# Webots Reference Manual

release 8.0.1



**www.cyberbotics.com**

November 17, 2014

# Trademark information

# Thanks

Cyberbotics is grateful to all the people who contributed to the development of Webots, Webots sample applications, the Webots User Guide, the Webots Reference Manual, and the Webots web site, including Yvan Bourquin, Fabien Rohrer, Jean-Christophe Fillion-Robin, Jordi Porta, Emanuele Ornella, Yuri Lopez de Meneses, Sébastien Hugues, Auke-Jan Ispeert, Jonas Buchli, Alessandro Crespi, Ludovic Righetti, Julien Gagnet, Lukas Hohl, Pascal Cominoli, Stéphane Mojon, Jérôme Braure, Sergei Poskriakov, Anthony Truchet, Alcherio Martinoli, Chris Cianci, Nikolaus Correll, Jim Pugh, Yizhen Zhang, Anne-Elisabeth Tran Qui, Grégory Mermoud, Lucien Epinet, Jean-Christophe Zufferey, Laurent Lessieux, Aude Billiard, Ricardo Tellez, Gerald Foliot, Allen Johnson, Michael Kertesz, Simon Garnieri, Simon Blanchoud, Manuel João Ferreira, Rui Picas, José Afonso Pires, Cristina Santos, Michal Pytasz and many others.

Many thanks are also due to Cyberbotics's Mentors: Prof. Jean-Daniel Nicoud (LAMI-EPFL), Dr. Francesco Mondada (EPFL), Dr. Takashi Gomi (Applied AI, Inc.).

Finally, thanks to Skye Legon and Nathan Yawn, who proofread this manual.

# Contents

# Chapter 1

# Introduction

This manual contains the specification of the nodes and fields of the `.wbt` world description language used in Webots. It also specifies the functions available to operate on these nodes from controller programs.

The Webots nodes and APIs are open specifications which can be freely reused without authorization from Cyberbotics. The Webots API can be freely ported and adapted to operate on any robotics platform using the remote-control and/or the cross-compilation frameworks. Cyberbotics offers support to help developers implementing the Webots API on real robots. This benefits to the robotics community by improving interoperability between different robotics applications.

## 1.1 Nodes and Functions

### 1.1.1 Nodes

Webots nodes listed in this reference are described using standard VRML syntax. Principally, Webots uses a subset of the VRML97 nodes and fields, but it also defines additional nodes and fields specific to robotic definitions. For example, the Webots `WorldInfo` and `Sphere` nodes have additional fields with respect to VRML97.

### 1.1.2 Functions

This manual covers all the functions of the controller API, necessary to program robots. The C prototypes of these functions are described under the *SYNOPSIS* tag. The prototypes for the other languages are available through hyperlinks or directly in chapter 9. The language-related particularities mentioned under the label called *C++ Note, Java Note, Python Note, Matlab Note*, etc.

## 1.2 ODE: Open Dynamics Engine

Webots relies on ODE, the Open Dynamics Engine, for physics simulation. Hence, some Webots parameters, structures or concepts refer to ODE. The Webots documentation does not, however, duplicate or replace the ODE documentation. Hence, it is recommended to consult the ODE documentation to understand these parameters, structures or concepts. This ODE documentation is available online from the ODE web site[1].

[1] http://www.ode.org

# Chapter 2

# Node Chart

## 2.1 Chart

The Webots Node Chart outlines all the nodes available to build Webots worlds.

In the chart, an arrow between two nodes represents an inheritance relationship. The inheritance relationship indicates that a derived node (at the arrow tail) inherits all the fields and API functions of a base node (at the arrow head). For example, the `Supervisor` node inherits from the `Robot` node, and therefore all the fields and functions available in the `Robot` node are also available in the `Supervisor` node.

Boxes depicted with a dashed line (`Light`, `Device` and `Geometry`) represent *abstract* nodes, that is, nodes that cannot be instantiated (either using the SceneTree or in a `.wbt` file). Abstract nodes are used to group common fields and functions that are shared by derived nodes.

A box with round corners represents a `Geometry` node; that is, a node that will be graphically depicted when placed in the `geometry` field of a `Shape` node.

A box with a grey background indicates a node that can be used directly (or composed using `Group` and `Transform` nodes) to build a *boundingObject* used to detect collisions between `Solid` objects. Note that not all geometry nodes can be used as boundingObjects, and that although `Group` and `Transform` can be used, not every combination of these will work correctly.

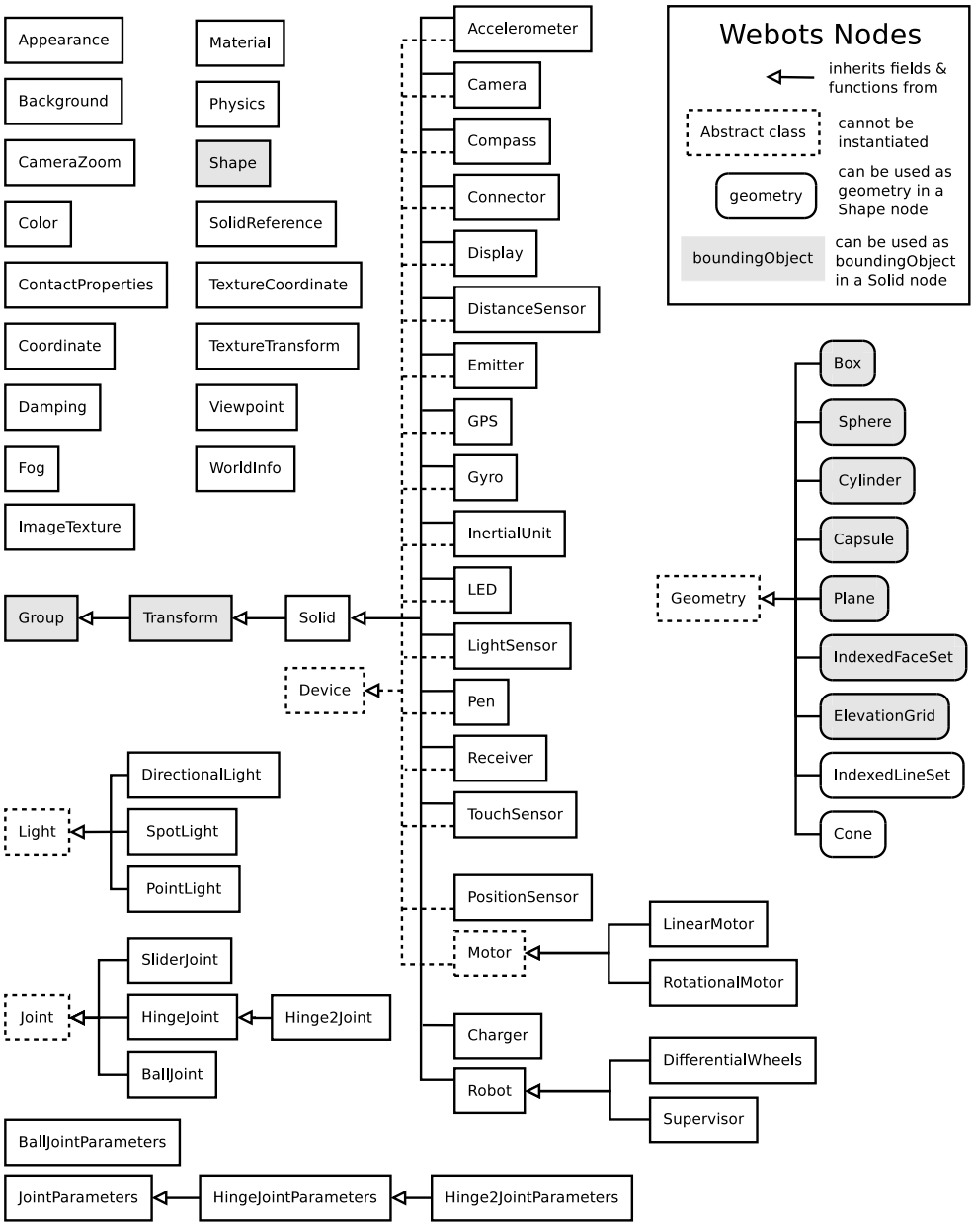


Figure 2.1: Webots Nodes Chart

# Chapter 3

# Nodes and API Functions

## 3.1 Accelerometer

Derived from Device.

```
Accelerometer {
  MFVec3f    lookupTable    []    # interpolation
  SFBool     xAxis          TRUE  # compute x-axis
  SFBool     yAxis          TRUE  # compute y-axis
  SFBool     zAxis          TRUE  # compute z-axis
  SFFloat    resolution     -1
}
```

### 3.1.1 Description

The Accelerometer node can be used to model accelerometer devices such as those commonly found in mobile electronics, robots and game input devices. The Accelerometer node measures acceleration and gravity induced reaction forces over 1, 2 or 3 axes. It can be used for example to detect fall, the up/down direction, etc.

### 3.1.2 Field Summary

- lookupTable: This field optionally specifies a lookup table that can be used for mapping the raw acceleration values [m/s$^2$] to device specific output values. With the lookup table it is also possible to add noise and to define the min and max output values. By default the lookup table is empty and therefore the raw acceleration values are returned (no mapping).

- `xAxis, yAxis, zAxis`: Each of these boolean fields enables or disables computation for the specified axis. If one of these fields is set to FALSE, then the corresponding vector element will not be computed and will return *NaN* (Not a Number). For example, if `zAxis` is FALSE, then `wb_accelerometer_get_values()[2]` will always return *NaN*. The default is that all three axes are enabled (TRUE). Modifying these fields makes it possible to choose between a single, dual or three-axis accelerometer and to specify which axes will be used.
- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. For example, if `resolution` is 0.2 instead of returning 1.767 the sensor will return 1.8. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

### 3.1.3 Accelerometer Functions

---

#### NAME

wb_accelerometer_enable,
wb_accelerometer_disable,
wb_accelerometer_get_sampling_period,
wb_accelerometer_get_values – *enable, disable and read the output of the accelerometer*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/accelerometer.h>

void wb_accelerometer_enable (WbDeviceTag tag, int ms);

void wb_accelerometer_disable (WbDeviceTag tag);

int wb_accelerometer_get_sampling_period (WbDeviceTag tag);

const double *wb_accelerometer_get_values (WbDeviceTag tag);
```

#### DESCRIPTION

The `wb_accelerometer_enable()` function allows the user to enable the acceleration measurement each `ms` milliseconds.

The `wb_accelerometer_disable()` function turns the accelerometer off, saving computation time.

The `wb_accelerometer_get_sampling_period()` function returns the period given into the `wb_accelerometer_enable()` function, or 0 if the device is disabled.

The wb_accelerometer_get_values() function returns the current values measured by the Accelerometer. These values are returned as a 3D-vector, therefore only the indices 0, 1, and 2 are valid for accessing the vector. Each element of the vector represents the acceleration along the corresponding axis of the Accelerometer node, expressed in meters per second squared [m/s$^2$]. The first element corresponds to the x-axis, the second element to the y-axis, etc. An Accelerometer at rest with earth's gravity will indicate 1 g (9.81 m/s$^2$) along the vertical axis. Note that the gravity can be specified in the gravity field in the WorldInfo node. To obtain the acceleration due to motion alone, this offset must be subtracted. The device's output will be zero during free fall when no offset is substracted.


**language: C, C++**
*The returned vector is a pointer to the internal values managed by the* Accelerometer *node, therefore it is illegal to free this pointer. Furthermore, note that the pointed values are only valid until the next call to* wb_robot_step() *or* Robot::step(). *If these values are needed for a longer period they must be copied.*


**language: Python**
getValues() *returns the 3D-vector as a list containing three floats.*

## 3.2 Appearance

```
Appearance {
  SFNode   material           NULL
  SFNode   texture            NULL
  SFNode   textureTransform   NULL
}
```

### 3.2.1 Description

The Appearance node specifies the visual properties of a geometric node. The value for each of the fields in this node may be NULL. However, if the field is non-NULL, it shall contain one node of the appropriate type.

### 3.2.2 Field Summary

- The material field, if specified, shall contain a Material node. If the material field is NULL or unspecified, lighting is off (all lights are ignored during rendering of the object that references this Appearance) and the unlit object color is (1,1,1).

- The `texture` field, if specified, shall contain an `ImageTexture` node. If the `texture` node is NULL or the `texture` field is unspecified, the object that references this `Appearance` is not textured.
- The `textureTransform` field, if specified, shall contain a `TextureTransform` node. If the `textureTransform` is NULL or unspecified, the `textureTransform` field has no effect.

## 3.3 Background

```
Background {
  MFColor   skyColor   [ 0 0 0 ]   # [0,1]
}
```

The `Background` node defines the background used for rendering the 3D world. The `skyColor` field defines the red, green and blue components of this color. Only the three first float values of the `skyColor` field are used.

## 3.4 BallJoint

Derived from `Joint`.

```
BallJoint {
}
```

### 3.4.1 Description

The `BallJoint` node can be used to model a ball joint. A ball joint, also called ball-and-socket, prevents translation motion while allowing rotation around its anchor (3 degrees of freedom). It supports spring and damping parameters which can be used to simulate the elastic deformation of ropes and flexible beams.

It inherits `Joint`'s `jointParameters` field. This field can be filled with a `BallJointParameters` node only. If empty, `BallJointParameters` default values apply.

## 3.5 BallJointParameters

```
BallJointParameters {
  field SFVec3f anchor 0 0 0      # point at which the bodies are
     connected (m)
  field SFFloat springConstant  0 # uniform rotational spring constant
      (Nm)
  field SFFloat dampingConstant 0 # uniform rotational damping
     constant (Nms)
}
```

### 3.5.1 Description

The BallJointJointParameters node can be used to specify the parameters of a ball joint. It contains the anchor position, i.e. the coordinates of the point where bodies under a ball joint constraints are kept attached. It can be used in the jointParameters field of BallJoint only.

### 3.5.2 Field Summary

- anchor: This field specifies the anchor position expressed in relative coordinates with respect to the center of the closest upper Solid's frame.
- springConstant and dampingConstant: These fields specify the uniform amount of rotational spring and damping effect around each of the the frame axis of the BallJoint's closest upper Solid (see Joint's "Springs and Dampers" section for more information on these constants). This is can be useful to simulate a retraction force that pulls the BallJoint solid endPoint back towards its initial orientation.

## 3.6 Box

```
Box {
  SFVec3f   size   2 2 2   # (-inf,inf)
}
```

### 3.6.1 Description

The Box node specifies a rectangular parallelepiped box centered at (0,0,0) in the local coordinate system and aligned with the local coordinate axes. By default, the box measures 2 meters in each dimension, from -1 to +1.

The size field specifies the extents of the box along the *x*-, *y*-, and *z*-axes respectively. See figure 3.1. Three positive values display the outside faces while three negative values display the inside faces.

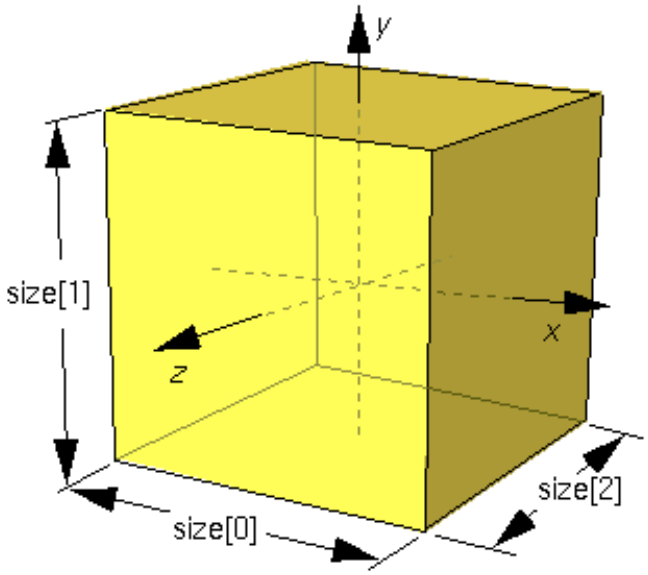


Figure 3.1: Box node

Textures are applied individually to each face of the box. On the front (+*z*), back (-*z*), right (+*x*), and left (-*x*) faces of the box, when viewed from the outside with the +*y*-axis up, the texture is mapped onto each face with the same orientation as if the image were displayed normally in 2D. On the top face of the box (+*y*), when viewed from above and looking down the *y*-axis toward the origin with the -*z*-axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. On the bottom face of the box (-*y*), when viewed from below looking up the *y*-axis toward the origin with the +Z-axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. `TextureTransform` affects the texture coordinates of the Box.

## 3.7 Brake

Derived from `Device`.

```
Brake {
}
```

### 3.7.1 Description

A `Brake` node can be used in a mechanical simulation in order to change the friction of a joint. The `Brake` node can be inserted in the `device` field of a `HingeJoint`, a `Hinge2Joint`,

or a SliderJoint.

### 3.7.2 Brake Functions

---

#### NAME

wb_brake_set_damping_constant,
wb_brake_get_type – *set the damping constant coefficient of the joint and get the type of brake*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/brake.h>

void wb_brake_set_damping_constant (WbDeviceTag tag, double damping_constant);

int wb_brake_get_type (WbDeviceTag tag);
```

#### DESCRIPTION

`wb_brake_set_damping_constant()` sets the value of the dampingConstant coefficient (Ns/m or Nms) of the joint. If any dampingConstant is already set using JointParameters the resulting dampingConstant coefficient is the sum of the one in the JointParameters and the one set using the `wb_brake_set_damping_constant()` function.

`wb_brake_get_type()` returns the type of the brake. It will return WB_ANGULAR if the sensor is associated with a HingeJoint or a Hinge2Joint node, and WB_LINEAR if it is associated with a SliderJoint.

## 3.8 Camera

Derived from Device.

```
Camera {
  SFFloat    fieldOfView      0.7854
  SFInt32    width            64
  SFInt32    height           64
  SFString   type             "color"
  SFBool     spherical        FALSE
  SFFloat    near             0.01
  SFFloat    maxRange         1.0
  SFVec2f    windowPosition   0 0
  SFFloat    pixelSize        1.0
```

```
  SFBool     antiAliasing     FALSE
  SFFloat    colorNoise       0.0
  SFFloat    rangeNoise       0.0
  SFFloat    rangeResolution -1.0
  SFNode     zoom             NULL
}
```

### 3.8.1 Description

The Camera node is used to model a robot's on-board camera, a range-finder, or both simultaneously. The resulted image can be displayed on the 3D window. Depending on its setup, the Camera node can model a linear camera, a lidar device, a Microsoft Kinect or even a biological eye which is spherically distorted.

### 3.8.2 Field Summary

- `fieldOfView`: horizontal field of view angle of the camera. The value ranges from *0* to $\pi$ radians. Since camera pixels are squares, the vertical field of view can be computed from the `width`, `height` and horizontal `fieldOfView`:

  *vertical FOV = fieldOfView * height / width*

- `width`: width of the image in pixels

- `height`: height of the image in pixels

- `type`: type of the camera: "color", "range-finder" or "both". The camera types are described precisely in the corresponding subsection below.

- `spherical`: switch between a planar or a spherical projection. A spherical projection can be used for example to simulate a biological eye or a lidar device. More information on spherical projection in the corresponding subsection below.

- The `near` field defines the distance from the camera to the near clipping plane. This plane is parallel to the camera retina (i.e. projection plane). The near field determines the precision of the OpenGL depth buffer. A too small value produces depth fighting between overlaid polygons, resulting in random polygon overlaps. More information on frustums in the corresponding subsection below.

- The `maxRange` field is used only when the camera is a range-finder. In this case, `maxRange` defines the distance between the camera and the far clipping plane. The far clipping plane is not set to infinity. This field defines the maximum range that a range-finder can achieve and so the maximum possible value of the range image (in meter).

- The `windowPosition` field defines a position in the main 3D window where the camera image will be displayed. The X and Y values for this position are floating point values between 0.0 and 1.0. They specify the position of the center of the camera image, relatively to the top left corner of the main 3D view. This position will scale whenever the main window is resized. Also, the user can drag and drop this camera image in the main Webots window using the mouse. This will affect the X and Y position values.

- The `pixelSize` field defines the zoom factor for camera images rendered in the main Webots window (see the `windowPosition` description). Setting a `pixelSize` value higher than 1 is useful to better see each individual pixel of the camera image. Setting it to 0 simply turns off the display of the camera image, thus saving computation time.

- The `antiAliasing` field switches on or off (the default) anti-aliasing effect on the camera images. Anti-aliasing is a technique that assigns pixel colors based on the fraction of the pixel's area that's covered by the primitives being rendered. Anti-aliasing makes graphics more smooth and pleasing to the eye by reducing aliasing artifacts. Aliasing artifacts can appear as jagged edges (or moiré patterns, strobing, etc.). Anti-aliasing will not be applied if it is not supported by the hardware.

- If the `colorNoise` field is greater than 0.0, this adds a gaussian noise to each RGB channel of a color image. This field is useless in case of range-finder cameras. A value of 0.0 corresponds to remove the noise and thus saving computation time. A value of 1.0 corresponds to a gaussian noise having a standard derivation of 255 in the channel representation. More information on noise in the corresponding subsection below.

- If the `rangeNoise` field is greater than 0.0, this adds a gaussian noise to each depth value of a range-finder image. This field is useless in case of color cameras. A value of 0.0 corresponds to remove the noise and thus saving computation time. A value of 1.0 corresponds to a gaussian noise having a standard derivation of `maxRange` meters. More information on noise in the corresponding subsection below.

- `rangeResolution`: This field allows to define the depth resolution of the camera, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field is used only when type is "range-finder" or "both" and accepts any value in the interval (0.0, inf).

- The `zoom` field may contain a `CameraZoom` node to provide the camera device with a controllable zoom system. If this field is set to NULL, then no zoom is available on the camera device.

### 3.8.3 Camera Type

The camera type can be setup by the `type` field described above.

#### Color

The color camera allows to get color information from the OpenGL context of the camera. This information can be get by the `wb_camera_get_image` function, while the red, green and blue channels (RGB) can be extracted from the resulted image by the `wb_camera_image_get_*`-like functions.

Internally when the camera is refreshed, an OpenGL context is created, and the color or depth information is copied into the buffer which can be get throught the `wb_camera_get_image` or the `wb_camera_get_range_image` functions. The format of these buffers are respectively BGRA (32 bits) and float (16 bits). We recommend to use the `wb_camera_image_get_*`-like functions to access the buffer because the internal format can changed.

#### Range-Finder

The range-finder camera allows to get depth information (in meters) from the OpenGL context of the camera. This information is obtained through the `wb_camera_get_range_image` function, while depth information can be extracted from the returned image by using the `wb_camera_range_image_get_depth` function.

Internally when the camera is refreshed, an OpenGL context is created, and the z-buffer is copied into a buffer of `float`. As the z-buffer contains scaled and logarithmic values, an algorithm linearizes the buffer to metric values between `near` and `maxRange`. This is the buffer which is accessible by the `wb_camera_get_range_image` function.

Range-finder cannot see transparent objects. An object can be semi-transparent either if its texture has an alpha channel, or if its `Material.transparency` field is not equal to 1.

#### Both

This type of camera allows to get both the color data and the range-finder data in the returned buffer using the same OpenGL context. This has been introduced for optimization reasons, mainly for the Microsoft Kinect device, as creating the OpenGL context is costly. The color image and the depth data are obtained by using the `wb_camera_get_image` and the `wb_camera_get_range_image` functions as described above.

### 3.8.4 Frustum

The frustum is the truncated pyramid defining what is visible from the camera. Any 3D shape completely outside this frustum won't be rendered. Hence, shapes located too close to the camera (standing between the camera and the near plane) won't appear. It can be displayed with magenta lines by enabling the `View|Optional Rendering|Show Camera Frustums` menu item. The `near` field defines the position of the near clipping plane (x, y, -near). The

fieldOfView field defines the horizontal angle of the frustum. The fieldOfView, width and height fields define the vertical angle of the frustum according to the above formula.

Generally speaking there is no far clipping plane while this is common in other OpenGL programs. In Webots, a camera can see as far as needed. Nevertheless, a far clipping plane is artificially added in the case of range-finder cameras (i.e. the resulted values are bounded by the maxRange field).

In the case of the spherical cameras, the frustum is quite different and difficult to represent. In comparison with the frustum description above, the near and the far planes are transformed to be sphere parts having their center at the camera position, and the fieldOfView can be greater than Pi.

### 3.8.5 Noise

It is possible to add quickly a white noise on the cameras by using the colorNoise and the rangeNoise fields (applied respectively on the color cameras and on the range-finder cameras). A value of 0.0 corresponds to an image without noise. For each channel of the image and at each camera refresh, a gaussian noise is computed and added to the channel. This gaussian noise has a standard deviation corresponding to the noise field times the channel range. The channel range is 256 for a color camera and maxRange for a range-finder camera.

### 3.8.6 Spherical projection

OpenGL is designed to have only planar projections. However spherical projections are very useful for simulating a lidar, a camera pointing on a curved mirror or a biological eye. Therefore we implemented a camera mode rendering spherical projections. It can be enabled simply by switching on the corresponding spherical field described above.

Internally, depending on the field of view, a spherical camera is implemented by using between 1 to 6 OpenGL cameras oriented towards the faces of a cube (the activated cameras are displayed by magenta squares when the View|Optional Rendering|Show Camera Frustums menu item is enabled). Moreover an algorithm computing the spherical projection is applied on the result of the subcameras.

So this mode is costly in terms of performance! Reducing the resolution of the cameras and using a fieldOfView which minimizes the number of activated cameras helps a lot to improve the performances if needed.

When the camera is spherical, the image returned by the wb_camera_get_image or the wb_camera_get_range_image functions is a 2-dimensional array (s,t) in spherical coordinates.

Let hFov be the horizontal field of view, and let theta be the angle in radian between the (0, 0, -z) relative coordinate and the relative coordinate of the target position along the xz plane

relative to the camera, then `s=0` corresponds to a `theta` angle of `-hFov/2`, `s=(width-1)/2` corresponds to a `theta` angle of 0, and `s=width-1` corresponds to a `theta` angle of `hFov/2`.

Similarly, let `vFov` be the vertical field of view (defined just above), and `phi` the angle in radian between the `(0, 0, -z)` relative coordinate and the relative coordinate of the target position along the `xy` plane relative to the camera, `t=0` corresponds to a `phi` angle of `-vFov/2`, `t=(height-1)/2` corresponds to a `phi` angle of 0, and `t=height-1` corresponds to a `phi` angle of `vFov/2`).

### 3.8.7 Camera Functions

---

#### NAME

wb_camera_enable,
wb_camera_disable,
wb_camera_get_sampling_period – *enable and disable camera updates*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

void wb_camera_enable (WbDeviceTag tag, int ms);

void wb_camera_disable (WbDeviceTag tag);

int wb_camera_get_sampling_period (WbDeviceTag tag);
```

#### DESCRIPTION

`wb_camera_enable()` allows the user to enable a camera update each `ms` milliseconds.

`wb_camera_disable()` turns the camera off, saving computation time.

The `wb_camera_get_sampling_period()` function returns the period given into the `wb_camera_enable()` function, or 0 if the device is disabled.

---

#### NAME

wb_camera_get_fov,
wb_camera_set_fov – *get and set field of view for a camera*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

double wb_camera_get_fov (WbDeviceTag tag);

void wb_camera_set_fov (WbDeviceTag tag, double fov);
```

### DESCRIPTION

These functions allow the controller to get and set the value for the field of view (fov) of a camera. The original value for this field of view is defined in the Camera node, as fieldOfView. Note that changing the field of view using wb_camera_set_fov() is possible only if the camera device has a CameraZoom node defined in its zoom field. The minimum and maximum values for the field of view are defined in this CameraZoom node.

---

### NAME

wb_camera_get_width,
wb_camera_get_height – *get the size of the camera image*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

int wb_camera_get_width (WbDeviceTag tag);

int wb_camera_get_height (WbDeviceTag tag);
```

### DESCRIPTION

These functions return the width and height of a camera image as defined in the corresponding Camera node.

---

### NAME

wb_camera_get_near – *get the near parameter of the camera device*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

double wb_camera_get_near (WbDeviceTag tag);
```

### DESCRIPTION

This function returns the near parameter of a camera device as defined in the corresponding `Camera` node.

---

### NAME

wb_camera_get_type – *get the type of the camera*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

int wb_camera_get_type ();
```

### DESCRIPTION

This function returns the type of the camera as defined by the `type` field of the corresponding `Camera` node. The constants defined in `camera.h` are summarized in table 3.1:

| Camera.type | return value |
|---|---|
| ”color” | WB_CAMERA_COLOR |
| ”range-finder” | WB_CAMERA_RANGE_FINDER |
| both” | WB_CAMERA_BOTH |

Table 3.1: Return values for the `wb_camera_get_type()` function


**language: C++, Java, Python**
*In the oriented-object APIs, the WB_CAMERA_* constants are available as static integers of the `Camera` class (for example, Camera::COLOR).*

---

### NAME

wb_camera_get_image,
wb_camera_image_get_red,
wb_camera_image_get_green,
wb_camera_image_get_blue,
wb_camera_image_get_grey – *get the image data from a camera*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>
```

```
const unsigned char *wb_camera_get_image (WbDeviceTag tag);

unsigned char wb_camera_image_get_red (const unsigned char *image, int wi-
dth, int x, int y);

unsigned char wb_camera_image_get_green (const unsigned char *image, int
width, int x, int y);

unsigned char wb_camera_image_get_blue (const unsigned char *image, int wi-
dth, int x, int y);

unsigned char wb_camera_image_get_grey (const unsigned char *image, int wi-
dth, int x, int y);
```

## DESCRIPTION

The `wb_camera_get_image()` function reads the last image grabbed by the camera. The image is coded as a sequence of three bytes representing the red, green and blue levels of a pixel. Pixels are stored in horizontal lines ranging from the top left hand side of the image down to bottom right hand side. The memory chunk returned by this function must not be freed, as it is handled by the camera itself. The size in bytes of this memory chunk can be computed as follows:

`byte_size` = `camera_width` * `camera_height` * 4

Internal pixel format of the buffer is BGRA (32 bits). Attempting to read outside the bounds of this chunk will cause an error.

The `wb_camera_image_get_red()`, `wb_camera_image_get_green()` and `wb_camera_image_get_blue()` macros can be used for directly accessing the pixel RGB levels from the pixel coordinates. The `wb_camera_image_get_grey()` macro works in a similar way but returns the grey level of the specified pixel by averaging the three RGB components. In the C version, these four macros return an `unsigned char` in the range [0..255]. Here is a C usage example:


**language: C**

```c
const unsigned char *image = wb_camera_get_image(
    camera);
for (int x = 0; x < image_width; x++)
  for (int y = 0; y < image_height; y++) {
    int r = wb_camera_image_get_red(image,
        image_width, x, y);
    int g = wb_camera_image_get_green(image,
        image_width, x, y);
    int b = wb_camera_image_get_blue(image,
        image_width, x, y);
    printf("red=%d, green=%d, blue=%d", r, g, b);
  }
```


**language: Java**

`Camera.getImage()` *returns an array of int (*`int[]`*). The length of this array corresponds to the number of pixels in the image, that is the width multiplied by the height of the image. Each* `int` *element of the array represents one pixel coded in BGRA (32 bits). For example red is* `0x0000ff00`*, green is* `0x00ff0000`*, etc. The* `Camera.pixelGetRed()`*,* `Camera.pixelGetGreen()` *and* `Camera.pixelGetBlue()` *functions can be used to decode a pixel value for the red, green and blue components. The* `Camera.pixelGetGrey()` *function works in a similar way, but returns the grey level of the pixel by averaging the three RGB components. Each of these four functions take an* `int` *pixel argument and return an* `int` *color/grey component in the range [0..255]. Here is an example:*

```java
int[] image = camera.getImage();
for (int i=0; i < image.length; i++) {
  int pixel = image[i];
  int r = Camera.pixelGetRed(pixel);
  int g = Camera.pixelGetGreen(pixel);
  int b = Camera.pixelGetBlue(pixel);
  System.out.println("red=" + r + " green=" + g +
      " blue=" + b);
}
```

**language: Python**

`getImage()` *returns a* `string`*. This* `string` *is closely related to the* `const char *` *of the C API.* `imageGet*`*-like functions can be used to get the channels of the camera Here is an example:*

```
#...
cameraData = camera.getImage()

# get the grey component of the pixel (5,10)
grey = Camera.imageGetGrey(cameraData, camera.
   getWidth(), 5, 10)
```


*Another way to use the camera in Python is to get the image by* `getImageArray()` *which returns a* `list<list<list<int>>>`*. This three dimensional list can be directly used for accessing to the pixels. Here is an example:*

```
image = camera.getImageArray()
# display the components of each pixel
for x in range(0,camera.getWidth()):
  for y in range(0,camera.getHeight()):
    red   = image[x][y][0]
    green = image[x][y][1]
    blue  = image[x][y][2]
    grey  = (red + green + blue) / 3
    print 'r='+str(red)+' g='+str(green)+' b='+
       str(blue)
```

**language: Matlab**
`wb_camera_get_image()` *returns a 3-dimensional array of* `uint(8)`*. The first two dimensions of the array are the width and the height of camera's image, the third being the RGB code: 1 for red, 2 for blue and 3 for green.* `wb_camera_get_range_image()` *returns a 2-dimensional array of* `float('single')`*. The dimensions of the array are the width and the length of camera's image and the float values are the metric distance values deduced from the OpenGL z-buffer.*


```
camera = wb_robot_get_device('camera');
wb_camera_enable(camera,TIME_STEP);
half_width = floor(wb_camera_get_width(camera) /
   2);
half_height = floor(wb_camera_get_height(camera)
   / 2);
% color camera image
image = wb_camera_get_image(camera);
red_middle_point = image(half_width,half_heigth
   ,1);% red color component of the pixel lying
   in the middle of the image
green_middle_line = sum(image(half_width,:,2));%
   sum of the green color over the vertical
   middle line of the image
blue_overall = sum(sum(image(:,:,3));% sum of the
    blue color over all the pixels in the image
fprintf('red_middle_point = %d, green_middle_line
   = %d, blue_overall = %d\n', red_middle_point,
    green_middle_line, blue_overall);
% range-finder camera image
image = wb_camera_get_range_image(camera);
imagesc(image,[0 1]);
colormap(gray);
drawnow;
distance = min(min(image))% distance to the
   closest point seen by the camera
```

## NAME

wb_camera_get_range_image,
wb_camera_range_image_get_depth,
wb_camera_get_max_range – *get the range image and range data from a range-finder camera*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

const float *wb_camera_get_range_image (WbDeviceTag tag);

float wb_camera_range_image_get_depth (const float *range_image, int width,
int x, int y);

double wb_camera_get_max_range (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_camera_get_range_image()` macro allows the user to read the contents of the last range image grabbed by a range-finder camera. The range image is computed using the depth buffer produced by the OpenGL rendering. Each pixel corresponds to the distance expressed in meter from the object to the plane defined by the equation $z = 0$ within the coordinates system of the camera. The bounds of the range image is determined by the near clipping plane (defined by the `near` field) and the far clipping plane (see the `maxRange` field). The range image is coded as an array of single precision floating point values corresponding to the range value of each pixel of the image. The precision of the range-finder values decreases when the objects are located farther from the near clipping plane. Pixels are stored in scan lines running from left to right and from top to bottom. The memory chunk returned by this function shall not be freed, as it is managed by the camera internally. The size in bytes of the range image can be computed as follows:

`size` = `camera_width` * `camera_height` * sizeof(float)

Attempting to read outside the bounds of this memory chunk will cause an error.

The `wb_camera_range_image_get_depth()` macro is a convenient way to access a range value, directly from its pixel coordinates. The `camera_width` parameter can be obtained from the `wb_camera_get_width()` function. The `x` and `y` parameters are the coordinates of the pixel in the image.

The `wb_camera_get_max_range()` function returns the value of the `maxRange` field.


**language: Python**
*The Camera class has two methods for getting the camera image. The* `getRangeImage()` *returns a one-dimensional list of floats, while the* `getRangeImageArray()` *returns a two-dimensional list of floats. Their content are identical but their handling is of course different.*

---

### NAME

wb_camera_save_image – *save a camera image in either PNG or JPEG format*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/camera.h>

int wb_camera_save_image (WbDeviceTag tag, const char *filename, int qual-
ity);
```

#### DESCRIPTION

The `wb_camera_save_image()` function allows the user to save a `tag` image which was previously obtained with the `wb_camera_get_image()` function. The image is saved in a file in either PNG or JPEG format. The image format is specified by the `filename` parameter. If `filename` is terminated by `.png`, the image format is PNG. If `filename` is terminated by `.jpg` or `.jpeg`, the image format is JPEG. Other image formats are not supported. The `quality` parameter is useful only for JPEG images. It defines the JPEG quality of the saved image. The `quality` parameter should be in the range 1 (worst quality) to 100 (best quality). Low quality JPEG files will use less disk space. For PNG images, the `quality` parameter is ignored.

The return value of the `wb_camera_save_image()` is 0 in case of success. It is -1 in case of failure (unable to open the specified file or unrecognized image file extension).

## 3.9 CameraZoom

```
CameraZoom {
  SFFloat     minFieldOfView 0.5 # (rad)
  SFFloat     maxFieldOfView 1.5 # (rad)
}
```

### 3.9.1 Description

The `CameraZoom` node allows the user to define a controllable zoom for a `Camera` device. The `CameraZoom` node should be set in the `zoom` field of a `Camera` node. The zoom level can be adjusted from the controller program using the `wb_camera_set_fov()` function.

### 3.9.2 Field Summary

- The `minFieldOfView` and the `maxFieldOfView` fields define respectively the minimum and maximum values for the field of view of the camera zoom (i.e., respectively the maxium and minimum zoom levels). Hence, they represent the minimum and maximum values that can be passed to the `wb_camera_set_fov()` function.

## 3.10 Capsule

```
Capsule {
  SFBool    bottom        TRUE
  SFFloat   height        2     # (-inf,inf)
  SFFloat   radius        1     # (-inf,inf)
  SFBool    side          TRUE
  SFBool    top           TRUE
  SFInt32   subdivision   12    # (2,inf)
}
```

### 3.10.1 Description

A Capsule node is like a Cylinder node except it has half-sphere caps at its ends. The capsule's height, not counting the caps, is given by the height field. The radius of the caps, and of the cylinder itself, is given by the radius field. Capsules are aligned along the local y-axis.

The capsule can be used either as a graphical or collision detection primitive (when placed in a boundingObject). The capsule is a particularly fast and accurate collision detection primitive.

A capsule has three optional parts: the side, the top and the bottom. Each part has an associated boolean field that indicates whether the part should be drawn or not. For collision detection, all parts are considered to be present, regardless of the value of these boolean fields.

If both height and radius are positive, the outside faces of the capsule are displayed while if they are negative, the inside faces are displayed. The values of height and radius must both be greater than zero when the capsule is used for collision detection.

The subdivision field defines the number of triangles that must be used to represent the capsule and so its resolution. More precisely, it corresponds to the number of faces that compose the capsule's side. This field has no effect on collision detection.

When a texture is mapped to a capsule, the texture map is vertically divided in three equally sized parts (e.g., like the German flag). The top part is mapped to the capsule's top. The middle part is mapped to the capsule's side (body). The bottom part is mapped to the capsule's bottom. On each part, the texture wraps counterclockwise (seen from above) starting from the intersection with the $y$- and negative $z$-plane.

## 3.11 Charger

Derived from Solid.

