rovercode Documentation

Release 0.4.0

Brady L. Hurlburt and other rovercode.org contributers

Jun 02, 2017

Contents:

License GPLv3

Source https://github.com/aninternetof/rovercode

Hosted at https://rovercode.com (master) and https://beta.rovercode.com (development)

CHAPTER 1

Welcome!

rovercode is an easy-to-use system for controlling robots (rovers) that can sense and react to their environment. The Blockly editor makes it easy to program and run your bot straight from your browser. Just drag and drop your commands to drive motors, read values from a variety of supported sensors, and see what your rover sees with the built in webcam viewer.

CHAPTER 2

Architecture

rovercode is made up of two parts:

- rovercode (the docs you're reading right now) is the service that runs on the rover.
- rovercode-web (a separate repo documented here) is the web app running at rovercode.com.

rovercode runs on the rover. The rover can be any single-board-computer supported by the Adafruit Python GPIO wrapper library, including the NextThingCo CHIP, Raspberry Pi, and BeagleBone Black.

rovercode-web is hosted on the Internet at rovercode.com. It has a Blockly-based editor (which we call Mission Control) for creating a routine. The routine executes in the browser (sandboxed, of course), and commands are sent to the rover for rovercode to execute (e.g. "stop motor, turn on light"). Events on the rover ("right eye detects something") are sent to the browser via a WebSocket connection.

The rover and the device running the browser must be on the same local network.

CHAPTER 3

Get Started

Check out the quickstart guide. Then see how to contribute.

CHAPTER 4

Contact

Please join the rovercode developer mailing list! Go here, then click "register".

Also, we'd love to chat with you! Join the the rovercode Slack channel.

You can also email brady@rovercode.com.

CHAPTER 5

Contents

quickstart

Standard Setup (on rover)

First, on your rover (CHIP, Raspberry Pi, BeagleBone, etc):

```
$ sudo apt install git
$ git clone --recursive https://github.com/aninternetof/rovercode.git && cd rovercode
$ sudo bash setup.sh #run this only once -- it will take some time
$ sudo bash start.sh #run this each time you boot the rover (or automatically start␣
↪if chosen in setup)
```

Then, on any PC or tablet, head to rovercode.com to connect to your rover. Start playing!

Development Setup (on development PC)

When developing rovercode, you may want to run rovercode on your PC instead of a CHIP/Raspberry Pi/Beaglebone. Below are instructions for how to install and run rovercode on your PC. Everything should work fine: rovercode will automatically detect that it is not running on target hardware and will stub out the calls to the motors and sensors.

First, on your development PC:

```
$ sudo apt install git
$ git clone --recursive https://github.com/aninternetof/rovercode.git && cd rovercode
$ sudo bash setup.sh #run this only once -- it will take some time
$ sudo bash start.sh #run this each time
```

Then, still on your development PC, head to rovercode.com and connect to your "rover" (your PC running the service).

Alternate Development Setup (on development PC using Docker)

Rather use Docker? First, on your development PC:

```
$ sudo apt install git docker.io
$ git clone --recursive https://github.com/aninternetof/rovercode.git && cd rovercode
$ sudo docker build -t rovercode .
$ sudo docker run --name rovercode -v $PWD:/var/www/rovercode -p 80:80 -d rovercode
```

Then, still on your development PC, head to rovercode.com and connect to your "rover" (your PC running the service).

detailed usage

using rovercode with a rovercode-web hosted somewhere other than rovercode.com

By default, when rovercode runs, it registers itself with *https://rovercode.com*. But what if you want to try your changes to rovercode with *https://beta.rovercode.com*? Or with your local instance of rovercode-web (as described in the next section)? You can specify the target rovercode-web url by creating a .env file in your rovercode directory.

```
# first, navigate to the rovercode root diretory (same level as the Dockerfile), then
$ echo ROVERCODE_WEB_URL=https://beta.rovercode.com/ > .env
```

When you start rovercode, it will register itself with *beta.rovercode.com*.

develop rovercode and rovercode-web on the same machine at the same time

Get, build, and bring up rovercode-web as usual:

```
$ git clone --recursive https://github.com/aninternetof/rovercode-web.git && cd␣
↪rovercode-web
$ sudo docker-compose -f dev.yml build
$ sudo docker-compose -f dev.yml up
$ google-chrome localhost:8000
```

