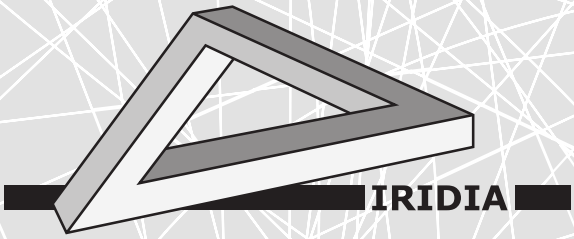

Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Software support for the Pi-Puck mobile robot

M. Allwright

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2022-001

March 2022
Last revision: October 2022

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:

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

Technical report number TR/IRIDIA/2022-001

Revision history:

TR/IRIDIA/2022-001.001 March 2022
TR/IRIDIA/2022-001.002 October 2022

Table of contents

Introduction **3**

ARGoS Plug-in for the Pi-Puck mobile robot **4**
Usage . . . 4
Controllers . . . 4
Physics Engines . . . 5
Media . . . 5
Sensors . . . 5
`pipuck_differential_drive` . . . 6
`pipuck_front_camera` . . . 6
`pipuck_ground` . . . 8
`pipuck_rangefinders` . . . 9
`pipuck_system` . . . 9
Actuators . . . 10
`pipuck_differential_drive` . . . 10
`pipuck_leds` . . . 10

The omni-directional camera plugin for the Pi-Puck **12**
Quick start . . . 12
Compilation . . . 12
Current state of the plugin . . . 12
Hardware . . . 13
Simulation . . . 13
Current limitations . . . 13
Focus control is not implemented . . . 13
Capture is limited to a single camera . . . 14
Tag detection is very slow . . . 14

The meta-pipuck layer for the Yocto build system **15**
Description . . . 15
Quick start . . . 15
Create the Docker image . . . 16
Create the Docker container . . . 16
Start the build . . . 16
Copying the image . . . 17
Booting the image and accessing the console . . . 17
Wifi configuration . . . 17

Acknowledgements **18**

Introduction

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

1. ARGoS Plugin for the Pi-Puck mobile robot
2. The omni-directional camera plugin for the Pi-Puck
3. The meta-pipuck layer for the Yocto build system

The following firmware is relevant to the software support documented in this technical report:

1. E-Puck1 firmware
2. FT903 firmware

ARGoS Plug-in for the Pi-Puck mobile robot

The Pi-Puck mobile robot is supported in the ARGoS simulator. The plugin in the ARGoS repository defines the control interface and provides an implementation of the Pi-Puck in simulation. This implementation provides a visualization for the `qt-opengl` plugin and models for the `dynamics2d` and `dynamics3d` physics engine plugins.

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

Usage

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

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

Controllers

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

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

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

Physics Engines

To create a Pi-Puck in ARGoS, it must be hosted by either the `dynamics2d` or `dynamics3d` physics engine plugin. The `dynamics2d` model is faster since it essentially models a Pi-Puck as a circle in the XY plane. The `dynamics3d` model is somewhat slower but more realistic. This model actually simulates the wheels of the Pi-Puck as cylindrical bodies that drive the Pi-Puck forwards as a result of the friction between the wheels and the floor of the arena. You can use either physics engine, however, entities in one physics engine will not collide with entities from another. The only exception to this rule are static objects such as the walls of the arena which can be hosted by multiple engines at the same time.

Media

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

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

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

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

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

Sensors

The following section lists the sensors that are specific to the Pi-Puck. There are also a few generic sensors (e.g., `simple_radios` for wifi) that can be added. These sensors are in part discoverable by running `argos3 -q sensors` in a terminal. The examples below make use of the Lua controller in ARGoS which allows writing control software in the form of a Lua script. The advantage of writing controllers in Lua is that there is no need for (cross-)compilation or

linking. That being said, all sensors and actuators do provide a C++ interface for which the code itself is the documentation.

pipuck_differential_drive

This sensor is designed to read back the current velocity from the wheels. It's accuracy is inherently tied to the implementation details of the physic model. The sensor can be added to the `sensors` section of a controller as follows:

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

This configuration shows the sensor being added to a Lua controller. With this configuration, the current forwards wheel velocities in centimeters per second can be read from a controller as follows:

```
function step()
  log('left speed = ' .. robot.differential_drive.encoders.left)
  log('right speed = ' .. robot.differential_drive.encoders.right)
end
```

This code will write the speeds of the wheels to the ARGoS logger.

pipuck_front_camera

This sensor will detect tags and LEDs in front of the Pi-Puck's front camera. The following configuration adds this sensor to the Pi-Puck.