```
Charger {
```

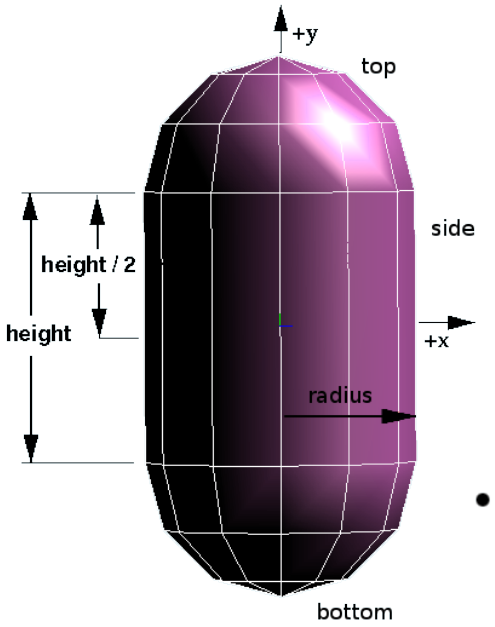


Figure 3.2: The Capsule node

```
  MFFloat    battery         []
  SFFloat    radius          0.04     # (0,inf)
  SFColor    emissiveColor   0 1 0    # [0,1]
  SFBool     gradual         TRUE
}
```

### 3.11.1 Description

The `Charger` node is used to model a special kind of battery charger for the robots. A robot has to get close to a charger in order to recharge itself. A charger is not like a standard battery charger connected to a constant power supply. Instead, it is a battery itself: it accumulates energy with time. It could be compared to a solar power panel charging a battery. When the robot comes to get energy, it can't get more than the charger has presently accumulated.

The appearance of the `Charger` node can be altered by its current energy. When the `Charger` node is full, the resulted color corresponds to its `emissiveColor` field, while when the `Charger` node is empty, its resulted color corresponds to its original one. Intermediate colors depend on the `gradual` field. Only the first child of the `Charger` node is affected by this alteration. The resulted color is applied only on the first child of the `Charger` node. If the first child is a `Shape` node, the `emissiveColor` field of its `Material` node is altered. If the first child is a `Light` node, its `color` field is altered. Otherwise, if the first child is a `Group` node, a recursive search is applied on this node and every `Light`, `Shape` and `Group` nodes are altered according to the two previous rules.

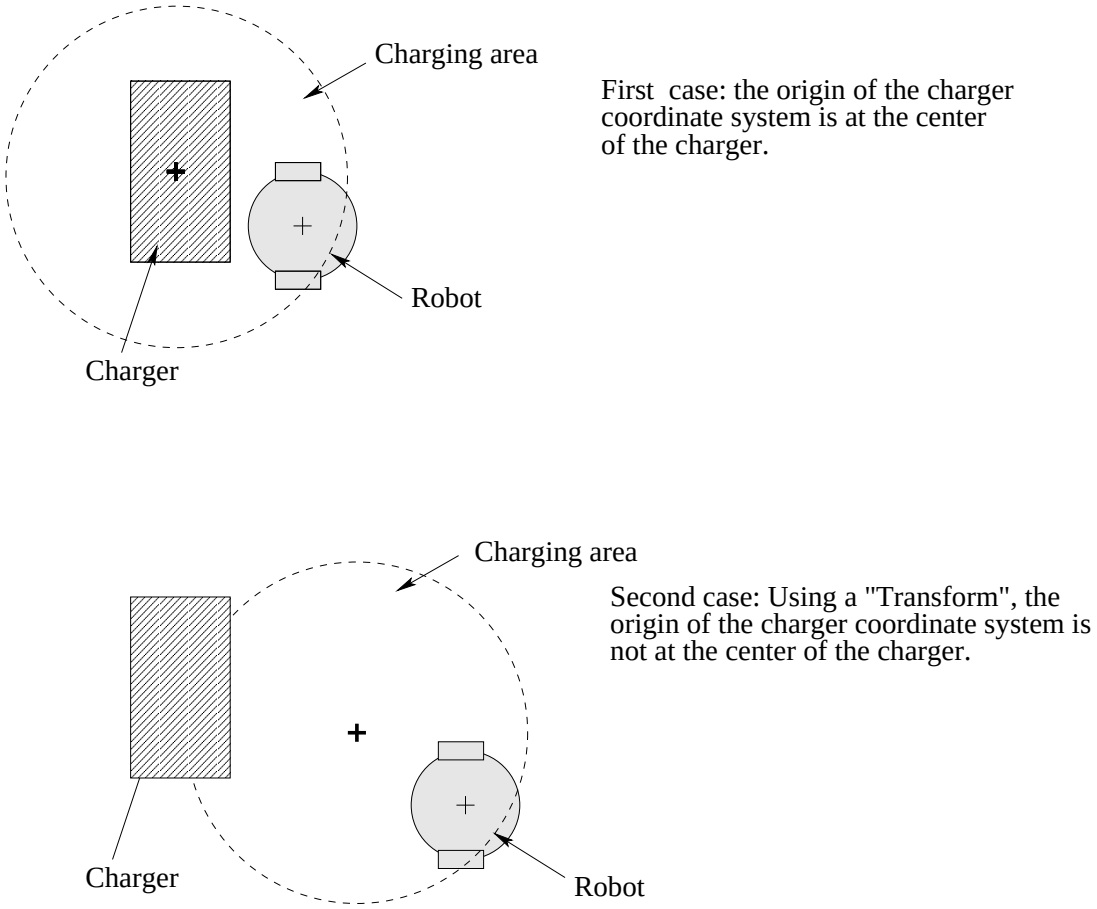


Figure 3.3: The sensitive area of a charger

### 3.11.2 Field Summary

The fields specific to the `Charger` node are:

- `battery`: this field should contain three values, namely the present energy of the charger (*J*), its maximum energy (*J*) and its charging speed (*W=J/s*).
- `radius`: radius of the charging area in meters. The charging area is a disk centered on the origin of the charger coordinate system. The robot can recharge itself if its origin is in the charging area (see figure 3.3).
- `emissiveColor`: color of the first child node (see above) when the charger is full.
- `gradual`: defines the behavior of the indicator. If set to TRUE, the indicator displays a progressive transition between its original color and the `emissiveColor` specified in the `Charger` node, corresponding to the present level of charge. If set to FALSE, the indicator remains its original color until the charger is fully charged (i.e., the present energy

level equals the maximum energy level). Then, it switches to the specified `emissiveColor`.

## 3.12 Color

```
Color {
  MFColor   color   []   # [0,1]
}
```

This node defines a set of RGB colors to be used in the fields of another node.

`Color` nodes are only used to specify multiple colors for a single geometric shape, such as colors for the faces or vertices of an `ElevationGrid`. A `Material` node is used to specify the overall material parameters of a geometric node. If both a `Material` node and a `Color` node are specified for a geometric shape, the colors shall replace the diffuse component of the material.

RGB or RGBA textures take precedence over colors; specifying both an RGB or RGBA texture and a Color node for a geometric shape will result in the Color node being ignored.

## 3.13 Compass

Derived from `Device`.

```
Compass {
  MFVec3f    lookupTable    []    # interpolation
  SFBool     xAxis          TRUE  # compute x-axis
  SFBool     yAxis          TRUE  # compute y-axis
  SFBool     zAxis          TRUE  # compute z-axis
  SFFloat    resolution     -1
}
```

### 3.13.1 Description

A `Compass` node can be used to model a 1, 2 or 3-axis digital compass (magnetic sensor). The `Compass` node returns a vector that indicates the direction of the *virtual north*. The *virtual north* is specified by the `northDirection` field in the `WorldInfo` node.

### 3.13.2 Field Summary

- `lookupTable`: This field optionally specifies a lookup table that can be used for mapping each vector component (between -1.0 and +1.0) to device specific output values. With the lookup table it is also possible to add noise and to define min and max output values. By default the lookup table is empty and therefore no mapping is applied.

- `xAxis, yAxis, zAxis`: Each of these boolean fields specifies if the computation should be enabled or disabled for the specified axis. If one of these fields is set to FALSE, then the corresponding vector element will not be computed and it will return *NaN* (Not a Number). For example if zAxis is FALSE, then calling wb_compass_get_values()[2] will always return *NaN*. The default is that all three axes are enabled (TRUE). Modifying these fields makes it possible to choose between a single, dual or a three-axis digital compass and to specify which axes will be used.

- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

### 3.13.3 Compass Functions

---

#### NAME

wb_compass_enable,
wb_compass_disable,
wb_compass_get_sampling_period,
wb_compass_get_values – *enable, disable and read the output values of the compass device*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/compass.h>

void wb_compass_enable (WbDeviceTag tag, int ms);

void wb_compass_disable (WbDeviceTag tag);

const double *wb_compass_get_values (WbDeviceTag tag);

int wb_compass_get_sampling_period (WbDeviceTag tag);
```

#### DESCRIPTION

The wb_compass_enable() function turns on the Compass measurement each ms milliseconds.

The wb_compass_disable() function turns off the Compass device.

The wb_compass_get_sampling_period() function returns the period given into the wb_compass_enable() function, or 0 if the device is disabled.

The wb_compass_get_values() function returns the current Compass measurement. The returned vector indicates the direction of the *virtual north* in the coordinate system of the Compass device. Here is the internal algorithm of wb_compass_get_values() in pseudo-code:


```
float[3] wb_compass_get_values() {
  float[3] n = getGlobalNorthDirection();
  n = rotateToCompassOrientation3D(n);
  n = normalizeVector3D(n);
  n[0] = applyLookupTable(n[0]);
  n[1] = applyLookupTable(n[1]);
  n[2] = applyLookupTable(n[2]);
  if (xAxis == FALSE) n[0] = 0.0;
  if (yAxis == FALSE) n[1] = 0.0;
  if (zAxis == FALSE) n[2] = 0.0;
  return n;
}
```

If the lookupTable is empty and all three xAxis, yAxis and zAxis fields are TRUE then the length of the returned vector is 1.0.

The values are returned as a 3D-vector, therefore only the indices 0, 1, and 2 are valid for accessing the vector. Let's look at one example. In Webots global coordinates system, the *xz*-plane represents the horizontal floor and the *y*-axis indicates the elevation. The default value of the northDirection field is [ 1 0 0 ] and therefore the north direction is horizontal and aligned with the x-axis. Now if the Compass node is in *upright* position, meaning that its y-axis is aligned with the global y-axis, then the bearing angle in degrees can be computed as follows:


**language: C**

```c
double get_bearing_in_degrees() {
const double *north = wb_compass_get_values(tag);
double rad = atan2(north[0], north[2]);
double bearing = (rad - 1.5708) / M_PI * 180.0;
if (bearing < 0.0)
  bearing = bearing + 360.0;
return bearing;
}
```


**language: C, C++**

*The returned vector is a pointer to the internal values managed by the* `Compass` *node, therefore it is illegal to free this pointer. Furthermore, note that the pointed values are only valid until the next call to* `wb_robot_step()` *or* `Robot::step()`*. If these values are needed for a longer period they must be copied.*


**language: Python**

`getValues()` *returns the vector as a list containing three floats.*

## 3.14 Cone

```
Cone {
  SFFloat    bottomRadius    1    # (-inf,inf)
  SFFloat    height          2    # (-inf,inf)
  SFBool     side            TRUE
  SFBool     bottom          TRUE
  SFInt32    subdivision     12   # (3,inf)
}
```

The `Cone` node specifies a cone which is centered in the local coordinate system and whose central axis is aligned with the local *y*-axis. The `bottomRadius` field specifies the radius of the cone's base, and the `height` field specifies the height of the cone from the center of the base to the apex. By default, the cone has a radius of 1 meter at the bottom and a height of 2 meters, with its apex at y = height/2 and its bottom at y = -height/2. See figure 3.4.

If both `bottomRadius` and `height` are positive, the outside faces of the cone are displayed while if they are negative, the inside faces are displayed.

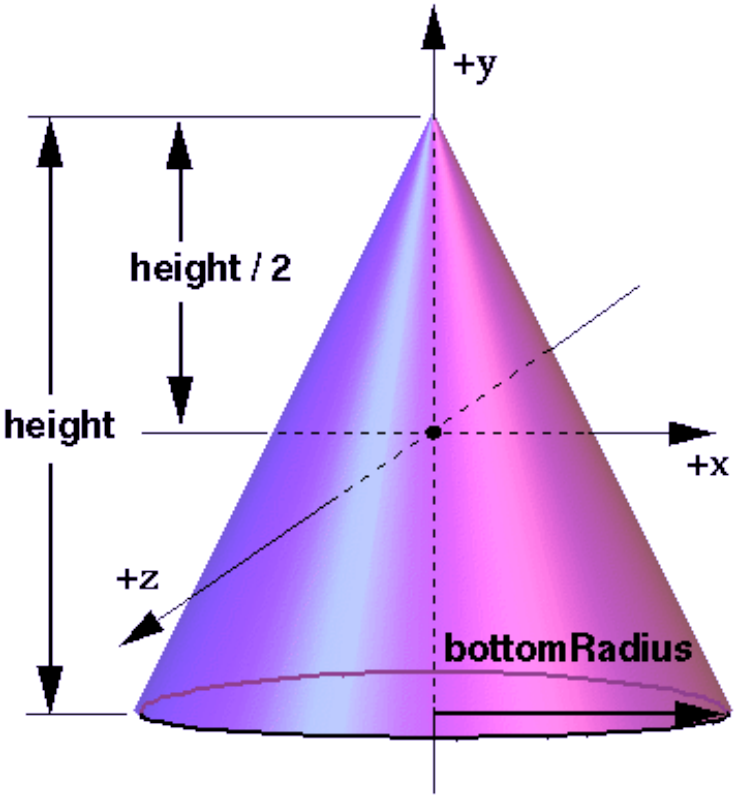


Figure 3.4: The Cone node

The `side` field specifies whether the sides of the cone are created, and the `bottom` field specifies whether the bottom cap of the cone is created. A value of `TRUE` specifies that this part of the cone exists, while a value of `FALSE` specifies that this part does not exist.

The `subdivision` field defines the number of polygons used to represent the cone and so its resolution. More precisely, it corresponds to the number of lines used to represent the bottom of the cone.

When a texture is applied to the sides of the cone, the texture wraps counterclockwise (from above) starting at the back of the cone. The texture has a vertical seam at the back in the *yz* plane, from the apex (0, `height`/2, 0) to the point (0, 0, -r). For the bottom cap, a circle is cut out of the unit texture square centered at (0, -`height`/2, 0) with dimensions (2 * `bottomRadius`) by (2 * `bottomRadius`). The bottom cap texture appears right side up when the top of the cone is rotated towards the -Z axis. `TextureTransform` affects the texture coordinates of the Cone.

`Cone` geometries cannot be used as primitives for collision detection in bounding objects.

## 3.15 Connector

Derived from `Device`.

```
Connector {
  SFString   type                "symmetric"
  SFBool     isLocked            FALSE
  SFBool     autoLock            FALSE
  SFBool     unilateralLock      TRUE
  SFBool     unilateralUnlock    TRUE
  SFFloat    distanceTolerance   0.01  # [0,inf)
  SFFloat    axisTolerance       0.2   # [0,pi)
  SFFloat    rotationTolerance   0.2   # [0,pi)
  SFInt32    numberOfRotations   4
  SFBool     snap                TRUE
  SFFloat    tensileStrength     -1
  SFFloat    shearStrength       -1
}
```

### 3.15.1 Description

Connector nodes are used to simulate mechanical docking systems, or any other type of device, that can dynamically create a physical link (or *connection*) with another device of the same type.

Connector nodes can only connect to other Connector nodes. At any time, each connection involves exactly two Connector nodes (peer to peer). The physical connection between two Connector nodes can be created and destroyed at run time by the robot's controller. The primary idea of Connector nodes is to enable the dynamic reconfiguration of modular robots, but more generally, Connector nodes can be used in any situation where robots need to be attached to other robots.

Connector nodes were designed to simulate various types of docking hardware:

- Mechanical links held in place by a latch
- Gripping mechanisms
- Magnetic links between permanent magnets (or electromagnets)
- Pneumatic suction systems, etc.

Connectors can be classified into two types, independent of the actual hardware system:

*Symmetric connectors*, where the two connecting faces are mechanically (and electrically) equivalent. In such cases both connectors are active.

*Asymmetric connectors*, where the two connecting interfaces are mechanically different. In asymmetric systems there is usually one active and one passive connector.

The detection of the presence of a peer Connector is based on simple distance and angle measurements, and therefore the Connector nodes are a computationally inexpensive way of simulating docking mechanisms.

### 3.15.2 Field Summary

- model: specifies the Connector's model. Two Connector nodes can connect only if their model strings are identical.
- type: specifies the connector's type, this must be one of: "symmetric", "active", or "passive". A "symmetric" connector can only lock to (and unlock from) another "symmetric" connector. An "active" connector can only lock to (and unlock from) a "passive" connector. A "passive" connector cannot lock or unlock.
- isLocked: represents the locking state of the Connector. The locking state can be changed through the API functions wb_connector_lock() and wb_connector_unlock(). The *locking state* means the current state of the locking hardware, it does not indicates whether or not an actual physical link exists between two connectors. For example, according to the hardware type, isLocked can mean that a mechanical latch or a gripper is closed, that electro-magnets are activated, that permanent magnets were moved to an attraction state, or that a suction pump was activated, etc. But the actual physical link exists only if wb_connector_lock() was called when a compatible peer was present (or if the Connector was auto-locked).
- autoLock: specifies if auto-locking is enabled or disabled. Auto-locking allows a connector to automatically lock when a compatible peer becomes present. In order to successfully auto-lock, both the autoLock and the isLocked fields must be TRUE when the peer becomes present, this means that wb_connector_lock() must have been invoked earlier. The general idea of autoLock is to allow passive locking. Many spring mounted latching mechanisms or magnetic systems passively lock their peer.
- unilateralLock: indicate that locking one peer only is sufficient to create a physical link. This field must be set to FALSE for systems that require both sides to be in the locked state in order to create a physical link. For example, symmetric connectors using rotating magnets fall into this category, because both connectors must be simultaneously in a magnetic "attraction" state in order to create a link. Note that this field should always be TRUE for "active" Connectors, otherwise locking would be impossible for them.
- unilateralUnlock: indicates that unlocking one peer only is sufficient to break the physical link. This field must be set to FALSE for systems that require both sides to be in an unlocked state in order to break the physical link. For example, connectors often use bilateral latching mechanisms, and each side must release its own latch in order for the link to break. Note that this field should always be TRUE for "active" Connectors, otherwise unlocking would be impossible for them.

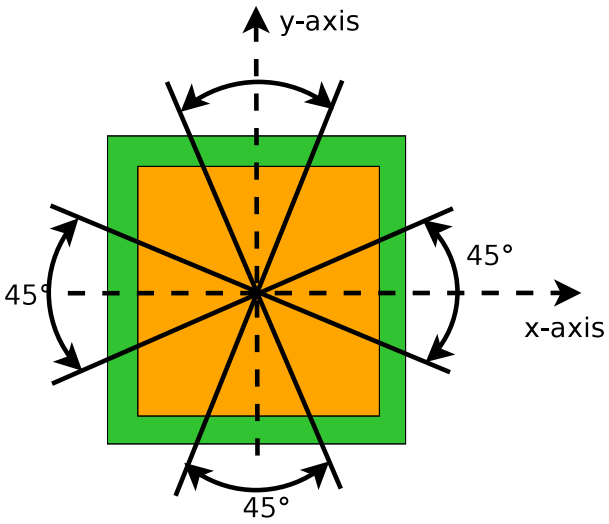


Figure 3.5: Example of rotational alignment (numberOfRotations=4 and rotationalTolerance=22.5 deg)

- `distanceTolerance`: the maximum distance [in meters] between two `Connectors` which still allows them to lock successfully. The distance is measured between the origins of the coordinate systems of the connectors.
- `axisTolerance`: the maximum angle [in radians] between the $z$-axes of two `Connectors` at which they may successfully lock. Two `Connector` nodes can lock when their $z$-axes are parallel (within tolerance), but pointed in opposite directions.
- `rotationTolerance`: the tolerated angle difference with respect to each of the allowed docking rotations (see figure 3.5).
- `numberOfRotations`: specifies how many different docking rotations are allowed in a full 360 degree rotation around the `Connector`'s $z$-axis. For example, modular robots' connectors are often 1-, 2- or 4-way dockable depending on mechanical and electrical interfaces. As illustrated in figure 3.5, if `numberOfRotations` is 4 then there will be 4 different docking positions (one every 90 degrees). If you don't wish to check the rotational alignment criterion this field should be set to zero.
- `snap`: when TRUE: the two connectors do automatically snap (align, adjust, etc.) when they become docked. The alignment is threefold: 1) the two bodies are rotated such that their z-axes become parallel (but pointed in opposite directions), 2) the two bodies are rotated such that their y-axes match one of the possible rotational docking position, 3) the two bodies are shifted towards each other such that the origin of their coordinate system match. Note that when the `numberOfRotations` field is 0, step 2 is omitted, and therefore the rotational alignment remains free. As a result of steps 1 and 3, the connector surfaces always become superimposed.

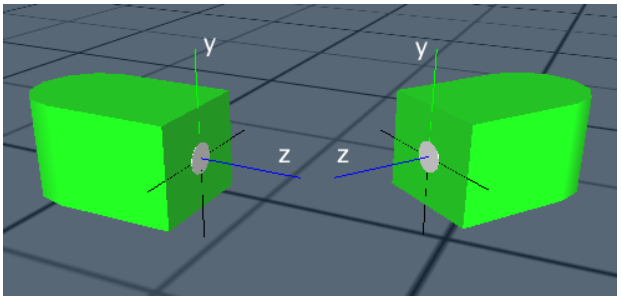


Figure 3.6: Connector axis system

- `tensileStrength`: maximum tensile force [in Newtons] that the docking mechanism can withstand before it breaks. This can be used to simulate the rupture of the docking mechanism. The tensile force corresponds to a force that pulls the two connectors apart (in the negative $z$-axes direction). When the tensile force exceeds the tensile strength, the link breaks. Note that if both connectors are locked, the effective tensile strength corresponds to the sum of both connectors' `tensileStrength` fields. The default value -1 indicates an infinitely strong docking mechanism that does not break no matter how much force is applied.

- `shearStrength`: indicates the maximum shear force [in Newtons] that the docking mechanism can withstand before it breaks. This can be used to simulate the rupture of the docking mechanism. The `shearStrength` field specifies the ability of two connectors to withstand a force that would makes them slide against each other in opposite directions (in the $xy$-plane). Note that if both connectors are locked, the effective shear strength corresponds to the sum of both connectors' `shearStrength` fields. The default value -1 indicates an infinitely strong docking mechanism that does not break no matter how much force is applied.

### 3.15.3 Connector Axis System

A `Connector`'s axis system is displayed by Webots when the corresponding robot is selected or when *Display Axes* is checked in Webots *Preferences*. The $z$-axis is drawn as a 5 cm blue line, the y-axis (a potential docking rotation) is drawn as a 5 cm red line, and each additional potential docking rotation is displayed as a 4 cm black line. The bounding objects and graphical objects of a `Connector` should normally be designed such that the docking surface corresponds exactly to $xy$-plane of the local coordinate system. Furthermore, the `Connector`'s z-axis should be perpendicular to the docking surface and point outward from the robot body. Finally, the bounding objects should allow the superposition of the origin of the coordinate systems. If these design criteria are not met, the `Connector` nodes will not work properly and may be unable to connect.


*To be functional, a* `Connector` *node requires the presence of a* `Physics` *node in its parent node. But it is not necessary to add a* `Physics` *node to the* `Connector` *itself.*

### 3.15.4 Connector Functions

---

#### NAME

wb_connector_enable_presence,
wb_connector_disable_presence,
wb_connector_get_presence – *detect the presence of another connector*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/connector.h>

void wb_connector_enable_presence (WbDeviceTag tag, int ms);

void wb_connector_disable_presence (WbDeviceTag tag);

int wb_connector_get_presence (WbDeviceTag tag);
```

#### DESCRIPTION

The `wb_connector_enable_presence()` function starts querying the `Connector`'s *presence* (see definition below) state each `ms` milliseconds. The `wb_connector_disable_presence()` function stops querying the `Connector`'s *presence*. The `wb_connector_get_presence()` function returns the current *presence* state of this connector, it returns:

- 1: in case of the *presence* of a peer connector
- 0: in case of the absence of a peer connector
- -1: not applicable (if this connector is of "passive" type)

The *presence* state is defined as the correct positioning of a compatible peer `Connector`.

Two connectors are in position if they are axis-aligned, rotation-aligned and near enough. To be axis-aligned, the angle between the $z$-axes of the two connectors must be smaller than the `axisTolerance` field. To be rotation-aligned, the angle between the $y$-axis of both `Connectors` must be within `distanceTolerance` of one of the possible `numberOfRotations` subdivisions of 360 degrees. Two `Connectors` are near enough if the distance between them

(measured between the origins of the coordinate systems) is smaller than `distanceTolerance`.

Two `Connectors` are compatible if both types are "symmetric" or if one is "active" and the other is "passive". A further requirement for the compatibility is that the `model` fields of the connectors must be identical. The conditions for detecting presence can be summarized this way:

```
presence          := in_position AND compatible
compatible        := type_compatible AND model_compatible
type_compatible   := both connectors are "symmetric" OR one connector
                     is "active" AND the other one is "passive"
model_compatible  := both models strings are equal
in_position       := near_enough AND axis_aligned AND rotation_aligned
near_enough       := the distance between the connectors < tolerance
axis_aligned      := the angle between the z-axes < tolerance
rotation_aligned  := the n-ways rotational angle is within tolerance
```

---

## NAME

wb_connector_lock,
wb_connector_unlock – *create / destroy the physical connection between two connector nodes*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/connector.h>

void wb_connector_lock (WbDeviceTag tag);

void wb_connector_unlock (WbDeviceTag tag);
```

## DESCRIPTION

The `wb_connector_lock()` and `wb_connector_unlock()` functions can be used to set or unset the `Connector`'s locking state (`isLocked` field) and eventually create or destroy the physical connection between two `Connector` nodes.

If `wb_connector_lock()` is invoked while a peer connector is *present* (see the definition of *presence* above), a physical link will be created between the two connectors. If both the `isLocked` and `autoLock` fields are TRUE, then the physical link will be created automatically as soon as the peer's *presence* is detected. If `wb_connector_lock()` succeeds in creating the link, the two connected bodies will keep a constant distance and orientation with respect to each other from this moment on.

If `wb_connector_unlock()` is invoked while there is a physical link between two `Connectors`, the link will be destroyed, unless `unilateralUnlock` is FALSE and the peer connector is still in the `isLocked` state.

# 3.16 ContactProperties

```
ContactProperties {
  SFString   material1           "default"
  SFString   material2           "default"
  MFFloat    coulombFriction     1          # [0,inf)
  SFVec2f    frictionRotation    0 0
  SFFloat    bounce              0.5        # [0,1]
  SFFloat    bounceVelocity      0.01       # (m/s)
  MFFloat    forceDependentSlip  0
  SFFloat    softERP             0.2
  SFFloat    softCFM             0.001
}
```

## 3.16.1 Description

ContactProperties nodes define the contact properties to use in case of contact between Solid nodes (or any node derived from Solid). ContactProperties nodes are placed in the contactProperties field of the WorldInfo node. Each ContactProperties node specifies the name of two *materials* for which these ContactProperties are valid.

When two Solid nodes collide, a matching ContactProperties node is searched in the WorldInfo.contactProperties field. A ContactProperties node will match if its material1 and material2 fields correspond (in any order) to the the contactMaterial fields of the two colliding Solids. The values of the first matching ContactProperties are applied to the contact. If no matching node is found, default values are used. The default values are the same as those indicated above.


*In older Webots versions, contact properties used to be specified in* Physics *nodes. For compatibility reasons, contact properties specified like this are still functional in Webots, but they trigger deprecation warnings. To remove these warning you need to switch to the new scheme described in this page. This can be done in three steps: 1. Add* ContactProperties *nodes in* WorldInfo*, 2. Define the* contactMaterial *fields of* Solid *nodes, 3. Reset the values of* coulombFriction, bounce, bounceVelocity *and* forceDependentSlip *in the* Physics *nodes.*

## 3.16.2 Field Summary

- The material1 and material2 fields specify the two *contact materials* to which this ContactProperties node must be applied. The values in this fields should match the contactMaterial fields of Solid nodes in the simulation. The values in material1 and material2 are exchangeable.

- The `coulombFriction` are the Coulomb friction coefficients. They must be in the range 0 to infinity (use -1 for infinity). 0 results in a frictionless contact, and infinity results in a contact that never slips. This field can held one to four values. If it has only one value, the friction is fully symmetric. With two values, the friction is fully asymmetric using the same coefficients for both solids. With three values, the first solid (corresponding to `material1`) uses asymmetric coefficients (first two values) and the other solid (corresponding to `material2`) uses a symmetric coefficient (last value). Finally, with four values, both solids use asymmetric coefficients, first two for the first solid and last two for the second solid. The two friction directions are defined for each faces of the geometric primitives and match with the U and V components used in the texture mapping. Only the `Box`, `Plane` and `Cylinder` primitives support asymmetric friction. If another primitive is used, only the first value will be used for symetric friction. WEBOTS_HOME/projects/sample/howto/worlds/asymmetric_friction1.wbt contains an example of fully asymmetric friction.

- The `frictionRotation` allows the user to rotate the friction directions used in case of asymmetric `coulombFriction` and/or asymmetric `forceDependentSlip`. By default, the directions are the same than the ones used for texture mapping (this can ease defining an asymmetric friction for a textured surface matching the rotation field of the corresponding TextureTransform node). WEBOTS_HOME/projects/sample/howto/worlds/asymmetric_friction2.wbt illustrates the use of this field.

- The `bounce` field is the coefficient of restitution (COR) between 0 and 1. The coefficient of restitution (COR), or *bounciness* of an object is a fractional value representing the ratio of speeds after and before an impact. An object with a COR of 1 collides elastically, while an object with a COR $< 1$ collides inelastically. For a COR = 0, the object effectively "stops" at the surface with which it collides, not bouncing at all. COR = (relative speed after collision) / (relative speed before collision).

- The `bounceVelocity` field represents the minimum incoming velocity necessary for bouncing. Solid objects with velocities below this threshold will have a `bounce` value set to 0.

- The `forceDependentSlip` field defines the *force dependent slip* (FDS) for friction, as explained in the ODE documentation: "FDS is an effect that causes the contacting surfaces to side past each other with a velocity that is proportional to the force that is being applied tangentially to that surface. Consider a contact point where the coefficient of friction mu is infinite. Normally, if a force f is applied to the two contacting surfaces, to try and get them to slide past each other, they will not move. However, if the FDS coefficient is set to a positive value k then the surfaces will slide past each other, building up to a steady velocity of k*f relative to each other. Note that this is quite different from normal frictional effects: the force does not cause a constant acceleration of the surfaces relative to each other - it causes a brief acceleration to achieve the steady velocity."

This field can held one to four values. If it has only one value, this coefficient is applied to both directions (force dependent slip is disabled if the value is 0). With two values, force dependent slip is fully asymmetric using the same coefficients for both solids (if one value is 0, force dependent slip is disabled in the corresponding direction). With three values, the first solid (corresponding to `material1`) uses asymmetric coefficients (first two values) and the other solid (corresponding to `material2`) uses a symmetric coefficient (last value). Finally, with four values, both solids use asymmetric coefficients, first two for the first solid and last two for the second solid. The friction directions and the supported geometric primitives are the same as the ones documented with the `coulombFriction` field.

- The `softERP` field defines the *Error Reduction Parameter* use by ODE to manage local contact joints. See `WorldInfo` for a description of the ERP concept.
- The `softCFM` field defines the soft *Constraint Force Mixing* use by ODE to manage local contacts joints. `WorldInfo` for a description of the CFM concept.


*The youBot robot is a good example of asymmetric coulombFriction and forceDependentSlip, it is located in WEBOTS_HOME/projects/robot/youbot/worlds/youbot.wbt.*

## 3.17 Coordinate

```
Coordinate {
  MFVec3f   point    []    # (-inf,inf)
}
```

This node defines a set of 3D coordinates to be used in the `coord` field of vertex-based `Geometry` nodes including `IndexedFaceSet` and `IndexedLineSet`.

## 3.18 Cylinder

```
Cylinder {
  SFBool    bottom        TRUE
  SFFloat   height        2      # (-inf,inf)
  SFFloat   radius        1      # (-inf,inf)
  SFBool    side          TRUE
  SFBool    top           TRUE
  SFInt32   subdivision   12     # (2,inf)
}
```

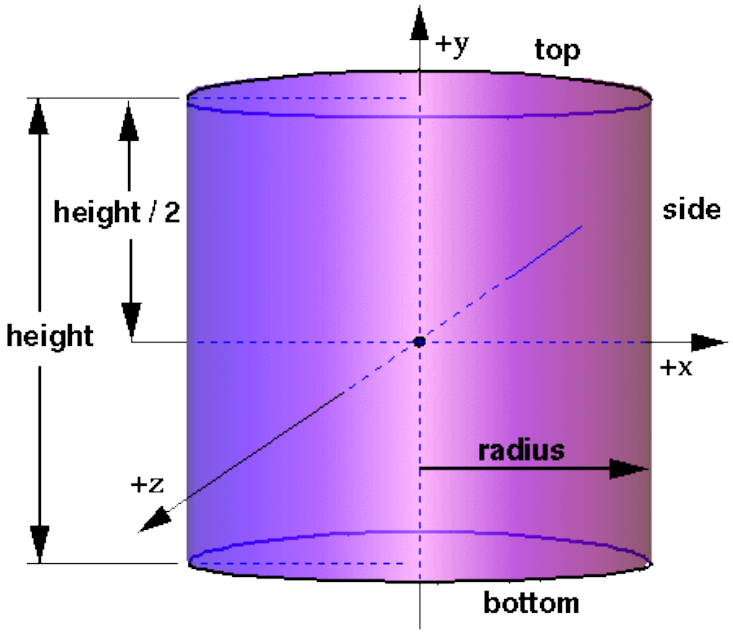


Figure 3.7: The Cylinder node

### 3.18.1 Description

The `Cylinder` node specifies a cylinder centered at (0,0,0) in the local coordinate system and with a central axis oriented along the local *y*-axis. By default, the cylinder spans -1 to +1 in all three dimensions. The `radius` field specifies the radius of the cylinder and the `height` field specifies the height of the cylinder along the central axis. See figure 3.7.

If both `height` and `radius` are positive, the outside faces of the cylinder are displayed while if they are negative, the inside faces are displayed.

The cylinder has three parts: the side, the top (y = +`height`/2) and the bottom (y = -`height`+/2). Each part has an associated `SFBool` field that indicates whether the part exists (`TRUE`) or does not exist (`FALSE`). Parts which do not exist are not rendered. However, all parts are used for collision detection, regardless of their associated `SFBool` field.

The `subdivision` field defines the number of polygons used to represent the cylinder and so its resolution. More precisely, it corresponds to the number of lines used to represent the bottom or the top of the cylinder.

When a texture is applied to a cylinder, it is applied differently to the sides, top, and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cylinder. The texture has a vertical seam at the back, intersecting the yz plane. For the top and bottom caps, a circle is cut out of the unit texture squares centered at (0, +/- `height`, 0) with dimensions 2*`radius` by 2*`radius`. The top texture appears right side up when the top of the cylinder is tilted toward the +*z* axis, and the bottom texture appears right side up when the top of the

cylinder is tilted toward the -*z* axis. TextureTransform affects the texture coordinates of the Cylinder.

## 3.19 Damping

```
Damping {
    SFFloat    linear         0.2      # [0,1]
    SFFloat    angular        0.2      # [0,1]
}
```

### 3.19.1 Description

A Damping node can be used to slow down a body (a Solid node with Physics). The speed of each body is reduced by the specified amount (between 0.0 and 1.0) every second. A value of 0.0 means "no slowing down" and value of 1.0 means a "complete stop", a value of 0.1 means that the speed should be decreased by 10 percent every second. A damped body will possibly come to rest and become disabled depending on the values specified in WorldInfo. Damping does not add any force in the simulation, it directly affects the velocity of the body. The damping effect is applied after all forces have been applied to the bodies. Damping can be used to reduce simulation instability.


*When several rigidly linked* Solid*s are merged (see* Physics*'s* solid merging *section) damping values of the aggregate body are averaged over the volumes of all* Solid *components. The volume of a* Solid *is the sum of the volumes of the geometries found in its* boundingObject*; overlaps are not handled.*

The linear field indicates the amount of damping that must be applied to the body's linear motion. The angular field indicates the amount of damping that must be applied to the body's angular motion. The linear damping can be used, e.g., to slow down a vehicule by simulating air or water friction. The angular damping can be used, e.g., to slow down the rotation of a rolling ball or the spin of a coin. Note that the damping is applied regardless of the shape of the object, so damping cannot be used to model complex fluid dynamics (use ImmersionProperties and Fluid nodes instead).

A Damping node can be specified in the defaultDamping field of the WorldInfo node; in this case it defines the default damping parameters that must be applied to every body in the simulation. A Damping node can be specified in the damping field of a Physics node; in this case it defines the damping parameters that must be applied to the Solid that contains the Physics node. The damping specified in a Physics node overrides the default damping.

## 3.20 Device

Abstract node, derived from `Solid`.

```
Device {
}
```

### 3.20.1 Description

This abstract node (not instanciable) represents a robot device (actuator and/or sensor).

### 3.20.2 Device Functions

---

#### NAME

wb_device_get_name – *convert WbDeviceTag to its corresponding device name*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/device.h>

const char *wb_device_get_name (WbDeviceTag tag);
```

#### DESCRIPTION

`wb_device_get_name()` convert the WbDeviceTag given as parameter (`tag`) to its corresponding name.

This function returns NULL if the WbDeviceTag does not match a valid device.

---

#### NAME

wb_device_get_node_type – *convert WbDeviceTag to its corresponding WbNodeType*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/device.h>

WbNodeType wb_device_get_node_type (WbDeviceTag tag);
```

#### DESCRIPTION

`wb_device_get_node_type()` convert the WbDeviceTag given as parameter (`tag`) to its corresponding WbNodeType (cf. the `Supervisor` API)

This function returns NULL if the WbDeviceTag does not match a valid device.

## 3.21 DifferentialWheels

Derived from `Robot`.

```
DifferentialWheels {
  SFFloat   motorConsumption    0       # [0,inf)
  SFFloat   axleLength          0.1     # (0,inf)
  SFFloat   wheelRadius         0.01    # (0,inf)
  SFFloat   maxSpeed            10      # (0,inf)
  SFFloat   maxAcceleration     10
  SFFloat   speedUnit           1
  SFFloat   slipNoise           0.1     # [0,inf)
  SFFloat   encoderNoise        -1
  SFFloat   encoderResolution   -1
  SFFloat   maxForce            0.3     # (0,inf)
}
```

### 3.21.1 Description

The `DifferentialWheels` node can be used as base node to build robots with two wheels and differential steering. Any other type of robot (legged, humanoid, vehicle, etc.) needs to use `Robot` as base node.

A `DifferentialWheels` robot will automatically take control of its wheels if they are placed in the `children` field. The wheels must be `Solid` nodes, and they must be named ”right wheel” and ”left wheel”. If the wheel objects are found, Webots will automatically make them rotate at the speed specified by the `wb_differential_wheels_set_speed()` function.

The origin of the robot coordinate system is the projection on the ground plane of the middle of the wheels' axle. The $x$ axis is the axis of the wheel axle, $y$ is the vertical axis and $z$ is the axis pointing towards the rear of the robot (the front of the robot has negative $z$ coordinates).

### 3.21.2 Field Summary

- `motorConsumption`: power consumption of the the motor in Watts.

- `axleLength`: distance between the two wheels (in meters). This field must be specified for "kinematics" based robot models. It will be ignored by "physics" based models.

- `wheelRadius`: radius of the wheels (in meters). Both wheels must have the same radius. This field must be specified for "kinematics" based robot models. It will be ignored by "physics" based models.

- `maxSpeed`: maximum speed of the wheels, expressed in *rad/s*.

- `maxAcceleration`: maximum acceleration of the wheels, expressed in *rad/s*$^2$. It is used only in "kinematics" mode.

- `speedUnit`: defines the unit used in the `wb_differential_wheels_set_speed()` function, expressed in *rad/s*.

- `slipNoise`: slip noise added to each move expressed in percent. If the value is 0.1, a noise component of +/- 10 percent is added to the command for each simulation step. The noise is, of course, different for each wheel. The noise has a uniform distribution, also known as as "white noise."

- `encoderNoise`: white noise added to the incremental encoders. If the value is -1, the encoders are not simulated. If the value is 0, encoders are simulated without noise. Otherwise a cumulative uniform noise is added to encoder values. At every simulation step, an increase value is computed for each encoder. Then, a random uniform noise is applied to this increase value before it is added to the encoder value. This random noise is computed in the same way as the slip noise (see above). When the robot encounters an obstacle, and if no physics simulation is used, the robot wheels do not slip, hence the encoder values are not incremented. This is very useful to detect that a robot has hit an obstacle. For each wheel, the angular velocity is affected by the `slipNoise` field. The angular speed is used to compute the rotation of the wheel for a basic time step (by default 32 ms). The wheel is actually rotated by this amount. This amount is then affected by the `encoderNoise` (if any). This means that a noise is added to the amount of rotation in a similar way as with the `slipNoise`. Finally, this amount is multiplied by the `encoderResolution` (see below) and used to increment the encoder value, which can be read by the controller program.

- `encoderResolution`: defines the number of encoder increments per radian of the wheel. An `encoderResolution` of *100* will make the encoders increment their value by (approximately) 628 each time the wheel makes a complete revolution. The -1 default value means that the encoder functionality is disabled as with `encoderNoise`.

- `maxForce`: defines the maximum torque used by the robot to rotate each wheel in a "physics" based simulation. It corresponds to the `dParamFMax` parameter of an ODE hinge joint. It is ignored in "kinematics" based simulations.

### 3.21.3 Simulation Modes

The DifferentialWheels's motion can be computed by different algorithms: "physics", "kinematics" or "Fast2D" depending on the structure of the world.

#### Physics mode

A DifferentialWheels is simulated in "physics" mode if it contains Physics nodes in its body and wheels. In this mode, the simulation is carried out by the ODE physics engine, and the robot's motion is caused by the friction forces generated by the contact of the wheels with the floor. The wheels can have any arbitrary shape (usually a cylinder), but their contact with the floor is necessary for the robot's motion. In "physics" mode the inertia, weight, etc. of the robot and wheels is simulated, so for example the robot will fall if you drop it. The friction is simulated with the Coulomb friction model, so a DifferentialWheels robot would slip on a wall with some friction coefficient that you can tune in the Physics nodes. The "physics" mode is the most realistic but also the slowest simulation mode.

#### Kinematics mode

When a DifferentialWheels does not have Physics nodes then it is simulated in "kinematics" mode. In the "kinematics" mode the robot's motion is calculated according to 2D kinematics algorithms and the collision detection is calculated with 3D algorithms. Friction is not simulated, so a DifferentialWheels does not actually require the contact of the wheels with the floor to move. Instead, its motion is controlled by a 2D kinematics algorithm using the axleLength, wheelRadius and maxAcceleration fields. Because friction is not simulated the DifferentialWheels will not slide on a wall or on another robot. The simulation will rather look as if obstacles (walls, robots, etc.) are very rough or harsh. However the robots can normally avoid to become blocked by changing direction, rotating the wheels backwards, etc. Unlike the "physics" mode, in the "kinematics" mode the gravity and other forces are not simulated therefore a DifferentialWheels robot will keep its initial elevation throughout the simulation.

