [onReady\(\)](#) function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

subscribe(subscription=None)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to [subscribe\(\)](#) returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method `put_nowait` (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from *subscribe()*.

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class rtcbot.connection.**ConnectionVideoHandler**(*rtc*)

Bases: *SubscriptionProducerConsumer*

Example

Allows usage of RTCTConnection as follows:

```
r = RTCTConnection()
frameSubscription = r.video.subscribe()

r.video.putSubscription(frameSubscription)
```

It uses the first incoming video stream for *subscribe()*, and creates a single outgoing video stream.

Subscribing to the tracks can be done

addTrack(*frameSubscription=None, fps=None, canSkip=True*)

Allows to send multiple video tracks in a single connection. Each call to *putTrack* *adds* the track to the connection. For simple usage, where you only have a single video stream, just use *putSubscription* - it automatically calls *putTrack* for you.

close()

Cleans up and closes the object.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual *Exception* that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using *onError()*, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to *subscribe().get()*. On the first call, creates a default subscription, and all subsequent calls to *get()* use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

offerToReceive(num=1)

Set the number of tracks that you can receive

onClose(subscription=None)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(subscription=None)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the `error` event is fired, with the associated `Exception`. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The `onError()` function behaves in the same way as a `subscribe()`, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(subscription=None)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a `ready` event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a `subscribe()`, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

onTrack(*callback=None*)

Callback that gets called each time a video track is received:

```
@r.video.onTrack
def onTrack(track):
    print(track)
```

The callback actually works exactly as a `subscribe()`, so you can do:

```
subscription = r.video.onTrack()
await subscription.get()
```

Note that if you have more than one track, you will need to tell rtcbot how many tracks to prepare to receive:

```
r.video.offerToReceive(2)
```

putSubscription(*subscription*)

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

put_nowait(data)

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

subscribe(subscription=None)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):


```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method `put_nowait` (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

unsubscribe(subscription=None)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from *subscribe()*.

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class rtcbot.connection.DataChannel(rtcDataChannel, json=True)

Bases: *SubscriptionProducerConsumer*

Represents a data channel. You can put_nowait messages into it, and subscribe to messages coming from it.

close()

Cleans up and closes the object.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual *Exception* that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using *onError()*, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to *subscribe().get()*. On the first call, creates a default subscription, and all subsequent calls to *get()* use that subscription.

If *unsubscribe()* is called, the subscription is deleted, so a subsequent call to *get()* will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

property name

`onClose(subscription=None)`

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

`onError(subscription=None)`

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The `onError()` function behaves in the same way as a `subscribe()`, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

`onReady(subscription=None)`

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a `subscribe()`, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

putSubscription(subscription)

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

put_nowait(data)

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property ready

This is `True` when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the `ready` state, since all functions for getting/putting data will work even if the class is still starting up in the background.

`stopSubscription()`

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

`subscribe(subscription=None)`

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters**subscription** (*optional*) –**An optional existing subscription to subscribe to. This can be one of 3 things:**

- 1) An object which has the method `put_nowait` (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by `get()`. If none exists, then does nothing.**Parameters****subscription** (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

```
class rtcbot.connection.RTCConnection(defaultChannelOrdered=True, loop=None,
                                     rtcConfiguration=aiortc.RTCConfiguration)
```

Bases: *SubscriptionProducerConsumer*

addDataChannel(name, ordered=True)

Adds a data channel to the connection. Note that the RTCConnection adds a “default” channel automatically, which you can subscribe to directly.

property audio

Convenience function - you can subscribe to it to get audio once a stream is received

close()

If the loop is running, returns a future that will close the connection. Otherwise, runs the loop temporarily to complete closing.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:",myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

getDataChannel(*name*)

Returns the data channel with the given name. Please note that the “default” channel is considered special, and is not returned.

async getLocalDescription(*description=None*)

Gets the description to send on. Creates an initial description if no remote description was passed, and creates a response if a remote was given,

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onDataChannel(*callback=None*)

Acts as a subscriber...

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

`putSubscription(subscription)`

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

`put_nowait(data)`

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property `ready`

This is `True` when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the `ready` state, since all functions for getting/putting data will work even if the class is still starting up in the background.

send(msg)

Send is an alias for `put_nowait` - makes it easier for people new to rtcbot to understand what is going on

async setRemoteDescription(description)

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

subscribe(subscription=None)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters**subscription** (*optional*) –**An optional existing subscription to subscribe to. This can be one of 3 things:**

- 1) An object which has the method *put_nowait* (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.**Parameters****subscription** (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

property video

Convenience function - you can subscribe to it to get video frames once they show up

1.3.2 Websocket

API

class rtcbot.websocket.**Websocket**(url_or_request, json=True, loop=None)

Bases: *SubscriptionProducerConsumer*

Wraps an aiohttp websocket to have an API matching RTCBot. The websocket can be given either a URL to connect to:

```
ws = Websocket("http://localhost:8080/ws")
msg = await ws.get()
```

It can also be used in a server context to complete the connection:

```
@routes.get("/ws")
async def websocketHandler(request):
    ws = Websocket(request)
    msg = await ws.get()
```

Naturally, just like all other parts of rtcbot, you can also *subscribe* and *putSubscription* instead of manually calling *get* and *put_nowait*.

close()

Cleans up and closes the object.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual *Exception* that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using *onError()*, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to *subscribe().get()*. On the first call, creates a default subscription, and all subsequent calls to *get()* use that subscription.

If *unsubscribe()* is called, the subscription is deleted, so a subsequent call to *get()* will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

putSubscription(subscription)

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

put_nowait(data)

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property ready

This is `True` when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the `ready` state, since all functions for getting/putting data will work even if the class is still starting up in the background.

`stopSubscription()`

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

`subscribe(subscription=None)`

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters**subscription** (*optional*) –**An optional existing subscription to subscribe to. This can be one of 3 things:**

- 1) An object which has the method *put_nowait* (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.**Parameters****subscription** (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

1.3.3 Camera

The Camera API allows you to subscribe to video frames coming in from a webcam. To use this API, you will need to either have OpenCV installed (for use with CVCamera and CVDisplay), have `picamera` installed to use PiCamera.

To install OpenCV on Ubuntu 18.04 or Raspbian Buster, use the following command:

```
sudo apt-get install python3-opencv
```

On Raspbian Stretch or older Ubuntu, you can install it with:

```
sudo apt-get install python-opencv
```

If using Windows or Mac, it is recommended that you use [Anaconda](#), and install OpenCV from there.

If on a Raspberry Pi, you don't need OpenCV at all to use the official Pi Camera.

CVCamera

The CVCamera uses a webcam connected to your computer, and gathers video frames using OpenCV:

```
import asyncio
from rtcbot import CVCamera, CVDisplay

camera = CVCamera()
display = CVDisplay()

display.putSubscription(camera)

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

The frames are gathered as BGR numpy arrays, so you can perform any OpenCV functions you'd like on them. For example, the following code shows the video in black and white:

```
import asyncio
from rtcbot import CVCamera, CVDisplay
import cv2

camera = CVCamera()
display = CVDisplay()

@camera.subscribe
def onFrame(frame):
    bwframe = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    display.put_nowait(bwframe)
```

(continues on next page)

(continued from previous page)

```
try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

Warning: There is currently an issue with threading in OpenCV that makes CVDdisplay not work on Mac.

PiCamera

This allows you to use the official raspberry pi camera. You can use it in exactly the same way as the OpenCV camera above, and it returns exactly the same data for the frame:

```
import asyncio
from rtcbot import PiCamera, CVDdisplay

camera = PiCamera()
display = CVDdisplay()

display.putSubscription(camera)

try:
    asyncio.get_event_loop().run_forever()
finally:
    camera.close()
    display.close()
```

This means that if not using CVDdisplay, you don't even need OpenCV installed to stream from you raspberry pi.

PiCamera2

This allows you to use the official raspberry pi camera, with libcamera stack (legacy camera interface disabled). This is default since Raspberry Pi OS bullseye, PiCamera2 also works with 64-bit OS. You can use the parameter hflip=1 to flip the camera horizontally, vflip=1 to flip vertically, or both to rotate 180 degrees. You can use it in exactly the same way as the OpenCV camera above, and it returns exactly the same data for the frame:

```
import asyncio
from rtcbot import PiCamera2, CVDdisplay

camera = PiCamera2()
display = CVDdisplay()

display.putSubscription(camera)

try:
    asyncio.get_event_loop().run_forever()
finally:
```

(continues on next page)

(continued from previous page)

```
camera.close()
display.close()
```

This means that if not using CVDisplay, you don't even need OpenCV installed to stream from you raspberry pi.

API

```
class rtcbot.camera.CVCamera(width=320, height=240, cameranumber=0, fps=30,
                             preprocessframe=<function CVCamera.<lambda>>, loop=None)
```

Bases: [ThreadedSubscriptionProducer](#)

Uses a camera supported by OpenCV.

When initializing, can give an optional function which preprocesses frames as they are read, and returns the modified versions thereof. Please note that the preprocessing happens synchronously in the camera capture thread, so any processing should be relatively fast, and should avoid pure python code due to the GIL. Numpy and openCV functions should be OK.

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using [onError\(\)](#), which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to [get\(\)](#) use that subscription.

If [unsubscribe\(\)](#) is called, the subscription is deleted, so a subsequent call to [get\(\)](#) will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(subscription=None)

Subscribe to new frames as they come in. By default returns a `MostRecentSubscription` object, which can be awaited to get the most recent frame, and skips missed frames.

Note that all subscribers get the same frame data numpy array, so if you are going to modify the values of the array itself, please do so in a copy!:

```
# Set up a camera and subscribe to new frames
cam = CVCamera()
subs = cam.subscribe()

async def mytask():

    # Wait for the next frame
    myframe = await subs.get()

    # Do stuff with the frame
```

If you want to have a different subscription type, you can pass anything which has a `put_nowait` method, which is called each time a frame comes in:

```
subs = cam.subscribe(asyncio.Queue()) # asyncio queue has a put_nowait method
await subs.get()
```

unsubscribe(subscription=None)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by `get()`. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from `subscribe()`.

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class `rtcbot.camera.CVDisplay(name=None, loop=None)`

Bases: `BaseSubscriptionConsumer`

Displays the frames in an openCV *imshow* window

Warning: Due to an issue with [threading in OpenCV on Mac](#), CVDDisplay does not work on Mac.

close()

Cleans up and closes the object.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

onClose(subscription=None)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(subscription=None)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated `Exception`. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ", err)
```

The `onError()` function behaves in the same way as a `subscribe()`, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(subscription=None)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a `subscribe()`, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

`putSubscription(subscription)`

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

`put_nowait(data)`

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the *onReady()* function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

class rtcbot.camera.PiCamera(rotation=0, **kwargs)

Bases: *CVCamera*

Instead of using OpenCV camera support, uses the picamera library for direct access to the Raspberry Pi's CSI camera.