Get and build rovercode as usual:

```
$ git clone --recursive https://github.com/aninternetof/rovercode.git && cd rovercode
$ sudo docker build -t rovercode .
```

Set the url of the rovercode-web target to *http://rovercodeweb:8000*. You will see in the next step that this is the hostname that we assign to our local rovercode-web container.

```
# first, navigate to the rovercode root diretory (same level as the Dockerfile), then
$ echo ROVERCODE_WEB_URL=http://rovercodeweb:8000/ > .env
```

Finally, when you bring up the rovercode container, add a *link* flag to allow access between this container and your rovercode-web container.

```
$ sudo docker run -t --link rovercodeweb_django_1:rovercodeweb --net rovercodeweb_
↪default --name rovercode -v $PWD:/var/www/rovercode -p 80:80 -d rovercode
```

docker-compose named it *rovercodeweb_django_1*, but notice that we used a colon to rename it simply *rovercodeweb*. This is necessary, because this becomes the hostname, and Django does not like underscores in hostname headers.

We also had to add a *net rovercodeweb_default* flag, because docker-compose put rovercode-web on its own network instead of on the default one. (If you're curious, you can find its name using the command *sudo docker network ls*.)

rovercode is now running, and you can see that it has registered itself with your local rovercodeweb container by going to http://localhost:8000/mission-control/rovers. You can now select this rover in the mission-control interface, and rover commands will be sent to your rovercode container.

Attribution DEIS blog post

contribute

There is lots of fun work to be done!

Head on over to the rovercode github. We use ZenHub to improve GitHub's agile management, so install it, then visit the *boards* tab to find a fun card in the backlog. Or submit a new card for a bug or cool new feature idea.

And remember, you can do all these same things for rovercode-web.

Chat with us on the rovercode Slack channel.

Follow the the code of conduct.

API

`app`	Rovercode app.

app

Rovercode app.

Functions

`MotorManager`(*args, **kwargs)	
`connect`()	Connect to the rovercode-web websocket.
`create_app`()	Creator of rovercode flask app.
`download_block_diagram`(id)	API: /download/<id> [GET].
`emit`(event, *args, **kwargs)	Emit a SocketIO event.
`find_dotenv`([filename, ...])	Search in increasingly higher folders for the given file
`get_block_diagram`(id)	API: /blockdiagrams/<id> [GET].
`get_block_diagrams`()	API: /blockdiagrams [GET].
`get_local_ip`()	Get the local area network IP address of the rover.
`init_rover_service`()	Initialize hardware pins and motor speeds.
`isfile`(path)	Test whether a path is a regular file
`join`(a, *p)	Join two or more pathname components, inserting '/' as needed.
`jsonify`(*args, **kwargs)	This function wraps `dumps()` to add a few enhancements that make life easier.
`listdir`((path) -> list_of_strings)	Return a list containing the names of the entries in the directory.
`load_dotenv`(dotenv_path)	Read a .env file and load into os.environ.
`run_command`(decoded)	Run the command specified by *decoded*.
`save_block_diagram`()	API: /blockdiagrams [POST].
`send_command`()	API: /sendcommand [POST].
`send_from_directory`(directory, filename, ...)	Send a file from a given directory with `send_file()`.
`sensors_thread`()	Scan each binary sensor and sends events based on changes.
`singleton`(class_)	Helper class for creating a singleton.

Continued on next page

Table 5.2 – continued from previous page

test_message(message)	Send a debug test message when status is received from rovercode-web.
upload_block_diagram()	API: /upload [POST].

Classes

BinarySensor(name, pin, rising_event, ...)	The binary sensor object contains information for each binary sensor.
CORS([app])	Initializes Cross Origin Resource sharing for the application.
Flask(import_name[, static_path, ...])	The flask object implements a WSGI application and acts as the central object.
HeartBeatManager(payload[, id])	A manager to register the rover with rovercode-web and periodically check in.
Response([response, status, headers, ...])	The response object that is used by default in Flask.
SocketIO([app])	Create a Flask-SocketIO server.

class app.**BinarySensor**(*name*, *pin*, *rising_event*, *falling_event*)

The binary sensor object contains information for each binary sensor.