#### Fast2D (Enki) mode

This mode is enabled when the string "enki" is specified in the WorldInfo.fast2d field. The "Fast2D" mode is implemented in a user-modifiable plugin which code can be found in this directory: webots/resources/projects/plugins/fast2d/enki. This is another implementation of 2D kinematics in which gravity, and other forces are also ignored simulated. However "Fast2D" mode the friction is simulated so a robot will smoothly slide over an obstacle or another robot. The "Fast2D" mode may be faster than "kinematics" in configurations where there are multiple DifferentialWheels with multiple DistanceSensors with multiple

| | Physics mode | Kinematics mode | Fast2D (Enki) mode |
|---|---|---|---|
| Motion triggered by | Wheels friction | 2d Webots kinematics | 2d Enki kinematics |
| Friction simulation | Yes, Coulomb model | No | Yes, Enki model |
| Inertia/Weight/Forces | Yes | No | No |
| Collision detection | 3D (ODE) | 3D (ODE) | 2D (Enki) |
| wheelRadius field | Ignored | Ignored | Used |
| axleLength field | Ignored | Ignored | Used |
| maxAcceleration field | Ignored | Ignored | Used |
| maxForce field | Used | Ignored | Ignored |
| Sensor rays shape | 3d cone | 3d cone | 2d fan |
| RGB sensitive | Yes | Yes | No |

Table 3.2: DifferentialWheels simulation modes

rays. However the "Fast2D" mode has severe limitations on the structure of the world and robots that it can simulate. More information on the "Fast2D" mode can be found here.

### 3.21.4 DifferentialWheels Functions

---

#### NAME

wb_differential_wheels_set_speed – *control the speed of the robot*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/differential_wheels.h>
void wb_differential_wheels_set_speed (double left, double right);
double wb_differential_wheels_get_left_speed ();
double wb_differential_wheels_get_right_speed ();
```

#### DESCRIPTION

The `wb_differential_wheels_set_speed` function allows the user to specify a speed for the `DifferentialWheels` robot. This speed will be sent to the motors of the robot at the beginning of the next simulation step. The speed unit is defined by the `speedUnit` field of the `DifferentialWheels` node. The default value is 1 radians per seconds. Hence a speed value of 2 will make the wheel rotate at a speed of 2 radians per seconds. The linear speed of the robot can then be computed from the angular speed of each wheel, the wheel radius and the noise added. Both the wheel radius and the noise are documented in the `DifferentialWheels` node.

The wb_differential_wheels_get_left_speed and wb_differential_wheels_get_right_speed functions allow to retrieve the last speed commands given as argument of the wb_differential_wheels_set_speed function.

---

## NAME

wb_differential_wheels_enable_encoders,
wb_differential_wheels_disable_encoders,
wb_differential_wheels_get_encoders_sampling_period – *enable or disable the incremental encoders of the robot wheels*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/differential_wheels.h>

void wb_differential_wheels_enable_encoders (int ms);

void wb_differential_wheels_disable_encoders ();

int wb_differential_wheels_get_encoders_sampling_period (WbDeviceTag tag);
```

## DESCRIPTION

These functions allow the user to enable or disable the incremental wheel encoders for both wheels of the DifferentialWheels robot. Incremental encoders are counters that increment each time a wheel turns. The amount added to an incremental encoder is computed from the angle the wheel rotated and from the encoderResolution parameter of the DifferentialWheels node. Hence, if the encoderResolution is 100 and the wheel made a whole revolution, the corresponding encoder will have its value incremented by about 628. Please note that in a kinematic simulation (with no Physics node set) when a DifferentialWheels robot encounters an obstacle while trying to move forward, the wheels of the robot do not slip, hence the encoder values are not increased. This is very useful to detect that the robot has hit an obstacle. On the contrary, in a physics simulation (when the DifferentialWheels node and its children contain appropriate Physics nodes), the wheels may slip depending on their friction parameters and the force of the motors (maxForce field of the DifferentialWheels node). If a wheel slips, then its encoder values are modified according to its actual rotation, even though the robot doesn't move.

The wb_differential_wheels_get_encoders_sampling_period() function returns the period given into the wb_differential_wheels_enable_encoders() function, or 0 if the device is disabled.

---

### NAME

wb_differential_wheels_get_left_encoder,
wb_differential_wheels_get_right_encoder,
wb_differential_wheels_set_encoders – *read or set the encoders of the robot wheels*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/differential_wheels.h>

double wb_differential_wheels_get_left_encoder ();

double wb_differential_wheels_get_right_encoder ();

void wb_differential_wheels_set_encoders (double left, double right);
```

### DESCRIPTION

These functions are used to read or set the values of the left and right encoders. The encoders must be enabled with `wb_differential_wheels_enable_encoders()`, so that the functions can read valid data. Additionally, the `encoderNoise` of the corresponding `DifferentialWheels` node should be positive. Setting the encoders' values will not make the wheels rotate to reach the specified value; instead, it will simply reset the encoders with the specified value.

---

### NAME

wb_differential_wheels_get_max_speed – *get the value of the maxSpeed field*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/differential_wheels.h>

double wb_differential_wheels_get_max_speed ();
```

### DESCRIPTION

The `wb_differential_wheels_get_max_speed` function allows the user to get the value of the `maxSpeed` field of the `DifferentialWheels` node.

---

### NAME

wb_differential_wheels_get_speed_unit – *get the value of the speedUnit field*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/differential_wheels.h>

double wb_differential_wheels_get_speed_unit ();
```

#### DESCRIPTION

The `wb_differential_wheels_get_speed_unit` function allows the user to get the value of the `speedUnit` field of the `DifferentialWheels` node.

## 3.22 DirectionalLight

Derived from `Light`.

```
DirectionalLight {
  SFVec3f    direction    0 0 -1    # (-inf,inf)
}
```

### 3.22.1 Description

The `DirectionalLight` node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector. Unlike `PointLight`, rays cast by `DirectionalLight` nodes do not attenuate with distance.

### 3.22.2 Field Summary

- The `direction` field specifies the direction vector of the illumination emanating from the light source in the global coordinate system. Light is emitted along parallel rays from an infinite distance away. The `direction` field is taken into account when computing the quantity of light received by a `LightSensor`.

## 3.23 Display

Derived from `Device`.

```
Display {
  SFInt32    width          64
  SFInt32    height         64
  SFVec2f    windowPosition 0 0
  SFFloat    pixelSize      1.0
}
```

### 3.23.1 Description

The Display node allows to handle a 2D pixel array using simple API functions, and render it into a 2D overlay on the 3D view, into a 2D texture of any Shape node, or both. It can model an embedded screen or it can display any graphical information such as graphs, text, robot trajectory, filtered camera images and so on.

If the first child of the Display node is or contains (recursive search if the first node is a Group) a Shape node having a ImageTexture, then the internal texture of the(se) ImageTexture node(s) is replaced by the texture of the Display.

### 3.23.2 Field Summary

- width: width of the display in pixels
- height: height of the display in pixels
- windowPosition: position in the 3D window where the Display image will be displayed. The X and Y values for this position are floating point values between 0.0 and 1.0. They specify the position of the center of the image, relatively to the top left corner of the 3D window. This position will scale whenever the 3D window is resized. Also, the user can drag and drop this display image in the 3D window using the mouse. This will affect the X and Y position values.
- pixelSize: scale factor for the Display image rendered in the 3D window (see the windowPosition description). Setting a pixelSize value higher than 1 is useful to better see each individual pixel of the image. Setting it to 0 simply turns off the display of the camera image.

### 3.23.3 Coordinates system

Internally, the Display image is stored in a 2D pixel array. The RGBA value (4x8 bits) of a pixel is dislayed in the status bar (the bar at the bottom of the console window) when the mouse hovers over the pixel in the Display. The 2D array has a fixed size defined by the width and height fields. The (0,0) coordinate corresponds to the top left pixel, while the (width-1,height-1) coordinate corresponds to the bottom right pixel.

### 3.23.4 Command stack

Each function call of the Display device API (except for wb_display_get_width() and wb_display_get_height()) is storing a specific command into an internal stack. This command stack is sent to Webots during the next call of the wb_robot_step() function, using a

FIFO scheme (First In, First Out), so that commands are executed in the same order as the corresponding function calls.

### 3.23.5 Context

The `Display` device has among other things two kinds of functions; the contextual ones which allow to set the current state of the display, and the drawing ones which allow to draw specific primitives. The behavior of the drawing functions depends on the display context. For example, in order to draw two red lines, the `wb_display_set_color` contextual function must be called for setting the display's internal color to red before calling twice the `wb_display_draw_line` drawing function to draw the two lines.

### 3.23.6 Display Functions

---

#### NAME

wb_display_get_width,
wb_display_get_height – *get the size of the display*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/display.h>

int wb_display_get_width (WbDeviceTag tag);

int wb_display_get_height (WbDeviceTag tag);
```

#### DESCRIPTION

These functions return respectively the values of the `width` and `height` fields.

---

#### NAME

wb_display_set_color,
wb_display_set_alpha,
wb_display_set_opacity – *set the drawing properties of the display*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/display.h>
```

$$Cn = (1 - opacity) * Co + opacity * Cn$$

Figure 3.8: Blending formula used to compute the new the color channels (Cn) of a pixel from the old color channels (Co) of the background pixel and from the opacity.

```
void wb_display_set_color (WbDeviceTag tag, int color);

void wb_display_set_alpha (WbDeviceTag tag, double alpha);

void wb_display_set_opacity (WbDeviceTag tag, double opacity);
```

## DESCRIPTION

These three functions define the context in which the subsequent drawing commands (see draw primitive functions) will be applied.

`wb_display_set_color()` defines the color for the subsequent drawing commands. It is expressed as a 3 bytes RGB integer, the most significant byte (leftmost byte in hexadecimal representation) represents the red component, the second most significant byte represents the green component and the third byte represents the blue component. For example, `0xFF00FF` (a mix of the red and blue components) represents the magenta color. Before the first call to `wb_display_set_color()`, the default color is white (`0xFFFFFF`).

`wb_display_set_alpha()` defines the alpha channel for the subsequent drawing commands. This function should be used only with special displays that can be transparent or semi-transparent (for which one can see through the display). The alpha channel defines the opacity of a pixel of the display. It is expressed as a floating point value between 0.0 and 1.0 representing respectively fully transparent and fully opaque. Intermediate values correspond to semi-transparent levels. Before the first call to `wb_display_set_alpha()`, the default value for alpha is 1 (opaque).

`wb_display_set_opacity()` defines with which opacity the new pixels will replace the old ones for the following drawing instructions. It is expressed as a floating point value between 0.0 and 1.0; while 0 means that the new pixel has no effect over the old one and 1 means that the new pixel replaces entirely the old one. Only the color channel is affected by the `opacity` according to the figure 3.8 formula.


**language: Matlab**
*In the Matlab version of* `wb_display_set_color()` *the* `color` *argument must be a vector containing the three RGB components:* `[RED GREEN BLUE]`. *Each component must be a value between 0.0 and 1.0. For example the vector* `[1 0 1]` *represents the magenta color.*

## NAME

wb_display_draw_pixel,
wb_display_draw_line,
wb_display_draw_rectangle,
wb_display_draw_oval,
wb_display_draw_polygon,
wb_display_draw_text,
wb_display_fill_rectangle,
wb_display_fill_oval,
wb_display_fill_polygon – *draw a graphic primitive on the display*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/display.h>

void wb_display_draw_pixel (WbDeviceTag tag, int x, int y);

void wb_display_draw_line (WbDeviceTag tag, int x1, int y1, int x2, int y2);

void wb_display_draw_rectangle (WbDeviceTag tag, int x, int y, int width,
int height);

void wb_display_draw_oval (WbDeviceTag tag, int cx, int cy, int a, int b);

void wb_display_draw_polygon (WbDeviceTag tag, const int *x, const int *y,
int size);

void wb_display_draw_text (WbDeviceTag tag, const char *txt, int x, int y);

void wb_display_fill_rectangle (WbDeviceTag tag, int x, int y, int width,
int height);

void wb_display_fill_oval (WbDeviceTag tag, int cx, int cy, int a, int b);

void wb_display_fill_polygon (WbDeviceTag tag, const int *x, const int *y,
int size);
```

## DESCRIPTION

These functions order the execution of a drawing primitive on the display. They depend on the context of the display as defined by the contextual functions (see `set context functions`).

`wb_display_draw_pixel()` draws a pixel at the (`x`,`y`) coordinate.

`wb_display_draw_line()` draws a line between the (`x1`,`y1`) and the (`x2`,`y2`) coordinates using the *Bresenham's line drawing algorithm*.

`wb_display_draw_rectangle()` draws the outline of a rectangle having a size of `width*height`. Its top left pixel is defined by the (`x`,`y`) coordinate.

`wb_display_draw_oval()` draws the outline of an oval. The center of the oval is specified by the (`cx`,`cy`) coordinate. The horizontal and vertical radius of the oval are specified by the (`a`,`b`) parameters. If `a` equals `b`, this function draws a circle.

`wb_display_draw_polygon()` draws the outline of a polygon having `size` vertices. The list of vertices must be defined into `px` and `py`. If the first pixel coordinates are not identical to the last ones, the loop is automatically closed. Here is an example :

```
const int px[] = {10,20,10, 0};
const int py[] = {0, 10,20,10};
wb_display_draw_polygon(display,px,py,4); // draw a diamond
```

`wb_display_draw_text()` draws an ASCII text from the (`x`,`y`) coordinate. The font used to display the characters has a size of 8x8 pixels. There is no extra space between characters.

`wb_display_fill_rectangle()` draws a rectangle having the same properties as the rectangle drawn by the `wb_display_draw_rectangle()` function except that it is filled instead of outlined.

`wb_display_fill_oval()` draws an oval having the same properties as the oval drawn by the `wb_display_draw_oval()` function except that it is filled instead of outlined.

`wb_display_fill_polygon()` draws a polygon having the same properties as the polygon drawn by the `wb_display_draw_polygon()` function except that it is filled instead of outlined.


**language: Java, Python, Matlab**
*The Java, Python and Matlab equivalent of* `wb_display_draw_polygon()` *and* `wb_display_fill_polygon()` *don't have a* `size` *argument because in these languages the size is determined directly from the* `x` *and* `y` *arguments.*

---

## NAME

wb_display_image_new,
wb_display_image_load,
wb_display_image_copy,
wb_display_image_paste,
wb_display_image_save,
wb_display_image_delete – *image manipulation functions*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/display.h>
```

```
WbImageRef wb_display_image_new (WbDeviceTag tag, int width, int height,
const void *data, int format);

WbImageRef wb_display_image_load (WbDeviceTag tag, const char *filename);

WbImageRef wb_display_image_copy (WbDeviceTag tag, int x, int y, int wi-
dth, int height);

void wb_display_image_paste (WbDeviceTag tag, WbImageRef ir, int x, int y);

void wb_display_image_save (WbDeviceTag tag, WbImageRef ir, const char *fi-
lename);

void wb_display_image_delete (WbDeviceTag tag, WbImageRef ir);
```

## DESCRIPTION

In addition to the main display image, each `Display` node also contains a list of clipboard images used for various image manipulations. This list is initially empty. The functions described below use a reference (corresponding to the `WbImageRef` data type) to refer to a specific image. Clipboard images can be created either with `wb_display_image_new()`, or `wb_display_image_load()`, or `wb_display_image_copy()`. They should be deleted with the `wb_display_image_delete()` function.when they are no more used. Finally, note that both the main display image and the clipboard images have an alpha channel.

`wb_display_image_new()` creates a new clipboard image, with the specified `with` and `height`, and loads the image `data` into it with respect to the defined image `format`. Three images format are supported: `WB_IMAGE_RGBA` which is similar to the image format returned by a `Camera` device and `WB_IMAGE_RGB` or `WB_IMAGE_ARGB`. `WB_IMAGE_RGBA` and `WB_IMAGE_ARGB` are including an alpha channel respectively after and before the color components.

`wb_display_image_load()` creates a new clipboard image, loads an image file into it and returns a reference to the new clipboard image. The image file is specified with the `filename` parameter (relatively to the controller directory). An image file can be in either PNG or JPEG format. Note that this function involves sending an image from the controller process to Webots, thus possibly affecting the overall simulation speed.

`wb_display_image_copy()` creates a new clipboard image and copies the specified sub-image from the main display image to the new clipboard image. It returns a reference to the new clipboard image containing the copied sub-image. The copied sub-image is defined by its top left coordinate (`x`,`y`) and its dimensions (`width`,`height`).

`wb_display_image_paste()` pastes a clipboard image referred to by the `ir` parameter to the main display image. The (`x`,`y`) coordinates define the top left point of the pasted image. The resulting pixels displayed in the main display image are computed using a blending operation (similar to the one depicted in the figure 3.8 formula but involving the alpha channels of the old and new pixels instead of the opacity).

`wb_display_image_save()` saves a clipboard image referred to by the `ir` parameter to a file. The file name is defined by the `filename` parameter (relatively to the controller directory). The image is saved in a file using either the PNG format or the JPEG format depending on the end of the `filename` parameter (respectively `.png` and `.jpg`). Note that this function involves sending an image from Webots to the controller process, thus possibly affecting the overall simulation speed.

`wb_display_image_delete()` releases the memory used by a clipboard image specified by the `ir` parameter. After this call the value of `ir` becomes invalid and should not be used any more. Using this function is recommended after a clipboard image is not needed any more.


**language: Java**
*The* `Display.imageNew()` *function can display the image returned by the* `Camera.getImage()` *function directly if the pixel format argument is set to ARGB.*

## 3.24 DistanceSensor

Derived from `Device`.

```
DistanceSensor {
  MFVec3f    lookupTable      [ 0 0 0, 0.1 1000 0 ]
  SFString   type             "generic"
  SFInt32    numberOfRays     1          # [1,inf)
  SFFloat    aperture         1.5708     # [0,2pi]
  SFFloat    gaussianWidth    1
  SFFloat    resolution      -1
}
```

### 3.24.1 Description

The `DistanceSensor` node can be used to model a generic sensor, an infra-red sensor, a sonar sensor, or a laser range-finder. This device simulation is performed by detecting the collisions between one or several sensor rays and objects in the environment. In case of generic, sonar and laser type the collision occurs with the bounding objects of `Solid` nodes, whereas infra-red rays collision detection uses the `Solid` nodes themselves.

The rays of the `DistanceSensor` nodes can be displayed by checking the menu **View** > **Optional Rendering** > **Show Distance Sensor Rays**. The red/green transition on the rays indicates the points of intersection with the bounding objects.

### 3.24.2 Field Summary

- `lookupTable`: a table used for specifying the desired response curve and noise of the device. This table indicates how the ray intersection distances measured by Webots must be mapped to response values returned by the function `wb_distance_sensor_get_value()`. The first column of the table specifies the input distances, the second column specifies the corresponding desired response values, and the third column indicates the desired standard deviation of the noise. The noise on the return value is computed according to a gaussian random number distribution whose range is calculated as a percent of the response value (two times the standard deviation is often referred to as the signal quality). Note that the input values of a lookup table must always be positive and sorted in increasing order.

  Let us consider a first example:

```
lookupTable [ 0     1000  0,
              0.1   1000  0.1,
              0.2    400  0.1,
              0.3     50  0.1,
              0.37    30  0   ]
```

  The above lookup table means that for a distance of 0 meters, the sensor will return a value of 1000 without noise (0); for a distance of 0.1 meter, the sensor will return 1000 with a noise of standard deviation of 10 percent (100); for a distance value of 0.2 meters, the sensor will return 400 with a standard deviation of 10 percent (40), etc. Distance values not directly specified in the lookup table will be linearly interpolated. This can be better understood in figure 3.9 below.

  A different graph is produced when the trend of the desired response value and the trend of the desired noise standard deviation have opposite sign. This is the case in the following example, where the response value increases with the input values but the noise decreases:

```
lookupTable [ 0     1023  0,
              0.02  1023  0.05,
              4        0  0.4  ]
```

  The resulting range of measured values is shown in figure 3.10.

- `type`: one of ”generic” (the default), ”infra-red”, ”sonar” or ”laser”. Sensors of type ”infra-red” are sensitive to the objects’ colors; light and red (RGB) obstacles have a higher response than dark and non-red obstacles (see below for more details).

  Sensors of type ”sonar” and ”laser” return the distance to the nearest object while ”generic” and ”infa-red” computes the average distance of all rays. Note however that sensors of type ”sonar” will return the sonar range for each ray whose angle of incidence is greater than $\pi/8$ radians (see below for more details).

  Sensors of type ”laser” can have only one ray and they have the particularity to draw a red spot at the point where this ray hits an obstacle. This red spot is visible on the camera

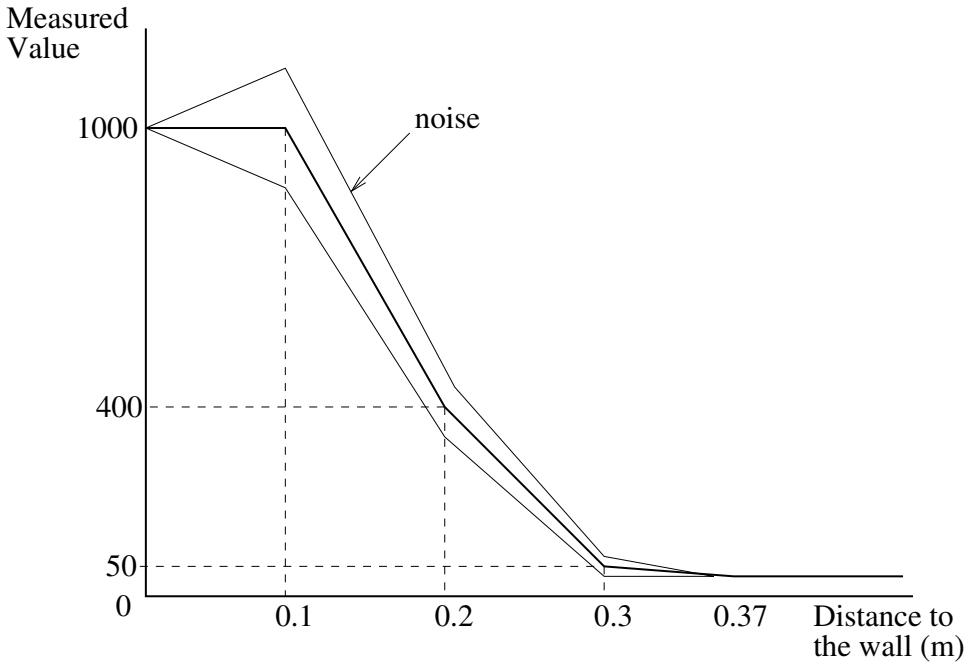


Figure 3.9: Sensor response versus obstacle distance

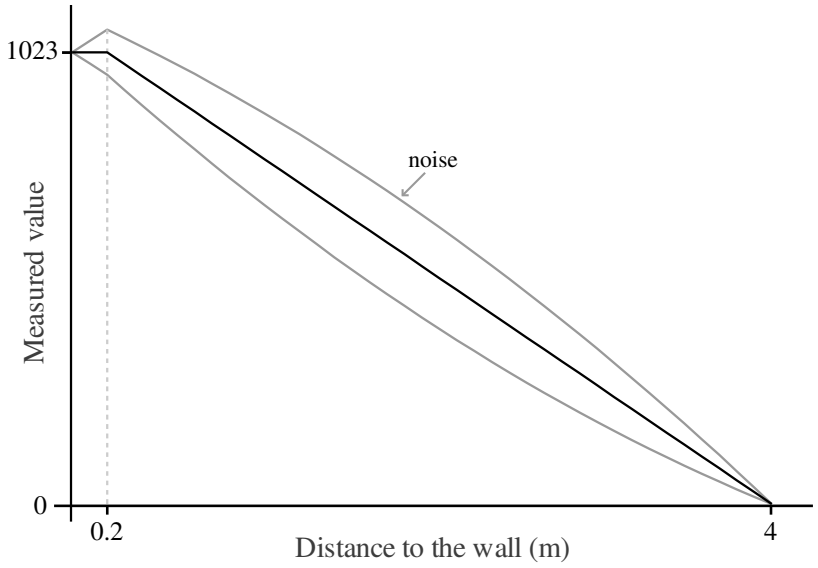


Figure 3.10: Sensor response versus obstacle distance with opposite response-noise increase

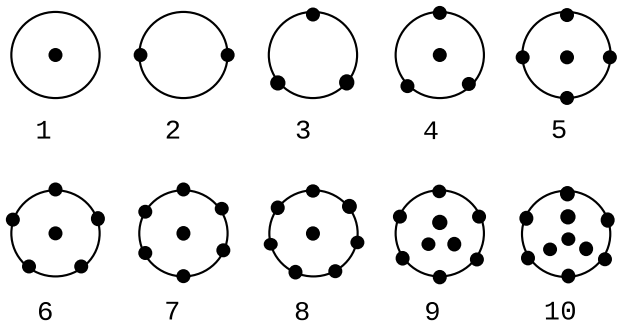


Figure 3.11: Predefined configurations for 1 through 10 sensor rays

$$v_i = exp\left(-\left(\frac{t_i}{a \cdot g}\right)^2\right)$$

$$w_i = \frac{v_i}{\sum_{j=1}^{n} v_j}$$

Figure 3.12: Weight distribution formulas

images. If the red spot disappears due to depth fighting, then it could help increasing the `lineScale` value in `WorldInfo` node that is used for computing its position offset.

- `numberOfRays`: number of rays cast by the sensor. The number of rays must be equal to, or greater than 1 for "infra-red" and "sonar" sensors. `numberOfRays` must be exactly 1 for "laser" sensors. If this number is larger than 1, then several rays are used and the sensor measurement value is computed from the weighted average of the individual rays' responses. By using multiple rays, a more accurate model of the physical infra-red or ultrasound sensor can be obtained. The sensor rays are distributed inside 3D-cones whose opening angles can be tuned through the `aperture` field. See figure 3.11 for the ray distributions from one to ten rays. The spacial distribution of the rays is as much as possible uniform and has a left/right symmetry. There is no upper limit on the number of rays; however, Webots' performance drops as the number of rays increases.

- `aperture`: sensor aperture angle or laser beam radius. For the "infra-red" and "sonar" sensor types, this field controls the opening angle (in radians) of the cone of rays when multiple rays are used. For the "laser" sensor type, this field specifies (in meters) the radius of the red spot drawn where the laser beam hits an obstacle.

- `gaussianWidth`: width of the Gaussian distribution of sensor ray weights (for "generic" and "infra-red" sensors). When averaging the sensor's response, the individual weight of each sensor ray is computed according to a Gaussian distribution as described in figure

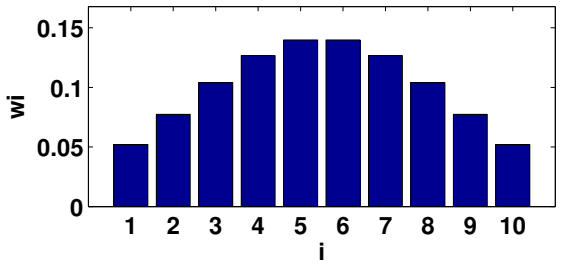


Figure 3.13: Example distribution for 10 rays using a Gaussian width of 0.5

3.12. where $w_i$ is the weight of the *i*th ray, $t_i$ is the angle between the *i*th ray and the sensor axis, *a* is the aperture angle of the sensor, *g* is the Gaussian width, and *n* is the number of rays. As depicted in figure 3.13, rays in the center of the sensor cone are given a greater weight than rays in the periphery. A wider or narrower distribution can be obtained by tuning the `gaussianWidth` field. An approximation of a flat distribution is obtained if a sufficiently large number is chosen for the `gaussianWidth`. This field is ignored for the "sonar" and "laser" DistanceSensor types.

- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).


*In* fast2d *mode, the sensor rays are arranged in* 2d-fans *instead of* 3D-cones *and the* `aperture` *field controls the opening angle of the fan. In* fast2d *mode, Gaussian averaging is also applied, and the* $t_i$ *parameter of the above formula corresponds to the 2D angle between the* ith *ray and the sensor axis.*

### 3.24.3 DistanceSensor types

This table summarizes the difference between the three types of DistanceSensor.

Two different methods are used for calculating the distance from an object. *Average* method computes the average of the distances measured by all the rays, whereas *Nearest* method uses the shortest distance measured by any of the rays.

### 3.24.4 Infra-Red Sensors

In the case of an "infra-red" sensor, the value returned by the lookup table is modified by a reflection factor depending on the color properties of the object hit by the sensor ray. The reflection

| type (field) | ”generic” | ”infra-red” | ”sonar” | ”laser” |
|---|---|---|---|---|
| numberOfRays (field) | $> 0$ | $> 0$ | $> 0$ | 1 |
| Distance calculation | Average | Average | Nearest | Nearest |
| gaussianWidth (field) | Used | Used | Ignored | Ignored |
| Sensitive to red objects | No | Yes | No | No |
| Draws a red spot | No | No | No | Yes |

Table 3.3: Summary of DistanceSensor types

factor is computed as follows: *f = 0.2 + 0.8 * red_level* where *red_level* is the level of red color of the object hit by the sensor ray. This level is evaluated combining the `diffuseColor` and `transparency` values of the object, the pixel value of the image texture and the paint color applied on the object with the `Pen` device. Then, the distance value computed by the simulator is divided by the reflection factor before the lookup table is used to compute the output value.


*Unlike other distance sensor rays, ”infra-red” rays can detect solid parts of the robot itself. It is thus important to ensure that no solid geometries interpose between the sensor and the area to inspect.*

### 3.24.5 Sonar Sensors

In the case of a ”sonar” sensor, the return value will be the last value entered in the lookup table, i.e. the value corresponding to sonar sensor’s range, if the angle of incidence is greater than 22.5 degrees ($\pi$/8 radians). In other words, sonar rays which lie outside the reflexion cone of aperture 45 degrees never return and thus are lost for distance computation (see figure 3.14).

### 3.24.6 Line Following Behavior

Some support for `DistanceSensor` nodes used for reading the red color level of a textured floor is implemented. This is useful to simulate line following behaviors. This feature is demonstrated in the `rover.wbt` example (see in the `projects/robots/mindstorms/worlds` directory of Webots). The ground texture must be placed in a `Plane`.

### 3.24.7 DistanceSensor Functions

---

#### NAME

wb_distance_sensor_enable,
wb_distance_sensor_disable,

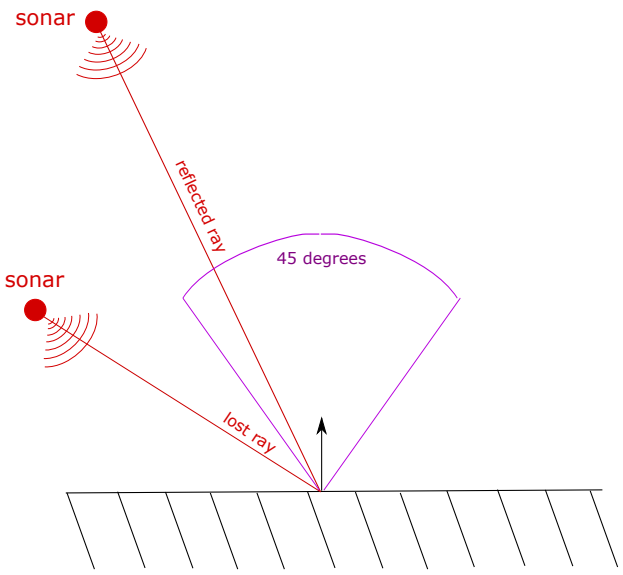


Figure 3.14: Sonar sensor

wb_distance_sensor_get_sampling_period,
wb_distance_sensor_get_value – *enable, disable and read distance sensor measurements*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/distance_sensor.h>

void wb_distance_sensor_enable (WbDeviceTag tag, int ms);

void wb_distance_sensor_disable (WbDeviceTag tag);

int wb_distance_sensor_get_sampling_period (WbDeviceTag tag);

double wb_distance_sensor_get_value (WbDeviceTag tag);
```

### DESCRIPTION

`wb_distance_sensor_enable()` allows the user to enable a distance sensor measurement each `ms` milliseconds.

`wb_distance_sensor_disable()` turns the distance sensor off, saving computation time.

The `wb_distance_sensor_get_sampling_period()` function returns the period given into the `wb_distance_sensor_enable()` function, or 0 if the device is disabled.

`wb_distance_sensor_get_value()` returns the last value measured by the specified distance sensor. This value is computed by the simulator according to the lookup table of the `DistanceSensor` node. Hence, the range of the return value is defined by this lookup table.

# 3.25 ElevationGrid

```
ElevationGrid {
  SFNode    color            NULL
  MFFloat   height           []      # (-inf,inf)
  SFBool    colorPerVertex   TRUE
  SFInt32   xDimension       0       # [0,inf)
  SFFloat   xSpacing         1       # (0,inf)
  SFInt32   zDimension       0       # [0,inf)
  SFFloat   zSpacing         1       # (0,inf)
  SFFloat   thickness        1       # [0,inf)
}
```

## 3.25.1 Description

The ElevationGrid node specifies a uniform rectangular grid of varying height in the *y=0* plane of the local coordinate system. The geometry is described by a scalar array of height values that specify the height of the surface above each point of the grid. The ElevationGrid node is the most appropriate to model an uneven terrain.

## 3.25.2 Field Summary

The xDimension and zDimension fields indicate the number of points in the grid height array in the *x* and *z* directions. Both xDimension and zDimension shall be greater than or equal to zero. If either the xDimension or the zDimension is less than two, the ElevationGrid contains no quadrilaterals. The vertex locations for the quadrilaterals are defined by the height field and the xSpacing and zSpacing fields:

- The height field is an xDimension by zDimension array of scalar values representing the height above the grid for each vertex.
- The xSpacing and zSpacing fields indicate the distance between vertices in the *x* and *z* directions respectively, and shall be greater than zero.

Thus, the vertex corresponding to the point P[i,j] on the grid is placed at:

```
P[i,j].x = xSpacing * i
P[i,j].y = height[ i + j * xDimension]
P[i,j].z = zSpacing * j

where 0 <= i < xDimension and 0 <= j < zDimension,
and P[0,0] is height[0] units above/below the origin of the local
coordinate system
```

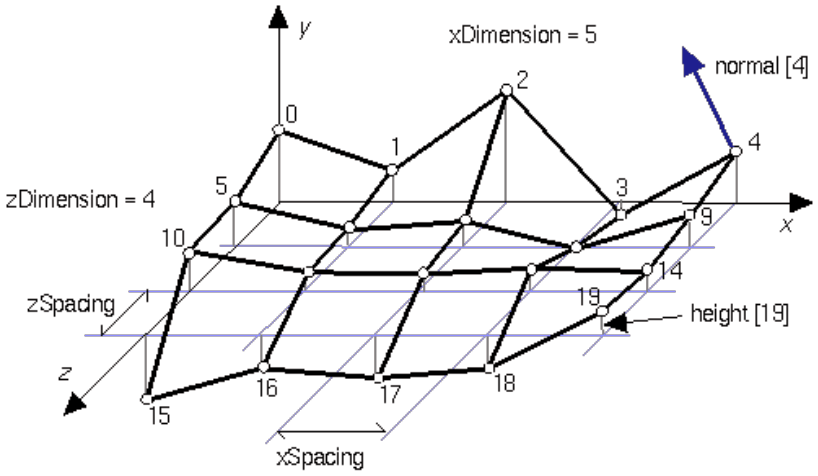


Figure 3.15: ElevationGrid node

The `color` field specifies per-vertex or per-quadrilateral colors for the `ElevationGrid` node depending on the value of `colorPerVertex`. If the `color` field is NULL, the `ElevationGrid` node is rendered with the overall attributes of the `Shape` node enclosing the `ElevationGrid` node. If only two colors are supplied, these two colors are used alternatively to display a checkerboard structure.

The `colorPerVertex` field determines whether colors specified in the color field are applied to each vertex or each quadrilateral of the `ElevationGrid` node. If `colorPerVertex` is `FALSE` and the `color` field is not NULL, the `color` field shall specify a `Color` node containing at least (`xDimension`-1) x (`zDimension`-1) colors.

If `colorPerVertex` is `TRUE` and the `color` field is not NULL, the `color` field shall specify a `Color` node containing at least `xDimension` x `zDimension` colors, one for each vertex.

The `thickness` field specifies the thickness of the bounding box which is added below the lowest point of the `height` field, to prevent objects from falling through very thin `ElevationGrid`s.

### 3.25.3 Texture Mapping

The default texture mapping produces a texture that is upside down when viewed from the positive *y*-axis. To orient the texture with a more intuitive mapping, use a `TextureTransform` node to reverse the texture coordinate, like this:

```
Shape {
  appearance Appearance {
    textureTransform TextureTransform {
```

```
      scale 1 -1
    }
  }
  geometry ElevationGrid {
    ...
  }
}
```

This will produce a compact ElevationGrid with texture mapping that aligns with the natural orientation of the image.

## 3.26 Emitter

Derived from Device.

```
Emitter {
  SFString   type         "radio"  # or "serial" or "infra-red"
  SFFloat    range        -1       # -1 or positive
  SFFloat    maxRange     -1       # -1 or positive
  SFFloat    aperture     -1       # -1 or between 0 and 2*pi
  SFInt32    channel      0
  SFInt32    baudRate     -1       # -1 or positive
  SFInt32    byteSize     8        # 8 or more
  SFInt32    bufferSize   -1       # -1 or positive
}
```

### 3.26.1 Description

The Emitter node is used to model radio, serial or infra-red emitters. An Emitter node must be added to the children of a robot or a supervisor. Please note that an emitter can send data but it cannot receive data. In order to simulate a unidirectional communication between two robots, one robot must have an Emitter while the other robot must have a Receiver. To simulate a bidirectional communication between two robots, each robot needs to have both an Emitter and a Receiver. Note that messages are never transmitted from one robot to itself.

### 3.26.2 Field Summary

- type: type of signals: ”radio”, ”serial” or ”infra-red”. Signals of type ”radio” (the default) and ”serial” are transmitted without taking obstacles into account. Signals of type ”infra-red,” however, do take potential obstacles between the emitter and the receiver into account. Any solid object (solid, robots, etc ...) with a defined bounding object is a potential obstacle

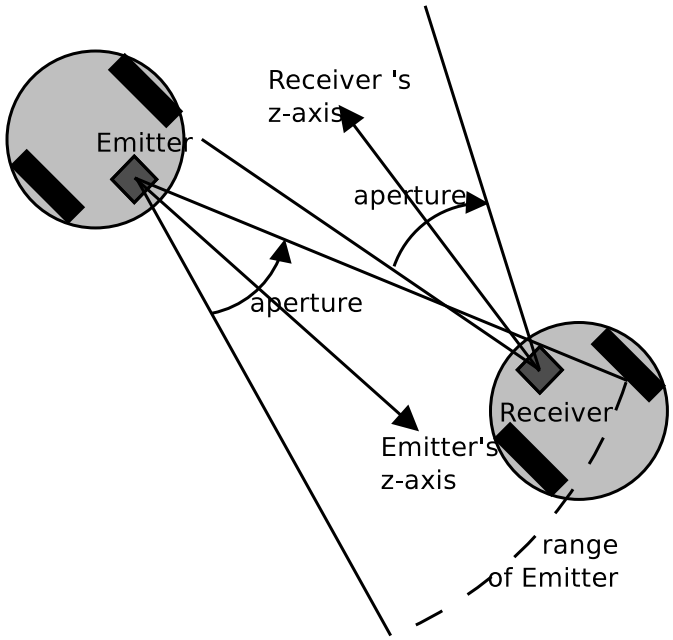


Figure 3.16: Illustration of aperture and range for ”infra-red” Emitter/Receiver

to an ”infra-red” communication. The structure of the emitting or receiving robot itself will not block an ”infra-red” transmission. Currently, there is no implementation difference between the ”radio” and ”serial” types.

- `range`: radius of the emission sphere (in meters). A receiver can only receive a message if it is located within the emission sphere. A value of -1 (the default) for `range` is considered to be an infinite range.

- `maxRange`: defines the maximum value allowed for `range`. This field defines the maximum value that can be set using `emitter_set_range()`. A value of -1 (the default) for `maxRange` is considered to be infinite.

- `aperture` opening angle of the emission cone (in radians); for ”infra-red” only. The cone’s apex is located at the origin ([0 0 0]) of the emitter’s coordinate system and the cone’s axis coincides with the $z$-axis of the emitter coordinate system. An ”infra-red” emitter can only send data to receivers currently located within its emission cone. An `aperture` of -1 (the default) is considered to be infinite, meaning that the emitted signals are omni-directional. For ”radio” and ”serial” emitters, this field is ignored. See figure 3.16 for an illustration of `range` and `aperture`.

- `channel`: transmission channel. This is an identification number for an ”infra-red” emitter or a frequency for a ”radio” emitter. Normally a receiver must use the same channel as an emitter to receive the emitted data. However, the special channel -1 allows broadcasting messages on all channels. Channel 0 (the default) is reserved for communicating with a physics plugin. For inter-robot communication, please use positive channel numbers.

- `baudRate`: the baud rate is the communication speed expressed in number of bits per second. A `baudRate` of -1 (the default) is regarded as infinite and causes the data to be transmitted immediately (within one control step) from emitter to receiver.
- `byteSize`: the byte size is the number of bits required to transmit one byte of information. This is usually 8 (the default), but can be more if control bits are used.
- `bufferSize`: specifies the size (in bytes) of the transmission buffer. The total number of bytes in the packets enqueued in the emitter cannot exceed this number. A `bufferSize` of -1 (the default) is regarded as unlimited buffer size.