The interface is identical to CVCamera. When testing code on a desktop computer, it can be useful to have the code automatically choose the correct camera:

```
try:
    import picamera # picamera import will fail if not on pi
    cam = PiCamera()
except ImportError:
    cam = CVCamera()
```

This enables simple drop-in replacement between the two.

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not `None`, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the *onReady()* function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(*subscription=None*)

Subscribe to new frames as they come in. By default returns a `MostRecentSubscription` object, which can be awaited to get the most recent frame, and skips missed frames.

Note that all subscribers get the same frame data numpy array, so if you are going to modify the values of the array itself, please do so in a copy!:

```
# Set up a camera and subscribe to new frames
cam = CVCamera()
subs = cam.subscribe()

async def mytask():

    # Wait for the next frame
    myframe = await subs.get()

    # Do stuff with the frame
```

If you want to have a different subscription type, you can pass anything which has a `put_nowait` method, which is called each time a frame comes in:

```
subs = cam.subscribe(asyncio.Queue()) # asyncio queue has a put_nowait method
await subs.get()
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by `get()`. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from `subscribe()`.

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class `rtcbot.camera.PiCamera2`(*hflip=False, vflip=False, **kwargs*)

Bases: `CVCamera`

Instead of using OpenCV camera support, uses the `picamera2` library for direct access to the Raspberry Pi's CSI camera.

The interface is identical to `CVCamera`. When testing code on a desktop computer, it can be useful to have the code automatically choose the correct camera:

```
try:
    import picamera2 # picamera2 import will fail if not on pi
    cam = PiCamera2()
except ImportError:
    cam = CVCamera()
```

This enables simple drop-in replacement between the two.

You can use the parameter `hflip=True` to flip the camera horizontally, `vflip=True` to flip vertically, or both to rotate 180 degrees.

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an `async` function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(subscription=None)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the *onReady()* function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(*subscription=None*)

Subscribe to new frames as they come in. By default returns a `MostRecentSubscription` object, which can be awaited to get the most recent frame, and skips missed frames.

Note that all subscribers get the same frame data numpy array, so if you are going to modify the values of the array itself, please do so in a copy!:

```
# Set up a camera and subscribe to new frames
cam = CVCamera()
subs = cam.subscribe()

async def mytask():

    # Wait for the next frame
    myframe = await subs.get()

    # Do stuff with the frame
```

If you want to have a different subscription type, you can pass anything which has a `put_nowait` method, which is called each time a frame comes in:

```
subs = cam.subscribe(asyncio.Queue()) # asyncio queue has a put_nowait method
await subs.get()
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by `get()`. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from `subscribe()`.

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

1.3.4 Audio

Audio support is built upon the `SoundCard` library. The provided API gives a simple asyncio-based wrapper of the library, which integrates directly with other components of `rtcbot`.

The library is made up of two objects: a `Speaker` and `Microphone`. The `Microphone` gathers audio at 48000 samples per second, and gives the data in chunks of 1024 samples. The data is returned as a numpy array of shape `(samples, channels)`.

The `speaker` performs the reverse operation: it is given numpy arrays containing audio samples, and it plays them on the computer's default audio output.

Basic Example

With the following code, you can listen to yourself. Make sure to wear headphones, so you don't get feedback:

```
import asyncio
from rtcbot import Microphone, Speaker

microphone = Microphone()
speaker = Speaker()

speaker.putSubscription(microphone)

try:
    asyncio.get_event_loop().run_forever()
finally:
    microphone.close()
    speaker.close()
```

Naturally, the raw data can be manipulated with numpy. For example, the following code makes the output five times as loud:

```
import asyncio
from rtcbot import Microphone, Speaker

microphone = Microphone()
speaker = Speaker()

@microphone.subscribe
def onData(data):
    data = data * 5
    if speaker.ready:
        speaker.put_nowait(data)

try:
    asyncio.get_event_loop().run_forever()
finally:
    microphone.close()
    speaker.close()
```

By checking if the speaker is ready, we don't queue up audio while it is initializing (if the microphone starts returning data before the speaker is prepared). This allows us to hear the audio with low latency. This effect was automatic in the first example, because a subscription to microphone was not created until `microphone.get` was called by the speaker.

Warning: This is one of the fundamental differences between video and audio in RTCBot - dropping a video frame is not a big deal, so the cameras automatically always return the most recent frame. However, dropping audio results in weird audio glitches. To avoid this, audio is *queued*. This means that a subscription that is not actively being read will keep queueing up data indefinitely. Make sure to unsubscribe the moment you stop using an audio subscription, or your code will eventually run out of memory!

API

class `rtcbot.audio.Microphone`(*samplerate=48000, channels=None, blocksize=1024, device=None, loop=None*)

Bases: [*ThreadedSubscriptionProducer*](#)

Reads microphone data, and writes audio output. This class allows you to output sound while reading it.

Parameters

- **samplerate** (*int, optional*) – The sampling rate in Hz. Default is 48000.
- **channels** (*int, list(int), optional*) – The index of channel to record. Allows a list of indices. Records on all available channels by default.
- **blocksize** (*int, optional*) – Records this many samples at a time. A lower block size will give lower latency, but higher CPU usage.
- **device** ([`soundcard._Microphone`](#)) – The [`soundcard`](#) device to record from. Uses default if not specified.

`close()`

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

property `closed`

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property `error`

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using [`onError\(\)`](#), which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async `get()`

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to [`get\(\)`](#) use that subscription.

If [`unsubscribe\(\)`](#) is called, the subscription is deleted, so a subsequent call to [`get\(\)`](#) will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:


```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the [onReady\(\)](#) function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(subscription=None)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to [subscribe\(\)](#) returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method *put_nowait* (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class `rtcbot.audio.Speaker`(*samplerate=48000, channels=None, blocksize=1024, device=None, loop=None*)

Bases: [ThreadedSubscriptionConsumer](#)

close()

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

Make sure to run close on exit, since sometimes Python has trouble exiting from multiple threads without having them closed explicitly.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:",myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using [onError\(\)](#), which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

putSubscription(*subscription*)

Given a subscription, such that *await subscription.get()* returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

put_nowait(data)

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property ready

This is `True` when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the `ready` state, since all functions for getting/putting data will work even if the class is still starting up in the background.

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

1.3.5 Inputs

The Inputs API is built as a thin wrapper over the identically-named library ([inputs](#)). If you are having issues, check whether they are coming from RTCBot or from the underlying library.

There are three input devices exposed. A Keyboard, a Mouse and a Gamepad.

Note: To get access to Keyboard and Mouse, you might need to either run as administrator, or on Linux, add your user to the `input` group.

Warning: Keyboard support is experimental - it only works in certain environments. You can try it, but don't be surprised if no events show up.

Mouse

To get mouse events, you can run the following:

```
import asyncio
from rtcbot import Mouse

m = Mouse()

@m.subscribe
def onkey(key):
    print(key)

try:
    asyncio.get_event_loop().run_forever()
finally:
    m.close()
```

This code gives the following results:

```
{'timestamp': 1552629001.833567, 'code': 'REL_X', 'state': 1, 'event': 'Relative'}
{'timestamp': 1552629001.833567, 'code': 'REL_Y', 'state': 1, 'event': 'Relative'}
{'timestamp': 1552629001.841518, 'code': 'REL_X', 'state': 2, 'event': 'Relative'}
{'timestamp': 1552629001.889522, 'code': 'REL_X', 'state': 2, 'event': 'Relative'}
{'timestamp': 1552629001.905525, 'code': 'REL_X', 'state': 3, 'event': 'Relative'}
{'timestamp': 1552629001.905525, 'code': 'REL_Y', 'state': -2, 'event': 'Relative'}
{'timestamp': 1552629002.16957, 'code': 'REL_X', 'state': 2, 'event': 'Relative'}
{'timestamp': 1552629004.233588, 'code': 'MSC_SCAN', 'state': 589825, 'event': 'Misc'}
{'timestamp': 1552629004.233588, 'code': 'BTN_LEFT', 'state': 1, 'event': 'Key'}
{'timestamp': 1552629004.361593, 'code': 'MSC_SCAN', 'state': 589825, 'event': 'Misc'}
{'timestamp': 1552629004.361593, 'code': 'BTN_LEFT', 'state': 0, 'event': 'Key'}
{'timestamp': 1552629005.361596, 'code': 'MSC_SCAN', 'state': 589826, 'event': 'Misc'}
{'timestamp': 1552629005.361596, 'code': 'BTN_RIGHT', 'state': 1, 'event': 'Key'}
```

The REL_X and REL_Y codes refer to relative mouse motion. Here, the mouse started by moving 1 unit to the right (REL_X).

Gamepad

The Gamepad usually refers to a wired Xbox controller. Connect it to your computer through USB. To use the gamepad, you probably don't need administrator access:

```
import asyncio
from rtcbot import Gamepad

g = Gamepad()

@g.subscribe
def onkey(key):
    print(key)

try:
    asyncio.get_event_loop().run_forever()
```

(continues on next page)

(continued from previous page)

```
finally:
    g.close()
```

This code gives the following results:

```
{'timestamp': 1552629513.7494, 'code': 'BTN_SOUTH', 'state': 1, 'event': 'Key'}
{'timestamp': 1552629513.7494, 'code': 'ABS_Y', 'state': -1, 'event': 'Absolute'}
{'timestamp': 1552629513.969403, 'code': 'BTN_SOUTH', 'state': 0, 'event': 'Key'}
{'timestamp': 1552629517.089424, 'code': 'ABS_X', 'state': -253, 'event': 'Absolute'}
{'timestamp': 1552629517.097385, 'code': 'ABS_X', 'state': -64, 'event': 'Absolute'}
{'timestamp': 1552629517.109388, 'code': 'ABS_X', 'state': -211, 'event': 'Absolute'}
{'timestamp': 1552629517.117379, 'code': 'ABS_X', 'state': -242, 'event': 'Absolute'}
```

The resulting events are all button presses and joystick control. For example, ABS_X here refers to the horizontal position of the right joystick on a wired Xbox controller.

API

class `rtcbot.inputs.Gamepad(eventFilter=<function <lambda>>, loop=None)`

Bases: `InputDevice`

`close()`

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

property `closed`

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property `error`

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not `None`, the object is considered crashed, and no longer processing data.

async `get()`

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
```