Parameters

- **name** – The human readable name of the sensor
- **pin** – The hardware pin connected to the sensor
- **rising_event** – The event name associated with a signal changing from low to high
- **falling_event** – The event name associated with a signal changing from high to low

Constructor for BinarySensor object.

class app.**HeartBeatManager**(*payload*, *id=None*)

A manager to register the rover with rovercode-web and periodically check in.

Parameters

- **run** – A flag for the state of the thread. Set to false to gracefully stop the thread.
- **thread** – The Thread object that performs the periodic check-in.
- **web_id** – The rovercode-web id of this rover.
- **payload** – The json-formatted information about the rover to send to rovercode-web.

Constructor for the HeartBeatManager.

register()

Regiser the rover with rovercode-web.

stopThread()

Gracefully stop the periodic check-in thread.

thread_func(*run_once=False*)

Thread function that periodically checks in with rovercode-web.

app.**connect**()

Connect to the rovercode-web websocket.

app.**create_app**()
Creator of rovercode flask app.

app.**download_block_diagram**(*id*)
API: /download/<id> [GET].

Starts a download of the block diagram specified by *id*

app.**get_block_diagram**(*id*)
API: /blockdiagrams/<id> [GET].

Replies with an XML formatted description of the block diagram specified by *id*

app.**get_block_diagrams**()
API: /blockdiagrams [GET].

Replies with a JSON formatted list of the block diagrams

app.**get_local_ip**()
Get the local area network IP address of the rover.

app.**init_rover_service**()
Initialize hardware pins and motor speeds.

app.**run_command**(*decoded*)
Run the command specified by *decoded*.

Parameters **decoded** – The command to run

app.**save_block_diagram**()
API: /blockdiagrams [POST].

Saves the posted block diagram

app.**send_command**()
API: /sendcommand [POST].

Executes the posted command

Available Commands::: START_MOTOR STOP_MOTOR

app.**sensors_thread**()
Scan each binary sensor and sends events based on changes.

app.**singleton**(*class_*)
Helper class for creating a singleton.

app.**test_message**(*message*)
Send a debug test message when status is received from rovercode-web.

app.**upload_block_diagram**()
API: /upload [POST].

Adds the posted block diagram

build a rover

future plans

We are developing a rover reference design that includes a custom daughter board, a Lexan chassis, and MEMS IR sensors. Join the #hardware channel of our Slack to follow our progress and suggest ideas.

In the meantime, you can still build a rover like the one we built as our first prototype. In that approach, you borrow the chassis, motors, and motor driver circuitry from an easily-hackable RC car! Instructions for doing so are below.

supply list

- Thunder Tumbler RC car (or at CVS)
- IR emitter receiver pairs
- jumpers
- assorted resistors
- two small proto boards
- 0.1-inch headers, male, vertical (we'll cut to desired length)
- C.H.I.P
- webcam
- powered USB hub
- USB battery
- soldering iron and solder

chassis, motors, wheels – the Thunder Tumbler

We'll use the chassis, motors, and wheels from the venerable Thunder Tumbler RC car. I get mine at Walgreens or CVS; sometimes they are called something else there, but whatever RC car they sell is likely to be pretty much a Thunder Tumbler. You can also order one from Amazon.

We've chosen this RC car because it's easy to hack. Specifically, it's easy to rip out the radio controller IC. This is the IC that receives messages from the wireless controller and drives the motors. We don't care about the wireless controller; instead, we'd like the C.H.I.P to drive the motors. So, we'll remove the radio controller IC, leaving empty its pads that connected it to the motors. Then we'll connect some GPIO from our C.H.I.P to those pads.

Preparing the Thunder Tumbler

Here are the instructions for preparing the Thunder Tumbler. Your end goal is to bring out these connections to a 5-pin header:

- Left motor forward
- Left motor backward
- Right motor forward
- Right motor backward
- Ground

Hot-glue this header somewhere convenient on the chassis. Later we'll run jumpers to it from the C.H.I.P.

Here are some tweaks/tips to augment the tutorial:

- Everywhere he says "Arduino", replace it with "C.H.I.P."
- Depending on what version of the Thunder Tumbler you happen to get, the radio controller IC could be through-hole or surface-mount. If it's surface mount, try your best not to rip off the pads when you remove the IC.