*`Emitter` nodes can also be used to communicate with the physics plugin (see chapter 6). In this case the channel must be set to 0 (the default). In addition it is highly recommended to choose -1 for the baudRate, in order to enable the fastest possible communication; the `type`, `range` and `aperture` will be ignored.*

### 3.26.3 Emitter Functions

---

#### NAME

wb_emitter_send – *send a data packet to potential receivers*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/emitter.h>

int wb_emitter_send (WbDeviceTag tag, const void *data, int size);
```

#### DESCRIPTION

The `wb_emitter_send()` function adds to the emitters's queue a packet of `size` bytes located at the address indicated by `data`. The enqueued data packets will then be sent to potential receivers (and removed from the emitter's queue) at the rate specified by the `baudRate` field of the `Emitter` node. Note that a packet will not be sent to its emitter robot. This function returns 1 if the message was placed in the sending queue, 0 if the sending queue was full. The queue is considered to be *full* when the sum of bytes of all the currently enqueued packets exceeds the buffer size specified by the `bufferSize` field. Note that a packet must have at least 1 byte.

The Emitter/Receiver API does not impose any particular format on the data being transmitted. Any user chosen format is suitable, as long as the emitter and receiver codes agree. The following example shows how to send a null-terminated ascii string using the C API:


**language: C**

```c
char message[128];
sprintf(message, "hello%d", i);
wb_emitter_send(tag, message, strlen(message) +
   1);
```

And here an example on how to send binary data with the C API:


**language: C**

```c
double array[5] = { 3.0, x, y, -1/z, -5.5 };
wb_emitter_send(tag, array, 5 * sizeof(double));
```


**language: Python**

*The* `send()` *function sends a string. For sending primitive data types into this string, the* struct *module can be used. This module performs conversions between Python values and C structs represented as Python strings. Here is an example:*

```python
import struct
#...
message = struct.pack("chd","a",45,120.08)
emitter.send(message)
```


**language: Java**

*The Java* `send()` *method does not have a* `size` *argument because the size is implicitly passed with the* `data` *argument. Here is an example of sending a Java string in a way that is compatible with a C string, so that it can be received in a C/C++ controller.*

```java
String request = "You␣are␣number␣" + num + "\0";
try {
  emitter.send(request.getBytes("US-ASCII"));
}
catch (java.io.UnsupportedEncodingException e) {
  System.out.println(e);
}
```

### NAME

wb_emitter_set_channel,
wb_emitter_get_channel – *set and get the emitter's channel.*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/emitter.h>

void wb_emitter_set_channel (WbDeviceTag tag, int channel);

int wb_emitter_get_channel (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_emitter_set_channel()` function allows the controller to change the transmission channel. This modifies the `channel` field of the corresponding `Emitter` node. Normally, an emitter can send data only to receivers that use the same channel. However, the special WB_CHANNEL_BROADCAST value can be used for broadcasting to all channels. By switching the channel number an emitter can selectively send data to different receivers. The `wb_emitter_get_channel()` function returns the current channel number of the emitter.


**language: C++, Java, Python**
*In the oriented-object APIs, the WB_CHANNEL_BROADCAST constant is available as static integer of the* `Emitter` *class (Emitter::CHANNEL_BROADCAST).*

### NAME

wb_emitter_set_range,
wb_emitter_get_range – *set and get the emitter's range.*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/emitter.h>

void wb_emitter_set_range (WbDeviceTag tag, double range);

double wb_emitter_get_range (WbDeviceTag tag);
```

### DESCRIPTION

The wb_emitter_set_range() function allows the controller to change the transmission range at run-time. Data packets can only reach receivers located within the emitter's range. This function modifies the range field of the corresponding Emitter node. If the specified range argument is larger than the maxRange field of the Emitter node then the current range will be set to maxRange. The wb_emitter_get_range() function returns the current emitter's range. For both the wb_emitter_set_range() and emitter_get_range() functions, a value of -1 indicates an infinite range.

---

### NAME

wb_emitter_get_buffer_size – *get the transmission buffer size*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/emitter.h>

int wb_emitter_get_buffer_size (WbDeviceTag tag);
```

### DESCRIPTION

The wb_emitter_get_buffer_size() function returns the size (in bytes) of the transmission buffer. This corresponds to the value specified by the bufferSize field of the Emitter node. The buffer size indicates the maximum number of data bytes that the emitter's queue can hold in total, if the size is -1, the number of data bytes is not limited. When the buffer is full, calls to wb_emitter_send() will fail and return 0.

## 3.27 Fluid

Derived from Transform.

```
Fluid {
  SFString  description     ""
  field     SFString         name           "fluid"   # used in
     ImmersionProperties
  field     SFString         model           ""       # generic name
     of the fluid (eg: "sea")
  field     SFString         description     ""       # a short (1
     line) of description of the fluid
  field     SFFloat          density         1000     # (kg/m^3) fluid
      density
  field     SFFloat          viscosity       0.001    # (kg/(ms))
     fluid's dynamic viscosity
```

```
  field     SFVec3f          streamVelocity  0 0 0    # (m/s) linear
     fluid velocity
  SFNode    boundingObject   NULL
  SFBool    locked           FALSE
}
```

### 3.27.1 Description

A Fluid node represents a possibly unbounded fluid volume with physical properties such as density and stream velocity. A Solid node which is partially or fully immersed in some Fluid's boundingObject will be subject to the static force (Archimedes'thrust) and the dynamic force (drag force) exerted by the Fluid provided it has a Physics node, a boundingObject and that its field immersionProperties contains an ImmersionProperties node referring to the given Fluid.

In the 3D window, Fluid nodes can be manipulated (dragged, lifted, rotated, etc) using the mouse.

### 3.27.2 Fluid Fields

Note that in the Fluid node, the scale field inherited from the Transform must always remain uniform, i.e., of the form x x x where x is any positive real number. This ensures that all primitive geometries will remain suitable for ODE immersion detection. Whenever a scale coordinate is changed, the two other ones are automatically changed to this new value. If a scale coordinate is assigned a non-positive value, it is automatically changed to 1.

- name: name of the fluid. This is the name used in a ImmersionProperties to refer to a given Fluid.
- model: generic name of the fluid, e.g., "sea".
- description: short description (1 line) of the fluid.
- density: density of the fluid expressed in kg/m$^3$; it defaults to water density. The fluid density is taken into account for the computations of Archimedes' thrust, drag forces and drag torques, see ImmersionProperties.
- viscosity: dynamic viscosity of the fluid expressed in kg/(ms). It defaults to viscosity of water at 20 degrees Celsius.
- streamVelocity: fluid linear velocity, the flow being assumed laminar. The fluid linear velocity is taken into account for the drag and viscous resistance computations, see ImmersionProperties.

- `boundingObject`: the bounding object specifies the geometrical primitives and their `Transform` offset used for immersion detection. If the `boundingObject` field is NULL, then no immersion detection is performed and that fluid will have no effect on immersed objects. A `Solid` will undergo static or dynamic forces exerted by a `Fluid` only if its `boundingObject` collides with the `Fluid`'s `boundingObject`. The intersection volume volume with an individual primitive geometry is approximated by the intersection volume of this geometry with a tangent plane of equation $y = c, c > 0$ in the geometry coordinate system. This volume is used to generates Archimedes'thrust.

  This field is subject to the same restrictions as a `Solid`'s `boundingObject`.

- `locked`: if `TRUE`, the fluid object cannot be moved using the mouse. This is useful to prevent moving an object by mistake.

## 3.28 Fog

```
Fog {
  SFColor    color              1 1 1     # [0,1]
  SFString   fogType            "LINEAR"
  SFFloat    visibilityRange    0         # [0,inf)
}
```

The `Fog` node provides a way to simulate atmospheric effects by blending objects with the color specified by the `color` field based on the distances of the various objects from the camera. The distances are calculated in the coordinate space of the `Fog` node. The `visibilityRange` specifies the distance in meters (in the local coordinate system) at which objects are totally obscured by the fog. Objects located beyond the `visibilityRange` of the camera are drawn with a constant specified by the `color` field. Objects very close to the viewer are blended very little with the fog `color`. A `visibilityRange` of 0.0 disables the `Fog` node.

The `fogType` field controls how much of the fog color is blended with the object as a function of distance. If `fogType` is "LINEAR", the amount of blending is a linear function of the distance, resulting in a depth cueing effect. If `fogType` is "EXPONENTIAL", an exponential increase in blending is used, resulting in a more natural fog appearance. If `fogType` is "EXPONENTIAL2", a square exponential increase in blending is used, resulting in an even more natural fog appearance (see the OpenGL documentation for more details about fog rendering).

## 3.29 GPS

Derived from `Device`.

```
GPS {
  SFString   type        "satellite"
```

```
  SFFloat    accuracy    0
  SFFloat    resolution -1
}
```

### 3.29.1 Description

The GPS node is used to model a Global Positioning Sensor (GPS) which can obtain information about its absolute position from the controller program.

### 3.29.2 Field Summary

- `type`: This field defines the type of GPS technology used like ”satellite” or ”laser” (currently ignored).
- `accuracy`: This field defines the precision of the GPS, that is the maximum error (expressed in meter) in the absolute position.
- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an ’infinite’ resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

### 3.29.3 GPS Functions

---

#### NAME

wb_gps_enable,
wb_gps_disable,
wb_gps_get_sampling_period,
wb_gps_get_values – *enable, disable and read the GPS measurements*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/gps.h>

void wb_gps_enable (WbDeviceTag tag, int ms);

void wb_gps_disable (WbDeviceTag tag);

int wb_gps_get_sampling_period (WbDeviceTag tag);

const double *wb_gps_get_values (WbDeviceTag tag);
```

### DESCRIPTION

`wb_gps_enable()` allows the user to enable a GPS measurement each `ms` milliseconds.

`wb_gps_disable()` turns the GPS off, saving computation time.

The `wb_gps_get_sampling_period()` function returns the period given into the `wb_gps_enable()` function, or 0 if the device is disabled.

The `wb_gps_get_values()` function returns the current `GPS` measurement. The values are returned as a 3D-vector, therefore only the indices 0, 1, and 2 are valid for accessing the vector. The returned vector indicates the absolute position of the `GPS` device.


**language: C, C++**
*The returned vector is a pointer to the internal values managed by the* `GPS` *node, therefore it is illegal to free this pointer. Furthermore, note that the pointed values are only valid until the next call to* `wb_robot_step()` *or* `Robot::step()`. *If these values are needed for a longer period they must be copied.*


**language: Python**
`getValues()` *returns the 3D-vector as a list containing three floats.*

## 3.30 Group

```
Group {
  MFNode    children    []
}
```

Direct derived nodes: `Transform`.

A `Group` node contains `children` nodes without introducing a new transformation. It is equivalent to a `Transform` node containing an identity transform.

A `Group` node may not contain subsequent `Solid`, device or robot nodes.

## 3.31 Gyro

Derived from `Device`.

```
Gyro {
  MFVec3f    lookupTable    []    # interpolation
  SFBool     xAxis          TRUE  # compute x-axis
  SFBool     yAxis          TRUE  # compute y-axis
  SFBool     zAxis          TRUE  # compute z-axis
  SFFloat    resolution     -1
}
```

### 3.31.1 Description

The Gyro node is used to model 1, 2 and 3-axis angular velocity sensors (gyroscope). The angular velocity is measured in radians per second [rad/s].

### 3.31.2 Field Summary

- `lookupTable`: This field optionally specifies a lookup table that can be used for mapping the raw angular velocity values [rad/s] to device specific output values. With the lookup table it is also possible to add noise and to define the min and max output values. By default the lookup table is empty and therefore the raw values are returned (no mapping).

- `xAxis, yAxis, zAxis`: Each of these boolean fields specifies if the computation should be enabled or disabled for the specified axis. If one of these fields is set to FALSE, then the corresponding vector element will not be computed and it will return *NaN* (Not a Number) For example if `zAxis` is FALSE, then `wb_gyro_get_values()[2]` returns *NaN*. The default is that all three axes are enabled (TRUE).

- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

### 3.31.3 Gyro Functions

---

#### NAME

wb_gyro_enable,
wb_gyro_disable,
wb_gyro_get_sampling_period,
wb_gyro_get_values – *enable, disable and read the output values of the gyro device*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/gyro.h>

void wb_gyro_enable (WbDeviceTag tag, int ms);

void wb_gyro_disable (WbDeviceTag tag);

int wb_gyro_get_sampling_period (WbDeviceTag tag);

const double *wb_gyro_get_values (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_gyro_enable()` function turns on the angular velocity measurement each `ms` milliseconds.

The `wb_gyro_disable()` function turns off the `Gyro` device.

The `wb_gyro_get_sampling_period()` function returns the period given into the `wb_gyro_enable()` function, or 0 if the device is disabled.

The `wb_gyro_get_values()` function returns the current measurement of the `Gyro` device. The values are returned as a 3D-vector therefore only the indices 0, 1, and 2 are valid for accessing the vector. Each vector element represents the angular velocity about one of the axes of the `Gyro` node, expressed in radians per second [rad/s]. The first element corresponds to the angular velocity about the *x*-axis, the second element to the *y*-axis, etc.


**language: C, C++**
*The returned vector is a pointer to the internal values managed by the* `Gyro` *node, therefore it is illegal to free this pointer. Furthermore, note that the pointed values are only valid until the next call to* `wb_robot_step()` *or* `Robot::step()`*. If these values are needed for a longer period they must be copied.*


**language: Python**
`getValues()` *returns the vector as a list containing three floats.*

## 3.32 HingeJoint

Derived from `Joint`.

```
HingeJoint {
  field MFNode device [ ] # RotationalMotor, PositionSensor and Brake
  hiddenField SFFloat position 0 # (rad) initial position
}
```

### 3.32.1 Description

The HingeJoint node can be used to model a hinge, i.e. a joint allowing only a rotational motion around a given axis (1 degree of freedom). It inherits Joint's jointParameters field. This field can be filled with a HingeJointParameters only. If empty, HingeJointParameters default values apply.

### 3.32.2 Field Summary

- device: This field optionally specifies a RotationalMotor, an angular PositionSensor and/or a Brake device. If no motor is specified, the joint is passive joint.
- position: see joint's hidden position field.

## 3.33 HingeJointParameters

Derived from JointParameters.

```
HingeJointParameters {
  field SFVec3f anchor                    0 0 0 # for the rotation
     axis (m)
  # the following field have different default values than the parent
     class
  field SFVec3f axis                      1 0 0 # rotation axis
  field SFFloat suspensionSpringConstant  0     # linear spring
     constant along the suspension axis (Ns/m)
  field SFFloat suspensionDampingConstant 0     # linear damping
     constant along the suspension axis (Ns/m)
  field SFVec3f suspensionAxis            1 0 0 # direction of the
     suspension axis
}
```

### 3.33.1 Description

The HingeJointParameters node can be used to specify the hinge rotation axis and various joint parameters (e.g., angular position, stop angles, spring and damping constants etc.) related to this rotation axis.

### 3.33.2 Field Summary

- `anchor`: This field specifies the anchor position, i.e. a point through which the hinge axis passes. Together with the `axis` field inherited from the `JointParameters` node, the `anchor` field determines the hinge rotation axis in a unique way. It is expressed in relative coordinates with respect to the the closest upper `Solid`'s frame.
- `suspensionSpringConstant`: This field specifies the suspension spring constant along the suspension axis.
- `suspensionDampingConstant`: This field specifies the suspension damping constant along the suspension axis.
- `suspensionAxis`: This field specifies the direction of the suspension axis.

The `suspensionSpringConstant` and `suspensionDampingConstant` fields can be used to add a linear spring and/or damping behavior *along* the axis defined in `suspensionAxis`. These fields are described in more detail in `JointParameters`'s Springs and Dampers" section.

## 3.34 Hinge2Joint

Derived from `HingeJoint`.

```
Hinge2Joint {
  field SFNode jointParameters2 NULL # JointParameters for second axis
  field SFNode device2 [ ] # RotationalMotor, PositionSensor and Brake
  hiddenField SFFloat position2  0 # initial position with respect to
     the second hinge (rad)
}
```

### 3.34.1 Description

The `Hinge2Joint` node can be used to model a hinge2 joint, i.e. a joint allowing only rotational motions around two intersecting axes (2 degrees of freedom). These axes cross at the `anchor` point and need not to be perpendicular. The suspension fields defined in a `HingeJointParameters` node allow for spring and damping effects along the suspension axis.

Note that a universal joint boils down to a hinge2 joint with orthogonal axes and (default) zero suspension.

Typically, `Hinge2Joint` can be used to model a steering wheel with suspension for a car, a shoulder or a hip for a humanoid robot.


*A Hinge2Joint will connect only Solids having a Physics node. In other words, this joint cannot be statically based.*

### 3.34.2 Field Summary

- jointParameters2: This field optionally specifies a HingeJointParameters node It contains, among others, the joint position, the axis position expressed in relative coordinates, the stop positions and suspension parameters. If the jointParameters field is left empty, default values of the HingeJointParameters node apply.
- device2: This field optionally specifies a RotationalMotor, an angular PositionSensor and/or a Brake device attached to the second axis. If no motor is specified, this part of the joint is passive.
- position2: see joint's hidden position field.

## 3.35 ImageTexture

```
ImageTexture {
  MFString   url        []
  SFBool     repeatS    TRUE
  SFBool     repeatT    TRUE
  SFBool     filtering  TRUE
}
```

### 3.35.1 Description

The ImageTexture node defines a texture map by specifying an image file and general parameters for mapping to geometry. Texture maps are defined in a 2D coordinate system *(s,t)* that ranges from 0.0 to 1.0 in both directions. The bottom edge of the image corresponds to the *s*-axis of the texture map, and left edge of the image corresponds to the *t*-axis of the texture map. The lower-left pixel of the image corresponds to *s=0, t=0*, and the top-right pixel of the image corresponds to *s=1, t=1*. These relationships are depicted below.

The texture is read from the file specified by the url field. The file should be specified with a relative path. Absolute paths work as well, but they are not recommended because they are not portable across different systems. Ideally, the texture file should lie next to the world file, possibly inside a textures subfolder. Supported image formats include both JPEG and PNG. The rendering of the PNG alpha transparency is supported. The texture image width and height should be a power of 2. For example, images with a resolution of 8x8, 8x16, 32x64, 1024x64,

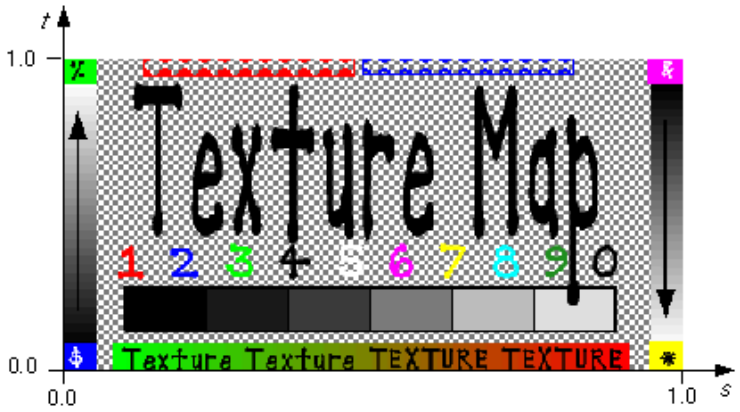


Figure 3.17: Texture map coordinate system

2048x512, 1024x1024 pixels are valid images. Images with a resolution of 100x100, 123x47, 1203x2336 are not valid images.

A PNG image may contain an alpha channel. If such an alpha channel exists, the texture becomes semi-transparent. This is useful to render for example a scissor cut texture. Semi-transparent objects are sorted according to their center (the local position of the parent Transform) and are rendered in the same rendering queue as the objects having a transparent material (see the `transparency` field of the `Material` node). Semi-transparent objects cannot receive and cannot cast shadows.

If the image contains an alpha channel no texture filtering is performed, otherwise both a trilinear interpolation and an anisotropic texture filtering is applied (the texture is subsampled according to the distance and the angle between the textured polygon and the camera).

The `repeatS` and `repeatT` fields specify how the texture wraps in the *s* and *t* directions. If `repeatS` is `TRUE` (the default), the texture map is repeated outside the [0.0,1.0] texture coordinate range in the *s* direction so that it fills the shape. If `repeatS` is `FALSE`, the texture coordinates are clamped in the *s* direction to lie within the [0.0,1.0] range. The `repeatT` field is analogous to the `repeatS` field.

The `filtering` field defines whether the texture will be displayed using a texture filtering or not. No filtering corresponds to a simple nearest-neighbor pixel interpolation filtering method. Filtering corresponds to both an anisotropic filtering method (using mipmapping) which chooses the smallest mipmap according to the texture orientation and to the texture distance, and a trilinear filtering method which smoothes the texture. Using filtering doesn't affect significantly the run-time performance, however it may increase slightly the initialization time because of the generation of the mipmaps.

## 3.36 ImmersionProperties

```
ImmersionProperties {
  field SFString   fluidName       ""
  field SFString   referenceArea   "immersed area"
  field SFVec3f    dragForceCoefficients  0 0 0 # dimensionless
     coefficient ranging in [0, infinity)
  field SFVec3f    dragTorqueCoefficients  0 0 0 # dimensionless
     coefficients ranging in [0, infinity)
  field SFFloat    viscousResistanceForceCoefficient  0 # (Ns/m)
     ranges in [0, infinity)
  field SFFloat    viscousResistanceTorqueCoefficient  0 # (Nm/s )
     ranges in [0, infinity)
}
```

### 3.36.1 Description

An ImmersionProperties node is used inside the immersionProperties field of a Solid node to specify its dynamical interactions with one or more Fluid nodes.

### 3.36.2 ImmersionProperties Fields

- fluidName: name of the fluid with which the dynamical interaction is enabled. The string value must coincide with the name field value of an existing Fluid node.
- referenceArea: this field defines the reference area(s) used to compute the drag force and drag torque of the submerging Fluid.

  If the referenceArea is set to "xyz-projected area", the $x$-coordinate of the drag force vector with respect to the the solid frame is given by:

  ```
  drag_force_x = - c_x * fluid_density * (rel_linear_velocity_x)^2
     * sign(rel_linear_velocity_x) * A_x
  ```

  where c_x is the $x$-coordinate of the dragForceCoefficients vector, linear_velocity_x the $x$-coordinate of the linear velocity of the solid with respect to the fluid expressed within the solid frame and A_x is the projected immersed area onto the plane $x = 0$. Analogous formulas hold for $y$ and $z$ coordinates.

  The $x$-coordinate of the drag torque vector with respect to the the solid frame is given by:

  ```
  drag_torque_x = - t_x * fluid_density * (rel_angular_velocity_x)
     ^2 * sign(rel_angular_velocity_y) * (A_y + A_z)
  ```

  where t_x is the $x$-coordinate of the dragTorqueCoefficients vector, angular_velocity_x the $x$-coordinate of the angular velocity of the solid expressed within the solid frame. Analogous formulas hold for $y$ and $z$ coordinates.

  If the referenceArea value is "immersed area" then the Solid boundingObject's immersed area is used for drag force and drag torque computations:

```
drag_force = - c_x * fluid_density * linear_velocity^2 *
   immersed_area,
drag_torque = - t_x * fluid_density * angular_velocity^2 *
   immersed_area
```

all vectors being expressed in world coordinates. Note that in this case the drag coefficients along the $y$ and $z$ axes are ignored.

- `dragForceCoefficients` and `dragTorqueCoefficients`: dimensionless non-negative coefficients used to compute the drag force and the drag torque exerted by the fluid on the solid. See above formulas.

- `viscousResistanceForceCoefficient` and `viscousResistanceTorque-Coefficient`: this non-negative coefficients, expressed respectively in Ns/m and Nm/s, are used to compute the viscous resistance force and the viscous resistance torque exerted by the fluid on the solid according the following formulas

  ```
  viscous_resistance_force = - immersion_ratio * fluid_viscosity *
     v_force * rel_linear_velocity
  viscous_resistance_torque = - immersion_ratio * fluid_viscosity *
      v_torque * angular_velocity
  ```

  where v_force (resp. v_torque) denotes the viscous resistance force (resp. torque) coefficient and immersion_ratio is obtained by dividing the immersed area by the full area.

  The viscous resistance (or linear drag) is appropriate for objects moving through a fluid at relatively low speed where there is no turbulences. By its linear nature it may offer a better numerical stability than the above quadratic drags when the immersed solids are subject to large external forces or torques.


*The "xyz-projected area" computation mode is implemented only for boundingObjects that contain fully or partially immersed* `Box` *nodes, fully immersed* `Cylinder`, `Capsule` *and* `Sphere` *nodes. The "immersed area" computation mode is implemented for every Geometry node.*

## 3.37 IndexedFaceSet

```
IndexedFaceSet {
  SFNode    coord           NULL
  SFNode    texCoord        NULL
  SFBool    solid           TRUE  # ignored and regarded as TRUE
  SFBool    ccw             TRUE
  SFBool    convex          TRUE
  MFInt32   coordIndex      []    # [-1,inf)
```

```
  MFInt32    texCoordIndex    []     # [-1,inf)
  SFFloat    creaseAngle      0      # [0,inf)
}
```

### 3.37.1 Description

The IndexedFaceSet node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the coord field. The IndexedFaceSet node can be used either as a graphical or as a collision detection primitive (in a boundingObject). IndexedFaceSet nodes can be easily imported from 3D modeling programs after a triangle mesh conversion.

### 3.37.2 Field Summary

The coord field contains a Coordinate node that defines the 3D vertices referenced by the coordIndex field. IndexedFaceSet uses the indices in its coordIndex field to specify the polygonal faces by indexing into the coordinates in the Coordinate node. An index of "-1" indicates that the current face has ended and the next one begins. The last face may be (but does not have to be) followed by a "-1" index. If the greatest index in the coordIndex field is N, the Coordinate node shall contain N+1 coordinates (indexed as 0 to N). Each face of the IndexedFaceSet shall have:

- at least three non-coincident vertices;
- vertices that define a planar polygon;
- vertices that define a non-self-intersecting polygon.

Otherwise, the results are undefined.

When used for collision detection (boundingObject), each face of the IndexedFaceSet must contain exactly three vertices, hence defining a triangle mesh (or trimesh).

If the texCoord field is not NULL, then it must contain a TextureCoordinate node. The texture coordinates in that node are applied to the vertices of the IndexedFaceSet as follows:

If the texCoordIndex field is not empty, then it is used to choose texture coordinates for each vertex of the IndexedFaceSet in exactly the same manner that the coordIndex field is used to choose coordinates for each vertex from the Coordinate node. The texCoordIndex field must contain at least as many indices as the coordIndex field, and must contain end-of-face markers (-1) in exactly the same places as the coordIndex field. If the greatest index in the texCoordIndex field is N, then there must be N+1 texture coordinates in the TextureCoordinate node.

The creaseAngle field, affects how default normals are generated. For example, when an IndexedFaceSet has to generate default normals, it uses the creaseAngle field to determine which edges should be smoothly shaded and which ones should have a sharp crease. The

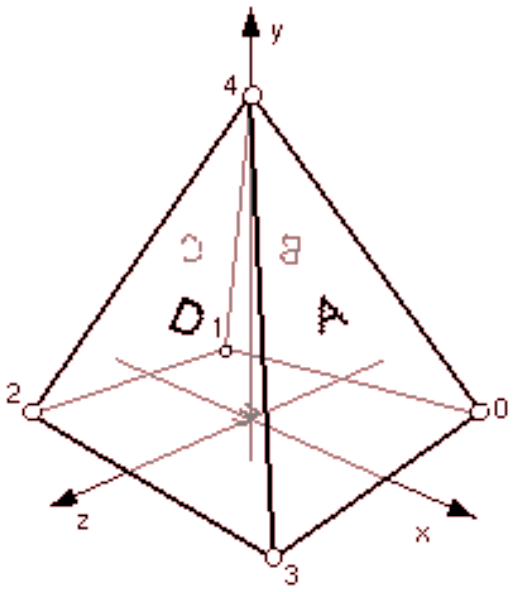


Figure 3.18: A simple IndexedFaceSet example

crease angle is the positive angle between surface normals on adjacent polygons. For example, a crease angle of .5 radians means that an edge between two adjacent polygonal faces will be smooth shaded if the normals to the two faces form an angle that is less than .5 radians (about 30 degrees). Otherwise, it will be faceted. Crease angles must be greater than or equal to 0.0.

### 3.37.3 Example

```
IndexedFaceSet {
  coord Coordinate {
    point [ 1 0 -1, -1 0 -1, -1 0 1, 1 0 1, 0 2 0 ]
  }
  coordIndex [ 0 4 3 -1   # face A, right
               1 4 0 -1   # face B, back
               2 4 1 -1   # face C, left
               3 4 2 -1   # face D, front
               0 3 2  1 ] # face E, bottom
}
```

## 3.38 IndexedLineSet

```
IndexedLineSet {
  SFNode    coord        NULL
```

```
  MFInt32    coordIndex    []     # [-1,inf)
}
```

The IndexedLineSet node represents a 3D geometry formed by constructing polylines from 3D vertices specified in the coord field. IndexedLineSet uses the indices in its coordIndex field to specify the polylines by connecting vertices from the coord field. An index of "-1" indicates that the current polyline has ended and the next one begins. The last polyline may be (but does not have to be) followed by a "-1". IndexedLineSet is specified in the local coordinate system and is affected by the transformations of its ancestors.

The coord field specifies the 3D vertices of the line set and contains a Coordinate node.

IndexedLineSets are not lit, are not texture-mapped and they do not cast or receive shadows. IndexedLineSets cannot be use for collision detection (boundingObject).

## 3.39 InertialUnit

Derived from Device.

```
InertialUnit {
  MFVec3f    lookupTable    []     # interpolation
  SFBool     xAxis          TRUE   # compute roll
  SFBool     zAxis          TRUE   # compute pitch
  SFBool     yAxis          TRUE   # compute yaw
  SFFloat    resolution     -1
}
```

### 3.39.1 Description

The InertialUnit node simulates an *Inertial Measurement Unit* (IMU). The InertialUnit computes and returns its *roll*, *pitch* and *yaw* angles with respect to a global coordinate system defined in the WorldInfo node. If you would like to measure an acceleration or an angular velocity, please use the Accelerometer or Gyro node instead. The InertialUnit node must be placed on the Robot so that its $x$-axis points in the direction of the Robot's forward motion (longitudinal axis). The positive $z$-axis must point towards the Robot's right side, e.g., right arm, right wing (lateral axis). The positive $y$-axis must point to the Robot's up/top direction. If the InertialUnit has this orientation, then the *roll*, *pitch* and *yaw* angles correspond to the usual automotive, aeronautics or spatial meaning.

More precisely, the InertialUnit measures the Tait-Bryan angles along $x$-axis (roll), $z$-axis (pitch) and $y$-axis (yaw). This convention is commonly referred to as the $x$-$z$-$y$ extrinsic sequence; it corresponds to the composition of elemental rotations denoted by YZX. The reference frame is made of the unit vector giving the north direction, the opposite of the normalized gravity vector and their cross-product (see WorldInfo to customize this frame).


*In a gimbal lock situation, i.e., when the pitch is -π/2 or π/2, the roll and the yaw are set to NaN (Not a Number).*

### 3.39.2 Field Summary

- `lookupTable`: This field optionally specifies a lookup table that can be used for changing the angle values [rad] into device specific output values, or for changing the units to degrees for example. With the lookup table it is also possible to define the min and max output values and to add noise to the output values. By default the lookup table is empty and therefore the returned angle values are expressed in radians and no noise is added.

- `xAxis, yAxis, zAxis`: Each of these boolean fields specifies if the computation should be enabled or disabled for the specified axis. The `xAxis` field defines whether the *roll* angle should be computed. The `yAxis` field defines whether the *yaw* angle should be computed. The `zAxis` field defines whether the *pitch* angle should be computed. If one of these fields is set to FALSE, then the corresponding angle element will not be computed and it will return *NaN* (Not a Number). For example if `zAxis` is FALSE, then `wb_inertial_unit_get_values()[2]` returns *NaN*. The default is that all three axes are enabled (TRUE).

- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

### 3.39.3 InertialUnit Functions

---

#### NAME

wb_inertial_unit_enable,
wb_inertial_unit_disable,
wb_inertial_unit_get_sampling_period,
wb_inertial_unit_get_roll_pitch_yaw – *enable, disable and read the output values of the inertial unit*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/inertial_unit.h>

void wb_inertial_unit_enable (WbDeviceTag tag, int ms);
```

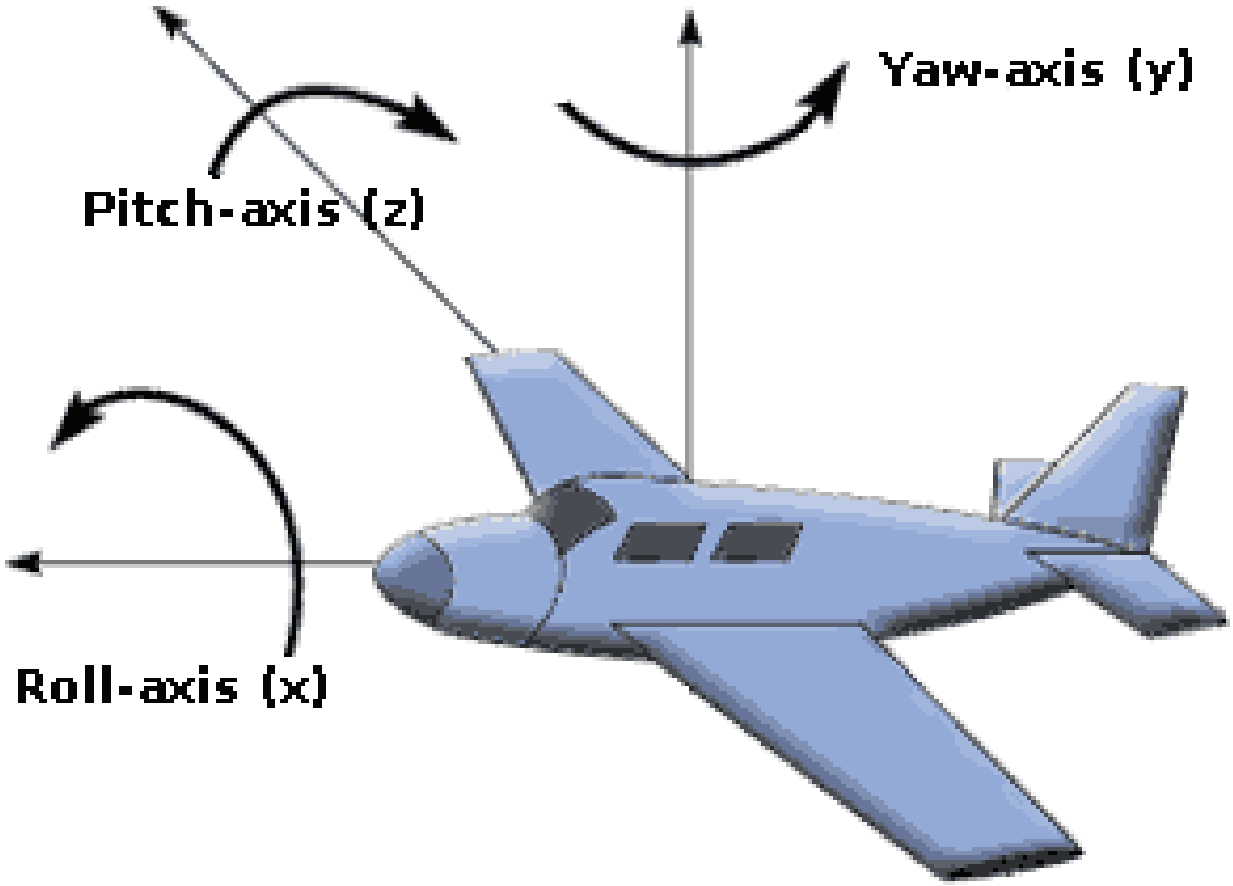


Figure 3.19: Roll, pitch and yaw angles in Webots' Inertial Unit

```
void wb_inertial_unit_disable (WbDeviceTag tag);
int wb_inertial_unit_get_sampling_period (WbDeviceTag tag);
const double *wb_inertial_unit_get_roll_pitch_yaw (WbDeviceTag tag);
```

## DESCRIPTION

The `wb_inertial_unit_enable()` function turns on the angle measurement each `ms` milliseconds.

The `wb_inertial_unit_disable()` function turns off the `InertialUnit` device.

The `wb_inertial_unit_get_sampling_period()` function returns the period given into the `wb_inertial_unit_enable()` function, or 0 if the device is disabled.

The `wb_inertial_unit_get_roll_pitch_yaw()` function returns the current *roll*, *pitch* and *yaw* angles of the `InertialUnit`. The values are returned as an array of 3 components therefore only the indices 0, 1, and 2 are valid for accessing the returned array. Note that the indices 0, 1 and 2 return the *roll*, *pitch* and *yaw* angles respectively.

The *roll* angle indicates the unit's rotation angle about its *x*-axis, in the interval [-$\pi$,$\pi$]. The *roll* angle is zero when the `InertialUnit` is horizontal, i.e., when its *y*-axis has the opposite direction of the gravity (`WorldInfo` defines the `gravity` vector).

The *pitch* angle indicates the unit's rotation angle about is *z*-axis, in the interval [-$\pi$/2,$\pi$/2]. The *pitch* angle is zero when the `InertialUnit` is horizontal, i.e., when its *y*-axis has the opposite direction of the gravity. If the `InertialUnit` is placed on the `Robot` with a standard orientation, then the *pitch* angle is negative when the `Robot` is going down, and positive when the robot is going up.

The *yaw* angle indicates the unit orientation, in the interval [-$\pi$,$\pi$], with respect to `WorldInfo.northDirection`. The *yaw* angle is zero when the `InertialUnit`'s *x*-axis is aligned with the north direction, it is $\pi$/2 when the unit is heading east, and -$\pi$/2 when the unit is oriented towards the west. The *yaw* angle can be used as a compass.


**language: C, C++**
*The returned vector is a pointer to internal values managed by the Webots, therefore it is illegal to free this pointer. Furthermore, note that the pointed values are only valid until the next call to* `wb_robot_step()` *or* `Robot::step()`. *If these values are needed for a longer period they must be copied.*


**language: Python**
`getRollPitchYaw()` *returns the angles as a list containing three floats.*

# 3.40 Joint

```
Joint {
  field SFNode jointParameters NULL # a joint parameters node
  field SFNode endPoint        NULL # Solid or SolidReference
}
```

## 3.40.1 Description

The Joint node is an abstract node (not instantiated) whose derived classes model various types of mechanichal joints: hinge (HingeJoint), slider (SliderJoint), ball joint (BallJoint), hinge2 (Hinge2Joint). Apart from the ball joint, joints can be motorized and endowed with PositionSensor nodes.

The Joint node creates a link between its Solid parent and the Solid placed into its endPoint field. Using a SolidReference inside endPoint enables you to close mechanical loops within a Robot or a passive mechanichal system.

## 3.40.2 Field Summary

- jointParameters: this field optionally specifies a JointParameters node or one of its derived classes. These nodes contain common joint paramaters such as position, stops, anchor or axis if existing. This field must be filled with an HingeJointParameters node for an HingeJoint or an Hinge2Joint, with a JointParameters node for a SliderJoint (anchor-less) and with a BallJointParameters node for a BallJoint.

  For an Hinge2Joint, the jointParameters field is related to the first rotation axis while an additional field called jointParameters2 refers to the second rotation axis.

  3D-vector parameters (e.g axis, anchor) are always expressed in relative coordinates with respect to the closest upper Solid's frame using the meter as unit. If the jointParameters field is not specified, parameters are set with the default values defined in the corresponding parameter node.

- endPoint: this field specifies which Solid will be subjected to the joint constraints. It must be either a Solid child, or a reference to an existing Solid, i.e. a SolidReference.

## 3.40.3 Joint's hidden position fields

If the jointParameters is set to NULL, joint positions are then not visible from the Scene Tree. In this case Webots keeps track of the initial positions of Joint nodes (except for the

BallJoint) by means of hidden position fields. These fields, which are not visible from the Scene Tree, are used to store inside the world file the current joint positions when the simulation is saved. As a result joint positions are restored when reloading the simulation just the same way they would be if JointParameters nodes were used.

For HingeJoint and SliderJoint nodes containing no JointParameters, Webots uses the hidden field named position. For a Hinge2Joint node, an additional hidden field named position2 is used to store the joint position with respect the the second hinge.

## 3.41 JointParameters