(continues on next page)

(continued from previous page)

```
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the *onReady()* function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(subscription=None)

Allows subscribing to new data as it comes in, returning a subscription (see *Subscriptions*):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to *subscribe()* returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method *put_nowait* (see *Subscriptions*):

```
q = asyncio.Queue()
myobj.subscribe(q)
```

(continues on next page)

(continued from previous page)

```
while True:
    data = await q.get()
    print(data)
```

2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

`unsubscribe(subscription=None)`

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by `get()`. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from `subscribe()`.

`unsubscribeAll()`

Removes all currently active subscriptions, including the default one if it was initialized.

class `rtcbot.inputs.InputDevice(device, eventFilter=<function <lambda>>, loop=None)`

Bases: `ProcessSubscriptionProducer`

A thin wrapper over `inputs`, which permits getting events in an asynchronous manner.

`close()`

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

property `closed`

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property `error`

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:",myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

`async get()`

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

`onClose(subscription=None)`

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

`onError(subscription=None)`

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The `onError()` function behaves in the same way as a `subscribe()`, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a [subscribe\(\)](#), meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the [onReady\(\)](#) function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(*subscription=None*)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to [subscribe\(\)](#) returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method *put_nowait* (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class rtcbot.inputs.Keyboard(*eventFilter=<function <lambda>>, loop=None*)

Bases: [InputDevice](#)

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not `None`, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(subscription=None)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the *onReady()* function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(*subscription=None*)

Allows subscribing to new data as it comes in, returning a subscription (see *Subscriptions*):


```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method `put_nowait` (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from *subscribe()*.

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

class rtcbot.inputs.**Mouse**(*eventFilter=<function <lambda>>, loop=None*)

Bases: *InputDevice*

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual *Exception* that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:",myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using *onError()*, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

async get()

Behaves similarly to *subscribe().get()*. On the first call, creates a default subscription, and all subsequent calls to *get()* use that subscription.

If *unsubscribe()* is called, the subscription is deleted, so a subsequent call to *get()* will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
```

(continues on next page)

(continued from previous page)

```
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the `ready` state, since all functions for getting/putting data will work even if the class is still starting up in the background.

subscribe(*subscription=None*)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters

subscription (*optional*) –

An optional existing subscription to subscribe to. This can be one of 3 things:

- 1) An object which has the method `put_nowait` (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.

Parameters

subscription (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

`rtcbot.inputs.defaultFilter(x)`

1.3.6 Arduino

The Pi can control certain hardware directly, but a dedicated microcontroller can be better for real-time tasks like controlling servos or certain sensors. The code here is dedicated to efficiently interfacing with microcontrollers, and was tested to work with both Arduino and ESP32, but should work with any hardware with a serial port and C compiler.

You can connect a Pi to an Arduino using a USB cable (easiest), or, with the help of a [level shifter](#), directly through the Pi's hardware serial pins ([BCM 14 and 15](#), see [here](#) for details). The `SerialConnection` class is provided to easily send and receive asynchronous commands as your robot is doing other processing.

Basic Communication

Assuming that you have connected the Pi to an Arduino with a USB cable, you can read and write to the serial port as follows:

```
import asyncio
from rtcbot.arduino import SerialConnection

conn = SerialConnection("/dev/ttyAMA1")

async def sendAndReceive(conn):
    conn.put_nowait("Hello world!")
    while True:
        msg = await conn.get().decode('ascii')
```

(continues on next page)

(continued from previous page)

```

    print(msg)
    await asyncio.sleep(1)

asyncio.ensure_future(sendAndReceive(conn))

asyncio.get_event_loop().run_forever()

```

This sends a Hello World to the Arduino, and then reads the incoming serial messages line by line. Given the corresponding Arduino code,

```

void setup() {
    Serial.begin(115200);
}
void loop() {
    if (Serial.available() > 0) {
        Serial.print("I received: ");
        Serial.println(Serial.read());
    }
}

```

you should get the messages:

```

I received: H
I received: e
...

```

By default, `SerialConnection` reads line by line. To get raw input as it comes in, you can set the `readFormat` to `None`:

```
conn = SerialConnection("/dev/ttyAMA0", readFormat=None)
```

While reading/writing strings is useful for debugging, for speed and robustness, it is recommended that communication with the Arduino be performed through C structs.

C Struct Messaging

When using a struct write format, a Python dict or tuple is directly encoded by the `SerialConnection`, and is read by the Arduino in a way that the values are directly available for use.

As an example, we will write control messages to the Arduino. On the arduino, you need to create an associated struct into which messages will be received:

```

#include <stdint.h>
typedef __attribute__((packed)) struct {
    int16_t value1;
    uint8_t value2;
} controlMessage;

```

The packed attribute ensures that the arduino's struct is compatible with the encoding performed by Python.

From Python, you need to give the `SerialConnection` the structure shape in the format expected by Python's [structure packing library](#). The arduino is little endian (each string should start with "<"). For example, we need to tell the `SerialConnection` that first element of the struct is called "value1", and is a 16 bit integer (the default int size on a standard Arduino). This corresponds to the format character "h" (see structure packing [table of format values](#)).

```
conn = SerialConnection(
    url="/dev/ttyAMA1",
    writeFormat="<hB",
    writeKeys=["value1", "value2"]
)
```

With this format, you can send messages to the Arduino as dicts:

```
conn.put_nowait({"value1": -23, "value2": 101})
```

To decode them on the Arduino, you can read:

```
controlMessage msg;
Serial.read((char*)&msg, sizeof(msg));
```

Similarly, you can also send structs to Python from the Arduino:

```
typedef __attribute__((packed)) struct {
    uint8_t sensorID;
    uint16_t measurement;
} sensorMessage;
```

and:

```
sensorMessage msg = { .sensorID = 12, .measurement=123 };
Serial.write((char*)&msg, sizeof(msg));
```

You can then get the message directly as a Python dict:

```
conn = SerialConnection(
    url="/dev/ttyAMA1",
    readFormat="<BH",
    readKeys=["sensorID", "measurement"]
)

# Run this in a coroutine
print(await conn.get() )
# {"sensorID": 12, "measurement": 123}
```

Full Example

The above can be demonstrated with a full example that sends and receives messages:

```
// The controlMessage comes from the pi
typedef __attribute__((packed)) struct {
    uint16_t value1;
    uint8_t value2;
} controlMessage;

// We write this back to the Pi
typedef __attribute__((packed)) struct {
    uint8_t value1;
    uint16_t value2;
```

(continues on next page)

(continued from previous page)

```

} sensorMessage;

// These are the specific message instances
controlMessage cMsg;
sensorMessage sMsg;

void setup() {
    Serial.begin(115200);
}
void loop() {

    // Read the control message
    Serial.readBytes((char*)&cMsg,sizeof(cMsg));

    // set up the sensor message
    sMsg.value1 = cMsg.value2;
    sMsg.value2 = cMsg.value1;

    // Send it back!
    Serial.write((char*)&sMsg,sizeof(sMsg));
}

```

The above code echoes the values sent to it, with value1 and value2 switched. The python code to read it is:

```

import asyncio
from rtcbot.arduino import SerialConnection

loop = asyncio.get_event_loop()

sc = SerialConnection(
    url="/dev/ttyAMA1",
    writeFormat="<HB",
    writeKeys=["value1", "value2"],
    readFormat="<BH",
    readKeys=["value1", "value2"],
    loop=loop
)

async def sendAndReceive(sc):
    while True:
        sc.put_nowait({"value1": 1003,"value2": 2})
        msg = await sc.get()
        print("Received:",msg)
        await asyncio.sleep(1)

asyncio.ensure_future(sendAndReceive(sc))

try:
    loop.run_forever()
finally:
    print("Exiting Event Loop")
    loop.run_until_complete(loop.shutdown_asyncgens())

```

(continues on next page)

(continued from previous page)

```
loop.close()
```

Running this program, you get:

```
Received: {"value1": 2,"value2": 1003}
Received: {"value1": 2,"value2": 1003}
Received: {"value1": 2,"value2": 1003}
```

API

```
class rtcbot.arduino.SerialConnection(url='/dev/ttyS0', readFormat='\n', writeFormat=None,
                                     baudrate=115200, writeKeys=None, readKeys=None,
                                     startByte=None, delayWriteStart=0, loop=None)
```

Bases: *SubscriptionProducerConsumer*

Handles sending and receiving commands to/from a serial port. Has built-in support for sending structs to/from Arduinos.

By default, reads and writes bytes from/to the serial port, splitting incoming messages by newline. To return raw messages (without splitting), use `readFormat=None`.

If a `writeFormat` or `readFormat` is given, they are interpreted as *struct format strings*, and all incoming or outgoing messages are assumed to conform to the given format. Without setting `readKeys` or `writeKeys`, the messages are assumed to be tuples or lists.

When given a list of strings for `readKeys` or `writeKeys`, the write or read formats are assumed to come from objects with the given keys. Using these, a `SerialConnection` can read/write python dicts to the associated structure formats.

`close()`

Cleans up and closes the object.

property `closed`

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property `error`

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:",myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not *None*, the object is considered crashed, and no longer processing data.