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <pipuck_front_camera implementation="default"
      rotation="0"
      resolution="640,480"
      principal_point="320,240"
      focal_length="1000,1000"
      tag_medium="tags"
      directional_led_medium="leds"
      show_tag_rays="false"
      show_led_rays="false"
      show_frustum="false" />
  </sensors>
```

```
  <params />
</lua_controller>
```

There are many attributes that can be used to configure the front camera sensor although most of them have defaults. The only two attributes which have to be supplied for the sensor to work are the `tag_medium` and the `directional_led_medium`. These attributes specify the indices in which the camera should try and find entities. The remaining attributes are optional and default to the values specified above (note that the principal point defaults to half the resolution).

Attribute	Description
`rotation`	Rotation of the Pi-Puck's front camera. Zero degrees of rotation has the camera rotated by -90 degrees as per the PO3030 variant
`resolution`	Resolution that the camera is running at. This only influences the pixel coordinates of the detected tags
`principal_point`	An intrinsic parameter of the camera sensor
`focal_length`	An intrinsic parameter of the camera sensor
`tag_medium`	The index in which tags can be found
`directional_led_medium`	The index in which LEDs can be found
`show_tag_rays`	Draws the rays from the camera to the corners of a detected tag
`show_led_rays`	Draws the rays from the camera to an LED
`show_frustum`	Draw the bounding frustum in which tags and LEDs can be detected

By default the camera is disabled and needs to be enabled as per the following Lua code:

```
function init()
  robot.front_camera.enable()
end
```

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

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

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

To detect an LED, you need to specify the position where you expect the LED to be. This design choice reflects how the camera sensor for the real robot works. The function `detect_led` will return an integer in the range from 1..4 which represents which quadrant of the UV color space in which the color of LED matches. In simulation, these colors can be one of four fixed values `magenta`, `orange`, `green`, or `blue` as rough approximations for the four quadrants of the UV color space. Other colors will not be detected in simulation. The following code demonstrates detecting an LED that is 2 centimeters to the left of a detected tag (ignoring rotation).

```
function step()
  tag = robot.front_camera.tags[1]
  if tag ~= nil then
    led = robot.front_camera.detect_led(tag.position - vector3(0, 0.02, 0))
    log('led color is in quadrant ' .. led)
  end
end
```

pipuck_ground

This is a sensor that is capable of reading the brightness of the floor at three locations near the front of the robot.

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

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

```
function step()
  for i, sensor in ipairs(robot.ground_sensors)
    log('ground sensor ' .. i .. ':')
    log('  reflected = ' .. sensor.reflected)
    log('  background = ' .. sensor.background)
    log('  transform:')
    log('    anchor = ' .. sensor.transform.anchor)
    log('    position = ' .. sensor.transform.position)
    log('    orientation = ' .. sensor.transform.orientation)
  end
end
```

pipuck_rangefinders

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

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

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

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

pipuck_system

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

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

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

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

```
end
```

Actuators

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

`pipuck_differential_drive`

The differential drive actuator controls the differential drive of the Pi-Puck and can be added to a controller as follows:

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

The actuator exports a single function in Lua called `set_linear_velocity` which takes two arguments: the target forwards velocity of the left and right wheel in centimeters per second.

```
function step()
  -- drive forwards at 5 centimeters per second
  robot.differential_drive.set_linear_velocity(5, 5)
end
```

`pipuck_leds`

The LED actuator turns the LEDs of Pi-Puck off and on. It can be added to a controller as follows:

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

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

Function	Arguments
`set_ring_led_index`	The index (integer between 1..8) and mode (on/off) (boolean)
`set_ring_leds`	The mode (on/off) (boolean)
`set_front_led`	The mode (on/off) (boolean)
`set_body_led`	The mode (on/off) (boolean)

For example, the following code turns off all LEDs on the Pi-Puck ring except for the LED with index 5.