```
JointParameters {
  field SFFloat position         0 # current position (m or rad)
  field SFVec3f axis         0 0 1 # displacement axis (m)
  field SFFloat minStop          0 # low stop position (m or rad)
  field SFFloat maxStop          0 # high stop position (m or rad)
  field SFFloat springConstant   0 # spring constant (N/m or Nm)
  field SFFloat dampingConstant  0 # damping constant (Ns/m or Nms)
  field SFFloat staticFriction   0 # friction constant (Ns/m or Nms)
}
```


*The default value of the axis field may change in a derived class. For instance, the axis default value of an* HingeJointParameters *is* 1 0 0.

### 3.41.1 Description

The JointParameters node is a concrete base node used to specify various joint parameters related to an axis along which, or around which, the motion is allowed. As an instantiated node it can be used within the jointParameters field of SliderJoint or within the jointParameters2 field of Hinge2Joint. Unlike the other joint parameters node, it has no anchor.

### 3.41.2 Field Summary

- The position field represents the current *position* of the joint, in radians or meters. For an hinge, position represents the current rotation angle in radians. For a slider, position represents the magnitude of the current translation in meters.

- The minPosition and maxPosition fields specify *soft limits* for the target position. These fields are described in more detail in the section ”Joint Limits”, see below.

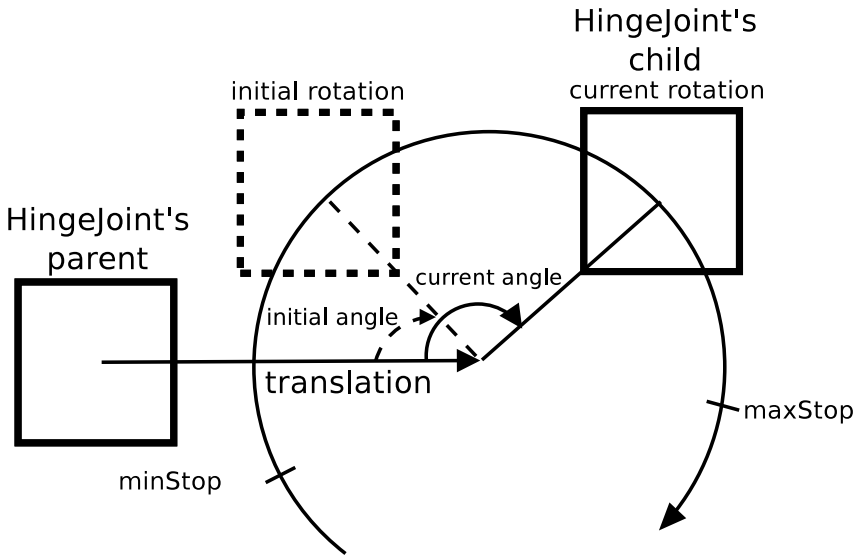


Figure 3.20: HingeJoint

- The `minStop` and `maxStop` fields specify the position of physical (or mechanical) stops. These fields are described in more detail in the section "Joint Limits", see below.
- The `springConstant` and `dampingConstant` fields allow the addition of spring and/or damping behavior to the joint. These fields are described in more detail in the section "Springs and Dampers", see below.
- The `staticFriction` allows to add a friction opposed to the joint motion. This field is described in more detail in the section "Friction", see below.

### 3.41.3 Units

Rotational joint units (`HingeJoint`, `Hinge2Joint`) are expressed in *radians* while linear joint units (`SliderJoint`) are expressed in *meters*. See table 3.4:

| | Rotational | Linear |
|---|---|---|
| Position | rad (radians) | m (meters) |

Table 3.4: Joint Units

### 3.41.4 Initial Transformation and Position

The `position` field is a scalar representing an angle (in radians) or a distance (in meters) computed with respect to the initial `translation` and `rotation` of the `Joint`'s `Solid` child.

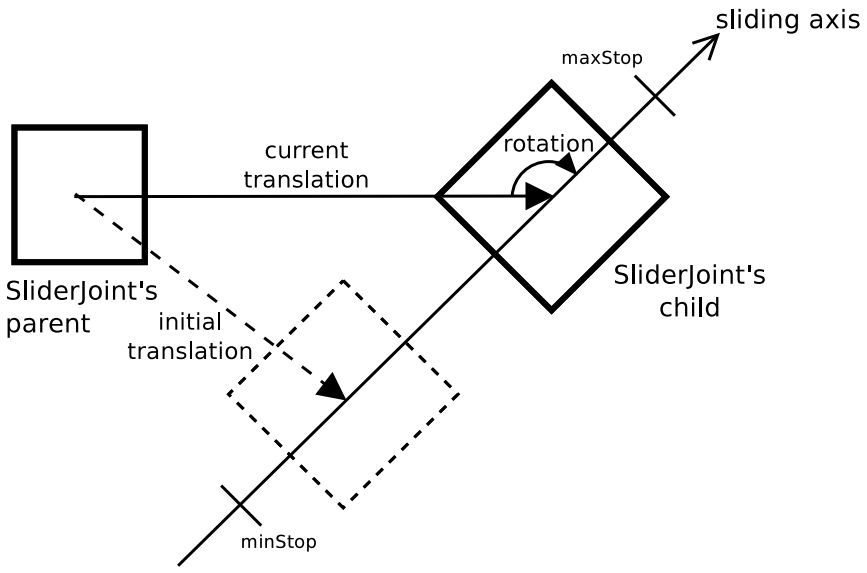


Figure 3.21: SliderJoint

If its value is zero, then the `Joint`'s child is *by definition* set with its initial `translation` and `rotation`. For a joint with one or two rotational degrees of freedom (e.g., `HingeJoint`, `Hinge2Joint`), the `position` field value is the rotation angle around one the joint axes that was applied to the `Joint`'s child initially in zero position. For a slider joint, `position` is the translation length along the sliding axis that was applied to the `Joint`'s child initially in zero position.

For example if we have a `HingeJoint` and a `position` field value of 1.5708, this means that this `HingeJoint` is 90 degrees from its initial rotation with respect to the hinge rotation axis. The values passed to the `wb_motor_set_position()` function are specified with respect to the zero position. The values of the `minStop` and `maxStop` fields are also defined with respect to the zero position.

### 3.41.5 Joint Limits

The `minStop` and `maxStop` fields define the *hard limits* of the joint. Hard limits represent physical (or mechanical) bounds that cannot be overrun by any force; they are defined with respect to the zero joint `position`. Hard limits can be used, for example, to simulate both end caps of a hydraulic or pneumatic piston or to restrict the range of rotation of a hinge. When used for a rotational motion the value of `minStop` must be in the range [-$\pi$, 0] and `maxStop` must be in the range [0, $\pi$]. When both `minStop` and `maxStop` are zero (the default), the hard limits are deactivated. The joint hard limits use ODE joint stops (for more information see the ODE documentation on `dParamLoStop` and `dParamHiStop`).

Finally, note that when both soft (`minPosition` and `maxPosition`, see the `Motor`'s "Motor

Limits" section) and hard limits (minStop and maxStop) are activated, the range of the soft limits must be included in the range of the hard limits, such that minStop <= minValue and maxStop>= maxValue.

### 3.41.6 Springs and Dampers

The springConstant field specifies the value of the spring constant (or spring stiffness), usually denoted as $K$. The springConstant must be positive or zero. If the springConstant is zero (the default), no spring torque/force will be applied to the joint. If the springConstant is greater than zero, then a spring force will be computed and applied to the joint in addition to the other forces (i.e., motor force, damping force). The spring force is calculated according to Hooke's law: $F = -Kx$, where $K$ is the springConstant and $x$ is the current joint position as represented by the position field. Therefore, the spring force is computed so as to be proportional to the current joint position, and to move the joint back to its initial position. When designing a robot model that uses springs, it is important to remember that the spring's resting position for each joint will correspond to the initial position of the joint.

The dampingConstant field specifies the value of the joint damping constant. The value of dampingConstant must be positive or zero. If dampingConstant is zero (the default), no damping torque/force will be added to the joint. If dampingConstant is greater than zero, a damping torque/force will be applied to the joint in addition to the other forces (i.e., motor force, spring force). This damping torque/force is proportional to the effective joint velocity: $F = -Bv$, where $B$ is the damping constant, and $v = dx/dt$ is the effective joint velocity computed by the physics simulator.

As you can see in (see figure 3.22), a Joint creates a joint between two masses $m_0$ and $m_1$. The mass $m_0$ is defined by the Physics node in the closest upper Solid of the Joint. The mass $m_1$ is defined by the Physics node of the Solid placed into the endPoint of the Joint. The value $x_0$ corresponds to the anchor position of the Joint defined in the anchor field of a JointParameters node. The position $x$ corresponds to the current position of the Joint defined in the position field of a JointParameters node.

## 3.42 LED

Derived from Device.

```
LED {
  MFColor   color    [ 1 0 0 ]  # [0,1]
  SFBool    gradual FALSE      # for gradual color display and RBG
     LEDs
}
```

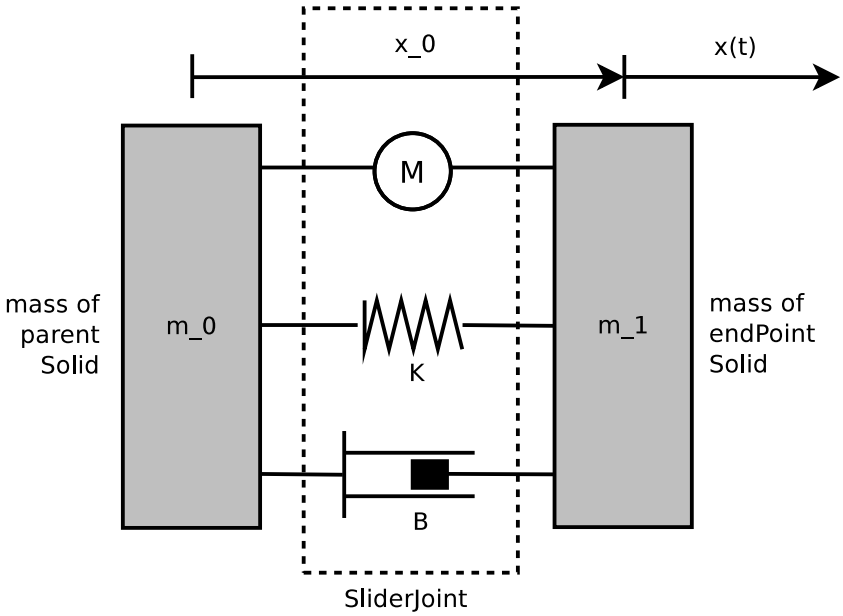


Figure 3.22: Mechanical Diagram of a Slider Joint

### 3.42.1 Description

The LED node is used to model a light emitting diode (LED). The light produced by an LED can be used for debugging or informational purposes. The resulted color is applied only on the first child of the LED node. If the first child is a Shape node, the emissiveColor field of its Material node is altered. If the first child is a Light node, its color field is altered. Otherwise, if the first child is a Group node, a recursive search is applied on this node in order to find which color field must be modified, so every Light, Shape and Group node is altered according to the previous rules.

### 3.42.2 Field Summary

- color: This defines the colors of the LED device. When off, an LED is always black. However, when on it may have different colors as specified by the LED programming interface. By default, the color defines only one color (red), but you can change this and add extra colors that could be selected from the LED programming interface. However, the number of colors defined depends on the value of the gradual field (see below).

- gradual: This defines the type of LED. If set to FALSE, the LED can take any of the color values defined in the color list. If set to TRUE, then the color list should either be empty or contain only one color value. If the color list is empty, then the LED is an RGB LED and can take any color in the R8G8B8 color space (16 million possibilities). If

the color list contains a single color, then the LED is monochromatic, and its intensity can be adjusted between 0 (off) and 255 (maximum intensity).

### 3.42.3 LED Functions

---

#### NAME

wb_led_set – *turn an LED on or off*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/led.h>

void wb_led_set (WbDeviceTag tag, int value);

int wb_led_get (WbDeviceTag tag);
```

#### DESCRIPTION

wb_led_set() switches an LED on or off, possibly changing its color. If the value parameter is 0, the LED is turned off. Otherwise, it is turned on.

In the case of a non-gradual LED (gradual field set to FALSE), if the value parameter is 1, the LED is turned on using the first color specified in the color field of the corresponding LED node. If the value parameter is 2, the LED is turned on using the second color specified in the color field of the LED node, and so on. The value parameter should not be greater than the size of the color field of the corresponding LED node.

In the case of a monochromatic LED (gradual field set to TRUE and color field containing exactly one color), the value parameter indicates the intensity of the LED in the range 0 (off) to 255 (maximum intensity).

In the case of an RGB LED (gradual field set to TRUE and color field containing an empty list), the value parameter indicates the RGB color of the LED in the range 0 (off or black) to 0xffffff (white). The format is R8G8B8: The most significant 8 bits (left hand side) indicate the red level (between 0x00 and 0xff). Bits 8 to 15 indicate the green level and the least significant 8 bits (right hand side) indicate the blue level. For example, 0xff0000 is red, 0x00ff00 is green, 0x0000ff is blue, 0xffff00 is yellow, etc.

The wb_led_get function returns the value given as argument of the last wb_led_set function call.

## 3.43 Light

```
Light {
  SFFloat    ambientIntensity    0         # [0,1]
  SFColor    color               1 1 1     # [0,1]
  SFFloat    intensity           1         # [0,1]
  SFBool     on                  TRUE
  SFBool     castShadows         FALSE
}
```

Direct derived nodes: PointLight, SpotLight, DirectionalLight.

### 3.43.1 Description

The Light node is abstract: only derived nodes can be instantiated. Lights have two purposes in Webots: (1) the are used to graphically illuminate objects and (2) they determine the quantity of light perceived by LightSensor nodes. Except for castShadows, every field of a Light node affects the light measurements made by LightSensor nodes.

### 3.43.2 Field Summary

- The intensity field specifies the brightness of the direct emission from the light, and the ambientIntensity specifies the intensity of the ambient emission from the light. Light intensity usually ranges from 0.0 (no light emission) to 1.0 (full intensity). However, when used together with LightSensors, and if real physical quantities such as Watts or lumen (lm) are desired, larger values of intensity and ambientIntensity can also be used. The color field specifies the spectral color properties of both the direct and ambient light emission as an RGB value.

- The on boolean value allows the user to turn the light on (TRUE) or off (FALSE).

- The castShadows field allows the user to turn on (TRUE) or off (FALSE) the casting of shadows for this Light. When activated, sharp shadows are casted from and received by any renderable object except for the semi-transparent objects, and the IndexedLineSet primitive. An object can be semi-transparent either if its texture has an alpha channel, or if its Material.transparency field is not equal to 1. Shadows are additive (Several lights can cast shadows). The darkness of a shadow depends on how the occluded part is lighted (either by an ambient light component or by another light). Activating the shadows of just one Light can have a significant impact on the global rendering performance, particularly if the world contains either lots of objects or complex meshes. Some shadow issues can occurs in closed spaces.

# 3.44 LightSensor

Derived from Device.

```
LightSensor {
  MFVec3f     lookupTable   [ 0 0 0, 1 1000 0 ]
  SFColor     colorFilter   1 1 1     # [0,1]
  SFBool      occlusion     FALSE
  SFFloat     resolution    -1
}
```

## 3.44.1 Description

LightSensor nodes are used to model photo-transistors, photo-diodes or any type of device that measures the irradiance of light in a given direction. *Irradiance* represents the radiant power incident on a surface in Watts per square meter (W/m$^2$), and is sometimes called *intensity*. The simulated irradiance is computed by adding the irradiance contributed by every light source (DirectionalLight, SpotLight and PointLight) in the world. Then the total irradiance is multiplied by a color filter and fed into a lookup table that returns the corresponding user-defined value.

The irradiance contribution of each light source is divided into *direct* and *ambient* contributions. The direct contribution depends on the position and the orientation of the sensor, the location and the direction of the light sources and (optionally) on the possible occlusion of the light sources. The ambient contribution ignores the possible occlusions, and it is not affected by the orientation of the sensor nor by the direction of a light source. The direct and ambient contributions of PointLights and SpotLights are attenuated according to the distance between the sensor and the light, according to specified attenuation coefficients. The light radiated by a DirectionalLight is not attenuated. See also DirectionalLight, SpotLight and PointLight node descriptions.

Note that the Webots lighting model does not take reflected light nor object colors into account.

## 3.44.2 Field Summary

- lookupTable: this table allows Webots to map simulated irradiance values to user-defined sensor output values and to specify a noise level. The first column contains the input irradiance values in W/m$^2$. The second column represents the corresponding sensor output values in user-defined units. The third column specifies the level of noise in percent of the corresponding output value. See the section on the DistanceSensor node for more explanation on how a lookupTable works.

$$E = \frac{1}{3}\vec{F} \cdot \sum_{i=1}^{n} \left(on[i] \times att[i] \times spot[i] \times (I_a[i] + I_d[i])\right) \vec{C}[i]$$

Figure 3.23: Light sensor irradiance formula

$$att[i] = \begin{cases} \frac{1}{a_1 + a_2 d + a_3 d^2} & \text{if (PointLight or SpotLight)} \\ 1 & \text{otherwise} \end{cases}$$

Figure 3.24: Light attenuation

- `colorFilter`: specifies an RGB filter that can be used to approximate a physical color filter or spectral response. The total RGB irradiance is multiplied by this filter (see formula below) in order to obtain a scalar irradiance value *E* that is then used as the input to the lookup table. The `colorFilter` field can, for example, be used to selectively detect light sources according to color.

- `occlusion`: specifies whether or not obstacles between the sensor and light sources should be taken into account in the calculation of irradiance. If the `occlusion` field is FALSE (the default), all potential obstacles (Walls, other Robots, etc.) are ignored and Webots behaves as if they were transparent. If the `occlusion` field is TRUE, Webots will detect which light sources are occluded (from the sensor's viewpoint) and it will ignore their direct contributions. Note that the `occlusion` flag affects only the *direct* light measurement, not the *ambient* light which is always added in. By default, the `occlusion` field is disabled because the occlusion detection is computationally expensive and should be avoided whenever possible. For example, in a setup where it is obvious that there will never be an obstacle between a particular sensor and the various light sources, the `occlusion` flag can be set to FALSE.

- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

Before being interpolated by the `lookupTable`, the total irradiance $E$ [W/m$^2$] seen by a sensor is computed according to the equation shown in figure 3.23:

The $F$ vector corresponds to the sensor's `colorFilter` field, $n$ is the total number of lights in the simulation, $on[i]$ corresponds to the `on` field of light $i$ (TRUE=1, FALSE=0), the $C[i]$ vector is the `color` field of light $i$, and $I_a[i]$ is the `ambientIntensity` field of light $i$. The value $att[i]$ is the attenuation of light $i$, and is calculated as shown in figure 3.24.

Variables $a_1, a_2$ and $a_3$ correspond to the `attenuation` field of light $i$, and $d$ is the distance between the sensor and the light. There is no attenuation for `DirectionalLight`s. $I_d[i]$ is the direct irradiance contributed by light $i$, and is calculated as shown in figure 3.25.

$$I_d[i] = \begin{cases} 0 & \text{if the light source is occluded} \\ I[i]\left(\frac{\vec{N} * \vec{L}}{|\vec{N}||\vec{L}|}\right) & \text{otherwise} \end{cases}$$

Figure 3.25: Direct irradiance

$$spot[i] = \begin{cases} (\cos\alpha)^{\log\frac{\log\frac{1}{2}}{2\cdot\log(\cos(beamWidth))}} & \text{if (SpotLight and } \alpha \leq \text{CutOffAngle)} \\ 0 & \text{else if (SpotLight and } \alpha > \text{CutOffAngle)} \\ 1 & \text{otherwise} \end{cases}$$

Figure 3.26: SpotLight factor

Finally, $spot[i]$ is a factor used only in case of a `SpotLight`, and that depends on its `cutOffAngle` and `beamWidth` fields, and is calculated as shown in figure 3.26, where the $alpha$ angle corresponds to the angle between $-L$ and the `direction` vector of the `SpotLight`.

The value $I[i]$ corresponds to the *intensity* field of light $i$, and $N$ is the normal axis ($x$-axis) of the sensor (see figure 3.27). In the case of a `PointLight`, $L$ is the sensor-to-light-source vector. In the case of a `DirectionalLight`, $L$ corresponds to the negative of the light's `direction` field. The * operation is a modified dot product: if dot $< 0$, then 0, otherwise, dot product. Hence, each light source contributes to the irradiance of a sensor according to the cosine of the angle between the $N$ and the $L$ vectors, as shown in the figure. The contribution is zero if the light source is located behind the sensor. This is derived from the physical fact that a photo-sensitive device is usually built as a surface of semiconductor material and therefore, the closer the angle of incidence is to perpendicular, the more photons will actually hit the surface and excite the device. When a light source is parallel to (or behind) the semiconductor surface, no photons actually reach the surface.

The "occlusion" condition is true if the light source is hidden by one or more obstacles. More precisely, "occlusion" is true if (1) the `occlusion` field of the sensor is set to TRUE and (2) there is an obstacle in the line of sight between the sensor and the light source. Note that

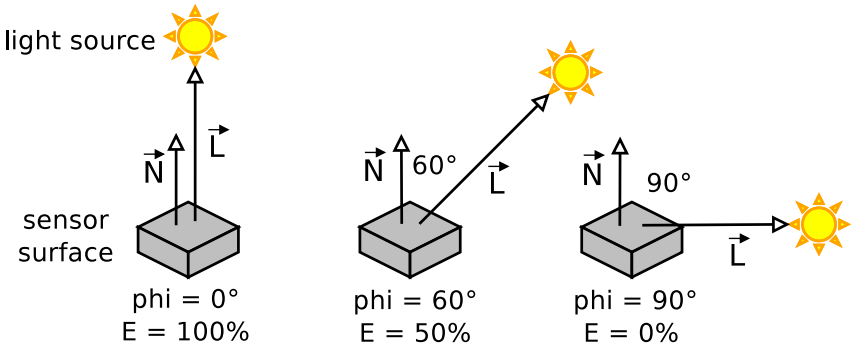


Figure 3.27: The irradiance (E) depends on the angle (phi) between the $N$ and $L$ vectors

`DirectionalLight` nodes don't have *location* fields; in this case Webots checks for obstacles between the sensor and an imaginary point located 1000m away in the direction opposite to the one indicated by the `direction` field of this `DirectionalLight`.

Like any other type of collision detection in Webots, the `LightSensor` occlusion detection is based on the `boundingObjects` of `Solid` nodes (or derived nodes). Therefore, even if it has a visible geometric structure, a `Solid` node cannot produce any occlusion if its `boundingObject` is not specified.


*The default value of the* `attenuation` *field of* `PointLight`*s and* `SpotLight`*s is* 1 0 0*. These values correspond to the VRML default, and are not appropriate for modeling the attenuation of a real lights. If a point or spot light radiates uniformly in all directions and there is no absorption, then the irradiance drops off in proportion to the square of the distance from the object. Therefore, for realistic modeling, the* `attenuation` *field of a light source should be changed to* 0 0 4*$\pi$*. If, in addition, the* `intensity` *field of the light is set to the radiant power [W] of a real point source (e.g., a light bulb), then the computed sensor irradiance* E *will approximate real world values in [W/m$^2$]. Finally, if the sensor's* `lookupTable` *is filled with correct calibration data, a fairly good approximation of the real world should be achieved.*


*If the calibration data for the* `lookupTable` *was obtained in lux (lx) or lumens per square meter (lm/m$^2$) instead of W/m$^2$, it makes sense to substitute the radiometry terms and units in this document with their photometry equivalents:* irradiance *becomes* illuminance*,* radiant power *becomes* luminous power *and* W *becomes* lm (lumen)*, etc.*

### 3.44.3 LightSensor Functions

---

#### NAME

wb_light_sensor_enable,
wb_light_sensor_disable,
wb_light_sensor_get_sampling_period,
wb_light_sensor_get_value – *enable, disable and read light sensor measurement*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/light_sensor.h>

void wb_light_sensor_enable (WbDeviceTag tag, int ms);

void wb_light_sensor_disable (WbDeviceTag tag);

int wb_light_sensor_get_sampling_period (WbDeviceTag tag);

double wb_light_sensor_get_value (WbDeviceTag tag);
```

#### DESCRIPTION

`wb_light_sensor_enable()` enables a light sensor measurement each `ms` milliseconds.

`wb_light_sensor_disable()` turns off the light sensor to save CPU time.

The `wb_light_sensor_get_sampling_period()` function returns the period given into the `wb_light_sensor_enable()` function, or 0 if the device is disabled.

`wb_light_sensor_get_value()` returns the most recent value measured by the specified light sensor. The returned value is the result of interpolating the irradiance $E$ as described above with the sensor's `lookupTable`.

## 3.45 LinearMotor

Derived from `Motor`.

```
LinearMotor {
  field SFString name       "linear motor" # used by
     wb_robot_get_device()
  field SFFloat  maxForce   10             # max force (N) : [0, inf)
}
```

### 3.45.1 Description

A `LinearMotor` node can be used to power a `SliderJoint`.

### 3.45.2 Field Summary

- The `name` field specifies the name identifier of the motor device. This the name to which `wb_robot_get_device()` can refer. It defaults to `"linear motor"`.
- The `maxForce` field specifies both the upper limit and the default value for the motor *available force*. The *available force* is the force that is available to the motor to perform the requested motions. The `wb_motor_set_available_force()` function can be

used to change the *available force* at run-time. The value of `maxForce` should always be zero or positive (the default is 10). A small `maxForce` value may result in a motor being unable to move to the target position because of its weight or other external forces.

## 3.46 Material

```
Material {
  SFFloat   ambientIntensity   0.2           # [0,1]
  SFColor   diffuseColor       0.8 0.8 0.8   # [0,1]
  SFColor   emissiveColor      0 0 0         # [0,1]
  SFFloat   shininess          0.2           # [0,1]
  SFColor   specularColor      0 0 0         # [0,1]
  SFFloat   transparency       0             # [0,1]
}
```

### 3.46.1 Description

The `Material` node specifies surface material properties for associated geometry nodes and is used by the VRML97 lighting equations during rendering. The fields in the `Material` node determine how light reflects off an object to create color.

### 3.46.2 Field Summary

- The `ambientIntensity` field specifies how much ambient light from the various light sources in the world this surface shall reflect. Ambient light is omni-directional and depends only on the number of light sources, not their positions with respect to the surface. Ambient color is calculated as `ambientIntensity` x `diffuseColor`.

- The `diffuseColor` field reflects all VRML97 light sources depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects.

- The `emissiveColor` field models ”glowing” objects. This can be useful for displaying pre-lit models (where the light energy of the room is computed explicitly), or for displaying scientific data.

- The `specularColor` and `shininess` fields determine the specular highlights (e.g., the shiny spots on an apple). When the angle from the light to the surface is close to the angle from the surface to the camera, the `specularColor` is added to the diffuse and ambient color calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.

- The transparency field specifies how "translucent" an object must be rendered: with 0.0 (the default) the object will appear completely opaque, and with 1.0 it will appear completely transparent. A transparent object doesn't cast or receive shadows. Webots performs dynamic alpha sorting according to the distance between the center of the objects (the local position of the parent Transform) and the viewpoint. Some occlusion issues can occur if two transparent objects intersect each other, or if the coordinate center of a transparent object is located outside the effectively rendered polygons, or if the sizes of nearby transparent objects differ significantly.

## 3.47 Motor

Derived from Device.

```
Motor {
  SFFloat maxVelocity  10 # (m/s or rad/s): (0,inf)
  SFVec3f controlPID   10 0 0 # PID gains: (0,inf), [0, inf), [0, inf)
  SFFloat acceleration -1 # (m/s^2 or rad/s^2): -1 or (0,inf)
  SFFloat minPosition  0  # (m or rad): (-inf,0] or [-pi, 0]
  SFFloat maxPosition  0  # (m or rad): [0,inf) or [0, pi]
}
```

### 3.47.1 Description

A Motor node is an abstract node (not instantiated) whose derived classes can be used in a mechanical simulation to power a joint hence producing a motion along, or around, one of its axes.

A RotationalMotor can power a HingeJoint (resp. a Hinge2Joint) when set inside the device (resp. device or device2) field of these nodes. It produces then a rotational motion around the choosen axis. Likewise, a LinearMotor can power a SliderJoint, producing a sliding motion along its axis.

### 3.47.2 Field Summary

- The maxVelocity field specifies both the upper limit and the default value for the motor *velocity*. The *velocity* can be changed at run-time with the wb_motor_set_velocity() function. The value should always be positive (the default is 10).

- The first coordinate of controlPID field specifies the initial value of the $P$ parameter, which is the *proportional gain* of the motor PID-controller. A high $P$ results in a large response to a small error, and therefore a more sensitive system. Note that by setting $P$ too

high, the system can become unstable. With a small *P*, more simulation steps are needed to reach the target position, but the system is more stable.

The second coordinate of `controlPID` field specifies the initial value of the *I* parameter, which is the *integral gain* of the motor PID-controller. The integral term of the PID controller is defined as the product of the error integral over time by *I*. This term accelerates the movement towards target position and eliminates the residual steady-state error which occurs with a pure proportional controller. However, since the integral term represents accumulated errors from the past, it can cause the present value to overshoot the target position.

The third coordinate `controlPID` field specifies the initial value of the *D* parameter, which is the *derivative gain* of the motor PID-controller. The derivative term of the PID-controller is defined as the product of the error derivative with respect to time by *D*. This term predicts system behavior and thus improves settling time and stability of the system.

The value of *P, I* and *D* can be changed at run-time with the `wb_motor_set_control_pid()` function.

- The `acceleration` field defines the default acceleration of the P-controller. A value of -1 (infinite) means that the acceleration is not limited by the P-controller. The acceleration can be changed at run-time with the `wb_motor_set_acceleration()` function.

- The `position` field represents the current *position* of the `Motor`, in radians or meters. For a rotational motor, `position` represents the current rotation angle in radians. For a linear motor, `position` represents the magnitude of the current translation in meters.

- The `minPosition` and `maxPosition` fields specify *soft limits* for the target position. These fields are described in more detail in the section "Motor Limits", see below.

### 3.47.3 Units

By *motor position*, we mean joint position as defined in `JointParameters`. Rotational motors units are expressed in *radians* while linear motors units are expressed in *meters*. See table 3.5:

| | Rotational | Linear |
|---|---|---|
| Position | rad (radians) | m (meters) |
| Velocity | rad/s (radians / second) | m/s (meters / second) |
| Acceleration | rad/s$^2$ (radians / second$^2$) | m/s$^2$ (meters / second$^2$) |

Table 3.5: Motor Units

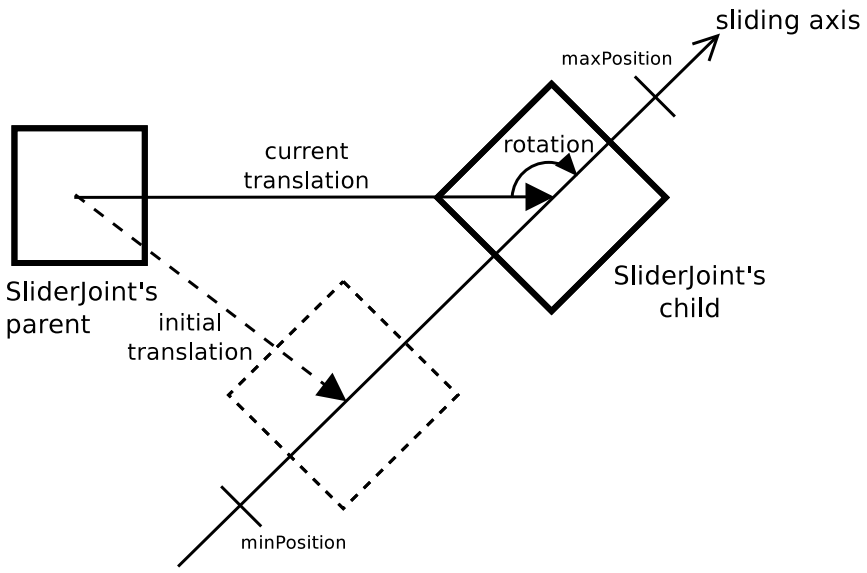


Figure 3.28: Linear Motor

### 3.47.4 Initial Transformation and Position

The `minPosition` and `maxPosition` are defined with respect to joint's zero position (see description of the `position` field in `JointParameters`).

### 3.47.5 Position Control

The standard way of operating a `Motor` is to control the position directly (*position control*). The user specifies a target position using the `wb_motor_set_position()` function, then the P-controller takes into account the desired velocity, acceleration and motor force in order to move the motor to the target position. See table 3.10.

In Webots, position control is carried out in three stages, as depicted in figure 3.30. The first stage is performed by the user-specified controller (1) that decides which position, velocity, acceleration and motor force must be used. The second stage is performed by the motor P-controller (2) that computes the current velocity of the motor $V_c$. Finally, the third stage (3) is carried out by the physics simulator (ODE joint motors).

At each simulation step, the PID-controller (2) recomputes the current velocity *Vc* according to following algorithm:

```
error = Pt - Pc;
error_integral += error * ts;
error_derivative = (previous_error - error) / ts;
Vc = P * error + D * error_derivative + I * error_integral ;
if (abs(Vc) > Vd)
```

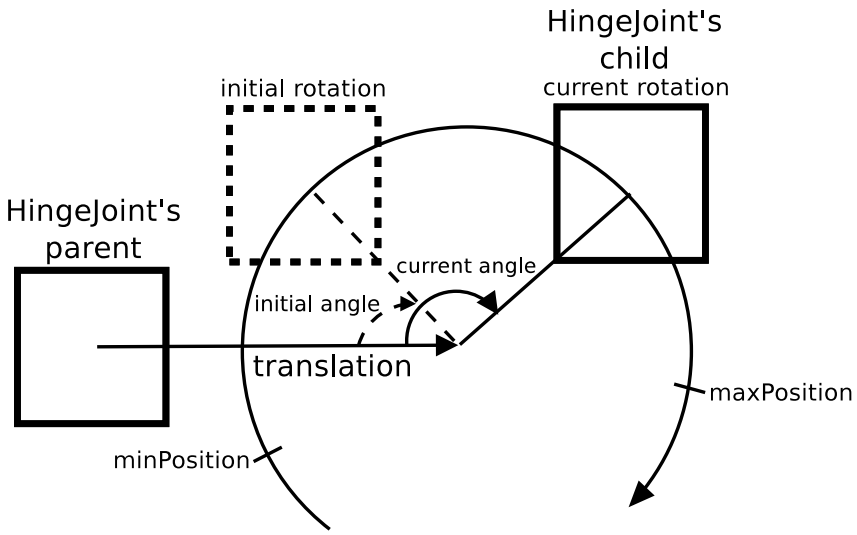


Figure 3.29: Rotational Motor

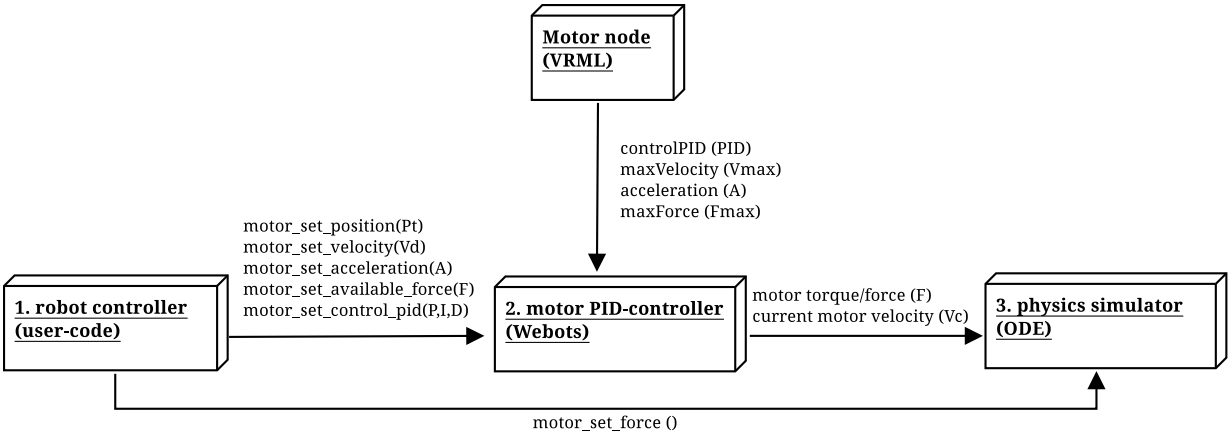


Figure 3.30: Motor control

```
  Vc = sign(Vc) * Vd;
if (A != -1) {
  a = (Vc - Vp) / ts;
  if (abs(a) > A)
    a = sign(a) * A;
  Vc = Vp + a * ts;
}
```

where $V_c$ is the current motor velocity in rad/s or m/s, $P, I$ and $D$ are the PID-control gains specified in the `controlPID` field, or set with `wb_motor_set_control_pid()`, $P_t$ is the *target position* of the motor set by the function `wb_motor_set_position()`, $P_c$ is the current motor position as reflected by the `position` field, $V_d$ is the desired velocity as specified by the `maxVelocity` field (default) or set with `wb_motor_set_velocity()`, $a$ is the acceleration required to reach *Vc* in one time step, $V_p$ is the motor velocity of the previous time step, $t_s$ is the duration of the simulation time step as specified by the `basicTimeStep` field of the `WorldInfo` node (converted in seconds), and $A$ is the acceleration of the motor as specified by the `acceleration` field (default) or set with `wb_motor_set_acceleration()`.


*error_integral and previous_error are both reset to* 0 *after every call of* `wb_motor_set_control_pid()`.

### 3.47.6 Velocity Control

The motors can also be used with *velocity control* instead of *position control*. This is obtained with two function calls: first the `wb_motor_set_position()` function must be called with `INFINITY` as a position parameter, then the desired velocity, which may be positive or negative, must be specified by calling the `wb_motor_set_velocity()` function. This will initiate a continuous motor motion at the desired speed, while taking into account the specified acceleration and motor force. Example:

```
wb_motor_set_position(motor, INFINITY);
wb_motor_set_velocity(motor, 6.28);  // 1 rotation per second
```

`INFINITY` is a C macro corresponding to the IEEE 754 floating point standard. It is implemented in the C99 specifications as well as in C++. In Java, this value is defined as `Double.POSITIVE_INFINITY`. In Python, you should use `float('inf')`. Finally, in Matlab you should use the `inf` constant.

### 3.47.7 Force and Torque Control

The position (resp. velocity) control described above are performed by the Webots PID-controller and ODE's joint motor implementation (see ODE documentation). As an alternative, Webots

does also allow the user to directly specify the amount of force (resp. torque) that must be applied by a Motor. This is achieved with the wb_motor_set_force() (resp. wb_motor_set_torque()) function which specifies the desired amount of forces (resp. torques) and switches off the PID-controller and motor force (resp. motor torque). A subsequent call to wb_motor_set_position() restores the original *position control*. Some care must be taken when using *force control*. Indeed the force (resp. torque) specified with wb_motor_set_force() (resp. wb_motor_set_torque()) is applied to the Motor continuously. Hence the Motor will infinitely accelerate its rotational or linear motion and eventually *explode* unless a functional force control (resp. torque control) algorithm is used.

| | position control | velocity control | force or torque control |
|---|---|---|---|
| uses PID-controller | yes | no | no |
| wb_motor_set_position() | * specifies the desired position | should be set to INFINITY | switches to position/velocity control |
| wb_motor_set_velocity() | specifies the max velocity | * specifies the desired velocity | is ignored |
| wb_motor_set_acceleration() | specifies the max acceleration | specifies the max acceleration | is ignored |
| wb_motor_set_available_force() (resp. wb_motor_set_available_torque()) | specifies the available force (resp. torque) | specifies the available force (resp. torque) | specifies the max force (resp. max torque) |
| wb_motor_set_force() (resp. wb_motor_set_torque()) | switches to force control (resp. torque control) | switches to force control (resp. torque control) | * specifies the desired force (resp. torque) |

Table 3.6: Motor Control Summary

### 3.47.8 Motor Limits

The minPosition and maxPosition fields define the *soft limits* of the motor. Motor zero position and joint zero position coincide (see description of the position field in JointParameters). Soft limits specify the *software* boundaries beyond which the PID-controller will not attempt to move. If the controller calls wb_motor_set_position() with a target position that exceeds the soft limits, the desired target position will be clipped in order to fit into the soft limit range. Since the initial position of the motor is always zero, minPosition must always be negative or zero, and maxPosition must always be positive or zero. When both minPosition and maxPosition are zero (the default), the soft limits are deactivated. Note that the soft limits can be overstepped when an external force which exceeds the motor force is applied to the motor. For example, it is possible that the weight of a robot exceeds the motor force that is required to hold it up.

Finally, note that when both soft (`minPosition` and `maxPosition`) and hard limits (`minStop` and `maxStop`, see `JointParameters`) are activated, the range of the soft limits must be included in the range of the hard limits, such that `minStop <= minValue` and `maxStop>= maxValue`. Moreover a simulation instability can appear if `position` is exactly equal to one of the bounds defined by the `minStop` and `maxStop` fields at the simulation startup. Warnings are displayed if theses rules are not respected.

### 3.47.9 Motor Functions

---

#### NAME

wb_motor_set_position,
wb_motor_set_velocity,
wb_motor_set_acceleration,
wb_motor_set_available_force,
wb_motor_set_available_torque,
wb_motor_set_control_pid,
wb_motor_get_min_position,
wb_motor_get_max_position – *change the parameters of the PID-controller*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/motor.h>

void wb_motor_set_position (WbDeviceTag tag, double position);

double wb_motor_get_target_position (WbDeviceTag tag);

void wb_motor_set_velocity (WbDeviceTag tag, double velocity);

void wb_motor_set_acceleration (WbDeviceTag tag, double acceleration);

void wb_motor_set_available_force (WbDeviceTag tag, double force);

void wb_motor_set_available_torque (WbDeviceTag tag, double torque);

void wb_motor_set_control_pid (WbDeviceTag tag, double p);

double wb_motor_get_min_position (WbDeviceTag tag);

double wb_motor_get_max_position (WbDeviceTag tag);
```

#### DESCRIPTION

The `wb_motor_set_position()` function specifies a new target position that the PID-controller will attempt to reach using the current velocity, acceleration and motor torque/force parameters.