`async get()`

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

onClose(*subscription=None*)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(*subscription=None*)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the *error* event is fired, with the associated *Exception*. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ",err)
```

The *onError()* function behaves in the same way as a *subscribe()*, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(*subscription=None*)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a *ready* event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a *subscribe()*, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

putSubscription(subscription)

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

put_nowait(data)

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The `put_nowait()` method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through `putSubscription()`, it is immediately stopped, and the object waits only for `put_nowait()`:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

property ready

This is `True` when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the `onReady()` function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the `ready` state, since all functions for getting/putting data will work even if the class is still starting up in the background.

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

subscribe(subscription=None)

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters**subscription** (*optional*) –**An optional existing subscription to subscribe to. This can be one of 3 things:**

- 1) An object which has the method *put_nowait* (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.**Parameters****subscription** (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).

unsubscribeAll()

Removes all currently active subscriptions, including the default one if it was initialized.

1.3.7 Javascript

The Javascript API for RTCBot is provided for simple interoperability of RTCBot and the browser. Wherever possible, the Javascript API mirrors the Python API, and can be used in exactly the same way.

Note: The Javascript API includes only a minimal subset of the functionality of RTCBot's Python version. While this may change in the future, many of the functions available in Python can't be used in javascript.

Basic Usage

To start using the Javascript API, all you need to do is include the *RTCBot.js* file in a script tag, and use the following javascript:

```
// The connection object
var conn = new rtcbot.RTCConnection();

// Here we set up the connection. We put it in an async function, since we will be
// waiting for results from the server (Promises).
async function connect() {
    // Get the information needed to connect from the server to the browser
    let offer = await conn.getLocalDescription();

    // POST the information to the server, which will respond with the corresponding
    // remote
    // connection's description
    let response = await fetch("/connect", {
        method: "POST",
        cache: "no-cache",
        body: JSON.stringify(offer)
    });

    // Set the remote server's information
    await conn.setRemoteDescription(await response.json());
}

connect(); // Run the async function in the background.
```

Next, to establish the connection with Python, you include the Python counterpart:

```
from aiohttp import web
routes = web.RouteTableDef()

from rtcbot import RTCConnection, getRTCBotJS
conn= None

@routes.get("/") # Serve the html file
async def index(request):
```

(continues on next page)

(continued from previous page)

```

with open("index.html", "r") as f:
    return web.Response(content_type="text/html", text=f.read())

# Serve the RTCBot javascript library at /rtcbot.js
@routes.get("/rtcbot.js")
async def rtcbotjs(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

@routes.post("/connect")
async def connect(request):
    global conn
    clientOffer = await request.json()
    conn = RTCConnection()

    response = await conn.getLocalDescription(clientOffer)
    return web.json_response(response)

async def cleanup(app=None):
    if conn is not None:
        await conn.close()

app = web.Application()
app.add_routes(routes)
app.on_shutdown.append(cleanup)
web.run_app(app, port=8080)

```

Python API

`rtcbot.javascript.getRTCBotJS()`

Returns the RTCBot javascript. This allows you to easily write self-contained scripts. You can serve it like this:

```

from rtcbot import getRTCBotJS
from aiohttp import web
routes = web.RouteTableDef()

@routes.get("/rtcbot.js")
async def rtcbotJS(request):
    return web.Response(content_type="application/javascript", text=getRTCBotJS())

app = web.Application()
app.add_routes(routes)
web.run_app(app, port=8000)

```

If you are writing a more complex application, you might want to bundle RTCBot's javascript with your code using rollup or webpack instead of including it in script tags. To do this, you can install the js library separately with npm, and bundle it however you'd like:

```
npm i rtcbot
```

Javascript API

class `RTCCConnection`(*defaultOrdered=true*, *rtcConfiguration*)

`RTCCConnection` mirrors the Python `RTCCConnection` in API. Whatever differences in functionality that may exist can be considered bugs unless explicitly documented as such.

For detailed documentation, see the `RTCCConnection` docs for Python.

Arguments

- **`defaultOrdered`** (*) –
- **`rtcConfiguration`** (*) – is the configuration given to the RTC connection

`RTCCConnection.audio`

The audio element allows you to directly access audio streams. The following functions are available:

- *`subscribe()`*: Unlike in Python, this is given a callback which is called *once*, when the stream is received.

```
conn.audio.subscribe(function(stream) {  
    document.querySelector("audio").srcObject = stream;  
});
```

- *`putSubscription()`*: Allows to send a video stream:

```
let streams = await navigator.mediaDevices.getUserMedia({audio: true, ↵  
↵video: false});  
conn.audio.putSubscription(streams.getAudioTracks()[0]);
```

`RTCCConnection.video`

Just like in the Python version, the video element allows you to directly access video streams. The following functions are available:

- *`subscribe()`*: Unlike in Python, this is given a callback which is called *once*, when the stream is received.

```
conn.video.subscribe(function(stream) {  
    document.querySelector("video").srcObject = stream;  
});
```

- *`putSubscription()`*: Allows to send a video stream:

```
let streams = await navigator.mediaDevices.getUserMedia({audio: false, ↵  
↵video: true});  
conn.video.putSubscription(streams.getVideoTracks()[0]);
```

`RTCCConnection.close()`

Close the connection

`RTCCConnection.getLocalDescription(description=null)`

Sets up the connection. If no description is passed in, creates an initial description. If a description is given, creates a response to it.

Arguments

- **description** (*) – (optional)

`RTCCConnection.put_nowait(msg)`

Send the given data over a data stream.

Arguments

- **msg** (*) – Message to send

`RTCCConnection.setRemoteDescription(description)`

When initializing a connection, this response reads the remote response to an initial description.

Arguments

- **description** (*) –

`RTCCConnection.subscribe(s)`

Subscribe to incoming messages. Unlike in the Python library, which can accept a wide variety of inputs, the *subscribe* function in javascript only allows simple callbacks.

Arguments

- **s** (*) – A function to call each time a new message comes in

class Keyboard()

Keyboard subscribes to keypresses on the keyboard. Internally, the *keydown* and *keyup* events are used to get keys.

```
var kb = new rtcbot.Keyboard();
kb.subscribe(function(event) {
  console.log(event); // prints the button and joystick events
})
```

`Keyboard.close()`

Stop listening to keypresses

`Keyboard.subscribe(s)`

Subscribe to the events. Unlike in the Python library, which can accept a wide variety of inputs, the *subscribe* function in javascript only allows simple callbacks.

Arguments

- **s** (*) – A function to call on each event

class Gamepad()

Gamepad allows you to use an Xbox controller. It uses the browser Gamepad API, polling at 10Hz by default. Use *rtcbot.setGamepadRate* to change polling frequency.

You must plug in the gamepad, and press a button on it for it to be recognized by the browser:

```
var gp = new rtcbot.Gamepad();
gp.subscribe(function(event) {
  console.log(event); // prints the button and joystick events
})
```

`Gamepad.close()`

Stop polling the gamepad.

`Gamepad.subscribe(s)`

Subscribe to the events. Unlike in the Python library, which can accept a wide variety of inputs, the *subscribe* function in javascript only allows simple callbacks.

Arguments

- **s** (*) – A function to call on each event

`setGamepadRate(rate)`

Gamepads are polled at 10Hz by default, so that when moving joystick axes a connection is not immediately flooded with every miniscule joystick change. To modify this behavior, you can set the rate in Hz, allowing lower latency, with the downside of potentially lots of data suddenly overwhelming a connection.

Arguments

- **rate** (*number*) – Rate at which gamepad is polled in Hz

`class Queue()`

A simple async queue. Useful for converting callbacks into async operations. The API imitates Python's `asyncio.Queue`, making it easy to avoid callback hell

1.3.8 Subscriptions

The subscriptions available here are quick solutions to common problems that come up with the async producer/consumer model.

API

`class rtcbot.subscriptions.CallbackSubscription(callback, loop=None, runDirect=False)`

Bases: object

Sometimes you don't want to await anything, you just want to run a callback upon an event. The `CallbackSubscription` allows you to do precisely that:

```
@CallbackSubscription
async def mycallback(value):
    print(value)

cam = CVCamera()
cam.subscribe(mycallback)
```

Note: This is no longer necessary: you can just pass a function to *subscribe*, and it will automatically be wrapped in a *CallbackSubscription*.

`put_nowait(element)`

`class rtcbot.subscriptions.DelayedSubscription(SubscriptionWriter, subscription=None)`

Bases: object

In some instances, you want to subscribe to something, but don't actually want to start gathering the data until the data is needed.

This is especially common in something like audio streaming: if you were to subscribe to an audio stream right now, and `get()` the data only after a certain time, then there would be a large audio delay, because by default the audio subscription queues data.

This is common in the audio of an `RTCCConnection`, where `get` is called only once the connection is established:

```
s = Microphone().subscribe()
conn = RTCCConnection()
conn.audio.putSubscription(s) # Big audio delay!
```

Instead, what you want to do is delay subscribing until `get` is called the first time, which would wait until the connection is ready to start sending data:

```
s = DelayedSubscription(Microphone())
conn = RTCCConnection()
conn.audio.putSubscription(s) # Calls Microphone.subscribe() on first get()
```

One caveat is that calling `unsubscribe` will not work on the `DelayedSubscription` - you must use `unsubscribe` as given in the `DelayedSubscription`! That means:

```
m = Microphone()
s = DelayedSubscription(m)
m.unsubscribe(s) # ERROR!

s.unsubscribe() # correct!
```

Parameters

- **SubscriptionWriter** (*BaseSubscriptionWriter*) – An object with a `subscribe` method
- **subscription** (*optional*) – The subscription to subscribe. If given, calls *SubscriptionWriter.subscribe(subscription)*

async `get()`

unsubscribe()

class `rtcbot.subscriptions.EventSubscription`

Bases: `object`

This is a subscription that is fired once - upon the first insert.

async `get()`

put_nowait(*value*)

class `rtcbot.subscriptions.GetterSubscription(callback)`

Bases: `object`

You might have a function which behaves like a `get()`, but it is just a function. The `GetterSubscription` is a wrapper that calls your function on `get()`:

```
@GetterSubscription
async def myfunction():
    asyncio.sleep(1)
    return "hello!"
```

(continues on next page)

(continued from previous page)

```
await myfunction.get()
# returns "hello!"
```

async get()

class rtcbot.subscriptions.MostRecentSubscription

Bases: object

The MostRecentSubscription always returns the most recently added element. If you get an element and immediately call get again, it will wait until the next element is received, it will not return elements that were already processed.

It is not threadsafe.

async get()

Gets the most recently added element

put_nowait(element)

Adds the given element to the subscription.

class rtcbot.subscriptions.RebatchSubscription(samples, axis=0, subscription=None)

Bases: object

In certain cases, data comes with a suboptimal batch size. For example, audio coming from an *RTCTConnection* is always of shape (960,2), with 2 channels, and 960 samples per batch. This subscription allows you to change the frame size by mixing and matching batches. For example:

```
s = RebatchSubscription(samples=1024,axis=0)
s.put_nowait(np.zeros((960,2)))

# asyncio.TimeoutError - the RebatchSubscription does
# not have enough data to create a batch of size 1024
rebatched = await asyncio.wait_for(s.get(),timeout=5)

