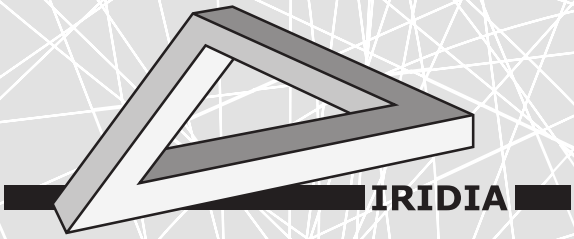

Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Software support for the BuilderBot

M. Allwright

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2022-003

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/2022-003

Revision history:

TR/IRIDIA/2022-003.001	March 2022
TR/IRIDIA/2022-003.002	October 2022
TR/IRIDIA/2022-003.003	October 2022

Table of contents

Introduction

This technical report covers the software support for the BuilderBot 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 BuilderBot construction robot
2. The meta-builderbot layer for the Yocto build system

The firmware for the BuilderBot construction robot is available in the Github repository iridia-ulb/builderbot-firmware. For more details about the BuilderBot hardware, please refer to the journal article published in HardwareX: An open-source multi-robot construction system.

ARGoS Plug-in for the BuilderBot construction robot

The BuilderBot construction robot is supported in the ARGoS simulator. The plugin in the ARGoS repository defines the control interface and provides an implementation of the BuilderBot in simulation. This implementation provides a visualization for the `qt-opengl` plugin and a model for the `dynamics3d` physics engine plugin.

The control interface defined in the ARGoS repository is also used on the real robot. The plugin for the real robot is available in the argos3-builderbot repository. This plugin can compiled as is, however, it is advisable to instead follow the instructions in the meta-builderbot 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 BuilderBot that includes support for ARGoS.

Usage

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

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

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 engine support

Since the BuilderBot needs to be able to collide with and pick up objects in the arena such as the stigmergic blocks, it can only be supported by the `dynamics3d` physics engine. To simulate the magnetism that enables the stigmergic blocks in the arena to be picked up and to be attracted to each other, a special plugin called `srocs` needs to be loaded as shown in the XML snippet below:

```
<dynamics3d id="dyn3d" iterations="25">
  <floor />
  <gravity g="9.8"/>
  <srocs />
</dynamics3d>
```

The `floor` and `gravity` plugins must also be loaded so that friction is generated between the wheels of the robot and the floor, which enables the robot to move around the environment. By default, when a robot is moved or rotated it does not move the block with it. This behavior can be changed by implementing a Qt-OpenGL user function that updates the position/orientation of a block being carried by a robot when it is moved or rotated.

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 BuilderBot's simulation model in ARGoS contains a crude approximation of wifi and NFC whereas messages are simply broadcasted to all nearby radios. The range of the wifi radio is 10 meters and the range of the NFC radio is 2 centimeters. For these radios to work, they need to be assigned to a `simple_radio` medium using the attributes `wifi_medium` and `nfc_medium`. There is also a tag on top of the BuilderBot that needs to be placed into a `tag` medium specified by the `tag_medium` attribute. The following XML provides an example of this configuration:

```
<arena size="1, 1, 1" positional_index="grid" positional_grid_size="1,1,1">
  <builderbot id="builderbot" wifi_medium="wifi" nfc_medium="nfc" tag_medium="tags">
    <body position="0,0,0" orientation="0,0,0"/>
    <controller config="my_controller"/>
  </builderbot>
</arena>

<media>
  <tag id="tags" index="grid" grid_size="5,5,5" />
  <simple_radio id="wifi" index="grid" grid_size="5,5,5" />
  <simple_radio id="nfc" index="grid" grid_size="5,5,5" />
</media>
```

Note that the LEDs on the BuilderBot are not currently added to any medium and hence are for visualization only. These LEDs can therefore not be detected by cameras in the simulation.

Sensors

The following section lists the sensors that are specific to the BuilderBot. There are also a few generic sensors (e.g., `simple_radios` for wifi and NFC) 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.

builderbot_differential_drive

This sensor is designed to read back the current velocity from the wheels. The sensor can be added to the `sensors` section of a controller as follows:

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

This configuration shows the sensor being added to a Lua controller. With this configuration, the current 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.

builderbot_camera_system

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

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <builderbot_camera_system implementation="default"
      show_tag_rays="false"
      show_led_rays="false"
      show_frustum="false" />
  </sensors>
  <params />
```

```
</lua_controller>
```

The camera sensor currently requires a `tag` medium called `tags` to be declared in the `media` section of the experiment configuration file. Moreover, a `directional_led` medium called `directional_leds` must also be declared for this sensor to work. The attributes `show_tag_rays`, `show_led_rays`, and `show_frustum` default to false but can be set to true to assist debugging. The sensor exports three Lua functions: `enable`, `disable`, and `detect_led`. The BuilderBot has a known issue in that the main CPU often overheats causing the CPU speed to be throttled. A workaround for this problem is to only enable the camera when it is strictly necessary to do so. By default the camera is disabled and needs to be enabled as per the following Lua code:

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