This function returns immediately (asynchronous) while the actual motion is carried out in the background by Webots. The target position will be reached only if the physics simulation allows it, that means, if the specified motor force is sufficient and the motion is not blocked by obstacles, external forces or the motor's own spring force, etc. It is also possible to wait until the Motor reaches the target position (synchronous) like this:


**language: C**

```c
void motor_set_position_sync(WbDeviceTag
   tag_motor, WbDeviceTag tag_sensor, double
   target, int delay) {
  const double DELTA = 0.001;  // max tolerated
     difference
  wb_motor_set_position(tag_motor, target);
  wb_position_sensor_enable(tag_sensor, TIME_STEP
     );
  double effective;  // effective position
  do {
    wb_robot_step(TIME_STEP);
    delay -= TIME_STEP;
    effective = wb_position_sensor_get_value(
       tag_sensor);
  }
  while (fabs(target - effective) > DELTA &&
     delay > 0);
  wb_position_sensor_disable(tag_sensor);
}
```

The INFINITY (*#include <math.h>*) value can be used as the second argument to the wb_motor_set_position() function in order to enable an endless rotational (or linear) motion. The current values for velocity, acceleration and motor torque/force are taken into account. So for example, wb_motor_set_velocity() can be used for controlling the velocity of the endless rotation:


**language: C**

```c
// velocity control
wb_motor_set_position(tag, INFINITY);
wb_motor_set_velocity(tag, desired_speed);  //
   rad/s
```


**language: C++**
*In C++ use* `std::numeric_limits<double>::infinity()` *instead of INFINITY*


**language: Java**
*In Java use* `Double.POSITIVE_INFINITY` *instead of INFINITY*


**language: Python**
*In Python use* `float('+inf')` *instead of INFINITY*


**language: Matlab**
*In MATLAB use* `inf` *instead of INFINITY*

The `wb_motor_get_target_position()` function allows to get the target position. This value matches with the argument given to the last `wb_motor_set_position()` function call.

The `wb_motor_set_velocity()` function specifies the velocity that motor should reach while moving to the target position. In other words, this means that the motor will accelerate (using the specified acceleration, see below) until the target velocity is reached. The velocity argument passed to this function cannot exceed the limit specified in the `maxVelocity` field.

The `wb_motor_set_acceleration()` function specifies the acceleration that the PID-controller should use when trying to reach the specified velocity. Note that an infinite acceleration is obtained by passing -1 as the `acc` argument to this function.

The `wb_motor_set_available_force()` (resp. `wb_motor_set_available_torque()`) function specifies the maximum force (resp. torque) that will be available to the motor to carry out the requested motion. The motor force/torque specified with this function cannot exceed the value specified in the `maxForce`/`maxTorque` field.

The `wb_motor_set_control_pid()` function changes the values of the gains $P, I$ and $D$ in the PID-controller. These parameters are used to compute the current motor velocity $V_c$ from the current position $P_c$ and target position $P_t$, such that $V_c = P * error + I * error\_integral + D * error\_derivative$ where $error = P_t - P_c$. With a small $P$, a long time is needed to reach the target position, while too large a $P$ can make the system unstable. The default value of $P, I$ and $D$ are specified by the `controlPID` field of the corresponding `Motor` node.

The `wb_motor_get_[min|max]_position()` functions allow to get the values of respectively the `minPosition` and the `maxPosition` fields.

## NAME

wb_motor_enable_force_feedback,
wb_motor_get_force_feedback,
wb_motor_get_force_feedback_sampling_period,
wb_motor_disable_force_feedback,
wb_motor_enable_torque_feedback,
wb_motor_get_torque_feedback,
wb_motor_get_torque_feedback_sampling_period,
wb_motor_disable_torque_feedback – *get the motor force or torque currently used by a motor*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/motor.h>

void wb_motor_enable_force_feedback (WbDeviceTag tag, int ms);

void wb_motor_disable_force_feedback (WbDeviceTag tag);

int wb_motor_get_force_feedback_sampling_period (WbDeviceTag tag);

double wb_motor_get_force_feedback (WbDeviceTag tag);

void wb_motor_enable_torque_feedback (WbDeviceTag tag, int ms);

void wb_motor_disable_torque_feedback (WbDeviceTag tag);

int wb_motor_get_torque_feedback_sampling_period (WbDeviceTag tag);

double wb_motor_get_torque_feedback (WbDeviceTag tag);
```

## DESCRIPTION

The `wb_motor_enable_force_feedback()` (resp. `wb_motor_enable_torque_feedback()`) function activates force (resp. torque) feedback measurements for the specified motor. A new measurement will be performed each `ms` milliseconds; the result must be retrieved with the `wb_motor_get_force_feedback()` (resp. `wb_motor_get_torque_feedback()`) function.

The `wb_motor_get_force_feedback()` (resp. `wb_motor_get_torque_feedback()`) function returns the most recent motor force (resp. torque) measurement. This function measures the amount of motor force (resp. torque) that is currently being used by the motor in order to achieve the desired motion or hold the current position. For a ”rotational” motor, the returned value is a torque [N*m]; for a ”linear” motor, the value is a force [N]. The returned value is an approximation computed by the physics engine, and therefore it may be inaccurate. The returned value normally does not exceed the available motor force (resp. torque) specified with `wb_motor_set_force()` (resp. `wb_motor_set_torque()`). The default value is provided by the `maxForce` (resp. `maxTorque` field. Note that this function measures the *current motor*

*force* (resp. *torque*) exclusively, all other external or internal forces (resp. torques) that may apply to the motor are ignored. In particular, `wb_motor_get_force_feedback()` (resp. `wb_motor_get_torque_feedback()`) does not measure:

- The spring and damping forces that apply when the `springConstant` or `dampingConstant` fields are non-zero.
- The force specified with the `wb_motor_set_force()` (resp. `wb_motor_set_torque()`) function.
- The *constraint forces or torques* that restrict the motor motion to one degree of freedom (DOF). In other words, the forces or torques applied outside of the motor DOF are ignored. Only the forces or torques applied in the DOF are considered. For example, in a "linear" motor, a force applied at a right angle to the sliding axis is completely ignored. In a "rotational" motor, only the torque applied around the rotation axis is considered.

Note that this function applies only to *physics-based* simulation. Therefore, the `physics` and `boundingObject` fields of the `Motor` node must be defined for this function to work properly.

If `wb_motor_get_force_feedback()` (resp. `wb_motor_get_torque_feedback()`) was not previously enabled, the return value is undefined.

The `wb_motor_get_force_feedback_sampling_period()` (resp. `wb_motor_get_torque_feedback_sampling_period()`) function returns the period given in the `wb_motor_enable_force_feedback()` (resp. `wb_motor_enable_force_feedback()`) function, or 0 if the device is disabled.

---

## NAME

wb_motor_set_force,
wb_motor_set_torque – *direct force or torque control*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/motor.h>

void wb_motor_set_force (WbDeviceTag tag, double force);

void wb_motor_set_torque (WbDeviceTag tag, double torque);
```

## DESCRIPTION

As an alternative to the PID-controller, the `wb_motor_set_force()` (resp. `wb_motor_set_torque()`) function allows the user to directly specify the amount of force (resp. torque)

that must be applied by a motor. This function bypasses the PID-controller and ODE joint motors; it adds the force to the physics simulation directly. This allows the user to design a custom controller, for example a PID controller. Note that when `wb_motor_set_force()` (resp. `wb_motor_set_torque()`) is invoked, this automatically resets the force previously added by the PID-controller.

In a "rotational" motor, the *torque* parameter specifies the amount of torque [N.m] that will be applied around the motor rotation axis. In a "linear" motor, the *force* parameter specifies the amount of force [N] that will be applied along the sliding axis. A positive *force* (resp. *torque*) will move the bodies in the positive direction, which corresponds to the direction of the motor when the `position` field increases. When invoking `wb_motor_set_force()` (resp. `wb_motor_set_torque()`), the specified *force* (resp. *torque*) parameter cannot exceed the current motor force (resp. torque) of the motor specified with `wb_motor_set_force()` (resp. `wb_motor_set_torque()`) and defaulting to the value of the `maxForce` (resp. `maxTorque`) field.

Note that this function applies only to *physics-based* simulation. Therefore, the `physics` and `boundingObject` fields of the `Motor` node must be defined for this function to work properly.

It is also possible, for example, to use this function to implement springs or dampers with controllable properties. The example in `projects/samples/howto/worlds/force_control.wbt` demonstrates the usage of `wb_motor_set_force()` for creating a simple spring and damper system.

---

### NAME

wb_motor_get_type – *get the motor type*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/motor.h>

int wb_motor_get_type (WbDeviceTag tag);
```

### DESCRIPTION

This function allows to retrieve the motor type defined by the `type` field. If the value of the `type` field is "linear", this function returns WB_LINEAR, and otherwise it returns WB_ANGULAR.

## 3.48 Pen

Derived from `Device`.

| Motor.type | return value |
|---|---|
| ”rotational” | WB_ANGULAR |
| ”linear” | WB_LINEAR |

Table 3.7: Return values for the `wb_motor_get_type()` function

```
Pen {
  SFColor   inkColor     0 0 0    # [0,1]
  SFFloat   inkDensity   0.5      # [0,1]
  SFFloat   leadSize     0.002
  SFFloat   maxDistance  0.0      # >= 0.0
  SFBool    write        TRUE
}
```

### 3.48.1 Description

The `Pen` node models a pen attached to a mobile robot, typically used to show the trajectory of the robot. The paint direction of the `Pen` device coincides with the -*y*-axis of the node. So, it can be adjusted by modifying the rotation and translation fields of the `Solid` node. By setting the `maxDistance` field is possible to define the range of the `Pen` and paint only on objects close to the device. For example with a small value of `maxDistance` you can simulate the real behaviour of a pen or pencil that writes only on physical contact. If `maxDistance` is set to 0 (default value), the range will be unlimited.

In order to be paintable, an object should be made up of a `Solid` node containing a `Shape` with a valid `Geometry`. Even if a `ImageTexture` is already defined, the painture is applied over the texture without modifying it.

The precision of the painting action mainly depends on the `subdivision` field of the `Geometry` node. A high `subdivision` value increases the number of polygons used to represent the geometry and thus allows a more precise texture mapping, but it will also slow down the rendering of the scene. On the other hand, with a poor texture mapping, the painted area could be shown at a different position than the expected one. In case of `IndexedFaceSet`, the precision can be improved by defining a texture mapping and setting the `texCoord` and `texCoordIndex` fields. In fact, if no texture mapping or an invalid one is given, the system will use a default general mapping.

An example of a textured floor used with a robot equipped with a pen is given in the `pen.wbt` example world (located in the `projects/samples/devices/worlds` directory of Webots).


*The* `inkEvaporation` *field of the* `WorldInfo` *node controls how fast the ink evaporates (disappears).*


*The drawings performed by a pen can be seen by infra-red distance sensors. Hence, it is possible to implement a robotics experiment where a robot draws a line on the floor with a pen and a second robot performs a line following behavior with the line drawn by the first robot.*

### 3.48.2 Field Summary

- `inkColor`: define the color of the pen's ink. This field can be changed from the pen API, using the `wb_pen_set_ink_color()` function.
- `inkDensity`: define the density of the color of the ink. The value should be in the range [0,1]. This field can also be changed from the pen API, using the `wb_pen_set_ink_color()` function.
- `leadSize`: define the width of the "tip" of the pen. This allows the robot to write a wider or narrower track.
- `maxDistance`: define the maximal distance between the `Pen` device and a paintable object and allows to simulate write-on-contact behaviors. A value smaller or equal 0 represents an unlimited painting range.
- `write`: this boolean field allows the robot to enable or disable writing with the pen. It is also switchable from the pen API, using the `wb_pen_write()` function.

### 3.48.3 Pen Functions

---

#### NAME

wb_pen_write – *enable or disable pen writing*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/pen.h>

void wb_pen_write (WbDeviceTag tag, bool write);
```

#### DESCRIPTION

`wb_pen_write()` allows the user to switch a pen device on or off to disable or enable writing. If the `write` parameter is *true*, the specified `tag` device will write; if `write` is *false*, it won't.

---

### NAME

wb_pen_set_ink_color – *change the color of a pen's ink*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/pen.h>

void wb_pen_set_ink_color (WbDeviceTag tag, int color, double density);
```

### DESCRIPTION

`wb_pen_set_ink_color()` changes the current ink color of the specified `tag` device. The `color` is a 32 bit integer value which defines the new color of the ink in the 0xRRGGBB hexadecimal format (i.e., 0x000000 is black, 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue, 0xFFA500 is orange, 0x808080 is grey 0xFFFFFF is white, etc.). The `density` parameter defines the ink density, with 0 meaning transparent ink and 1 meaning completely opaque ink.

### EXAMPLE

```
wb_pen_set_ink_color(pen,0xF01010,0.9);
```

The above statement will change the ink color of the indicated pen to some red color.


**language: Matlab**
*In the Matlab version of* `wb_pen_set_ink_color()`*, the* `color` *argument must be a vector containing the three RGB components:* `[RED GREEN BLUE]`*. Each component must be a value between 0.0 and 1.0. For example the vector* `[1 0 1]` *represents the magenta color.*

## 3.49 Physics

```
Physics {
  SFFloat     density             1000      # (kg/m^3) -1 or > 0
  SFFloat     mass                -1        # (kg) -1 or > 0
  SFVec3f     centerOfMass        0 0 0     # (-inf,inf)
  MFVec3f     inertiaMatrix       []        # empty or 2 values
  SFNode      damping             NULL      # optional damping node
}
```

### 3.49.1 Description

The Physics node allows to specify parameters for the physics simulation engine. Physics nodes are used in most Webots worlds with the exception of some purely kinematics-based simulations. The Physics node specifies the mass, the center of gravity and the mass distribution, thus allowing the physics engine to create a *body* and compute realistic forces.

A Physics node can be placed in a Solid node (or any node derived from Solid). The presence or absence of a Physics node in the physics field of a Solid defines whether the Solid will have a *physics* or a *kinematic* behavior.


*In older Webots versions,* coulombFriction, bounce, bounceVelocity *and* forceDependentSlip *fields used to be specified in* Physics *nodes. Now these values must be specified in* ContactProperties *nodes. For compatibility reasons, these fields are still present in the* Physics *but they should no longer be used.*

### 3.49.2 Field Summary

- The density field can be used to define the density of the containing Solid. The value of the density field should be a positive number number or -1. A -1 value indicates that the dentity is not known, in this case the mass field (see below) must be specified. If the density is specified (different from -1) then the total mass of the Solid is calculated by multiplying the specified density with the total volume of the geometrical primitives composing the boundingObject. Note that Webots ignores if the geometrical primitives intersect or not, the volume of each primitive is simply added to the total volume and finally multiplied by the density.

- The mass field can be used to specify the total mass of the containing Solid. The value of the mass field should be a positive number or -1. A -1 value indicates that the total mass is not known, in this case the density field (see above) must be specified. If the mass is known, e.g., indicated in the specifications of the robot, then it is more accurate to specify the mass rather than the density.

- The centerOfMass field defines the position of the center of mass of the solid. It is expressed in meters in the relative coordinate system of the Solid node. If centerOfMass field is different from [0 0 0], then the center of mass is depicted as a dark red/green/blue cross in Webots 3D-window.

- The inertiaMatrix field can be used to manually specify the inertia matrix of the Solid. This field can either be empty (the default) or contain exactly 2 vectors. If this field is empty, Webots will compute the inertia matrix automatically according to the position and orientation of the geometrical primitives in boundingObject.

If this field contains 2 vectors, these values specify the inertia matrix of the Solid. If the inertia matrix is specified then the mass field must also be specified. The first vector [I11, I22, I33] represents the *principals moments of inertia* and the second vector [I12, I13, I23] represents the *products of inertia*. Together these values form a 3x3 inertia matrix:

```
[ I11 I12 I13 ]
[ I12 I22 I23 ]
[ I13 I23 I33 ]
```

The Ixx values are expressed in kg*m$^2$. The principals moments of inertia must be positive. The inertia matrix is defined with respect to the centerOfMass of the Solid. Internally, these 6 values are passed unchanged to the dMassSetParameters() ODE function.

- The damping field allows to specify a Damping node that defines the velocity damping parameters to be applied to the Solid.

### 3.49.3 How to use Physics nodes?

If it contains a Physics node, a Solid object will be simulated in *physics* mode. The *physics* simulation mode takes into account the simulation of the forces that act on the bodies and the properties of these bodies, e.g., mass and moment of inertia. On the contrary, if its physics field is NULL, then the Solid will be simulated in *kinematics* mode. The *kinematics* mode simulates the objects motions without considering the forces that cause the motion. For example in *kinematics* mode an object can reach the desired speed immediately while in *physics* mode the inertial resistance will cause this object to accelerate progressively. It is usually not necessary to specify all the Physics nodes in a Webots world. Whether to use or not a Physics node in a particular case depends on what aspect of the real world your want to model in your simulation.

#### In passive objects

If a passive object should never move during a simulation then you should leave its physics field empty. In this case no contact force will be simulated on this object and hence it will never move. This is perfect for modeling walls or the floor. Furthermore the floor should always be designed without Physics node anyway, because otherwise it would fall under the action of gravity.

On the contrary, if a passive object needs to be pushed, kicked, dropped, etc. then it should have a Physics node. So for example, if you want to design a soccer game where the ball needs to be kicked and roll, then you will need to add a Physics node to the ball. Similarly, in a box pushing or stacking simulation, you will need to specify the Physics nodes for the boxes so that the friction and gravity forces are applied to these objects.

### In robots

Articulated robot, humanoids, vehicles and so on, are built as hierarchies of Solid nodes (or subclasses of Solid). The contact and friction forces generated by legs or wheels are usually a central aspect of the simulation of robot locomotion. Similarly, the contact and friction forces of a grasping robotic hand or gripper is crucial for the simulation of such devices. Therefore the mechanical body parts of robots (eg., legs, wheels, arms, hands, etc) need in general to have Physics nodes.


*It is possible to set the* physics *field of a* Robot *or a top* Solid *to* NULL *in order to pin its base to the static environment. This can be useful for the simulation of a robot arm whose base segment is anchored in a fixed place. More generally, you can define a larger* static base *rooted at a given top* Solid*. Indeed you can define a subtree starting from this top* Solid *and whose all* Solid *nodes have no* Physics *nodes.*


*The* DifferentialWheels *robot is a special case: it can move even if it does not have* Physics *nodes. That's because Webots uses a special* kinematics *algorithm for* DifferentialWheels *robots without* Physics*. However, if the* Physics *nodes are present then Webots uses the regular* physics *simulation algorithms.*

### Implicit solid merging and joints

By Solid *child* of a given Solid node, we mean either a node directly placed into the children list or a Solid node located into the endPoint field of a Joint placed in the children list. We extend this definition to nested Groups starting from the Solid children list and containing Joints or Solids.

If a Solid child in the above sense is not related to its Solid parent by a joint while both have a Physics node, they are *merged at the physics engine level*: ODE will be given only one body to represent both parent and child. This process is recursive and stops at the highest ancestor which have a joint pointing to an upper Solid or just before the highest ancestor without Physics node. This way modelling a rigid assembly of Solids won't hurt physics simulation speed even if it aggregates numerous components.


*When solid merging applies, only the highest ancestor of the rigid assembly has a body (a non null* dBodyID *in ODE terms) which holds the physical properties of the assembly. This may impact the way you design a* physics plugin*s.*

When designing the robot tree structure, there is one important rule to remember about the Physics nodes: *If a Solid node has a parent and a child with a Physics node then it must also have a Physics node* (1). A consequence of this rule is that, in a robot tree structure, only leaf nodes and nodes included in the *static basis* (see first note above) can have a NULL physics field. In addition top nodes (Robot, DifferentialWheels or Supervisor) do usually have Physics because this is required to allow any of their children to use the *physics* simulation.

Note that each Physics node adds a significant complexity to the world: as a consequence the simulation speed decreases. Therefore the number of Physics nodes should be kept as low as possible. Fortunately, even with a complex wheeled or articulated robot some of the physics fields can remain empty (NULL). This is better explained with an example. Let's assume that you want to design an articulated robot with two legs. Your robot model may look like this (very simplified):

```
Robot {
  ...
  children [
    DEF LEG1_HINGE HingeJoint  {
      ...
      endPoint DEF LEG1 Solid {
        physics Physics {
        }
      }
    }
    DEF LEG2_HINGE HingeJoint {
      ...
      endPoint DEF LEG2 Solid {
        physics Physics {
        }
      }
    }
  ]
  physics Physics {
  }
}
```

The legs need Physics nodes because the forces generated by their contact with the floor will allow the robot to move. If you would leave the legs without Physics, then no contact forces would be generated and therefore the robot would not move. Now, according to rule (1), because the legs have Physics nodes, their parent (the Robot node) must also have a Physics node. If the Physics node of the Robot was missing, the simulation would not work, the legs would fall off, etc.

Now suppose you would like to add a Camera to this robot. Let's also assume that the physical properties of this camera are not relevant for this simulation, say, because the mass of the camera is quite small and because we want to ignore potential collisions of the camera with other objects.

In this case, you should leave the `physics` field of the camera empty. So the model with the camera would look like this:

```
Robot {
  ...
  children [
    DEF CAM Camera {
      ...
    }
    DEF LEG1_HINGE HingeJoint  {
      ...
      endPoint DEF LEG1 Solid {
        ...
        physics Physics {
        }
      }
    }
    DEF LEG2_HINGE HingeJoint {
      ...
      endPoint DEF LEG2 Solid {
        physics Physics {
        }
      }
    }
  ]
  physics Physics {
  }
}
```

Now suppose that the camera needs to be motorized, e.g., it should rotate horizontally. Then the camera must simply be placed in the `endPoint` field of `HingeJoint` node that controls its horizontal position. This time again, the physical properties of the camera motor are apparently unimportant. If we assume that the mass of the camera motor is small and that its inertia is not relevant, then the camera `Physics` node can also be omitted:

```
Robot {
  ...
  children [
    DEF CAMERA_HINGE HingeJoint {
      ...
      device DEF CAM_MOTOR RotationalMotor {
          ...
      }
      endPoint DEF CAM Camera {
        ...
      }
    }
```

```
    DEF LEG1_HINGE HingeJoint  {
      ...
      endPoint DEF LEG1 Solid {
        ...
        physics Physics {
        }
      }
    }
    DEF LEG2_HINGE HingeJoint {
      ...
      endPoint DEF LEG2 Solid {
        physics Physics {
        }
      }
    }
  ]
  physics Physics {
  }
}
```

### Devices

Most device nodes work without Physics node. But a Physics node can optionally be used if one wishes to simulate the weight and inertia of the device. So it is usually recommended to leave the physics field of a device empty, unless it represents a significant mass or volume in the simulated robot. This is true for these devices: Accelerometer, Camera, Compass, DistanceSensor, Emitter, GPS, LED, LightSensor, Pen, and Receiver.


*The InertialUnit and Connector nodes work differently. Indeed, they require the presence of a Physics node in their parent node to be functional. It is also possible to specify a Physics node of the device but this adds an extra body to the simulation.*
*The TouchSensor is also a special case: it needs a Physics node when it is used as "force" sensor; it does not necessarily need a Physics node when it is only used as "bumper" sensor.*

## 3.50 Plane

```
Plane {
  SFVec2f     size        1 1     # (-inf,inf)
}
```

### 3.50.1 Description

The Plane node defines a plane in 3D-space. The plane's normal vector is the *y*-axis of the local coordinate system. The plane can be used as graphical object or as collision detection object.

When a plane is used as graphical object, the size field specifies the dimensions of the graphical representation. Just like the other graphical primitives, it is possible to apply a Material (e.g., a texture) to a plane.

When a plane is used as collision detection object (in a boundingObject) then the size field is ignored and the plane is considered to be infinite. The Plane node is the ideal primitive to simulate, e.g., the floor or infinitely high walls. Unlike the other collision detection primitives, the Plane can only be used in static objects (a static object is an object without a Physics node). Note that Webots ignores collision between planes, so planes can safely cut each other. Note that a Plane node is in fact not really a plane: it's a half-space. Anything that is moving inside the half-space will be ejected out of it. This means that planes are only planes from the perspective of one side. If you want your plane to be reversed, rotate it by $\pi$ using a Transform node.

## 3.51 PointLight

Derived from Light.

```
PointLight {
  SFVec3f   attenuation   1 0 0     # [0,inf)
  SFVec3f   location      0 0 0     # (-inf,inf)
  SFFloat   radius        100       # [0,inf)
}
```

### 3.51.1 Description

The PointLight node specifies a point light source at a 3D location in the local coordinate system. A point light source emits light equally in all directions. It is possible to put a PointLight on board a mobile robot to have the light move with the robot.

A PointLight node's illumination drops off with distance as specified by three attenuation coefficients. The final attenuation factor is calculated as follows: *att = 1/(attenuation[0] + attenuation[1] * r + attenuation[2] * $r^2$)*, where *r* is the distance from the light to the surface being illuminated. The default is no attenuation. When PointLight nodes are used together with LightSensor, it is recommended to change the default attenuation to a more realistic [*0 0 4*$\pi$] in order to more accurately model physical reality. Indeed, if a point source radiates light uniformly in all directions and there is no absorption, then the irradiance drops off in proportion to the square of the distance from the surface.

Contrary to the VRML specifications, the `attenuation` and the `ambientIntensity` fields cannot be set simultaneously.

## 3.52 PositionSensor

Derived from `Device`.

```
PositionSensor {
  SFFloat resolution  -1
}
```

### 3.52.1 Description

A `PositionSensor` node can be used in a mechanical simulation to monitor a joint position. The position sensor can be inserted in the `device` field of a `HingeJoint`, a `Hinge2Joint`, or a `SliderJoint`. Depending on the `Joint` type, it will measure the angular position in radians or the linear position in meters.

### 3.52.2 Field Summary

- `resolution`: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field accepts any value in the interval (0.0, inf).

### 3.52.3 PositionSensor Functions

---

#### NAME

wb_position_sensor_enable,
wb_position_sensor_disable,
wb_position_sensor_get_sampling_period,
wb_position_sensor_get_value,
wb_position_sensor_get_type – *enable, disable and read position sensor measurement*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/position_sensor.h>
```

```
void wb_position_sensor_enable (WbDeviceTag tag, int ms);
void wb_position_sensor_disable (WbDeviceTag tag);
int wb_position_sensor_get_sampling_period (WbDeviceTag tag);
double wb_position_sensor_get_value (WbDeviceTag tag);
int wb_position_sensor_get_type (WbDeviceTag tag);
```

#### DESCRIPTION

`wb_position_sensor_enable()` enables a measurement of the joint position each `ms` milliseconds.

`wb_position_sensor_disable()` turns off the position sensor to save CPU time.

The `wb_position_sensor_get_sampling_period()` function returns the period given into the `wb_position_sensor_enable()` function, or 0 if the device is disabled.

`wb_position_sensor_get_value()` returns the most recent value measured by the specified position sensor. Depending on the type, it will return a value in radians (angular position sensor) or in meters (linear position sensor).

`wb_position_sensor_get_type()` returns the type of the position sensor. It will return `WB_ANGULAR` if the sensor is associated with a `HingeJoint` or a `Hinge2Joint` node, and `WB_LINEAR` if it is associated with a `SliderJoint`.

## 3.53 Propeller

```
Propeller {
  field SFVec3f shaftAxis       1 0 0 # (m)
  field SFVec3f centerOfThrust  0 0 0 # (m)
  field SFVec2f thrustConstants 1 0   # Ns^2/rad : (-inf, inf), Ns^2/(
     m*rad) : (-inf, inf)
  field SFVec2f torqueConstants 1 0   # Nms^2/rad : (-inf, inf), Ns^2/
     rad : (-inf, inf)
  field SFNode device NULL            # RotationalMotor
  field SFNode fastHelix NULL         # Solid node containing a
     graphical representation for rotation speed > 24    rad/s (720
     rpm)
  field SFNode slowHelix NULL         # Solid node containing a
     graphical representation for rotation speed <= 24    rad/s
}
```

### 3.53.1 Description

The `Propeller` node can be used to model a marine or an aircraft propeller. When its `device` field is set with a `RotationalMotor`, the propeller turns the motor angular velocity into a

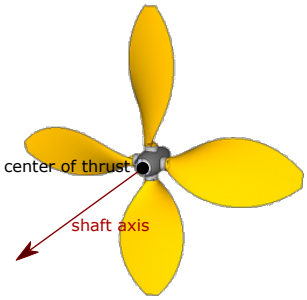


Figure 3.31: Propeller

thrust and a (resistant) torque. The resultant thrust is the product of a real number T by the unit length shaft axis vector defined in the `shaftAxis` field, with T given by the formula

```
T = t1 * |omega| * omega - t2 * |omega| * V
```

and where t1 and t2 are the constants specified in the `thrustConstants` field, omega is the motor angular velocity and V is the component of the linear velocity of the center of thrust along the shaft axis. The thrust is applied at the point specified within the `centerOfThrust` field.

The resultant torque is the product of a real number Q by the unit length shaft axis vector, with Q given by the formula

```
Q = q1 * |omega| * omega - q2 * |omega| * V
```

and where q1 and q2 are the constants specified in the `torqueConstants` field.

The above formulae are based on "Guidance and Control of Ocean Vehicles" from Thor I. Fossen and "Helicopter Performance, Stability, and Control" from Raymond W. Prouty.

The example `propeller.wbt` located in the `projects/samples/devices/worlds` directory of Webots shows three different helicopters modeled with `Propeller` nodes.

### 3.53.2 Field Summary

- `shaftAxis`: defines the axis along which the resultant thrust and torque will be exerted, see figure 3.31.
- `centerOfThrust`: defines the point where the generated thrust applies, see figure 3.31.
- `thrustConstants` and `torqueConstants`: coefficients used to define the resultant thrust and torque as functions of the motor angular velocity and the linear speed of adavance, see above formulae.
- `device`: this field has to be set with a `RotationalMotor` in order to control the propeller.

- `fastHelix` and `slowHelix`: if not NULL, these fields must be set with `Solid` nodes. The corresponding `Solid` nodes define the graphical representation of the propeller according to its motor's angular velocity omega: if —omega— $> 24\,\pi$ rad /s, only the `Solid` defined in `fastHelix` is visible, otherwise only the `Solid` defined in `slowHelix` is visible.

## 3.54 Receiver

Derived from `Device`.

```
Receiver {
  SFString  type                "radio"  # or "serial" or "infra-red"
  SFFloat   aperture            -1       # -1 or [0,2pi]
  SFInt32   channel             0        # [-1,inf)
  SFInt32   baudRate            -1       # -1 or [0,inf)
  SFInt32   byteSize            8        # [8,inf)
  SFInt32   bufferSize          -1       # -1 or [0,inf)
  SFFloat   signalStrengthNoise 0        # [0,inf)
  SFFloat   directionNoise      0        # [0,inf)
}
```

### 3.54.1 Description

The `Receiver` node is used to model radio, serial or infra-red receivers. A `Receiver` node must be added to the children of a robot or supervisor. Please note that a `Receiver` can receive data but it cannot send it. In order to achieve bidirectional communication, a robot needs to have both an `Emitter` and a `Receiver` on board.

### 3.54.2 Field Summary

- `type`: type of signal: "radio", "serial" or "infra-red". Signals of type "radio" (the default) and "serial" are transmitted without taking obstacles into account. Signals of type "infra-red," however, do take potential obstacles between the emitter and the receiver into account. Any solid object (solid, robots, etc ...) with a defined bounding object is a potential obstacle for an "infra-red" communication. The structure of the emitting or receiving robot itself will not block an "infra-red" transmission. Currently, there is no implementation difference between the "radio" and "serial" types.

- `aperture`: opening angle of the reception cone (in radians); for "infra-red" only. The receiver can only receive messages from emitters currently located within its reception cone. The cone's apex is located at the origin ([0 0 0]) of the receiver's coordinate system and the

cone's axis coincides with the $z$-axis of the receiver coordinate system (see figure 3.16 in section 3.26). An `aperture` of -1 (the default) is considered to be infinite, meaning that a signal can be received from any direction. For "radio" receivers, the `aperture` field is ignored.

- `channel`: reception channel. The value is an identification number for an "infra-red" receiver or a frequency for a "radio" receiver. Normally, both emitter and receiver must use the same channel in order to be able to communicate. However, the special -1 channel number allows the receiver to listen to all channels.

- `baudRate`: the baud rate is the communication speed expressed in bits per second. It should be the same as the speed of the emitter. Currently, this field is ignored.

- `byteSize`: the byte size is the number of bits used to represent one byte of transmitted data (usually 8, but may be more if control bits are used). It should be the same size as the emitter byte size. Currently, this field is ignored.

- `bufferSize`: size (in bytes) of the reception buffer. The size of the received data should not exceed the buffer size at any time, otherwise data may be lost. A `bufferSize` of -1 (the default) is regarded as unlimited buffer size. If the previous data have not been read when new data are received, the previous data are lost.

- `signalStrengthNoise`: standard deviation of the gaussian noise added to the signal strength returned by `wb_receiver_get_signal_strength`. The noise is proportionnal to the signal strength, e.g., a `signalStrengthNoise` of 0.1 will add a noise with a standard deviation of 0.1 for a signal strength of 1 and 0.2 for a signal strength of 2.

- `directionNoise`: standard deviation of the gaussian noise added to each of the components of the direction returned by `wb_receiver_get_emitter_direction`. The noise is not dependent on the distance between emitter-receiver.

### 3.54.3 Receiver Functions

---

#### NAME

wb_receiver_enable,
wb_receiver_disable,
wb_receiver_get_sampling_period – *enable and disable receiver*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/receiver.h>
```

```
void wb_receiver_enable (WbDeviceTag tag, int ms);

void wb_receiver_disable (WbDeviceTag tag);

int wb_receiver_get_sampling_period (WbDeviceTag tag);
```

### DESCRIPTION

`wb_receiver_enable()` starts the receiver listening for incoming data packets. Data reception is activated in the background of the controller's loop at a rate of once every `ms` milliseconds. Incoming data packet are appended to the tail of the reception queue (see figure 3.32). Incoming data packets will be discarded if the receiver's buffer size (specified in the `Receiver` node) is exceeded. To avoid buffer overflow, the data packets should be read at a high enough rate by the controller program. The function `wb_receiver_disable()` stops the background listening.

The `wb_receiver_get_sampling_period()` function returns the period given into the `wb_receiver_enable()` function, or 0 if the device is disabled.

---

### NAME

wb_receiver_get_queue_length,
wb_receiver_next_packet – *check for the presence of data packets in the receivers queue*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/receiver.h>

int wb_receiver_get_queue_length (WbDeviceTag tag);

void wb_receiver_next_packet (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_receiver_get_queue_length()` function returns the number of data packets currently present in the receiver's queue (see figure 3.32).

The `wb_receiver_next_packet()` function deletes the head packet. The next packet in the queue, if any, becomes the new head packet. The user must copy useful data from the head packet, before calling `wb_receiver_next_packet()`. It is illegal to call `wb_receiver_next_packet()` when the queue is empty (`wb_receiver_get_queue_length()` == 0). Here is a usage example:

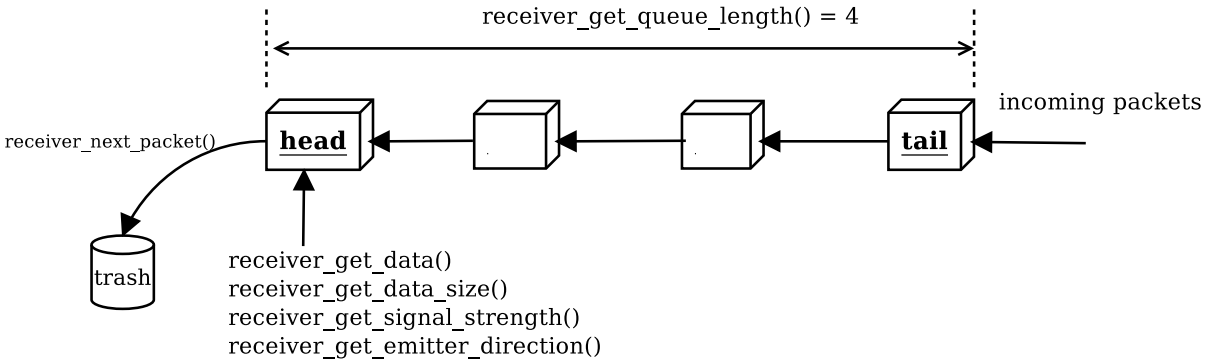


Figure 3.32: Receiver's packet queue

**language: C**

```c
while (wb_receiver_get_queue_length(tag) > 0) {
  const char *message = wb_receiver_get_data(tag);
  const double *dir = wb_receiver_get_emitter_direction(tag);
  double signal = wb_receiver_get_signal_strength(tag);
  printf("received: %s (signal=%g, dir=[%g %g %g])\n",
         message, signal, dir[0], dir[1], dir[2]);
  wb_receiver_next_packet(tag);
}
```

This example assumes that the data (*message*) was sent in the form of a null-terminated string. The Emitter/Receiver API does not put any restriction on the type of data that can be transmitted. Any user chosen format is suitable, as long as emitters and receivers agree.

*Webots' Emitter/Receiver API guarantees that:*

- Packets will be received in the same order they were sent
- Packets will be transmitted atomically (no byte-wise fragmentation)

However, the Emitter/Receiver API does not guarantee a specific schedule for the transmission. Sometimes several packets may be bundled and received together. For example, imagine a simple

setup where two robots have an Emitter and a Receiver on board. If both robots use the same controller time step and each one sends a packet at every time step, then the Receivers will receive, on average, one data packet at each step, but they may sometimes get zero packets, and sometimes two! Therefore it is recommend to write code that is tolerant to variations in the transmission timing and that can deal with the eventuality of receiving several or no packets at all during a particular time step. The wb_receiver_get_queue_length() function should be used to check how many packets are actually present in the Receiver's queue. Making assumptions based on timing will result in code that is not robust.

---

## NAME

wb_receiver_get_data,
wb_receiver_get_data_size – *get data and size of the current packet*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/receiver.h>

const void *wb_receiver_get_data (WbDeviceTag tag);

int wb_receiver_get_data_size (WbDeviceTag tag);
```

## DESCRIPTION

The wb_receiver_get_data() function returns the data of the packet at the head of the reception queue (see figure 3.32). The returned data pointer is only valid until the next call to the function wb_receiver_next_packet(). It is illegal to call wb_receiver_get_data() when the queue is empty (wb_receiver_get_queue_length() == 0). The Receiver node knows nothing about that structure of the data being sent but its byte size. The emitting and receiving code is responsible to agree on a specific format.

The wb_receiver_get_data_size() function returns the number of data bytes present in the head packet of the reception queue. The *data size* is always equal to the *size* argument of the corresponding emitter_send_packet() call. It is illegal to call wb_receiver_get_data_size() when the queue is empty (wb_receiver_get_queue_length() == 0).


**language: Python**

*The* `getData()` *function returns a string. Similarly to the* `sendPacket()` *function of the* `Emitter` *device, using the functions of the struct module is recommended for sending primitive data types. Here is an example for getting the data:*

```
1 import struct
2 #...
3 message=receiver.getData()
4 dataList=struct.unpack("chd",message)
```

**language: Matlab**
*The Matlab* `wb_receiver_get_data()` *function returns a MATLAB* libpointer. *The receiving code is responsible for extracting the data from the* libpointer *using MATLAB's* `setdatatype()` *and* `get()` *functions. Here is an example on how to send and receive a 2x3 MATLAB matrix.*

```
% sending robot
emitter = wb_robot_get_device('emitter');

A = [1, 2, 3; 4, 5, 6];
wb_emitter_send(emitter, A);
```

```
% receiving robot
receiver = wb_robot_get_device('receiver');
wb_receiver_enable(receiver, TIME_STEP);

while wb_receiver_get_queue_length(receiver) > 0
  pointer = wb_receiver_get_data(receiver);
  setdatatype(pointer, 'doublePtr', 2, 3);
  A = get(pointer, 'Value');
  wb_receiver_next_packet(receiver);
end
```


*The MATLAB* `wb_receiver_get_data()` *function can also take a second argument that specifies the type of the expected data. In this case the function does not return a* libpointer *but an object of the specified type, and it is not necessary to call* `setdatatype()` *and* `get()`. *For example* `wb_receiver_get_data()` *can be used like this:*

```
% receiving robot
receiver = wb_robot_get_device('receiver');
wb_receiver_enable(receiver, TIME_STEP);

while wb_receiver_get_queue_length(receiver) > 0
  A = wb_receiver_get_data(receiver, 'double');
  wb_receiver_next_packet(receiver);
end
```

*The available types are 'uint8', 'double' and 'string'. More sophisticated data typed must be accessed explicitly using* `setdatatype()` *and* `get()`.

## NAME

wb_receiver_get_signal_strength,

wb_receiver_get_emitter_direction – *get signal strength and emitter direction*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/receiver.h>

double wb_receiver_get_signal_strength (WbDeviceTag tag);

const double *wb_receiver_get_emitter_direction (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_receiver_get_signal_strength()` function operates on the head packet in the receiver's queue (see figure 3.32). It returns the simulated signal strength at the time the packet was transmitted. This signal strength is equal to the inverse of the distance between the emitter and the receiver squared. In other words, $s = 1 / r^2$, where $s$ is the signal strength and $r$ is the distance between emitter and receiver. It is illegal to call this function if the receiver's queue is empty (`wb_receiver_get_queue_length()` == 0).

The function `wb_receiver_get_emitter_direction()` also operates on the head packet in the receiver's queue. It returns a normalized (length=1) vector that indicates the direction of the emitter with respect to the receiver's coordinate system. The three vector components indicate the $x$, $y$ , and $z$-directions of the emitter, respectively. For example, if the emitter was exactly in front of the receiver, then the vector would be [0 0 1]. In the usual orientation used for 2D simulations (robots moving in the $xz$-plane and the $y$ -axis oriented upwards), a positive $x$ -component indicates that the emitter is located to the left of the receiver while a negative $x$ -component indicates that the emitter is located to the right. The returned vector is valid only until the next call to `wb_receiver_next_packet()`. It is illegal to call this function if the receiver's queue is empty (`wb_receiver_get_queue_length()` == 0).