# After adding another batch of 960, get returns a frame of goal shape
s.put_nowait(np.zeros((960,2)))
rebatched = await s.get()
print(rebatched.shape) # (1024,2)
```

The RebatchSubscription takes samples from the second data frame's dimension 1 to create a new batch of the correct size.

async get()

put_nowait(data)

1.3.9 Base

RTCBot is heavily based upon the concept of data producers, and data consumers. To that end, all classes that produce data, such as cameras, microphones, and incoming data streams are considered producers, and all classes that consume data, such as speakers, video displays or outgoing data streams are considered consumers.

This section of the documentation is built to describe the backend base classes upon which all of the data streams are based, and to help you create your own producers and consumers with an API compatible with the rest of RTCBot.

There are 3 main base classes types

- 1) BaseSubscriptionProducer and BaseSubscriptionConsumer
- 2) ThreadedSubscriptionProducer and ThreadedSubscriptionConsumer
- 3) MultiprocessSubscriptionProducer

The three types allow setting up your own data acquisition and processing code loops without needing to worry about the asyncio loop (Threaded) or even the GIL (Multiprocess), but also come with the downside of increasing complexity and communication overhead.

API

Note: Unlike elsewhere in RTCBot's documentation, inherited members are not shown here, so some functions available from a class might be hidden if they were defined in a parent.

class `rtcbot.base.events.baseEventHandler(logger)`

Bases: object

This class handles base events

_setError(*value*)

Sets the error state of the class to an error that was caught while processing data.

After the error is set, the class is assumed to be in a closed state, meaning that any background processes either crashed or were shut down.

Warning: Only call this if you are subclassing `baseEventHandler`.

_setReady(*value=True*)

Sets the ready to a given value, and fires all subscriptions created with `onReady()`. Call this when your producer/consumer is fully initialized.

Warning: Only call this if you are subclassing `baseEventHandler`.

close()

Fires the onClose event

property closed

Returns whether the object was closed. This includes both thrown exceptions, and clean exits.

property error

If there is an error that causes the underlying process to crash, this property will hold the actual `Exception` that was thrown:

```
if myobject.error is not None:
    print("Oh no! There was an error:", myobject.error)
```

This property is offered for convenience, but usually, you will want to subscribe to the error by using `onError()`, which will notify your app when the issue happens.

Note: If the error is not `None`, the object is considered crashed, and no longer processing data.

onClose(subscription=None)

This is mainly useful for connections - they can be closed remotely. This allows handling the close event.

```
@myobj.onClose
def closeCallback():
    print("Closed!")
```

Be aware that this is equivalent to explicitly awaiting the object:

```
await myobj
```

onError(subscription=None)

Since most data processing happens in the background, the object might encounter an error, and the data processing might crash. If there is a crash, the object is considered dead, and no longer gathering data.

To catch these errors, when an unhandled exception happens, the `error` event is fired, with the associated `Exception`. This function allows you to subscribe to these events:

```
@myobj.onError
def error_happened(err):
    print("Crap, stuff just crashed: ", err)
```

The `onError()` function behaves in the same way as a `subscribe()`, which means that you can pass it a coroutine, or even directly await it:

```
err = await myobj.onError()
```

onReady(subscription=None)

Creating the class does not mean that the object is ready to process data. When created, the object starts an initialization procedure in the background, and once this procedure is complete, and any spawned background workers are ready to process data, it fires a `ready` event.

This function allows you to listen for this event:

```
@myobj.onReady
def readyCallback():
    print("Ready!")
```

The function works in exactly the same way as a `subscribe()`, meaning that you can pass it a coroutine, or even await it directly:

```
await myobj.onReady()
```

Note: The object will automatically handle any subscriptions or inserts that happen while it is initializing, so you generally don't need to worry about the ready event, unless you need exact control.

property ready

This is *True* when the class has been fully initialized, and is ready to process data:

```
if not myobject.ready:
    print("Not ready to process data")
```

This property is offered for convenience, but if you want to be notified when ready to process data, you will want to use the *onReady()* function, which will allow you to set up a callback/coroutine to wait until initialized.

Note: You usually don't need to check the *ready* state, since all functions for getting/putting data will work even if the class is still starting up in the background.

class rtcbot.base.events.threadedEventHandler(*logger, loop=None*)

Bases: *baseEventHandler*

A threadsafe version of *baseEventHandler*.

_setError(*err*)

Threadsafe version of *baseEventHandler._setError()*.

_setReady(*value*)

Threadsafe version of *baseEventHandler._setReady()*.

class rtcbot.base.base.BaseSubscriptionConsumer(*directPutSubscriptionType=<class 'asyncio.queues.Queue'>, logger=None*)

Bases: *baseEventHandler*

A base class upon which consumers of subscriptions can be built.

The BaseSubscriptionConsumer class handles the logic of switching incoming subscriptions mid-stream and all the other annoying stuff.

async **_get**()

Warning: Only call this if you are subclassing *BaseSubscriptionConsumer*.

This function is to be awaited by a subclass to get the next datapoint from the active subscription. It internally handles the subscription for you, and transparently manages the user switching a subscription during runtime:

```
myobj.putSubscription(x)
# await self._get() waits on next datapoint from x
myobj.putSubscription(y)
# _get transparently switched to waiting on y
```

Raises

SubscriptionClosed – If *close()* was called, this error is raised, signalling your data processing function to clean up and exit.

Returns

The next datapoint that was put or subscribed to from the currently active subscription.

close()

Cleans up and closes the object.

putSubscription(subscription)

Given a subscription, such that *await subscription.get()* returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

put_nowait(data)

This function allows you to directly send data to the object, without needing to go through a subscription:

```
while True:
    data = get_data()
    myobj.put_nowait(data)
```

The *put_nowait()* method is the simplest way to process a new chunk of data.

Note: If there is currently an active subscription initialized through *putSubscription()*, it is immediately stopped, and the object waits only for *put_nowait()*:

```
myobj.putSubscription(s)
myobj.put_nowait(mydata) # unsubscribes from s

assert myobj.subscription is None
```

stopSubscription()

Stops reading the current subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)

assert myobj.subscription == q

myobj.stopSubscription()

assert myobj.subscription is None

# You can then subscribe again (or put_nowait)
myobj.putSubscription(q)
assert myobj.subscription == q
```

The object is not affected, other than no longer listening to the subscription, and not processing new data until something is inserted.

property subscription

Returns the currently active subscription:

```
q = asyncio.Queue()
myobj.putSubscription(q)
assert myobj.subscription == q

myobj.stopSubscription()
assert myobj.subscription is None

myobj.put_nowait(data)
assert myobj.subscription is None
```

```
class rtcbot.base.base.BaseSubscriptionProducer(defaultSubscriptionClass=<class
                                             'asyncio.queue.Queue'>,
                                             defaultAutosubscribe=False, logger=None)
```

Bases: *baseEventHandler*

This is a base class upon which all things that emit data in RTCBot are built.

This class offers all the machinery necessary to keep track of subscriptions to the incoming data. The most important methods from a user's perspective are the *subscribe()*, *get()* and *close()* functions, which manage subscriptions to the data, and finally close everything.

From an subclass's perspective, the most important pieces are the *_put_nowait()* method, and the *_shouldClose* and *_ready* attributes.

Once the subclass is ready, it should set *_ready* to True, and when receiving data, it should call *_put_nowait()* to insert it. Finally, it should either listen to *_shouldClose* or override the close method to stop producing data.

Example

A sample basic class that builds on the `BaseSubscriptionProvider`:

```
class MyProvider(BaseSubscriptionProvider):
    def __init__(self):
        super().__init__()

        # Add data in the background
        asyncio.ensure_future(self._dataProducer)

    async def _dataProducer(self):
        self._ready = True
        while not self._shouldClose:
            data = await get_data_here()
            self._put_nowait(data)
        self._ready = False
    def close():
        super().close()
        stop_gathering_data()

# you can now subscribe to the data
s = MyProvider().subscribe()
```

Parameters

- **defaultSubscriptionClass** (*optional*) – The subscription type to return by default if `subscribe()` is called without arguments. By default, it uses `asyncio.Queue`:

```
sp = SubscriptionProducer(defaultSubscriptionClass=asyncio.Queue)
q = sp.subscribe()

q is asyncio.Queue # True
```

- **defaultAutosubscribe** (*bool, optional*) – Calling `get()` creates a default subscription on first time it is called. Sometimes the data is very critical, and you want the default subscription to be created right away, so it never misses data. Be aware, though, if your `defaultSubscriptionClass` is `asyncio.Queue`, if `get()` is never called, such as when someone just uses `subscribe()`, it will just keep piling up queued data! To avoid this, it is *False* by default.
- **logger** (*optional*) – Your class logger - it gets a child of this logger for debug messages. If nothing is passed, creates a root logger for your class, and uses a child for that.
- **ready** (*bool, optional*) – Your producer probably doesn't need setup time, so this is set to *True* automatically, which automatically sets `_ready`. If you need to do background tasks, set this to *False*.

`_close()`

This function allows closing from the handler itself. Don't call `close()` directly when implementing producers or consumers. call `_close` instead.

`_put_nowait(element)`

Used by subclasses to add data to all subscriptions. This method internally calls all registered callbacks for you, so you only need to worry about the single function call.

Warning: Only call this if you are subclassing `BaseSubscriptionProducer`.

`_shouldClose`

Whether or not `close()` was called, and the user wants the class to stop gathering data. Should only be accessed from a subclass.

`close()`

Shuts down the data gathering, and removes all subscriptions.

`async get()`

Behaves similarly to `subscribe().get()`. On the first call, creates a default subscription, and all subsequent calls to `get()` use that subscription.

If `unsubscribe()` is called, the subscription is deleted, so a subsequent call to `get()` will create a new one:

```
data = await myobj.get() # Creates subscription on first call
data = await myobj.get() # Same subscription
myobj.unsubscribe()
data2 = await myobj.get() # A new subscription
```

The above code is equivalent to the following:

```
defaultSubscription = myobj.subscribe()
data = await defaultSubscription.get()
data = await defaultSubscription.get()
myobj.unsubscribe(defaultSubscription)
newDefaultSubscription = myobj.subscribe()
data = await newDefaultSubscription.get()
```

`subscribe(subscription=None)`

Allows subscribing to new data as it comes in, returning a subscription (see [Subscriptions](#)):

```
s = myobj.subscribe()
while True:
    data = await s.get()
    print(data)
```

There can be multiple subscriptions active at the same time, each of which get identical data. Each call to `subscribe()` returns a new, independent subscription:

```
s1 = myobj.subscribe()
s2 = myobj.subscribe()
while True:
    assert await s1.get() == await s2.get()
```

This function can also be used as a callback:

```
@myobj.subscribe
def newData(data):
    print("Got data:", data)
```

If passed an argument, it attempts to use the given callback/coroutine/subscription to notify of incoming data.