```
function step()
  robot.leds.set_ring_leds(false)
  robot.leds.set_ring_led_index(5, true)
end
```

The omni-directional camera plugin for the Pi-Puck

The Pi-Puck omni-directional camera repository contains external plugins for use with the Pi-Puck in ARGoS3. These plugins provides an omnidirectional camera sensor capable of locating Apriltags and detecting nearby LEDs.

Quick start

Compilation

If you are working with the hardware, the meta-pipuck layer will automatically install this plugin for you. If you are working in simulation, you will need to follow the steps to make the camera available to ARGoS. These steps assume that the main ARGoS simulator has been installed. If you get an error about FindARGoS.cmake not being in `CMAKE_MODULE_PATH`, it is likely that the main simulator is not properly installed.

```
# clone this repository
git clone https://github.com/iridia-ulb/argos3-pipuck-omnidirectional-camera.git
# compile
cd argos3-pipuck-omnidirectional-camera
mkdir build
cd build
cmake ../src
make
# run test
argos3 -c testing/test_omnidirectional_camera.argos
```

Current state of the plugin

At the time of writing, the sensor is capable of detecting tags directly in front of it but the mirror and its optical properties have not been implemented. Incorporating the mirror into this plugin requires work on both the hardware and simulation implementations.

Hardware

Assuming the Apriltags library is capable of detecting tags in the distorted image from the reflection of the mirror, the first step will be to update the corner and center pixel locations of each detected tag such that they reflect how the tag would have been detected through a normal camera. This should be done after Line 266 of hardware / pipuck_omnidirectional_camera_default_sensor.cpp. Specifically, the coordinates in `ptDetection->p` and `ptDetection->c` need to be updated.

If detecting the color of the LEDs nearby a tag is of interest, the `DetectLed` member function should be updated. This method (i) maps a 3D location (relative to the robot) to pixel coordinates in the captured image and (ii) samples those pixels to determine the color of an LED in that location. There is likely some overlap between the mapping component of this operation and the operation that updates the pixel coordinates of the detected tags.

Simulation

To keep the simulation fast, the tags nearby the robots are simply converted into readings in the simulated sensor. The sensor does not simulate an image from the camera. The conversion of tags into readings is a two step process. The first step is a broad query against the simulator's indices to see if there are any nearby tags. This broad query occurs using the index method `ForEntitiesInBoxRange`. This function takes an axis-aligned bounding box that represents a rough upper approximation of what the camera/mirror can see. It is currently calculated using a frustum, however, it is unlikely that this will apply once the mirror has been taken into consideration.

The second step is a refining query that iterates over the results of the first query to determine whether or not the tags can really be seen. This refining query is implemented in an `operator()` member function that takes a `CTagEntity&` as an argument. This member function is called by `ForEntitiesInBoxRange` for each tag detected in the broad query and is responsible for writing the readings into the control interface. The function applies several checks to determine whether or not the tag really can be seen by the camera. For example, it currently checks, among other things, whether there is anything blocking the rays between the camera and the corners of the tags. This check is used since if more than one of these rays are blocked, it is unlikely that the tag would have been detected in the real world. Such checks will need to be updated to compensate for the optics of the mirror, i.e., to check the path between the corner of a tag and the mirror instead of checking the path between the corner of the tag and the camera.

For the detecting of nearby LEDs, it is possible that no changes will need to be made to the `DetectLed` method. It should only be necessary to pass the same bounding box to the LED index as was passed to the tag index on Line 198 of simulator / pipuck_omnidirectional_camera_default_sensor.cpp.

Current limitations

Focus control is not implemented

The ability to adjust camera focus requires adding an auxiliary or V4L2 I2C subdevice to the OV5647 driver. These changes will mostly need to be made in the meta-pipuck layer and

include: 1. Enabling the video core I2C bus in config.txt 2. Creating an appropriate device tree overlay 3. Patching the OV5647 driver to exposure a control to focus motor

Once these steps are complete, the plugin in this repository can be updated to include the following code to control the camera focus:

```
struct v4l2_control sFocusControl;
memset(&sFocusControl, 0, sizeof (sFocusControl));
sFocusControl.id = V4L2_CID_FOCUS_ABSOLUTE;
sFocusControl.value = m_unCameraFocusAbsolute;
if (::ioctl(m_nCameraHandle, VIDIOC_S_CTRL, &sFocusControl) < 0)
   THROW_ARGOSEXCEPTION("Could not set camera focus");
```

The simulation implementation should also be updated to reflect how changing the focus influences tag detection so that there is reasonable correspondence between how the sensor works in the simulation and how it works on the real robot.

Capture is limited to a single camera

On the real Pi-Puck, it is currently only possible to capture from either the front camera or the omnidirectional camera. This limitation is due to libcamera and its V4L2 compatibility layer which is loaded automatically to support the omnidirectional camera. The solution to this problem would be to use gstreamer in the camera sensors instead of directly using V4L2 (which is how the current camera sensors are implemented).

Tag detection is very slow

The CPU on the Raspberry Pi Zero W is not the fastest (approximately the speed of the first generation of Raspberry Pi boards) and it can take almost a second to detect tags from a 640x480 resolution image. The solution to this problem would be to upgrade to the Raspberry Pi Zero 2 W, which contains a quad-core processor and is probably capable of reducing the processing time down to around 200 milliseconds. The Raspberry Pi Zero 2 W appears to be mechanically compatible with the Raspberry Pi Zero W, so it should be compatible with the existing Pi-Puck expansion PCBs.

The meta-pipuck layer for the Yocto build system

Description

The meta-pipuck repository contains a layer for the Yocto build system, which generates a complete, bootable Linux OS ready to be run on the Pi-Puck mobile robot. This layer is based on the meta-raspberrypi layer. The system comes preinstalled with:

- ARGoS3 and a plugin for the Pi-Puck
- Python3

Quick start

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

To get started, you first need to decide where you want to build the system for the Pi-Puck. Keep in mind that while the resulting image will be between 200-300 megabytes in size, the build system itself will require around **50 GB** of free disk space. The following steps will set up the build system. These steps assume that the build location is `/home/$(id -un)/yocto-pipuck` where `$(id -un)` will be replaced with the current username.

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

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

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

```
git clone https://github.com/iridia-ulb/meta-pipuck.git
```

Create the Docker image

The following command will execute the Dockerfile in the meta-pipuck repository and create a Docker image based on Ubuntu 20.04 LTS. The image will contain a user and a group, which match the identifiers of current user and group. Setting the user and group in this way enables trivial access to the build system from the host.

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

Create the Docker container

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

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

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

```
sudo docker start yocto-pipuck
sudo docker attach yocto-pipuck
```

Start the build

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

```
bitbake pipuck-image-base
```

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

Copying the image

The most straightforward way to burn a bootable image to the SD card is to use `bmaptool` from Intel. On Ubuntu, this package can be installed with `sudo apt install bmap-tools`. Most distributions of Linux should have a similar package that can be installed.

To burn the image, you need to locate the output image from the build system and to identify the device to which you would like to copy the image. The output image should be located under `yocto-pipuck/poky/build/tmp/deploy/images/raspberrypi0-wifi`. The device (probably an SD card) that you want to write to will usually be something like `/dev/sdX` or `/dev/mmcblkX`. The easiest way to find out is to inspect the output of `dmesg` before and after inserting the SD card into your computer. You will need to unmount the device before burning the image. Be careful – writing the image to the wrong device will result in data loss.

```
umount /dev/DEVICE*
sudo bmaptool copy PATH/TO/pipuck-image-base-raspberrypi0-wifi.wic.bmap \
 /dev/DEVICE
```

Booting the image and accessing the console

The easiest and most reliable way to get access to the Pi-Puck is by using the on-board serial-to-USB converter. You can then connect to the board using a terminal application such as Picocom as follows:

```
picocom -b 115200 /dev/ttyUSBX
```

Where `ttyUSBX` is the serial-to-USB converter. Check `dmesg` while attaching the cable to confirm that you have the right device. Note that to access the serial port, you will either have to (i) use `sudo`, (ii) switch to the root user, or (iii) add yourself to the `dialout` group (do not forget to restart afterwards).

Wifi configuration

The wireless connection is controlled using the `iwctl` command. This interactive command makes the process of connecting to a wireless network relatively painless. Once you are connected, the wireless network is saved on the Pi-Puck under `/var/lib/iwd/SSID.KEY_TYPE`. By default, the Pi-Puck will connect to a network with the name `MergeableNervousSystem` using PSK authentication with the password `uprising`. The network should automatically connect on boot and fetch an IP address using DHCP.

Acknowledgements

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