**language: Python**
`getEmitterDirection()` *returns the vector as a list containing three floats.*

---

### NAME

wb_receiver_set_channel,
wb_receiver_get_channel – *set and get the receiver's channel.*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/receiver.h>
```

```
void wb_receiver_set_channel (WbDeviceTag tag, int channel);
int wb_receiver_get_channel (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_receiver_set_channel()` function allows a receiver to change its reception channel. It modifies the `channel` field of the corresponding `Receiver` node. Normally, a receiver can only receive data packets from emitters that use the same channel. However, the special WB_CHANNEL_BROADCAST value can be used to listen simultaneously to all channels.

The `wb_receiver_get_channel()` function returns the current channel number of the receiver.


**language: C++, Java, Python**
*In the oriented-object APIs, the WB_CHANNEL_BROADCAST constant is available as static integer of the* `Receiver` *class (Receiver::CHANNEL_BROADCAST).*

## 3.55 Robot

Derived from `Solid`.

```
Robot {
  SFString   controller         "void"
  SFString   controllerArgs     ""
  SFString   data               ""
  SFBool     synchronization    TRUE
  MFFloat    battery            []
  SFFloat    cpuConsumption     0    # [0,inf)
  SFBool     selfCollision      FALSE
  SFBool     showRobotWindow    FALSE
  SFString   robotWindow        ""
  SFString   remoteControl      ""
}
```

Direct derived nodes: `DifferentialWheels`, `Supervisor`.

### 3.55.1 Description

The `Robot` node can be used as basis for building a robot, e.g., an articulated robot, a humanoid robot, a wheeled robot... If you want to build a two-wheels robot with differential-drive you should also consider the `DifferentialWheels` node. If you would like to build a robot with supervisor capabilities use the `Supervisor` node instead (Webots PRO license required).

### 3.55.2 Field Summary

- `controller`: name of the controller program that the simulator must use to control the robot. This program is located in a directory whose name is equal to the field's value. This directory is in turn located in the `controllers` subdirectory of the current project directory. For example, if the field value is "my_controller" then the controller program should be located in `my_project/controllers/my_controller/my_controller[.exe]`. The `.exe` extension is added on the Windows platforms only.

- `controllerArgs`: string containing the arguments (separated by space characters) to be passed to the `main()` function of the C/C++ controller program or the `main()` method of the Java controller program.

- `data`: this field may contain any user data, for example parameters corresponding to the configuration of the robot. It can be read from the robot controller using the `wb_robot_get_data()` function and can be written using the `wb_robot_set_data()` function. It may also be used as a convenience for communicating between a robot and a supervisor without implementing a Receiver / Emitter system: The supervisor can read and write in this field using the generic supervisor functions for accessing fields.

- `synchronization`: if the value is `TRUE` (default value), the simulator is synchronized with the controller; if the value is `FALSE`, the simulator runs as fast as possible, without waiting for the controller. The `wb_robot_get_synchronization()` function can be used to read the value of this field from a controller program.

- `battery`: this field should contain three values: the first one corresponds to the present energy level of the robot in Joules (*J*), the second is the maximum energy the robot can hold in Joules, and the third is the energy recharge speed in Watts (*[W]=[J]/[s]*). The simulator updates the first value, while the other two remain constant. *Important:* when the current energy value reaches zero, the corresponding controller process terminates and the simulated robot stops all motion.

  Note: *[J]=[V].[A].[s] and [J]=[V].[A.h]/3600*

- `cpuConsumption`: power consumption of the CPU (central processing unit) of the robot in Watts.

- `selfCollision`: setting this field to TRUE will enable the detection of collisions within the robot and apply the corresponding contact forces, so that the robot limbs cannot cross each other (provided that they have a `Physics` node). This is useful for complex articulated robots for which the controller doesn't prevent inner collisions. Enabling self collision is, however, likely to decrease the simulation speed, as more collisions will be generated during the simulation. Note that only collisions between non-consecutive solids will be detected. For consecutive solids, e.g., two solids attached to each other with a joint, no collision detection is performed, even if the self collision is enabled. The reason is that this type of collision detection is usually not wanted by the user, because a very accurate

design of the bounding objects of the solids would be required. To prevent two consecutive solid nodes from penetrating each other, the `minStop` and `maxStop` fields of the corresponding joint node should be adjusted accordingly. Here is an example of a robot leg with self collision enabled:

```
Thigh (solid)
  |
Knee (joint)
  |
Leg (solid)
  |
Ankle (joint)
  |
Foot (solid)
```

In this example, no collision is detected between the "Thigh" and the "Leg" solids because they are consecutive, e.g., directly joined by the "Knee". In the same way no collision detection takes place between the "Leg" and the "Foot" solids because they are also consecutive, e.g., directly joined by the "Ankle". However, collisions may be detected between the "Thigh" and the "Foot" solids, because they are non-consecutive, e.g., they are attached to each other through an intermediate solid ("Leg"). In such an example, it is probably a good idea to set `minStop` and `maxStop` values for the "Knee" and "Ankle" joints.

- `showRobotWindow`: defines whether the robot window should be shown at the startup of the controller. If yes, the related entry point function of the robot window controller plugin (`wbw_show()`) is called as soon as the controller is initialized.

- `robotWindow`: defines the path of the robot window controller plugin used to display the robot window. If the `robotWindow` field is empty, the default generic robot window is loaded. The search algorithm works as following: Let $(VALUE) be the value of the `robotWindow` field, let $(EXT) be the shared library file extension of the OS (".so", ".dll" or ".dylib"), let $(PREFIX) be the shared library file prefix of the OS ("" on windows and "lib" on other OS), let $(PROJECT) be the current project path, let $(WEBOTS) be the webots installation path, and let $(...) be a recursive search, then the first existing file will be used as absolute path:

  $(PROJECT)/plugins/robot_windows/$(VALUE)/$(PREFIX)$(VALUE)$(EXT)

  $(WEBOTS)/resources/$(...)/plugins/robot_windows/$(VALUE)/$(PREFIX)$(VALUE)$(EXT)

- `remoteControl`: defines the path of the remote-control controller plugin used to remote control the real robot. The search algorithm is identical to the one used for the robotWindow field, except that the subdirectory of `plugins` is `remote_controls` rather than `robot_windows`.

### 3.55.3 Synchronous versus Asynchronous controllers

The `synchronization` field specifies if a robot controller must be synchronized with the simulator or not.

If `synchronization` is `TRUE` (the default), the simulator will wait for the controller's `wb_robot_step()` whenever necessary to keep the simulation and the controller synchronized. So for example if the simulation step (`WorldInfo.basicTimeStep`) is 16 ms and the control step (`wb_robot_step()`) is 64 ms, then Webots will always execute precisely 4 simulation steps during one control step. After the 4th simulation step, Webots will wait for the controller's next control step (call to `wb_robot_step(64)`).

If `synchronization` is `FALSE`, the simulator will run as fast a possible without waiting for the control step. So for example, with the same simulation step (16 ms) and control step (64 ms) as before, if the simulator has finished the 4th simulation step but the controller has not yet reached the call to `wb_robot_step(64)`, then Webots will not wait; instead it will continue the simulation using the latest actuation commands. Hence, if `synchronization` is `FALSE`, the number of simulation steps that are executed during a control step may vary; this will depend on the current simulator and controller speeds and on the current CPU load, and hence the outcome of the simulation may also vary. Note that if the number of simulation steps per control step varies, this will appear as a variations of the "speed of the physics" in the controller's point of view, and this will appear as a variation of the robot's reaction speed in the user's point of view.

So generally the `synchronization` field should be set to `TRUE` when robust control is required. For example if a motion (or `.motionfile`) was designed in synchronous mode then it may appear completely different in asynchronous mode. The asynchronous mode is currently used only for the robot competitions, because in this case it is necessary to limit the CPU time allocated to each participating controller. Note that it is also possible to combine synchronous and asynchronous controllers, e.g., for the robot competitions generally the `Supervisor` controller is synchronous while the contestants controllers are asynchronous. Asynchronous controllers may also be recommended for networked simulations involving several robots distributed over a computer network with an unpredictable delay (like the Internet).

### 3.55.4 Robot Functions

---

#### NAME

wb_robot_step,
wb_robot_init,
wb_robot_cleanup – *controller step, initialization and cleanup functions*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

int wb_robot_step (int ms);

void wb_robot_init ();

void wb_robot_cleanup ();
```

## DESCRIPTION

The `wb_robot_step()` function is crucial and must be used in every controller. This function synchronizes the sensor and actuator data between Webots and the controllers. If the `wb_robot_step()` function is not called then there will be no actuation in Webots and no update of the sensors in the controller.

The `ms` parameter specifies the number of milliseconds that must be simulated until the `wb_robot_step()` function returns. Note that this is not real time but virtual (simulation) time, so this is not like calling the system's `sleep()`. In fact the function may return immediately, however the important point is that when it returns `ms` milliseconds of simulation will have elapsed. In other words the physics will have run for `ms` milliseconds and hence the motors may have moved, the sensor values may have changed, etc. Note that `ms` parameter must be a multiple of the `WorldInfo.basicTimeStep`.

If this function returns -1, this indicates that Webots wishes to terminate the controller. This happens when the user hits the **Revert** button or quits Webots. So if your code needs to do some cleanup, e.g., flushing or closing data files, etc., it is necessary to test this return value and take proper action. The controller termination cannot be vetoed: one second later the controller is killed by Webots. So only one second is available to do the cleanup.

If the `synchronization` field is TRUE, this function always returns 0 (or -1 to indicate termination). If the `synchronization` field is FALSE, the return value can be different from 0: Let `controller_time` be the current time of the controller and let `dt` be the return value. Then `dt` may be interpreted as follows:

- if `dt` = 0, then the asynchronous behavior was equivalent to the synchronous behavior.
- if 0 <= `dt` <= `ms`, then the actuator values were set at `controller_time` + `dt`, and the sensor values were measured at `controller_time` + `ms`, as requested. It means that the step actually lasted the requested number of milliseconds, but the actuator commands could not be executed on time.
- if `dt` > `ms`, then the actuators values were set at `controller_time` + `dt`, and the sensor values were also measured at `controller_time` + `dt`. It means that the requested step duration could not be respected.

The C API has two additional functions `wb_robot_init()` and `wb_robot_cleanup()`. There is not equivalent of the `wb_robot_init()` and `wb_robot_cleanup()` functions in

the Java, Python, C++ and MATLAB APIs. In these languages the necessary initialization and cleanup of the controller library is done automatically.

The `wb_robot_init()` function is used to initialize the Webots controller library and enable the communication with the Webots simulator. Note that the `wb_robot_init()` function must be called before any other Webots API function.

Calling the `wb_robot_cleanup()` function is the clean way to terminate a C controller. This function frees the various resources allocated by Webots on the controller side. In addition `wb_robot_cleanup()` signals the termination of the controller to the simulator. As a consequence, Webots removes the controller from the simulation which can continue normally with the execution of the other controllers (if any). If a C controller exits without calling `wb_robot_cleanup()`, then its termination will not be signalled to Webots. In this case the simulation will remain blocked (sleeping) on the current step (but only if this `Robot`'s `synchronization` field is TRUE). Note that the call to the `wb_robot_cleanup()` function must be the last API function call in a C controller. Any subsequent Webots API function call will give unpredictable results.

### SIMPLE C CONTROLLER EXAMPLE

**language: C**


```c
#include <webots/robot.h>

#define TIME_STEP 32

static WbDeviceTag my_sensor, my_led;

int main() {
  /* initialize the webots controller library */
  wb_robot_init();

  // get device tags
  my_sensor = wb_robot_get_device("my_distance_sensor");
  my_led = wb_robot_get_device("my_led");

  /* enable sensors to read data from them */
  wb_distance_sensor_enable(my_sensor, TIME_STEP);

  /* main control loop: perform simulation steps of 32 milliseconds */
  /* and leave the loop when the simulation is over */
  while (wb_robot_step(TIME_STEP) != -1) {

    /* Read and process sensor data */
    double val = wb_distance_sensor_get_value(my_sensor);

    /* Send actuator commands */
    wb_led_set(my_led, 1);
  }

  /* Add here your own exit cleanup code */

  wb_robot_cleanup();

  return 0;
}
```

### NAME

wb_robot_get_device – *get a unique identifier to a device*

### SYNOPSIS [Matlab]

```
#include <webots/robot.h>

WbDeviceTag wb_robot_get_device (const char *name);
```

### DESCRIPTION

This function returns a unique identifier for a device corresponding to a specified `name`. For example, if a robot contains a `DistanceSensor` node whose `name` field is "ds1", the function will return the unique identifier of that device. This `WbDeviceTag` identifier will be used subsequently for enabling, sending commands to, or reading data from this device. If the specified device is not found, the function returns 0.

### SEE ALSO

`wb_robot_step`.

---

### NAME

Robot::getAccelerometer,
Robot::getCamera,
Robot::getCompass,
Robot::getConnector,
Robot::getDistanceSensor,
Robot::getDisplay,
Robot::getEmitter,
Robot::getGPS,
Robot::getGyro,
Robot::getInertialUnit,
Robot::getLED,
Robot::getLightSensor,
Robot::getMotor,
Robot::getPen,
Robot::getPositionSensor,
Robot::getReceiver,
Robot::getServo,
Robot::getTouchSensor – *get the instance of a robot's device*

### SYNOPSIS [C++] [Java] [Python]

```
#include <webots/Robot.hpp>
Accelerometer *Robot::getAccelerometer (const std::string &name);
Camera *Robot::getCamera (const std::string &name);
Compass *Robot::getCompass (const std::string &name);
Connector *Robot::getConnector (const std::string &name);
Display *Robot::getDisplay (const std::string &name);
DistanceSensor *Robot::getDistanceSensor (const std::string &name);
Emitter *Robot::getEmitter (const std::string &name);
GPS *Robot::getGPS (const std::string &name);
Gyro *Robot::getGyro (const std::string &name);
InertialUnit *Robot::getInertialUnit (const std::string &name);
LightSensor *Robot::getLightSensor (const std::string &name);
Motor *Robot::getMotor (const std::string &name);
Pen *Robot::getPen (const std::string &name);
PositionSensor *Robot::getPositionSensor (const std::string &name);
Receiver *Robot::getReceiver (const std::string &name);
Servo *Robot::getServo (const std::string &name);
TouchSensor *Robot::getTouchSensor (const std::string &name);
```

### DESCRIPTION

These functions return a reference to an object corresponding to a specified `name`. Depending on the called function, this object can be an instance of a `Device` subclass. For example, if a robot contains a `DistanceSensor` node whose `name` field is "ds1", the function `getDistanceSensor` will return a reference to a `DistanceSensor` object. If the specified device is not found, the function returns `NULL` in C++, `null` in Java or the `none` in Python.

### SEE ALSO

`wb_robot_step`.

---

### NAME

wb_robot_get_device_by_index – *get the devices by introspection*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

WbDeviceTag wb_robot_get_device_by_index (int index);

int wb_robot_get_number_of_devices ();
```

### DESCRIPTION

These functions allows to get the robot devices by introspection. Indeed they allow to get the devices from an internal flat list storing the devices. The size of this list matches with the number of devices. The order of this list matches with their declaration in the scene tree.

If `index` is out of the bounds of the list index (from 0 to wb_robot_get_number_of_devices() - 1) then the returned WbDeviceTag is equal to 0.

The following example shows a typical example of introspection. It is used with the device API allowing to retrieve some information from a WbDeviceTag.

```
1  int n_devices = wb_robot_get_number_of_devices();
2  int i;
3  for(i=0; i<n_devices; i++) {
4    WbDeviceTag tag = wb_robot_get_device_by_index(i);
5
6    const char *name = wb_device_get_name(tag);
7    WbNodeType type = wb_device_get_node_type(tag);
8
9    // do something with the device
10   printf("Device #%d name = %s\n", i, name);
11
12   if (type == WB_NODE_CAMERA) {
13     // do something with the camera
14     printf("Device #%d is a camera\n", i);
15   }
16 }
```

---

### NAME

wb_robot_battery_sensor_enable,
wb_robot_battery_sensor_disable,
wb_robot_get_battery_sampling_period,
wb_robot_battery_sensor_get_value – *battery sensor function*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

void wb_robot_battery_sensor_enable (int ms);

void wb_robot_battery_sensor_disable ();

double wb_robot_battery_sensor_get_value ();

int wb_robot_get_battery_sampling_period (WbDeviceTag tag);
```

### DESCRIPTION

These functions allow you to measure the present energy level of the robot battery. First, it is necessary to enable battery sensor measurements by calling the `wb_robot_battery_sensor_enable()` function. The `ms` parameter is expressed in milliseconds and defines how frequently measurements are performed. After the battery sensor is enabled a value can be read from it by calling the `wb_robot_battery_sensor_get_value()` function. The returned value corresponds to the present energy level of the battery expressed in Joules (*J*).

The `wb_robot_battery_sensor_disable()` function should be used to stop battery sensor measurements.

The `wb_robot_get_battery_sampling_period()` function returns the period given into the `wb_robot_battery_sensor_enable()` function, or 0 if the device is disabled.

---

### NAME

wb_robot_get_basic_time_step – *returns the value of the basicTimeStep field of the WorldInfo node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

double wb_robot_get_basic_time_step ();
```

### DESCRIPTION

This function returns the value of the `basicTimeStep` field of the `WorldInfo` node.

---

### NAME

wb_robot_get_mode – *get operating mode, simulation vs. real robot*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

int wb_robot_get_mode ();

void wb_robot_set_mode (int mode, void *arg);
```

### DESCRIPTION

The `wb_robot_get_mode` function returns an integer value indicating the current operating mode for the controller.

The `wb_robot_set_mode` function allows to switch between the simulation and the remote control mode. When switching to the remote-control mode, the `wbr_start` function of the remote control plugin is called. The argument `arg` is passed directly to the `wbr_start` function (more information in the user guide).

The integers can be compared to the following enumeration items:

| Mode | Purpose |
|---|---|
| WB_MODE_SIMULATION | simulation mode |
| WB_MODE_CROSS_COMPILATION | cross compilation mode |
| WB_MODE_REMOTE_CONTROL | remote control mode |

Table 3.8: Helper enumeration to interpret the integer argument and return value of the `wb_robot_[gs]et_mode()` functions

---

### NAME

wb_robot_get_name – *return the name defined in the robot node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

const char *wb_robot_get_name ();
```

### DESCRIPTION

This function returns the name as it is defined in the name field of the robot node (Robot, DifferentialWheels, Supervisor, etc.) in the current world file. The string returned should not be deallocated, as it was allocated by the `libController` shared library and will be deallocated when the controller terminates. This function is very useful to pass some arbitrary parameter from a world file to a controller program. For example, you can have the same controller code behave differently depending on the name of the robot. This is illustrated in the `soccer.wbt`

sample demo, where the goal keeper robot runs the same control code as the other soccer players, but its behavior is different because its name was tested to determine its behavior (in this sample world, names are ”b3” for the blue goal keeper and ”y3” for the yellow goal keeper, whereas the other players are named ”b1”, ”b2”, ”y1” and ”y2”). This sample world is located in the `projects/samples/demos/worlds` directory of Webots.

---

### NAME

wb_robot_get_model – *return the model defined in the robot node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

const char *wb_robot_get_model ();
```

### DESCRIPTION

This function returns the model string as it is defined in the model field of the robot node (Robot, DifferentialWheels, Supervisor, etc.) in the current world file. The string returned should not be deallocated, as it was allocated by the `libController` shared library and will be deallocated when the controller terminates.

---

### NAME

wb_robot_get_data – *return the data defined in the robot node* ,
wb_robot_set_data – *set the data defined in the robot node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

const char * wb_robot_get_data ();

void wb_robot_set_data (const char *data);
```

### DESCRIPTION

The `wb_robot_get_data` function returns the string contained in the `data` field of the robot node.

The `wb_robot_set_data` function set the string contained in the `data` field of the robot node.

---

### NAME

wb_robot_get_type – *return the type of the robot node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/nodes.h> #include <webots/robot.h>

WbNodeType wb_robot_get_type ();
```

### DESCRIPTION

This function returns the type of the current mode (WB_NODE_ROBOT, WB_NODE_SUPERVISOR or WB_NODE_DIFFERENTIAL_WHEELS).

---

### NAME

wb_robot_get_project_path – *return the full path of the current project*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

const char *wb_robot_get_project_path ();
```

### DESCRIPTION

This function returns the full path of the current project, that is the directory which contains the worlds and controllers subdirectories (among others) of the current simulation world. It doesn't include the final directory separator char (slash or anti-slash). The returned pointer is a UTF-8 encoded char string. It should not be deallocated.

---

### NAME

wb_robot_get_controller_name,
wb_robot_get_controller_arguments – *return the content of the Robot::controller and Robot::controllerArgs fields*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>
```

```
const char *wb_robot_get_controller_name ();

const char *wb_robot_get_controller_arguments ();
```

### DESCRIPTION

These functions return the content of respectively the Robot::controller and the Robot::controllerArgs fields.

---

### NAME

wb_robot_get_synchronization – *return the value of the synchronization field of the Robot node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

bool wb_robot_get_synchronization ();
```

### DESCRIPTION

This function returns the boolean value corresponding to the synchronization field of the Robot node.

---

### NAME

wb_robot_get_time – *return the current simulation time in seconds*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

double wb_robot_get_time ();
```

### DESCRIPTION

This function returns the current simulation time in seconds. This correspond to the simulation time displayed in the speedometer located in the main toolbar. It does not matter whether the controller is synchronized or not.

---

## NAME

wb_robot_keyboard_enable,
wb_robot_keyboard_disable,
wb_robot_keyboard_get_key – *keyboard reading function*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/robot.h>

void wb_robot_keyboard_enable (int ms);

void wb_robot_keyboard_disable ();

int wb_robot_keyboard_get_key ();
```

## DESCRIPTION

These functions allow you to read a key pressed on the computer keyboard from a controller program while the main window of Webots is selected and the simulation is running. First, it is necessary to enable keyboard input by calling the `wb_robot_keyboard_enable()` function. The `ms` parameter is expressed in milliseconds, and defines how frequently readings are updated. After the enable function is called, values can be read by calling the `wb_robot_keyboard_get_key()` function repeatedly until this function returns 0. The returned value, if non-null, is a key code corresponding to a key currently pressed. If no modifier (shift, control or alt) key is pressed, the key code is the ASCII code of the corresponding key or a special value (e.g., for the arrow keys). However, if a modifier key was pressed, the ASCII code (or special value) can be obtained by applying a binary AND between to the `WB_ROBOT_KEYBOARD_KEY` mask and the returned value. In this case, the returned value is the result of a binary OR between one of `WB_ROBOT_KEYBOARD_SHIFT`, `WB_ROBOT_KEYBOARD_CONTROL` or `WB_ROBOT_KEYBOARD_ALT` and the ASCII code (or the special value) of the pressed key according to which modifier key was pressed simultaneously.

If no key is currently pressed, the function will return 0. Calling the `wb_robot_keyboard_get_key()` function a second time will return either 0 or the key code of another key which is currently simultaneously pressed. The function can be called up to 7 times to detect up to 7 simultaneous keys pressed. The `wb_robot_keyboard_disable()` function should be used to stop the keyboard readings.


**language: C++**
*The keyboard predefined values are located into a (static) enumeration of the Robot class. For example,* `Robot.KEYBOARD_CONTROL` *corresponds to the* Control *key stroke.*


**language: Java**
*The keyboard predefined values are final integers located in the Robot class. For example,* Ctrl+B *can be tested like this:*

```
1 int key=robot.keyboardGetKey()
2 if (key==Robot.KEYBOARD_CONTROL+'B')
3   System.out.Println("Ctrl+B is pressed");
```


**language: Python**
*The keyboard predefined values are integers located into the Robot class. For example,* Ctrl+B *can be tested like this:*

```
1 key=robot.keyboardGetKey()
2 if (key==Robot.KEYBOARD_CONTROL+ord('B')):
3   print 'Ctrl+B is pressed'
```

---

## NAME

wb_robot_task_new – *start a new thread of execution*

## SYNOPSIS

```
#include <webots/robot.h>

void wb_robot_task_new (void (*task)(void *), void *param);
```

## DESCRIPTION

This function creates and starts a new thread of execution for the robot controller. The `task` function is immediately called using the `param` parameter. It will end only when the `task` function returns. The Webots controller API is thread safe, however, some API functions use or return pointers to data structures which are not protected outside the function against asynchronous access from a different thread. Hence you should use mutexes (see below) to ensure that such data is not accessed by a different thread.

## SEE ALSO

wb_robot_mutex_new.

---

### NAME

wb_robot_mutex_new,
wb_robot_mutex_delete,
wb_robot_mutex_lock,
wb_robot_mutex_unlock – *mutex functions*

### SYNOPSIS

```
#include <webots/robot.h>

WbMutexRef wb_robot_mutex_new ();

void wb_robot_mutex_delete (WbMutexRef mutex);

void wb_robot_mutex_lock (WbMutexRef mutex);

void wb_robot_mutex_unlock (WBMutexRef mutex);
```

### DESCRIPTION

The `wb_robot_mutex_new()` function creates a new mutex and returns a reference to that mutex to be used with other mutex functions. A newly created mutex is always initially unlocked. Mutexes (mutual excluders) are useful with multi-threaded controllers to protect some resources (typically variables or memory chunks) from being used simultaneously by different threads.

The `wb_robot_mutex_delete()` function deletes the specified `mutex`. This function should be used when a mutex is no longer in use.

The `wb_robot_mutex_lock()` function attempts to lock the specified `mutex`. If the mutex is already locked by another thread, this function waits until the other thread unlocks the mutex, and then locks it. This function returns only after it has locked the specified `mutex`.

The `wb_robot_mutex_unlock()` function unlocks the specified `mutex`, allowing other threads to lock it.

### SEE ALSO

`wb_robot_task_new`.

Users unfamiliar with the mutex concept may wish to consult a reference on multi-threaded programming techniques for further information.

## 3.56 RotationalMotor

Derived from `Motor`.

```
RotationalMotor {
  field SFString name          "rotational motor" # for
     wb_robot_get_device()
  field SFFloat  maxTorque    10                  # max torque (Nm) :
     [0, inf)
}
```

### 3.56.1 Description

A RotationalMotor node can be used to power either a HingeJoint or a Hinge2Joint to produce a rotational motion around the choosen axis.

### 3.56.2 Field Summary

- The name field specifies the name identifier of the motor device. This the name to which wb_robot_get_device() can refer. It defaults to "rotational motor".
- The maxTorque field specifies both the upper limit and the default value for the motor *available torque*. The *available torque* is the torque that is available to the motor to perform the requested motions. The wb_motor_set_available_torque() function can be used to change the *available torque* at run-time. The value of maxTorque should always be zero or positive (the default is 10). A small maxTorque value may result in a motor being unable to move to the target position because of its weight or other external forces.

## 3.57 Servo


*As of Webots 7.2.0, the* Servo *node is deprecated and should not be used in any new simulation models. It is kept for backwards compatibility only. The functionality of the* Servo *node is replaced by the one provided by the* HingeJoint, RotationalMotor *and* LinearMotor *nodes. Therefore, you should use these nodes instead of the* Servo *node.*

Derived from Device.

```
Servo {
  SFString   type             "rotational"
  SFFloat    maxVelocity      10      # (0,inf)
  SFFloat    maxForce         10      # [0,inf)
  SFFloat    controlP         10      # (0,inf)
  SFFloat    acceleration     -1      # -1 or (0,inf)
```

```
  SFFloat    position          0
  SFFloat    minPosition       0        # (-inf,0]
  SFFloat    maxPosition       0        # [0,inf)
  SFFloat    minStop           0        # [-pi,0]
  SFFloat    maxStop           0        # [0,pi]
  SFFloat    springConstant    0        # [0,inf)
  SFFloat    dampingConstant   0        # [0,inf)
  SFFloat    staticFriction    0        # [0,inf)
}
```

### 3.57.1 Description

A Servo node is used to add a joint (1 degree of freedom (DOF)) in a mechanical simulation. The joint can be active or passive; it is placed between the parent and children nodes (.wbt hierarchy) of the Servo and therefore it allows the children to move with respect to the parent. The Servo can be of type "rotational" or "linear". A "rotational" Servo is used to simulate a rotating motion, like an electric motor or a hinge. A "linear" Servo is used to simulate a sliding motion, like a linear motor, a piston, a hydraulic/pneumatic cylinder, a spring, or a damper.

### 3.57.2 Field Summary

- The type field is a string which specifies the Servo type, and may be either "rotational" (default) or "linear".

- The maxVelocity field specifies both the upper limit and the default value for the servo *velocity*. The *velocity* can be changed at run-time with the wb_servo_set_velocity() function. The value should always be positive (the default is 10).

- The maxForce field specifies both the upper limit and the default value for the servo *motor force*. The *motor force* is the torque/force that is available to the motor to perform the requested motions. The wb_servo_set_motor_force() function can be used to change the *motor force* at run-time. The value of maxForce should always be zero or positive (the default is 10). A small maxForce value may result in a servo being unable to move to the target position because of its weight or other external forces.

- The controlP field specifies the initial value of the $P$ parameter, which is the *proportional gain* of the servo P-controller. A high $P$ results in a large response to a small error, and therefore a more sensitive system. Note that by setting $P$ too high, the system can become unstable. With a small $P$, more simulation steps are needed to reach the target position, but the system is more stable. The value of $P$ can be changed at run-time with the wb_servo_set_control_p() function.

- The `acceleration` field defines the default acceleration of the P-controller. A value of -1 (infinite) means that the acceleration is not limited by the P-controller. The acceleration can be changed at run-time with the `wb_servo_set_acceleration()` function.
- The `position` field represents the current *position* of the `Servo`, in radians or meters. For a "rotational" servo, `position` represents the current rotation angle in radians. For a "linear" servo, `position` represents the magnitude of the current translation in meters.
- The `minPosition` and `maxPosition` fields specify *soft limits* for the target position. These fields are described in more detail in the section "Servo Limits", see below.
- The `minStop` and `maxStop` fields specify the position of physical (or mechanical) stops. These fields are described in more detail in the section "Servo Limits", see below.
- The `springConstant` and `dampingConstant` fields allow the addition of spring and/or damping behavior to the `Servo`. These fields are described in more detail in the section "Springs and Dampers", see below.
- The `staticFriction` allows to add a friction opposed to the `Servo` movement. This field is described in more detail in the section "Friction", see below.

### 3.57.3 Units

Rotational servos units are expressed in *radians* while linear servos units are expressed in *meters*. See table 3.9:

| | Rotational | Linear |
|---|---|---|
| Position | rad (radians) | m (meters) |
| Velocity | rad/s (radians / second) | m/s (meters / second) |
| Acceleration | rad/s$^2$ (radians / second$^2$) | m/s$^2$ (meters / second$^2$) |
| Torque/Force | N*m (Newtons * meters) | N (Newtons) |

Table 3.9: Servo Units

### 3.57.4 Initial Transformation and Position

The `Servo` node inherits the `translation` and `rotation` fields from the `Transform` node. These two fields represent the initial coordinate system transformation between the `Servo` parent and children.

In a "rotational" `Servo`, these fields have the following meaning: The `translation` field specifies the translation of the axis of rotation. The `rotation` field specifies the orientation of the axis of rotation. See figure 3.33.

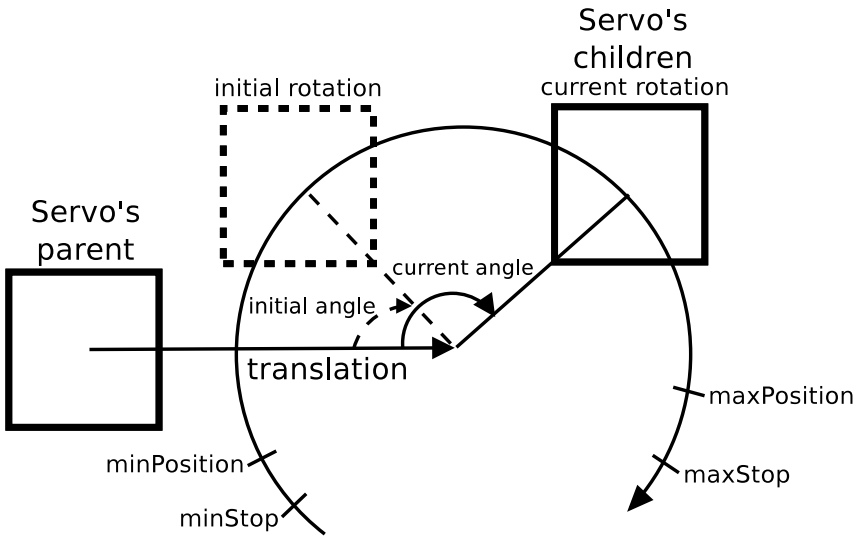


Figure 3.33: Rotational servo

In a "linear" `Servo`, these fields have the following meaning: The `translation` field specifies the translation of the sliding axis. The `rotation` field specifies the direction of the sliding axis. See figure 3.34.

The `position` field represents the current angle difference (in radians) or the current distance (in meters) with respect to the initial `translation` and `rotation` of the `Servo`. If `position` field is zero then the `Servo` is at its initial `translation` and `rotation`. For example if we have a "rotational" `Servo` and the value of the `position` field is 1.5708, this means that this `Servo` is 90 degrees from its initial rotation. The values passed to the `wb_servo_set_position()` function are specified with respect to the position zero. The values of the `minPosition`, `maxPosition`, `minStop` and `maxStop` fields are also defined with respect to the position zero.

### 3.57.5 Position Control

The standard way of operating a `Servo` is to control the position directly (*position control*). The user specifies a target position using the `wb_servo_set_position()` function, then the P-controller takes into account the desired velocity, acceleration and motor force in order to move the servo to the target position. See table 3.10.

In Webots, position control is carried out in three stages, as depicted in figure 3.35. The first stage is performed by the user-specified controller (1) that decides which position, velocity, acceleration and motor force must be used. The second stage is performed by the servo P-controller (2) that computes the current velocity of the servo $V_c$. Finally, the third stage (3) is carried out by the physics simulator (ODE joint motors).

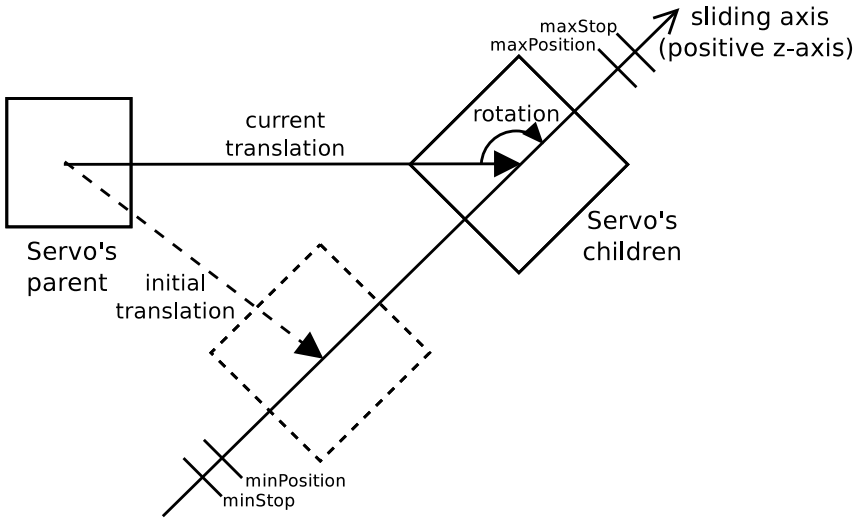


Figure 3.34: Linear servo

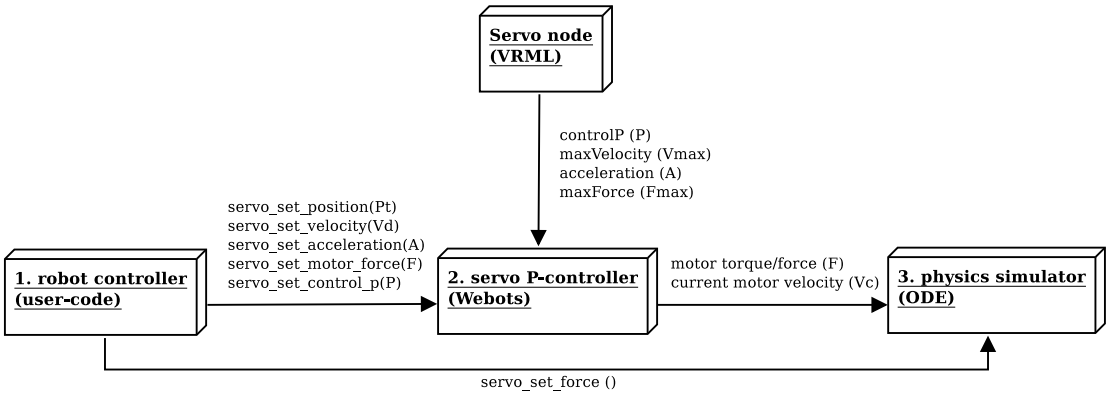


Figure 3.35: Servo control

At each simulation step, the P-controller (2) recomputes the current velocity *Vc* according to the following algorithm:

```
Vc = P * (Pt - Pc);
if (abs(Vc) > Vd)
  Vc = sign(Vc) * Vd;
if (A != -1) {
  a = (Vc - Vp) / ts;
  if (abs(a) > A)
    a = sign(a) * A;
  Vc = Vp + a * ts;
}
```

where $V_c$ is the current servo velocity in rad/s or m/s, $P$ is the P-control parameter specified in `controlP` field or set with `wb_servo_set_control_p()`, $P_t$ is the *target position* of the servo set by the function `wb_servo_set_position()`, $P_c$ is the current servo position as reflected by the `position` field, $V_d$ is the desired velocity as specified by the `maxVelocity` field (default) or set with `wb_servo_set_velocity()`, $a$ is the acceleration required to reach *Vc* in one time step, $V_p$ is the motor velocity of the previous time step, $t_s$ is the duration of the simulation time step as specified by the `basicTimeStep` field of the `WorldInfo` node (converted in seconds), and $A$ is the acceleration of the servo motor as specified by the `acceleration` field (default) or set with `wb_servo_set_acceleration()`.

### 3.57.6 Velocity Control

The servos can also be used with *velocity control* instead of *position control*. This is obtained with two function calls: first the `wb_servo_set_position()` function must be called with `INFINITY` as a position parameter, then the desired velocity, which may be positive or negative, must be specified by calling the `wb_servo_set_velocity()` function. This will initiate a continuous servo motion at the desired speed, while taking into account the specified acceleration and motor force. Example:

```
wb_servo_set_position(servo, INFINITY);
wb_servo_set_velocity(servo, 6.28);  // 1 rotation per second
```

`INFINITY` is a C macro corresponding to the IEEE 754 floating point standard. It is implemented in the C99 specifications as well as in C++. In Java, this value is defined as `Double.POSITIVE_INFINITY`. In Python, you should use `float('inf')`. Finally, in Matlab you should use the `inf` constant.

### 3.57.7 Force Control

The position/velocity control described above are performed by the Webots P-controller and ODE's joint motor implementation (see ODE documentation). As an alternative, Webots does

also allow the user to directly specify the amount of torque/force that must be applied by a Servo. This is achieved with the wb_servo_set_force() function which specifies the desired amount of torque/forces and switches off the P-controller and motor force. A subsequent call to wb_servo_set_position() restores the original *position control*. Some care must be taken when using *force control*. Indeed the torque/force specified with wb_servo_set_force() is applied to the Servo continuously. Hence the Servo will infinitely accelerate its rotational or linear motion and eventually *explode* unless a functional force control algorithm is used.

| | position control | velocity control | force control |
|---|---|---|---|
| uses P-controller | yes | no | no |
| wb_servo_set_position() | * specifies the desired position | should be set to INFINITY | switches to position/velocity control |
| wb_servo_set_velocity() | specifies the max velocity | * specifies the desired velocity | is ignored |
| wb_servo_set_acceleration() | specifies the max acceleration | specifies the max acceleration | is ignored |
| wb_servo_set_motor_force() | specifies the available force | specifies the available force | specifies the max force |
| wb_servo_set_force() | switches to force control | switches to force control | * specifies the desired force |

Table 3.10: Servo Control Summary

### 3.57.8 Servo Limits

The position field is a scalar value that represents the current servo "rotational" or "linear" position. For a rotational servo, position represents the difference (in radians) between the initial and the current angle of its rotation field. For a linear servo, position represents the distance (in meters) between the servo's initial and current translation (translation field).

The minPosition and maxPosition fields define the *soft limits* of the servo. Soft limits specify the *software* boundaries beyond which the P-controller will not attempt to move. If the controller calls wb_servo_set_position() with a target position that exceeds the soft limits, the desired target position will be clipped in order to fit into the soft limit range. Since the initial position of the servo is always zero, minPosition must always be negative or zero, and maxPosition must always be positive or zero. When both minPosition and maxPosition are zero (the default), the soft limits are deactivated. Note that the soft limits can be overstepped when an external force which exceeds the motor force is applied to the servo. For example, it is possible that the weight of a robot exceeds the motor force that is required to hold it up.