Parameters**subscription** (*optional*) –**An optional existing subscription to subscribe to. This can be one of 3 things:**

- 1) An object which has the method *put_nowait* (see [Subscriptions](#)):

```
q = asyncio.Queue()
myobj.subscribe(q)
while True:
    data = await q.get()
    print(data)
```

- 2) A callback function - this will be called the moment new data is inserted:

```
@myobj.subscribe
def myfunction(data):
    print(data)
```

- 3) An coroutine callback - A future of this coroutine is created on each insert:

```
@myobj.subscribe
async def myfunction(data):
    await asyncio.sleep(5)
    print(data)
```

Returns

A subscription. If one was passed in, returns the passed in subscription:

```
q = asyncio.Queue()
ret = thing.subscribe(q)
assert ret==q
```

unsubscribe(*subscription=None*)

Removes the given subscription, so that it no longer gets updated:

```
subs = myobj.subscribe()
myobj.unsubscribe(subs)
```

If no argument is given, removes the default subscription created by *get()*. If none exists, then does nothing.**Parameters****subscription** (*optional*) – Anything that was passed into/returned from [subscribe\(\)](#).**unsubscribeAll()**

Removes all currently active subscriptions, including the default one if it was initialized.

class rtcbot.base.base.NoClosedSubscription(*awaitable*)

Bases: object

NoClosedSubscription wraps a callback, and doesn't pass forward SubscriptionClosed errors - it converts them to asyncio.CancelledError. This allows exiting the application in a clean way.

exception rtcbot.base.base.SubscriptionClosed

Bases: Exception

This error is returned internally by *_get()* in all subclasses of [BaseSubscriptionConsumer](#) when *close()* is called, and signals the consumer to shut down. For more detail, see [BaseSubscriptionConsumer._get\(\)](#).

```
class rtcbot.base.base.SubscriptionConsumer(directPutSubscriptionType=<class
                                           'asyncio.queues.Queue'>, logger=None)
```

Bases: [BaseSubscriptionConsumer](#)

```
class rtcbot.base.base.SubscriptionProducer(defaultSubscriptionClass=<class 'asyncio.queues.Queue'>,
                                           defaultAutosubscribe=False, logger=None)
```

Bases: [BaseSubscriptionProducer](#)

```
class rtcbot.base.base.SubscriptionProducerConsumer(directPutSubscriptionType=<class
                                                    'asyncio.queues.Queue'>,
                                                    defaultSubscriptionType=<class
                                                    'asyncio.queues.Queue'>, logger=None,
                                                    defaultAutosubscribe=False)
```

Bases: [BaseSubscriptionConsumer](#), [BaseSubscriptionProducer](#)

This base class represents an object which is both a producer and consumer. This is common with two-way connections.

Here, you call `_get()` to consume the incoming data, and `_put_nowait()` to produce outgoing data.

`_close()`

This function allows closing from the handler itself. Don't call `close()` directly when implementing producers or consumers. call `_close` instead.

`close()`

Cleans up and closes the object.

```
class rtcbot.base.thread.ThreadedSubscriptionConsumer(directPutSubscriptionType=<class
                                                    'asyncio.queues.Queue'>, logger=None,
                                                    loop=None, daemonThread=True)
```

Bases: [BaseSubscriptionConsumer](#), [threadedEventHandler](#)

`close()`

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

Make sure to run `close` on exit, since sometimes Python has trouble exiting from multiple threads without having them closed explicitly.

`putSubscription(subscription)`

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

```
class rtcbot.base.thread.ThreadedSubscriptionProducer(defaultSubscriptionType=<class
                                                    'asyncio.queues.Queue'>, logger=None,
                                                    loop=None, daemonThread=True)
```

Bases: [BaseSubscriptionProducer](#), [threadedEventHandler](#)

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when exiting the program.

```
class rtcbot.base.multiprocess.ProcessSubscriptionConsumer(directPutSubscriptionType=<class
                                                         'asyncio.queues.Queue'>,
                                                         logger=None, loop=None,
                                                         daemonProcess=True, joinTimeout=1)
```

Bases: [BaseSubscriptionConsumer](#)

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

putSubscription(subscription)

Given a subscription, such that *await subscription.get()* returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1
```

(continues on next page)

(continued from previous page)

```
q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

```
class rtcbot.base.multiprocess.ProcessSubscriptionProducer(defaultSubscriptionType=<class
    'asyncio.queues.Queue'>,
    logger=None, loop=None,
    daemonProcess=True, joinTimeout=1)
```

Bases: [BaseSubscriptionProducer](#)

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

```
class rtcbot.base.multiprocess.ProcessSubscriptionProducerConsumer(directPutSubscriptionType=<class
    'asyncio.queues.Queue'>,
    defaultSubscription-
    Type=<class
    'asyncio.queues.Queue'>,
    logger=None,
    defaultAutosubscribe=False,
    loop=None,
    daemonProcess=True,
    joinTimeout=1)
```

Bases: [BaseSubscriptionConsumer](#), [BaseSubscriptionProducer](#)

This base class represents an object which is both a producer and consumer, run as a separate process. This is common with two-way connections. Here, you call `_get()` to consume the incoming data, and `_put_nowait()` to produce outgoing data.

close()

Shuts down data gathering, and closes all subscriptions. Note that it is not recommended to call this in an async function, since it waits until the background thread joins.

The object is meant to be used as a singleton, which is initialized at the start of your code, and is closed when shutting down.

putSubscription(subscription)

Given a subscription, such that `await subscription.get()` returns successive pieces of data, keeps reading the subscription forever:

```
q = asyncio.Queue() # an asyncio.Queue has a get() coroutine
myobj.putSubscription(q)

q.put_nowait(data)
```

Equivalent to doing the following in the background:

```
while True:
    myobj.put_nowait(await q.get())
```

You can replace a currently running subscription with a new one at any point in time:

```
q1 = asyncio.Queue()
myobj.putSubscription(q1)

assert myobj.subscription == q1

q2 = asyncio.Queue()
myobj.putSubscription(q2)

assert myobj.subscription == q2
```

class rtcbot.base.multiprocess.**internalSubscriptionMessage**(*type, value*)

Bases: object

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

r

- `rtcbot.arduino`, 117
- `rtcbot.audio`, 92
- `rtcbot.base`, 126
 - `rtcbot.base.base`, 131
 - `rtcbot.base.events`, 129
 - `rtcbot.base.multiprocess`, 138
 - `rtcbot.base.thread`, 137
- `rtcbot.camera`, 79
- `rtcbot.connection`, 51
- `rtcbot.inputs`, 100
- `rtcbot.javascript`, 123
- `rtcbot.subscriptions`, 126
- `rtcbot.websocket`, 72

Symbols

- `_close()` (*rtcbot.base.base.BaseSubscriptionProducer* method), 134
- `_close()` (*rtcbot.base.base.SubscriptionProducerConsumer* method), 137
- `_get()` (*rtcbot.base.base.BaseSubscriptionConsumer* method), 131
- `_put_nowait()` (*rtcbot.base.base.BaseSubscriptionProducer* method), 134
- `_setError()` (*rtcbot.base.events.baseEventHandler* method), 129
- `_setError()` (*rtcbot.base.events.threadedEventHandler* method), 131
- `_setReady()` (*rtcbot.base.events.baseEventHandler* method), 129
- `_setReady()` (*rtcbot.base.events.threadedEventHandler* method), 131
- `_shouldClose` (*rtcbot.base.base.BaseSubscriptionProducer* attribute), 135
- A**
- `addDataChannel()` (*rtcbot.connection.RTCCConnection* method), 67
- `addTrack()` (*rtcbot.connection.ConnectionAudioHandler* method), 52
- `addTrack()` (*rtcbot.connection.ConnectionVideoHandler* method), 57
- `audio` (*rtcbot.connection.RTCCConnection* property), 67
- B**
- `baseEventHandler` (class in *rtcbot.base.events*), 129
- `BaseSubscriptionConsumer` (class in *rtcbot.base.base*), 131
- `BaseSubscriptionProducer` (class in *rtcbot.base.base*), 133
- C**
- `CallbackSubscription` (class in *rtcbot.subscriptions*), 126
- `close()` (*rtcbot.arduino.SerialConnection* method), 117
- `close()` (*rtcbot.audio.Microphone* method), 92
- `close()` (*rtcbot.audio.Speaker* method), 95
- `close()` (*rtcbot.base.base.BaseSubscriptionConsumer* method), 132
- `close()` (*rtcbot.base.base.BaseSubscriptionProducer* method), 135
- `close()` (*rtcbot.base.base.SubscriptionProducerConsumer* method), 137
- `close()` (*rtcbot.base.events.baseEventHandler* method), 129
- `close()` (*rtcbot.base.multiprocess.ProcessSubscriptionConsumer* method), 138
- `close()` (*rtcbot.base.multiprocess.ProcessSubscriptionProducer* method), 139
- `close()` (*rtcbot.base.multiprocess.ProcessSubscriptionProducerConsumer* method), 139
- `close()` (*rtcbot.base.thread.ThreadedSubscriptionConsumer* method), 137
- `close()` (*rtcbot.base.thread.ThreadedSubscriptionProducer* method), 138
- `close()` (*rtcbot.camera.CVCamera* method), 79
- `close()` (*rtcbot.camera.CVDisplay* method), 82
- `close()` (*rtcbot.camera.PiCamera* method), 85
- `close()` (*rtcbot.camera.PiCamera2* method), 88
- `close()` (*rtcbot.connection.ConnectionAudioHandler* method), 52
- `close()` (*rtcbot.connection.ConnectionVideoHandler* method), 57
- `close()` (*rtcbot.connection.DataChannel* method), 62
- `close()` (*rtcbot.connection.RTCCConnection* method), 67
- `close()` (*rtcbot.inputs.Gamepad* method), 100
- `close()` (*rtcbot.inputs.InputDevice* method), 103
- `close()` (*rtcbot.inputs.Keyboard* method), 106
- `close()` (*rtcbot.inputs.Mouse* method), 110
- `close()` (*rtcbot.websocket.Websocket* method), 72
- `closed` (*rtcbot.arduino.SerialConnection* property), 117
- `closed` (*rtcbot.audio.Microphone* property), 92
- `closed` (*rtcbot.audio.Speaker* property), 95
- `closed` (*rtcbot.base.events.baseEventHandler* property), 129
- `closed` (*rtcbot.camera.CVCamera* property), 79
- `closed` (*rtcbot.camera.CVDisplay* property), 82
- `closed` (*rtcbot.camera.PiCamera* property), 85
- `closed` (*rtcbot.camera.PiCamera2* property), 88

closed (*rtcbot.connection.ConnectionAudioHandler* property), 52
 closed (*rtcbot.connection.ConnectionVideoHandler* property), 57
 closed (*rtcbot.connection.DataChannel* property), 62
 closed (*rtcbot.connection.RTCConnection* property), 67
 closed (*rtcbot.inputs.Gamepad* property), 100
 closed (*rtcbot.inputs.InputDevice* property), 103
 closed (*rtcbot.inputs.Keyboard* property), 107
 closed (*rtcbot.inputs.Mouse* property), 110
 closed (*rtcbot.websocket.Websocket* property), 72
 ConnectionAudioHandler (class in *rtcbot.connection*), 51
 ConnectionVideoHandler (class in *rtcbot.connection*), 57
 CVCamera (class in *rtcbot.camera*), 79
 CVDisplay (class in *rtcbot.camera*), 81