- To figure out which pads are the variable left/right forward/backward motor pads, I recommend connecting a wire to the 3.3V supply on your C.H.I.P, then poking the other end around on all the pads. Observe which wheel turns and in which direction, and write it down!
- You can get ground from anywhere you want; you don't need to use a pad from the radio control IC. Use a multimeter to find a spot that reads zero resistance with the negative terminal of the RC car's battery holder.

Connecting to the C.H.I.P

Use your jumpers to connect the signals on your new 5-pin header to the C.H.I.P. Ground connects to ground, and the motor control signals can connect to any *XIO-P[0-7]* pin. Right now the pins are hard-coded in blockly-api.js (booooo, I know), so to avoid having to edit the code, use these pins:

Motor Signal	C.H.I.P Pin
left, forward	XIO-P0
left, backward	XIO-P1
right, forward	XIO-P6
right, backward	XIO-P7
ground	any ground

infrared sensors – the ears

We call the infrared sensors the ears of the rover. They might be better called eyes since they operate using light (albeit invisible light). But, the rover already has an eye (the webcam), and the IR sensor boards stick off the the sides like ears, so we go with it.

Building the circuit

The rover has two ears: two IR sensor boards. They are identical. Each has an IR transmitter and an IR receiver. This is the circuit; create it on two of your proto boards:

Each ear has a header with three things on it:

- 3.3V
- ground
- signal (this is what varies to indicate something is detected)

We just want a binary signal out of the sensors, so even though we have a continuous analog signal coming out of the sensor, we won't hook it up to an analog input of the C.H.I.P. We'll just hook it up to a regular GPIO input, and let the input hardware of the pin serve as a rough comparator.

Just like the motor signal pins, the pins for the IR ear signals are hardcoded at the moment (this time in app.py – we are really gonna make this configurable soon). So to avoid having to change code, connect this like this:

Connecting to the C.H.I.P

IR Ear Signal	C.H.I.P Pin
left	XIO-P2
right	XIO-P4

Note: These sensor circuits are not great. Their detection range is only of a couple of inches. Our future reference design will include a PCB with a Silicon Labs I2C MEMS IR sensor, which should work much better.

webcam – the eye

important note The default CHIP kernel does not enable the USB Video Class (UVC) driver. In the future we hope to provide a ready-to-use eMMC image with this driver included, but for now you'll have to rebuild the kernel with the UVC driver included. This is a more advanced task. Your best bet is this tutorial. If you're not up for this, don't worry – just stay tuned for an update to this page telling you where you can get a ready-to-use eMMC image.

At the moment, the webcam streaming service is not installed or started with the main rovercode service (we have an issue card to fix this). So, you'll need to get and run mjpg-streamer yourself for now.

Get and build mjpg-streamer by following steps 1 through 5 in these instructions.

To make mjpg-start on boot, add this line to */etc/rc.local*. Replace {BUILD_DIR} with the absolute path to the directory where you built mjpg-streamer.

```
{BUILD_DIR}/mjpg_streamer -i "{BUILD_DIR}/input_uvc.so" -o "{BUILD_DIR}/output_http.
↪so -w {BUILD_DIR}/www"
```

Restart the rover. You can check that mjpg-streamer has started by pointing your PC's browser at *{ip-address-of-your-rover}:8080*. You should see the mjpg-streamer demo page.

assembly

Here is how it all hooks together:

Put everything on the chassis how you see fit. Below are some photos of how we did it. Hot glue is your friend.

Note that the motors are still powered by the Thunder Tumbler AA battery pack, so make sure there are batteries in there and that the switch on the bottom is turned on when in use.

The webcam draws too much current to be directly connected to the C.H.I.P's USB host port. So, we use a powered USB hub.

install rovercode service

Connect to the C.H.I.P. via serial or SSH.

Follow the Standard Setup on the quickstart page.

play

Go to https://rovercode.com, sign up for an account, then go to Mission Control. Click *Connect to a Rover*. Choose your rover, whose name is hardcoded here, sadly. You should see a message in the console bar on the right saying that it has connected to a a rover and listing its local IP address.

Drag in some commands, hit play, and have fun!

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index