```
  end
end
```

builderbot_electromagnet_system

This is a sensor that is capable of reading the state of the electromagnet system. The electromagnet system essentially charges three large capacitors on the BuilderBot that can be discharged through four semi-permanent electromagnets in the front of the BuilderBot. This discharging generates a pulse which can improve the alignment of a nearby stigmergic block and also attach/detach it from the semi-permanent electromagnets. The sensor can be added as follows:

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

The current accumulated voltage can be written to the ARGoS log as follows:

```
function step()
  log('voltage = ' .. robot.electromagnet_system.voltage)
end
```

builderbot_lift_system

The lift system sensor is capable of reporting the current position of the lift, the controller's state (`inactive`, `position_control`, `speed_control`, `calibration_search_top`, or `calibration_search_bottom`), and whether or not the limit switches at the top and the bottom of the lift are being pressed. The sensor can be added as follows:

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <builderbot_lift_system 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()
  log('state = ' .. robot.lift_system.state)
  log('position = ' .. robot.lift_system.position)
  log('limit_switches.top = ' .. robot.lift_system.limit_switches.top)
  log('limit_switches.bottom = ' .. robot.lift_system.limit_switches.bottom)
end
```

builderbot_rangefinders

The rangefinders sensor is used to detect nearby obstacles around the BuilderBot and its end effector. The following configuration demonstrates how to use this sensor in a controller.

```
<lua_controller id="my_controller">
  <actuators />
  <sensors>
    <builderbot_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 id, sensor in pairs(robot.rangefinders)
    log('rangefinder ' .. id .. ':')
    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
```

builderbot_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>
    <builderbot_system implementation="default" />
  </sensors>
```

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

The current time and temperature of the CPU are available from a Lua controller and can be printed to the ARGoS log as follows:

```
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 BuilderBot. There are also a few generic actuators (e.g., `simple_radios` for wifi and NFC) that can be added. These actuators are in part discoverable by running `argos3 -q actuators` in a terminal.

`builderbot_differential_drive`

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

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

The actuator exports a single function in Lua called `set_target_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_target_velocity(5, 5)
end
```

`builderbot_electromagnet_system`

The electromagnet system actuator controls the discharge mode of the electromagnet system. The discharge mode can be constructive (the magnetic field of semi-permanent magnets is increased), destructive (the magnetic field of the semi-permanent magnets is decreased), or disabled (no change in magnetic field, capacitors are able to recharge). The actuator can be added as follows:

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

The actuator exports a single function in Lua called `set_discharge_mode` which takes a single argument, a string matching either: `constructive`, `destructive`, or `disabled`.

```
function step()
  -- drop the stigmergic block
  robot.electromagnet_system.set_discharge_mode('destructive')
end
```

builderbot_lift_system

The lift system actuator controls the lift system of the BuilderBot and can be added to a controller as follows:

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

The actuator exports three functions to Lua called `calibrate`, `stop`, and `set_position`. The `calibrate` function moves the left all the way to the top and all the way to the bottom to calculate the valid range of the lift. The function `stop`, shuts down the actuator immediately. Finally, the function `set_position` sets the position of the lift actuator in meters.

```
function step()
  -- move the lift to the height of a stigmergic block
  robot.lift_system.set_position(0.055)
end
```

builderbot_leds

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

```
<lua_controller id="my_controller">
  <actuators>
```

```
    <builderbot_leds implementation="default" />
  </actuators>
  <sensors />
  <params />
</lua_controller>
```

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

Function	Arguments
`set_led_index`	The index (integer between 1..12) and the color (either a string or three RGB colors)
`set_leds`	the color (either a string or three RGB colors)

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

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

The meta-builderbot layer for the Yocto build system

Description

The meta-builderbot repository contains a layer for the Yocto build system, which generates a complete, bootable Linux OS ready to be run on the BuilderBot. This layer is based on the meta-duovero layer by Scott Ellis. The system comes preinstalled with:

- ARGoS3 and a plugin for the BuilderBot
- 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 BuilderBot. 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/$(id -un)/yocto-builderbot` where `$(id -un)` will be replaced with the current username.

```
# Create a directory for the build system
mkdir /home/`id -un`/yocto-builderbot
```

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

```
# Switch to the build location
cd /home/`id -un`/yocto-builderbot
# Clone the Yocto repository
git clone git://git.yoctoproject.org/poky --branch sumo --single-branch
# Clone this layer inside the poky directory
cd poky
git clone https://github.com/iridia-ulb/meta-builderbot.git
```

Create the Docker image

The following command will execute the Dockerfile in the meta-builderbot repository and create a Docker image based on Ubuntu 16.04 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-builderbot:latest \
 https://github.com/iridia-ulb/meta-builderbot.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-builderbot` under the current user's home directory on the host.

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

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 BuilderBot, just run the following command:

```
bitbake console-image-builderbot
```

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 instructions for creating a bootable microSD card are available on the Gumstix website.

Booting the image and accessing the console

The easiest and most reliable way to get access to the BuilderBot 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

Configuration for the wireless connection can be made by adding networks to `wpa_supplicant.conf-sane` before building the image. By default, the BuilderBot will connect to the network `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).