D

DataChannel (class in *rtcbot.connection*), 62
 defaultFilter() (in module *rtcbot.inputs*), 113
 DelayedSubscription (class in *rtcbot.subscriptions*), 126

E

error (*rtcbot.arduino.SerialConnection* property), 117
 error (*rtcbot.audio.Microphone* property), 92
 error (*rtcbot.audio.Speaker* property), 95
 error (*rtcbot.base.events.baseEventHandler* property), 129
 error (*rtcbot.camera.CVCamera* property), 79
 error (*rtcbot.camera.CVDisplay* property), 82
 error (*rtcbot.camera.PiCamera* property), 85
 error (*rtcbot.camera.PiCamera2* property), 88
 error (*rtcbot.connection.ConnectionAudioHandler* property), 52
 error (*rtcbot.connection.ConnectionVideoHandler* property), 57
 error (*rtcbot.connection.DataChannel* property), 62
 error (*rtcbot.connection.RTCConnection* property), 67
 error (*rtcbot.inputs.Gamepad* property), 100
 error (*rtcbot.inputs.InputDevice* property), 103
 error (*rtcbot.inputs.Keyboard* property), 107
 error (*rtcbot.inputs.Mouse* property), 110
 error (*rtcbot.websocket.Websocket* property), 72
 EventSubscription (class in *rtcbot.subscriptions*), 127

G

Gamepad (class in *rtcbot.inputs*), 100
 Gamepad() (class), 125
 Gamepad.close() (*Gamepad* method), 125
 Gamepad.subscribe() (*Gamepad* method), 126
 get() (*rtcbot.arduino.SerialConnection* method), 117
 get() (*rtcbot.audio.Microphone* method), 92

get() (*rtcbot.base.base.BaseSubscriptionProducer* method), 135
 get() (*rtcbot.camera.CVCamera* method), 79
 get() (*rtcbot.camera.PiCamera* method), 85
 get() (*rtcbot.camera.PiCamera2* method), 88
 get() (*rtcbot.connection.ConnectionAudioHandler* method), 52
 get() (*rtcbot.connection.ConnectionVideoHandler* method), 57
 get() (*rtcbot.connection.DataChannel* method), 62
 get() (*rtcbot.connection.RTCConnection* method), 67
 get() (*rtcbot.inputs.Gamepad* method), 100
 get() (*rtcbot.inputs.InputDevice* method), 104
 get() (*rtcbot.inputs.Keyboard* method), 107
 get() (*rtcbot.inputs.Mouse* method), 110
 get() (*rtcbot.subscriptions.DelayedSubscription* method), 127
 get() (*rtcbot.subscriptions.EventSubscription* method), 127
 get() (*rtcbot.subscriptions.GetterSubscription* method), 128
 get() (*rtcbot.subscriptions.MostRecentSubscription* method), 128
 get() (*rtcbot.subscriptions.RebatchSubscription* method), 128
 get() (*rtcbot.websocket.Websocket* method), 72
 getDataChannel() (*rtcbot.connection.RTCConnection* method), 67
 getLocalDescription() (*rtcbot.connection.RTCConnection* method), 68
 getRTCBotJS() (in module *rtcbot.javascript*), 123
 GetterSubscription (class in *rtcbot.subscriptions*), 127

I

InputDevice (class in *rtcbot.inputs*), 103
 internalSubscriptionMessage (class in *rtcbot.base.multiprocess*), 140

K

Keyboard (class in *rtcbot.inputs*), 106
 Keyboard() (class), 125
 Keyboard.close() (*Keyboard* method), 125
 Keyboard.subscribe() (*Keyboard* method), 125

M

Microphone (class in *rtcbot.audio*), 92
 module
 rtcbot.arduino, 117
 rtcbot.audio, 92
 rtcbot.base, 126
 rtcbot.base.base, 131
 rtcbot.base.events, 129

rtcbot.base.multiprocess, 138
 rtcbot.base.thread, 137
 rtcbot.camera, 79
 rtcbot.connection, 51
 rtcbot.inputs, 100
 rtcbot.javascript, 123
 rtcbot.subscriptions, 126
 rtcbot.websocket, 72
 MostRecentSubscription (class in *rtcbot.subscriptions*), 128
 Mouse (class in *rtcbot.inputs*), 110

N

name (*rtcbot.connection.DataChannel* property), 63
 NoClosedSubscription (class in *rtcbot.base.base*), 136

O

offerToReceive() (*rtcbot.connection.ConnectionAudioHandler* method), 52
 offerToReceive() (*rtcbot.connection.ConnectionVideoHandler* method), 58
 onClose() (*rtcbot.arduino.SerialConnection* method), 118
 onClose() (*rtcbot.audio.Microphone* method), 93
 onClose() (*rtcbot.audio.Speaker* method), 96
 onClose() (*rtcbot.base.events.baseEventHandler* method), 130
 onClose() (*rtcbot.camera.CVCamera* method), 80
 onClose() (*rtcbot.camera.CVDisplay* method), 82
 onClose() (*rtcbot.camera.PiCamera* method), 85
 onClose() (*rtcbot.camera.PiCamera2* method), 88
 onClose() (*rtcbot.connection.ConnectionAudioHandler* method), 53
 onClose() (*rtcbot.connection.ConnectionVideoHandler* method), 58
 onClose() (*rtcbot.connection.DataChannel* method), 63
 onClose() (*rtcbot.connection.RTCCConnection* method), 68
 onClose() (*rtcbot.inputs.Gamepad* method), 101
 onClose() (*rtcbot.inputs.InputDevice* method), 104
 onClose() (*rtcbot.inputs.Keyboard* method), 107
 onClose() (*rtcbot.inputs.Mouse* method), 111
 onClose() (*rtcbot.websocket.Websocket* method), 73
 onDataChannel() (*rtcbot.connection.RTCCConnection* method), 68
 onError() (*rtcbot.arduino.SerialConnection* method), 118
 onError() (*rtcbot.audio.Microphone* method), 93
 onError() (*rtcbot.audio.Speaker* method), 96
 onError() (*rtcbot.base.events.baseEventHandler* method), 130
 onError() (*rtcbot.camera.CVCamera* method), 80
 onError() (*rtcbot.camera.CVDisplay* method), 82
 onError() (*rtcbot.camera.PiCamera* method), 86
 onError() (*rtcbot.camera.PiCamera2* method), 89
 onError() (*rtcbot.connection.ConnectionAudioHandler* method), 53
 onError() (*rtcbot.connection.ConnectionVideoHandler* method), 58
 onError() (*rtcbot.connection.DataChannel* method), 63
 onError() (*rtcbot.connection.RTCCConnection* method), 68
 onError() (*rtcbot.inputs.Gamepad* method), 101
 onError() (*rtcbot.inputs.InputDevice* method), 104
 onError() (*rtcbot.inputs.Keyboard* method), 107
 onError() (*rtcbot.inputs.Mouse* method), 111
 onError() (*rtcbot.websocket.Websocket* method), 73
 onReady() (*rtcbot.arduino.SerialConnection* method), 118
 onReady() (*rtcbot.audio.Microphone* method), 93
 onReady() (*rtcbot.audio.Speaker* method), 96
 onReady() (*rtcbot.base.events.baseEventHandler* method), 130
 onReady() (*rtcbot.camera.CVCamera* method), 80
 onReady() (*rtcbot.camera.CVDisplay* method), 82
 onReady() (*rtcbot.camera.PiCamera* method), 86
 onReady() (*rtcbot.camera.PiCamera2* method), 89
 onReady() (*rtcbot.connection.ConnectionAudioHandler* method), 53
 onReady() (*rtcbot.connection.ConnectionVideoHandler* method), 58
 onReady() (*rtcbot.connection.DataChannel* method), 63
 onReady() (*rtcbot.connection.RTCCConnection* method), 68
 onReady() (*rtcbot.inputs.Gamepad* method), 101
 onReady() (*rtcbot.inputs.InputDevice* method), 105
 onReady() (*rtcbot.inputs.Keyboard* method), 108
 onReady() (*rtcbot.inputs.Mouse* method), 111
 onReady() (*rtcbot.websocket.Websocket* method), 73
 onTrack() (*rtcbot.connection.ConnectionAudioHandler* method), 53
 onTrack() (*rtcbot.connection.ConnectionVideoHandler* method), 59

P

PiCamera (class in *rtcbot.camera*), 84
 PiCamera2 (class in *rtcbot.camera*), 87
 ProcessSubscriptionConsumer (class in *rtcbot.base.multiprocess*), 138
 ProcessSubscriptionProducer (class in *rtcbot.base.multiprocess*), 139
 ProcessSubscriptionProducerConsumer (class in *rtcbot.base.multiprocess*), 139
 put_nowait() (*rtcbot.arduino.SerialConnection* method), 119
 put_nowait() (*rtcbot.audio.Speaker* method), 97
 put_nowait() (*rtcbot.base.base.BaseSubscriptionConsumer* method), 132

`put_nowait()` (*rtcbot.camera.CVDisplay* method), 83
`put_nowait()` (*rtcbot.connection.ConnectionAudioHandler* method), 54
`put_nowait()` (*rtcbot.connection.ConnectionVideoHandler* method), 59
`put_nowait()` (*rtcbot.connection.DataChannel* method), 64
`put_nowait()` (*rtcbot.connection.RTCConnection* method), 69
`put_nowait()` (*rtcbot.subscriptions.CallbackSubscription* method), 126
`put_nowait()` (*rtcbot.subscriptions.EventSubscription* method), 127
`put_nowait()` (*rtcbot.subscriptions.MostRecentSubscription* method), 128
`put_nowait()` (*rtcbot.subscriptions.RebatchSubscription* method), 128
`put_nowait()` (*rtcbot.websocket.Websocket* method), 74
`putSubscription()` (*rtcbot.arduino.SerialConnection* method), 119
`putSubscription()` (*rtcbot.audio.Speaker* method), 96
`putSubscription()` (*rtcbot.base.base.BaseSubscriptionConsumer* method), 132
`putSubscription()` (*rtcbot.base.multiprocess.ProcessSubscriptionHandler* method), 138
`putSubscription()` (*rtcbot.base.multiprocess.ProcessSubscriptionPreloaderConsumer* method), 139
`putSubscription()` (*rtcbot.base.thread.ThreadedSubscriptionHandler* method), 137
`putSubscription()` (*rtcbot.camera.CVDisplay* method), 83
`putSubscription()` (*rtcbot.connection.ConnectionAudioHandler* method), 54
`putSubscription()` (*rtcbot.connection.ConnectionVideoHandler* method), 59
`putSubscription()` (*rtcbot.connection.DataChannel* method), 64
`putSubscription()` (*rtcbot.connection.RTCConnection* method), 69
`putSubscription()` (*rtcbot.websocket.Websocket* method), 74

Q

`Queue()` (class), 126

R

`ready` (*rtcbot.arduino.SerialConnection* property), 119
`ready` (*rtcbot.audio.Microphone* property), 93
`ready` (*rtcbot.audio.Speaker* property), 97
`ready` (*rtcbot.base.events.baseEventHandler* property), 131
`ready` (*rtcbot.camera.CVCamera* property), 80
`ready` (*rtcbot.camera.CVDisplay* property), 84
`ready` (*rtcbot.camera.PiCamera* property), 86
`ready` (*rtcbot.camera.PiCamera2* property), 89
`ready` (*rtcbot.connection.ConnectionAudioHandler* property), 55
`ready` (*rtcbot.connection.ConnectionVideoHandler* property), 60
`ready` (*rtcbot.connection.DataChannel* property), 64
`ready` (*rtcbot.connection.RTCConnection* property), 69
`ready` (*rtcbot.inputs.Gamepad* property), 102
`ready` (*rtcbot.inputs.InputDevice* property), 105
`ready` (*rtcbot.inputs.Keyboard* property), 108
`ready` (*rtcbot.inputs.Mouse* property), 111
`ready` (*rtcbot.websocket.Websocket* property), 74
`RebatchSubscription` (class in *rtcbot.subscriptions*), 128
`rtcbot.arduino` module, 117
`rtcbot.audio` module, 92
`rtcbot.base` module, 126
`rtcbot.base.base` module, 131
`rtcbot.base.events` module, 129
`rtcbot.base.multiprocess` module, 138
`rtcbot.base.thread` module, 137
`rtcbot.camera` module, 79
`rtcbot.connection` module, 51
`rtcbot.inputs` module, 100
`rtcbot.javascript` module, 123
`rtcbot.subscriptions` module, 126
`rtcbot.websocket` module, 72
`RTCConnection` (class in *rtcbot.connection*), 67
`RTCConnection()` (class), 124
`RTCConnection.audio` (*RTCConnection* attribute), 124
`RTCConnection.close()` (*RTCConnection* method), 124
`RTCConnection.getLocalDescription()` (*RTCConnection* method), 124
`RTCConnection.put_nowait()` (*RTCConnection* method), 125
`RTCConnection.setRemoteDescription()` (*RTCConnection* method), 125
`RTCConnection.subscribe()` (*RTCConnection* method), 125
`RTCConnection.video` (*RTCConnection* attribute), 124

S

[send\(\)](#) (*rtcbot.connection.RTCConnection* method), 70
[SerialConnection](#) (class in *rtcbot.arduino*), 117
[setGamepadRate\(\)](#) (built-in function), 126
[setRemoteDescription\(\)](#) (*rtcbot.connection.RTCConnection* method), 70
[Speaker](#) (class in *rtcbot.audio*), 95
[stopSubscription\(\)](#) (*rtcbot.arduino.SerialConnection* method), 120
[stopSubscription\(\)](#) (*rtcbot.audio.Speaker* method), 97
[stopSubscription\(\)](#) (*rtcbot.base.base.BaseSubscriptionConsumer* method), 132
[stopSubscription\(\)](#) (*rtcbot.camera.CVDisplay* method), 84
[stopSubscription\(\)](#) (*rtcbot.connection.ConnectionAudioHandler* method), 55
[stopSubscription\(\)](#) (*rtcbot.connection.ConnectionVideoHandler* method), 60
[stopSubscription\(\)](#) (*rtcbot.connection.DataChannel* method), 65
[stopSubscription\(\)](#) (*rtcbot.connection.RTCConnection* method), 70
[stopSubscription\(\)](#) (*rtcbot.websocket.Websocket* method), 75
[subscribe\(\)](#) (*rtcbot.arduino.SerialConnection* method), 120
[subscribe\(\)](#) (*rtcbot.audio.Microphone* method), 94
[subscribe\(\)](#) (*rtcbot.base.base.BaseSubscriptionProducer* method), 135
[subscribe\(\)](#) (*rtcbot.camera.CVCamera* method), 81
[subscribe\(\)](#) (*rtcbot.camera.PiCamera* method), 87
[subscribe\(\)](#) (*rtcbot.camera.PiCamera2* method), 89
[subscribe\(\)](#) (*rtcbot.connection.ConnectionAudioHandler* method), 55
[subscribe\(\)](#) (*rtcbot.connection.ConnectionVideoHandler* method), 60
[subscribe\(\)](#) (*rtcbot.connection.DataChannel* method), 65
[subscribe\(\)](#) (*rtcbot.connection.RTCConnection* method), 70
[subscribe\(\)](#) (*rtcbot.inputs.Gamepad* method), 102
[subscribe\(\)](#) (*rtcbot.inputs.InputDevice* method), 105
[subscribe\(\)](#) (*rtcbot.inputs.Keyboard* method), 108
[subscribe\(\)](#) (*rtcbot.inputs.Mouse* method), 112
[subscribe\(\)](#) (*rtcbot.websocket.Websocket* method), 75
[subscription](#) (*rtcbot.arduino.SerialConnection* property), 121
[subscription](#) (*rtcbot.audio.Speaker* property), 98
[subscription](#) (*rtcbot.base.base.BaseSubscriptionConsumer* property), 133
[subscription](#) (*rtcbot.camera.CVDisplay* property), 84

[subscription](#) (*rtcbot.connection.ConnectionAudioHandler* property), 56
[subscription](#) (*rtcbot.connection.ConnectionVideoHandler* property), 61
[subscription](#) (*rtcbot.connection.DataChannel* property), 66
[subscription](#) (*rtcbot.connection.RTCConnection* property), 71
[subscription](#) (*rtcbot.websocket.Websocket* property), 76
[SubscriptionClosed](#), 136
[SubscriptionConsumer](#) (class in *rtcbot.base.base*), 137
[SubscriptionProducer](#) (class in *rtcbot.base.base*), 137
[SubscriptionProducerConsumer](#) (class in *rtcbot.base.base*), 137

T

[threadedEventHandler](#) (class in *rtcbot.base.events*), 131
[ThreadedSubscriptionConsumer](#) (class in *rtcbot.base.thread*), 137
[ThreadedSubscriptionProducer](#) (class in *rtcbot.base.thread*), 138

U

[unsubscribe\(\)](#) (*rtcbot.arduino.SerialConnection* method), 121
[unsubscribe\(\)](#) (*rtcbot.audio.Microphone* method), 95
[unsubscribe\(\)](#) (*rtcbot.base.base.BaseSubscriptionProducer* method), 136
[unsubscribe\(\)](#) (*rtcbot.camera.CVCamera* method), 81
[unsubscribe\(\)](#) (*rtcbot.camera.PiCamera* method), 87
[unsubscribe\(\)](#) (*rtcbot.camera.PiCamera2* method), 90
[unsubscribe\(\)](#) (*rtcbot.connection.ConnectionAudioHandler* method), 56
[unsubscribe\(\)](#) (*rtcbot.connection.ConnectionVideoHandler* method), 62
[unsubscribe\(\)](#) (*rtcbot.connection.DataChannel* method), 66
[unsubscribe\(\)](#) (*rtcbot.connection.RTCConnection* method), 71
[unsubscribe\(\)](#) (*rtcbot.inputs.Gamepad* method), 103
[unsubscribe\(\)](#) (*rtcbot.inputs.InputDevice* method), 106
[unsubscribe\(\)](#) (*rtcbot.inputs.Keyboard* method), 109
[unsubscribe\(\)](#) (*rtcbot.inputs.Mouse* method), 113
[unsubscribe\(\)](#) (*rtcbot.subscriptions.DelayedSubscription* method), 127
[unsubscribe\(\)](#) (*rtcbot.websocket.Websocket* method), 76
[unsubscribeAll\(\)](#) (*rtcbot.arduino.SerialConnection* method), 122
[unsubscribeAll\(\)](#) (*rtcbot.audio.Microphone* method), 95

`unsubscribeAll()` (*rtcbot.base.base.BaseSubscriptionProducer method*), [136](#)
`unsubscribeAll()` (*rtcbot.camera.CVCamera method*), [81](#)
`unsubscribeAll()` (*rtcbot.camera.PiCamera method*), [87](#)
`unsubscribeAll()` (*rtcbot.camera.PiCamera2 method*), [90](#)
`unsubscribeAll()` (*rtcbot.connection.ConnectionAudioHandler method*), [57](#)
`unsubscribeAll()` (*rtcbot.connection.ConnectionVideoHandler method*), [62](#)
`unsubscribeAll()` (*rtcbot.connection.DataChannel method*), [67](#)
`unsubscribeAll()` (*rtcbot.connection.RTCCConnection method*), [72](#)
`unsubscribeAll()` (*rtcbot.inputs.Gamepad method*), [103](#)
`unsubscribeAll()` (*rtcbot.inputs.InputDevice method*), [106](#)
`unsubscribeAll()` (*rtcbot.inputs.Keyboard method*), [110](#)
`unsubscribeAll()` (*rtcbot.inputs.Mouse method*), [113](#)
`unsubscribeAll()` (*rtcbot.websocket.Websocket method*), [77](#)

V

`video` (*rtcbot.connection.RTCCConnection property*), [72](#)

W

`Websocket` (*class in rtcbot.websocket*), [72](#)