The minStop and maxStop fields define the *hard limits* of the servo. Hard limits represent physical (or mechanical) bounds that cannot be overrun by any force. Hard limits can be used, for example, to simulate both end caps of a hydraulic or pneumatic piston or to restrict the range of rotation of a hinge. The value of minStop must be in the range [-$\pi$, 0] and maxStop must be in the range [0, $\pi$]. When both minStop and maxStop are zero (the default), the hard limits are deactivated. The servo hard limits use ODE joint stops (for more information see the ODE documentation on dParamLoStop and dParamHiStop).

Finally, note that when both soft and hard limits are activated, the range of the soft limits must be included in the range of the hard limits, such that minStop <= minValue and maxStop>= maxValue.

### 3.57.9 Springs and Dampers

The springConstant field specifies the value of the spring constant (or spring stiffness), usually denoted as $K$. The springConstant must be positive or zero. If the springConstant is zero (the default), no spring torque/force will be applied to the servo. If the springConstant is greater than zero, then a spring force will be computed and applied to the servo in addition to the other forces (i.e., motor force, damping force). The spring force is calculated according to Hooke's law: $F = -Kx$, where $K$ is the springConstant and $x$ is the current servo position as represented by the position field. Therefore, the spring force is computed so as to be proportional to the current servo position, and to move the servo back to its initial position. When designing a robot model that uses springs, it is important to remember that the spring's resting position for each servo will correspond to the initial position of the servo.

The dampingConstant field specifies the value of the servo damping constant. The value of dampingConstant must be positive or zero. If dampingConstant is zero (the default), no damping torque/force will be added to the servo. If dampingConstant is greater than zero, a damping torque/force will be applied to the servo in addition to the other forces (i.e., motor force, spring force). This damping torque/force is proportional to the effective servo velocity: $F = -Bv$, where $B$ is the damping constant, and $v = dx/dt$ is the effective servo velocity computed by the physics simulator.

As you can see in (see figure 3.36), a Servo creates a joint between two masses $m_0$ and $m_1$. $m_0$ is defined by the Physics node in the parent of the Servo. The mass $m_1$ is defined by the Physics node of the Servo. The value $x_0$ corresponds to the initial translation of the Servo defined by the translation field. The position $x$ corresponds to the current position of the Servo defined by the position field.

### 3.57.10 Servo Forces

Altogether, three different forces can be applied to a Servo: the motor force, the spring force and the damping force. These three forces are applied in parallel and can be switched on and off

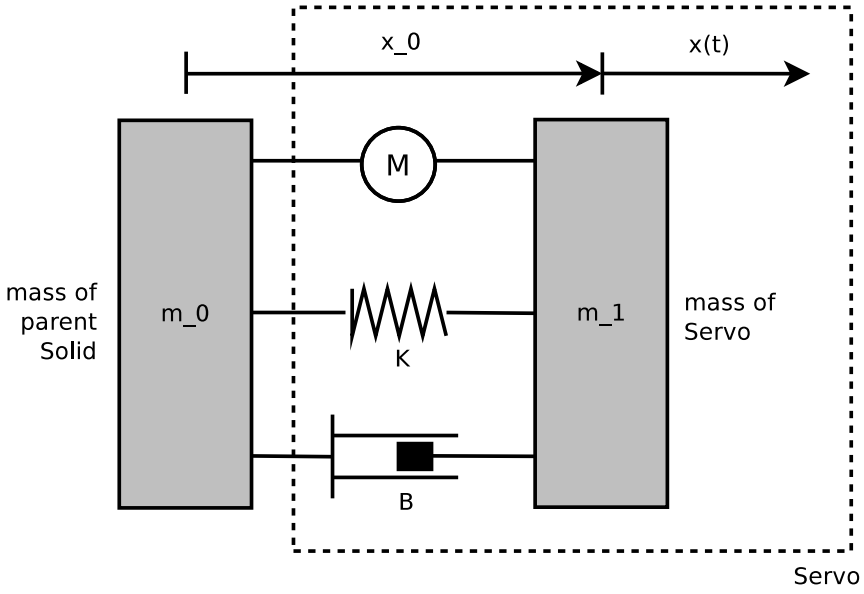


Figure 3.36: Mechanical Diagram of a Servo

independently (by default only the motor force is on). For example, to turn off the motor force and obtain a passive `Servo`, you can set the `maxForce` field to zero.

| Force | motor force | spring force | damping force |
|---|---|---|---|
| Turned on when: | maxForce $> 0$ | springConstant $> 0$ | dampingConstant $> 0$ |
| Turned off when: | maxForce = 0 | springConstant = 0 | dampingConstant = 0 |
| regular motor (the default) | on | off | off |
| regular spring & damper | off | on | on |
| damper (without spring) | off | off | on |
| motor with friction | on | off | on |
| spring without any friction | off | on | off |

Table 3.11: Servo Forces

To obtain a spring & damper element, you can set `maxForce` to zero and `springConstant` and `dampingConstant` to non-zero values. A pure spring is obtained when both `maxForce` and `dampingConstant` but not `springConstant` are set to zero. However in this case the spring may oscillate forever because Webots will not simulate the air friction. So it is usually wise to associate some damping to every spring.

### 3.57.11 Friction

The friction applied on the `Servo` to slow down its velocity is computed as the maximum between the `maxForce` and the `staticFriction` values. The static friction is particularily

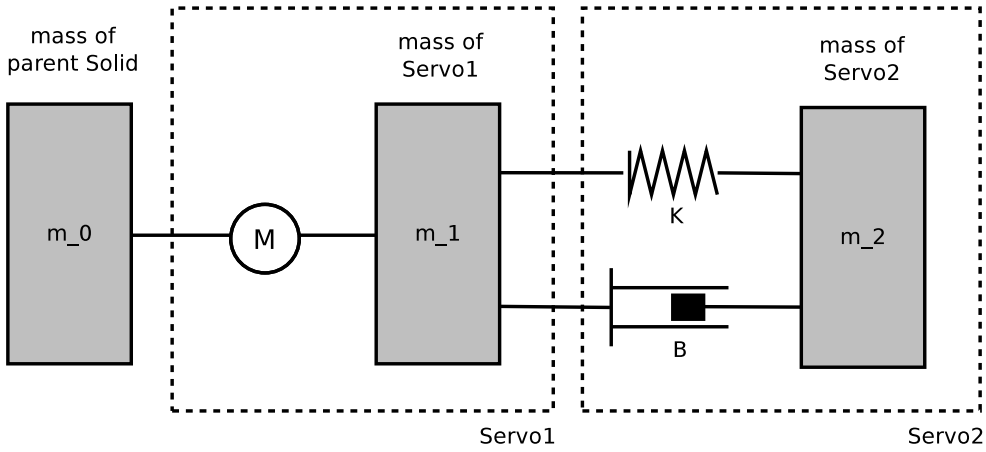


Figure 3.37: Example of serial connection of two Servo nodes

useful to add a friction for a passive Servo.

### 3.57.12 Serial Servos

Each instance of a Servo simulates a mechanical system with optional motor, spring and damping elements, mounted in parallel. Sometimes it is necessary to have such elements mounted serially. With Webot, serially mounted elements must be modeled by having Servo nodes used as children of other Servo nodes. For example if you wish to have a system where a motor controls the resting position of a spring, then you will need two Servo nodes, as depicted in figure 3.37. In this example, the parent Servo will have a motor force (maxForce $> 0$) and the child Servo will have spring and damping forces (springConstant $> 0$ and dampingConstant $> 0$).

This is equivalent to this .wbt code, where, as you can notice, *Servo2* is a child of *Servo1*:

```
DEF Servo1 Servo {
  ...
  children [
    DEF Servo2 Servo {
      ...
      children [
        ...
      ]
      boundingObject ...
      physics Physics {
        mass {m2}
      }
      maxForce 0
      springConstant {K}
```

```
      dampingConstant {B}
    }
  ]
  boundingObject ...
  physics Physics {
    mass {m1}
  }
  maxForce {M}
  springConstant 0
  dampingConstant 0
}
```

Note that it is necessary to specify the Physics and the boundingObject of *Servo1*. This adds the extra body $m_1$ in the simulation, between the motor and the spring and damper.

### 3.57.13 Simulating Overlayed Joint Axes

Sometimes it is necessary to simulate a joint with two or three independent but overlayed rotation axes (e.g., a shoulder joint with a *pitch* axis and a *roll* axis). As usually with Webots, each axis must be implemented as a separate Servo node. So for two axes you need two Servo nodes, for three axes you need three Servo nodes, etc.

With overlayed axes (or very close axes) the mass and the shape of the body located between these axes is often unknown or negligible. However, Webots requires all the intermediate boundingObject and physics fields to be defined. So the trick is to use dummy values for these fields. Usually the dummy boundingObject can be specified as a Sphere with a radius of 1 millimeter. A Sphere is the preferred choice because this is the cheapest shape for the collision detection. And the physics field can use a Physics node with default values.

This is better explained with an example. Let's assume that we want to build a pan/tilt robot head. For this we need two independent (and perpendicular) rotation axes: *pan* and *tilt*. Now let's assume that these axes cross each other but we don't know anything about the shape and the mass of the body that links the two axes. Then this can be modeled like this:

```
DEF PAN Servo {
  ...
  children [
    DEF TILT Servo {
      translation 0 0 0  # overlayed
      children [
        DEF HEAD_TRANS Transform {
          # head shape
        }
        # head devices
      ]
      boundingObject USE HEAD_TRANS
```

```
      physics Physics {
      }
    }
  ]
  boundingObject DEF DUMMY_BO Sphere {
    radius 0.001
  }
  physics DEF DUMMY_PHYSICS Physics {
  }
}
```

Please note the dummy `Physics` and the 1 millimeter `Sphere` as dummy `boundingObject`.

### 3.57.14 Servo Functions

---

#### NAME

wb_servo_set_position,
wb_servo_set_velocity,
wb_servo_set_acceleration,
wb_servo_set_motor_force,
wb_servo_set_control_p,
wb_servo_get_min_position,
wb_servo_get_max_position – *change the parameters of the P-controller*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/servo.h>

void wb_servo_set_position (WbDeviceTag tag, double position);

double wb_servo_get_target_position (WbDeviceTag tag);

void wb_servo_set_velocity (WbDeviceTag tag, double velocity);

void wb_servo_set_acceleration (WbDeviceTag tag, double acceleration);

void wb_servo_set_motor_force (WbDeviceTag tag, double force);

void wb_servo_set_control_p (WbDeviceTag tag, double p);

double wb_servo_get_min_position (WbDeviceTag tag);

double wb_servo_get_max_position (WbDeviceTag tag);
```

#### DESCRIPTION

The wb_servo_set_position() function specifies a new target position that the P-controller will attempt to reach using the current velocity, acceleration and motor torque/force parameters. This function returns immediately (asynchronous) while the actual motion is carried out in the background by Webots. The target position will be reached only if the physics simulation allows it, that means, if the specified motor force is sufficient and the motion is not blocked by obstacles, external forces or the servo's own spring force, etc. It is also possible to wait until the Servo reaches the target position (synchronous) like this:


**language: C**

```c
void servo_set_position_sync(WbDeviceTag tag,
   double target, int delay) {
  const double DELTA = 0.001;  // max tolerated
     difference
  wb_servo_set_position(tag, target);
  wb_servo_enable_position(tag, TIME_STEP);
  double effective;  // effective position
  do {
    wb_robot_step(TIME_STEP);
    delay -= TIME_STEP;
    effective = wb_servo_get_position(tag);
  }
  while (fabs(target - effective) > DELTA &&
     delay > 0);
  wb_servo_disable_position(tag);
}
```

The INFINITY (*#include <math.h>*) value can be used as the second argument to the wb_servo_set_position() function in order to enable an endless rotational (or linear) motion. The current values for velocity, acceleration and motor torque/force are taken into account. So for example, wb_servo_set_velocity() can be used for controlling the velocity of the endless rotation:


**language: C**

```c
// velocity control
wb_servo_set_position(tag, INFINITY);
wb_servo_set_velocity(tag, desired_speed);  //
   rad/s
```


**language: C++**
*In C++ use* `std::numeric_limits<double>::infinity()` *instead of INFINITY*


**language: Java**
*In Java use* `Double.POSITIVE_INFINITY` *instead of INFINITY*


**language: Python**
*In Python use* `float('+inf')` *instead of INFINITY*


**language: Matlab**
*In MATLAB use* `inf` *instead of INFINITY*

The `wb_servo_get_target_position()` function allows to get the target position. This value matches with the argument given to the last `wb_servo_set_position()` function call.

The `wb_servo_set_velocity()` function specifies the velocity that servo should reach while moving to the target position. In other words, this means that the servo will accelerate (using the specified acceleration, see below) until the target velocity is reached. The velocity argument passed to this function cannot exceed the limit specified in the `maxVelocity` field.

The `wb_servo_set_acceleration()` function specifies the acceleration that the P-controller should use when trying to reach the specified velocity. Note that an infinite acceleration is obtained by passing -1 as the `acc` argument to this function.

The `wb_servo_set_motor_force()` function specifies the max torque/force that will be available to the motor to carry out the requested motion. The motor torque/force specified with this function cannot exceed the value specified in the `maxForce` field.

The `wb_servo_set_control_p()` function changes the value of the $P$ parameter in the P-controller. $P$ is a parameter used to compute the current servo velocity $V_c$ from the current position $P_c$ and target position $P_t$, such that $V_c = P * (P_t - P_c)$. With a small $P$, a long time is needed to reach the target position, while too large a $P$ can make the system unstable. The default value of $P$ is specified by the `controlP` field of the corresponding `Servo` node.

The `wb_servo_get_[min|max]_position()` functions allow to get the values of respectively the `minPosition` and the `maxPosition` fields.

### NAME

wb_servo_enable_position,
wb_servo_disable_position,
wb_servo_get_position_sampling_period,
wb_servo_get_position – *get the effective position of a servo*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/servo.h>

void wb_servo_enable_position (WbDeviceTag tag, int ms);

void wb_servo_disable_position (WbDeviceTag tag);

int wb_servo_get_position_sampling_period (WbDeviceTag tag);

double wb_servo_get_position (WbDeviceTag tag);
```

### DESCRIPTION

The `wb_servo_enable_position()` function activates position measurements for the specified servo. A new position measurement will be performed each `ms` milliseconds; the result must be obtained with the `wb_servo_get_position()` function. The returned value corresponds to the most recent measurement of the servo position. The `wb_servo_get_position()` function measures the *effective position* of the servo which, under the effect of external forces, is usually different from the *target position* specified with `wb_servo_set_position()`. For a rotational servo, the returned value is expressed in radians, for a linear servo, the value is expressed in meters. The returned value is valid only if the corresponding measurement was previously enabled with `wb_servo_enable_position()`.

The `wb_servo_disable_position()` function deactivates position measurements for the specified servo. After a call to `wb_servo_disable_position()`, `wb_servo_get_position()` will return undefined values.

The `wb_servo_get_position_sampling_period()` function returns the period given into the `wb_servo_enable_position()` function, or 0 if the device is disabled.

---

### NAME

wb_servo_enable_motor_force_feedback,
wb_servo_get_motor_force_feedback,
wb_servo_get_motor_force_feedback_sampling_period,
wb_servo_disable_motor_force_feedback – *get the motor force currently used by a servo*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/servo.h>

void wb_servo_enable_motor_force_feedback (WbDeviceTag tag, int ms);

void wb_servo_disable_motor_force_feedback (WbDeviceTag tag);

int wb_servo_get_motor_force_feedback_sampling_period (WbDeviceTag tag);

double wb_servo_get_motor_force_feedback (WbDeviceTag tag);
```

## DESCRIPTION

The `wb_servo_enable_motor_force_feedback()` function activates torque/force feedback measurements for the specified servo. A new measurement will be performed each `ms` milliseconds; the result must be retrieved with the `wb_servo_get_motor_force_feedback()` function.

The `wb_servo_get_motor_force_feedback()` function returns the most recent motor force measurement. This function measures the amount of motor force that is currently being used by the servo in order to achieve the desired motion or hold the current position. For a "rotational" servo, the returned value is a torque [N*m]; for a "linear" servo, the value is a force [N]. The returned value is an approximation computed by the physics engine, and therefore it may be inaccurate. The returned value normally does not exceed the available motor force specified with `wb_servo_set_motor_force()` (the default being the value of the `maxForce` field). Note that this function measures the *current motor force* exclusively, all other external or internal forces that may apply to the servo are ignored. In particular, `wb_servo_get_motor_force_feedback()` does not measure:

- The spring and damping forces that apply when the `springConstant` or `dampingConstant` fields are non-zero.
- The force specified with the `wb_servo_set_force()` function.
- The *constraint forces* that restrict the servo motion to one degree of freedom (DOF). In other words, the forces applied outside of the servo DOF are ignored. Only the forces applied in the DOF are considered. For example, in a "linear" servo, a force applied at a right angle to the sliding axis is completely ignored. In a "rotational" servo, only the torque applied around the rotation axis is considered.

Note that this function applies only to *physics-based* simulation. Therefore, the `physics` and `boundingObject` fields of the `Servo` node must be defined for this function to work properly.

If `wb_servo_get_motor_force_feedback()` was not previously enabled, the return value is undefined.

The wb_servo_get_motor_force_feedback_sampling_period() function returns the period given into the wb_servo_enable_motor_force_feedback() function, or 0 if the device is disabled.

---

### NAME

wb_servo_set_force – *direct force control*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/servo.h>

void wb_servo_set_force (WbDeviceTag tag, double force);
```

### DESCRIPTION

As an alternative to the P-controller, the wb_servo_set_force() function allows the user to directly specify the amount of torque/force that must be applied by a servo. This function bypasses the P-controller and ODE joint motors; it adds the force to the physics simulation directly. This allows the user to design a custom controller, for example a PID controller. Note that when wb_servo_set_force() is invoked, this automatically resets the force previously added by the P-controller.

In a "rotational" servo, the *force* parameter specifies the amount of torque that will be applied around the servo rotation axis. In a "linear" servo, the parameter specifies the amount of force [N] that will be applied along the sliding axis. A positive *torque/force* will move the bodies in the positive direction, which corresponds to the direction of the servo when the position field increases. When invoking wb_servo_set_force(), the specified *force* parameter cannot exceed the current *motor force* of the servo (specified with wb_servo_set_motor_force() and defaulting to the value of the maxForce field).

Note that this function applies only to *physics-based* simulation. Therefore, the physics and boundingObject fields of the Servo node must be defined for this function to work properly.

It is also possible, for example, to use this function to implement springs or dampers with controllable properties. The example in projects/samples/howto/worlds/force_control.wbt demonstrates the usage of wb_servo_set_force() for creating a simple spring and damper system.

---

### NAME

wb_servo_get_type – *get the servo type*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/servo.h>

int wb_servo_get_type (WbDeviceTag tag);
```

### DESCRIPTION

This function allows to retrieve the servo type defined by the `type` field. If the value of the `type` field is "linear", this function returns WB_SERVO_LINEAR, and otherwise it returns WB_SERVO_ROTATIONAL.

| Servo.type | return value |
|---|---|
| "rotational" | WB_SERVO_ROTATIONAL |
| "linear" | WB_SERVO_LINEAR |

Table 3.12: Return values for the `wb_servo_get_type()` function

## 3.58 Shape

```
Shape {
  SFNode   appearance   NULL
  SFNode   geometry     NULL
}
```

The `Shape` node has two fields, `appearance` and `geometry`, which are used to create rendered objects in the world. The `appearance` field contains an `Appearance` node that specifies the visual attributes (e.g., material and texture) to be applied to the geometry. The `geometry` field contains a `Geometry` node: `Box`, `Capsule`, `Cone`, `Cylinder`, `ElevationGrid`, `IndexedFaceSet`, `IndexedLineSet`, `Plane` or `Sphere`. The specified `Geometry` node is rendered with the specified appearance nodes applied.

## 3.59 SliderJoint

Derived from `Joint`.

```
SliderJoint {
  field SFNode     device             [ ]          # linear motor or
     linear position sensor
  hiddenField SFFloat position         0           # initial position (m
     )
}
```

### 3.59.1 Description

The SliderJoint node can be used to model a slider, i.e. a joint allowing only a translation motion along a given axis (1 degree of freedom). It inherits Joint's jointParameters field. This field can be filled with a JointParameters only. If empty, JointParameters default values apply.

### 3.59.2 Field Summary

- device: This field optionally specifies a LinearMotor, a linear PositionSensor and/or a Brake device. If no motor is specified, the joint is passive joint.
- position: see joint's hidden position field.

## 3.60 Slot

```
Slot {
  field     SFString type      ""
  vrmlField SFNode   endPoint  NULL
}
```

### 3.60.1 Description

Slot nodes always works with pairs, only a second Slot can be added in the endPoint field of a Slot before to be able to add any node in the endPoint field of the second Slot. Furthermore, the second Slot can be added only if it has the same type as the first one.

The Slot node is particularly usefull with PROTOs, it allows the user to constrain the type of nodes that can be added in an extension field of the PROTO. Imagine for example that you have an armed robot in which you can plug different kinds of hands. In order to do so you will put the hand as an extension field of your robot, you will then be able to add all the different PROTOs of hand that you have made. But nothing prevent you to add a PROTO of table in the hand extension field. The Slot is made for preventing this kind of problems. By encapsulating your extension field in a Slot and using the Slot node as base node for all your hands PROTOs and defining the same type for the field Slot and the PROTO Slot, only hands can be inserted in the extension field. This is illustrated in the example section.

### 3.60.2 Field Summary

- type: defines the type of the Slot. Two Slot nodes can be connected only if their types match. It is possible to specify a gender by ending the string with a '+' or a '−'. In this

case, two Slot nodes can be connected only if they are of opposite gender (e.g. a Slot with a type ending with '+' can only be connected to a Slot with the same type, except that it ends with '−' instead of '+'). The default empty type matches any type.

- endPoint: The node inserted in the endPoint of a Slot should be another Slot if this Slot is not already connected to another Slot (i.e., its parent is a Slot). If the pair of Slot nodes is already connected, any node that can usually be inserted in a children field can be inserted in the endPoint field of the second Slot.

### 3.60.3 Example

If you want to write a proto of a robot called MyRobot that accepts only hands in its field handExtension, you have to set the field handExtension to be the endPoint of a Slot.

```
PROTO MyRobot [
  field SFNode handExtension NULL
]
Robot {
  children [
    Slot {
      type "robotHand"
      endPoint IS handExtension
    }
    ...
  ]
}
```

Then any PROTO of a hand needs to use the Slot as base node and the type of this Slot should match the one in MyRobot.

```
PROTO RobotHand [
]
{
  Slot {
    type "robotHand"
    endPoint Solid {
      ...
    }
  }
}
```

## 3.61 Solid

Derived from Transform.

```
Solid {
  SFString   name               "solid"
  SFString   model              ""
  SFString   description        ""
  SFString   contactMaterial    "default"
  MFNode     immersionProperties []
  SFNode     boundingObject     NULL
  SFNode     physics            NULL
  SFBool     locked             FALSE
  SFFloat    translationStep    0.01        # m
  SFFloat    rotationStep       0.261799387 # pi/12 rad
  # hidden fields
  hiddenField SFVec3f linearVelocity 0 0 0 # initial linear velocity
  hiddenField SFVec3f angularVelocity 0 0 0 # initial angular velocity
}
```

Direct derived nodes: Accelerometer, Camera, Charger, Compass, Connector, Display, DistanceSensor, Emitter, GPS, Gyro, InertialUnit, LED, LightSensor, Pen, Receiver, Robot, Servo, TouchSensor.

### 3.61.1 Description

A Solid node represents an object with physical properties such as dimensions, a contact material and optionally a mass. The Solid class is the base class for collision-detected objects. Robots and device classes are subclasses of the Solid class. In the 3D window, Solid nodes can be manipulated (dragged, lifted, rotated, etc) using the mouse.

### 3.61.2 Solid Fields

Note that in the Solid node, the scale field inherited from the Transform must always remain uniform, i.e., of the form x x x where x is any positive real number. This ensures that all primitive geometries will remain suitable for ODE collision detection. Whenever a scale coordinate is changed, the two other ones are automatically changed to this new value. If a scale coordinate is assigned a non-positive value, it is automatically changed to 1.

- name: name of the solid. In derived device classes this corresponds to the device name argument used by wb_robot_get_device().
- model: generic name of the solid (e.g., "chair").
- description: short description (1 line) of the solid.

- contactMaterial: name of the contact material. When the boundingObjects of Solid nodes intersect, the contactMaterial is used to define which ContactProperties must be applied at the contact points.

- immersionProperties: list of immersionProperties nodes. It is used to specify dynamic interactions of the Solid node with one or more Fluid nodes.

- boundingObject: the bounding object specifies the geometrical primitives used for collision detection. If the boundingObject field is NULL, then no collision detection is performed and that object can pass through any other object, e.g., the floor, obstacles and other robots. Note that if the boundingObject field is NULL then the physics field (see below) must also be NULL. You will find more explanations about the boundingObject field below.

- physics: this field can optionally contain a Physics node that is used to model the physical properties of this Solid. A Physics node should be added when effects such as gravity, inertia, frictional and contact forces need to be simulated. If the physics field is NULL then Webots simulates this object in *kinematics* mode. Note that if this field is not NULL then the boundingObject field must be specified. Please find more info in the description of the Physics node.

- locked: if TRUE, the solid object cannot be moved using the mouse. This is useful to prevent moving an object by mistake.

- translationStep and rotationStep: these fields specify the minimum step size that will be used by the translate and rotate handles appearing in the 3D window when selecting a top solid. Continuous increment is obtained by setting the step value to -1.

- linearVelocity and angularVelocity: these fields, which aren't visible from the Scene Tree, are used by Webots when saving a world file to store the initial linear and angular velocities of a Solid with a non-NULL Physics node. If the Solid node is merged into a solid assembly (see implicit solid merging), then these fields will be effective only for the Solid at the top of the assembly. Hidden velocity fields allow you to save and restore the dynamics of your simulation or to define initial velocities for every physical objects in the scene.

### 3.61.3 How to use the boundingObject field?

boundingObjects are used to define the bounds of a Solid as geometrical primitive. Each boundingObject can hold one or several geometrical primitives, such as Box, Capsule, Cylinder, etc. These primitives should normally be chosen such as to represent the approximate bounds of the Solid. In the usual case, the graphical representation of a robot is composed of many complex shapes, e.g., IndexedFaceSets, placed in the children field of

the Solid nodes. However this graphical representation is usually too complex to be used directly for detecting collisions. If there are too many faces the simulation becomes slow and error-prone. For that reason, it is useful to be able to approximate the graphical representation by simpler primitives, e.g., one or more Box or Capsules, etc. This is the purpose of the boundingObject field.

Various combinations of primitives can be used in a boundingObject: it can contain either:

1. A Box node,
2. A Capsule node,
3. A Cylinder node,
4. An ElevationGrid node,
5. An IndexedFaceSet node,
6. A Plane node,
7. A Sphere node,
8. A Shape node with one of the above nodes in its geometry field,
9. A Transform node with one of the above nodes in its children field, or
10. A Group node with several children, each being one of the above.

The boundingObject, together with the Physics node, are used to compute the inertia matrix of the Solid. Such a computation assumes a uniform mass distribution in the primitives composing the boundingObject. Note that the center of mass of the Solid does not depend on its boundingObject. The center of mass of is specified by the centerOfMass field of the Physics node (in coordinates relative to the center of the Solid).

## 3.62 SolidReference

```
SolidReference {
  field  SFString  solidName  "" # name of an existing solid or <
     static environment>
}
```

### 3.62.1 Description

A SolidReference can be used inside the endPoint field of a Joint node to refer either to an existing Solid or to the static environment. Mechanical loops can be closed this way. The only constraint when referring to a Solid is that both Solid and Joint must be descendants of a common upper Solid.

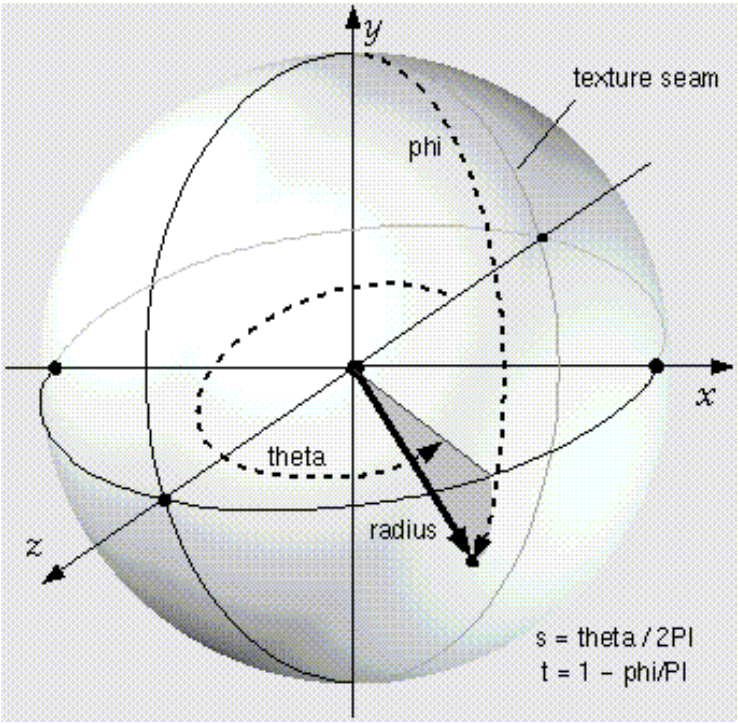


Figure 3.38: Sphere node

### 3.62.2 Field Summary

- `solidName`: This field specifies either the static environment or the name of an existing `Solid` node to be linked with the `Joint`'s closest upper `Solid` node. Referring to the `Joint` closest upper `Solid` node or to a `Solid` node which has no common upper `Solid` with the `Joint` is prohibited.

## 3.63 Sphere

```
Sphere {
  SFFloat    radius         1    # (-inf,inf)
  SFInt32    subdivision    1    # [0,5] or 10
}
```

The `Sphere` node specifies a sphere centered at (0,0,0) in the local coordinate system. The `radius` field specifies the radius of the sphere (see figure 3.38).

If `radius` is positive, the outside faces of the sphere are displayed while if it is negative, the inside faces are displayed.

The `subdivision` field controls the number of faces of the rendered sphere. Spheres are rendered as icosahedrons with 20 faces when the subdivision field is set to 0. If the subdivision

field is 1 (default value), then each face is subdivided into 4 faces, making 80 faces. With a subdivision field set to 2, 320 faces will be rendered, making the sphere very smooth. A maximum value of 5 (corresponding to 20480 faces) is allowed for this subdivision field to avoid a very long rendering process. A value of 10 will turn the sphere appearance into a black and white soccer ball.

When a texture is applied to a sphere, the texture covers the entire surface, wrapping counter-clockwise from the back of the sphere. The texture has a seam at the back where the *yz*-plane intersects the sphere. `TextureTransform` affects the texture coordinates of the Sphere.

## 3.64 SpotLight

Derived from `Light`.

```
SpotLight {
  SFFloat ambientIntensity  0         # [0,1]
  SFVec3f attenuation       1 0 0     # [0,inf)
  SFFloat beamWidth         1.570796  # [0,pi/2)
  SFColor color             1 1 1     # [0,1]
  SFFloat cutOffAngle       0.785398  # [0,pi/2)
  SFVec3f direction         0 0 -1    # (-inf,inf)
  SFFloat intensity         1         # [0,1]
  SFVec3f location          0 0 10    # (-inf,inf)
  SFBool  on                TRUE
  SFFloat radius            100       # [0,inf)
  SFBool  castShadows       FALSE
}
```

### 3.64.1 Description

The `SpotLight` node defines a light source that emits light from a specific point along a specific direction vector and constrained within a solid angle. Spotlights may illuminate `Geometry` nodes that respond to light sources and intersect the solid angle. Spotlights are specified in their local coordinate system and are affected by parent transformations.

The `location` field specifies a translation offset of the center point of the light source from the light's local coordinate system origin. This point is the apex of the solid angle which bounds light emission from the given light source. The `direction` field specifies the direction vector of the light's central axis defined in its own local coordinate system. The `on` field specifies whether the light source emits light–if TRUE, then the light source is emitting light and may illuminate geometry in the scene, if FALSE it does not emit light and does not illuminate any geometry. The `radius` field specifies the radial extent of the solid angle and the maximum distance from

location that may be illuminated by the light source - the light source does not emit light outside this radius. The radius must be >= 0.0.

The cutOffAngle field specifies the outer bound of the solid angle. The light source does not emit light outside of this solid angle. The beamWidth field specifies an inner solid angle in which the light source emits light at uniform full intensity. The light source's emission intensity drops off from the inner solid angle (beamWidth) to the outer solid angle (cutOffAngle). The drop off function from the inner angle to the outer angle is a cosine raised to a power function:

```
intensity(angle) = intensity * (cosine(angle) ** exponent)

where exponent = 0.5*log(0.5)/log(cos(beamWidth)),
      intensity is the SpotLight's field value,
      intensity(angle) is the light intensity at an arbitrary
          angle from the direction vector,
      and angle ranges from 0.0 at central axis to cutOffAngle.
```

If beamWidth > cutOffAngle, then beamWidth is assumed to be equal to cutOffAngle and the light source emits full intensity within the entire solid angle defined by cutOffAngle. Both beamWidth and cutOffAngle must be greater than 0.0 and less than or equal to $\pi/2$. See figure below for an illustration of the SpotLight's field semantics (note: this example uses the default attenuation).

The light's illumination falls off with distance as specified by three attenuation coefficients. The attenuation factor is 1/(attenuation[0]+attenuation[1]*r+attenuation[2]*r$^2$), where r is the distance of the light to the surface being illuminated. The default is no attenuation. An attenuation value of 0 0 0 is identical to 1 0 0. Attenuation values must be >= 0.0.

Contrary to the VRML specifications, the attenuation and the ambientIntensity fields cannot be set simultaneously.

## 3.65 Supervisor

Derived from Robot.

```
Supervisor {
  # no additional fields
}
```

### 3.65.1 Description

A Supervisor is a special kind of Robot which is specially designed to control the simulation. A Supervisor has access to extra functions that are not available to a regular Robot. If

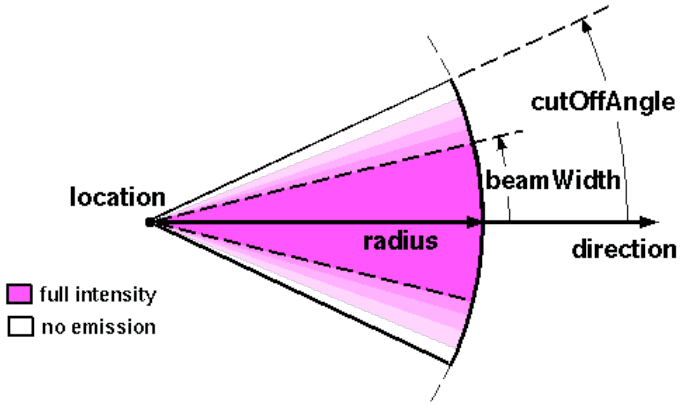


Figure 3.39: Spot light

a Supervisor contains devices then the Supervisor controller can use them. Webots PRO is required to use the Supervisor node.


*Note that in some special cases the* Supervisor *functions might return wrong values and it might not be possible to retrieve fields and nodes. This occurs when closing a world and quitting its controllers, i.e. reverting the current world, opening a new world, or closing Webots. In this case the output will be a NULL pointer or a default value. For functions returning a string, an empty string is returned instead of a NULL pointer.*


**language: C++, Java, Python**
*It is a good practice to check for a NULL pointer after calling a* Supervisor *function.*

### 3.65.2 Supervisor Functions

As for a regular Robot controller, the `wb_robot_init()`, `wb_robot_step()`, etc. functions must be used in a Supervisor controller.

---

#### NAME

wb_supervisor_export_image – *save the current 3D image of the simulator into a JPEG file, suitable for building a webcam system*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_export_image (const char *filename, int quality);
```

### DESCRIPTION

The `wb_supervisor_export_image()` function saves the current image of Webots main window into a JPEG file as specified by the `filename` parameter. The `quality` parameter defines the JPEG quality (in the range 1 - 100). The `filename` parameter should specify a valid (absolute or relative) file name, e.g., `snapshot.jpg` or `/var/www/html/images/snapshot.jpg`. In fact, a temporary file is first saved, and then renamed to the requested `filename`. This avoids having a temporary unfinished (and hence corrupted) file for webcam applications.

### EXAMPLE

The `projects/samples/howto/worlds/supervisor.wbt` world provides an example on how to use the `wb_supervisor_export_image()` function. In this example, the `Supervisor` controller takes a snapshot image each time a goal is scored.

---

### NAME

wb_supervisor_node_get_from_def,
wb_supervisor_node_get_root,
wb_supervisor_node_get_self – *get a handle to a node in the world*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

WbNodeRef wb_supervisor_node_get_from_def (const char *def);

WbNodeRef wb_supervisor_node_get_root ();

WbNodeRef wb_supervisor_node_get_self ();
```

### DESCRIPTION

The `wb_supervisor_node_get_from_def()` function retrieves a handle to a node in the world from its DEF name. The return value can be used for subsequent calls to functions which require a `WbNodeRef` parameter. If the requested node does not exist in the current world file, the function returns NULL, otherwise, it returns a non-NULL handle.

The wb_supervisor_node_get_root() function returns a handle to the root node which is actually a Group node containing all the nodes visible at the top level in the scene tree window of Webots. Like any Group node, the root node has a MFNode field called "children" which can be parsed to read each node in the scene tree. An example of such a usage is provided in the supervisor.wbt sample worlds (located in the projects/samples/devices/worlds directory of Webots.

The wb_supervisor_node_get_self() function returns a handle to the Supervisor node itself on which the controller is run. This is a utility function that simplifies the task of retrieving the base node without having to define a DEF name for it.

---

### NAME

wb_supervisor_node_get_type,
wb_supervisor_node_get_type_name – *get information on a specified node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

WbNodeType wb_supervisor_node_get_type (wbNodeRef node);

const char *wb_supervisor_node_get_type_name (wbNodeRef node);
```

### DESCRIPTION

The wb_supervisor_node_get_type() function returns a symbolic value corresponding the type of the node specified as an argument. If the argument is NULL, it returns WB_NODE_NO_NODE. A list of all node types is provided in the webots/nodes.h include file. Node types include WB_NODE_DIFFERENTIAL_WHEELS, WB_NODE_APPEARANCE, WB_NODE_LIGHT_SENSOR, etc.

The wb_supervisor_node_get_type_name() function returns a text string corresponding to the name of the node, like "DifferentialWheels", "Appearance", "LightSensor", etc. If the argument is NULL, the function returns NULL.


**language: C++, Java, Python**
*In the oriented-object APIs, the WB_NODE_* constants are available as static integers of the* Node *class (for example, Node::DIFFERENTIAL_WHEELS). These integers can be directly compared with the output of the* Node::getType()

---

### NAME

wb_supervisor_node_get_field – *get a field reference from a node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

WbFieldRef wb_supervisor_node_get_field (WbNodeRef node, const char *field_name);
```

### DESCRIPTION

The `wb_supervisor_node_get_field()` function retrieves a handler to a node field. The field is specified by its name in `field_name` and the `node` it belongs to. It can be a single field (SF) or a multiple field (MF). If no such field name exist for the specified node, the return value is NULL. Otherwise, it returns a handler to a field.

---

### NAME

wb_supervisor_node_get_position,
wb_supervisor_node_get_orientation – *get the global (world) position/orientation of a node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

const double *wb_supervisor_node_get_position (WbNodeRef node);

const double *wb_supervisor_node_get_orientation (WbNodeRef node);
```

### DESCRIPTION

The `wb_supervisor_node_get_position()` function returns the position of a node expressed in the global (world) coordinate system. The `node` argument must be a `Transform` node (or a derived node), otherwise the function will print a warning message and return 3 `NaN` (Not a Number) values. This function returns a vector containing exactly 3 values.

The `wb_supervisor_node_get_orientation()` function returns a matrix that represents the rotation of the node in the global (world) coordinate system. The `node` argument must be a `Transform` node (or a derived node), otherwise the function will print a warning message and return 9 `NaN` (Not a Number) values. This function returns a matrix containing exactly 9 values that shall be interpreted as a 3 x 3 orthogonal rotation matrix:

```
[ R[0] R[1] R[2] ]
[ R[3] R[4] R[5] ]
[ R[6] R[7] R[8] ]
```

Each column of the matrix represents where each of the three main axes ($x$, $y$ and $z$) is pointing in the node's coordinate system. The columns (and the rows) of the matrix are pairwise orthogonal unit vectors (i.e., they form an orthonormal basis). Because the matrix is orthogonal, its transpose is also its inverse. So by transposing the matrix you can get the inverse rotation. Please find more info here[1].

By multiplying the rotation matrix on the right with a vector and then adding the position vector you can express the coordinates of a point in the global (world) coordinate system knowing its coordinates in a local (node) coordinate system. For example:

```
p' = R * p + T
```

where $p$ is a point whose coordinates are given with respect to the local coordinate system of a node, $R$ the the rotation matrix returned by `wb_supervisor_node_get_orientation(node)`, $T$ is the position returned by `wb_supervisor_node_get_position(node)` and $p'$ represents the same point but this time with coordinates expressed in the global (world) coordinate system.

The `WEBOTS_HOME/projects/robots/ipr/worlds/ipr_cube.wbt` project shows how to use these functions to do this.


*The returned pointers are valid during one time step only as memory will be deallocated at the next time step.*

---

## NAME

wb_supervisor_node_get_center_of_mass – *get the global position of a solid's center of mass*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

const double *wb_supervisor_node_get_center_of_mass (WbNodeRef node);
```

## DESCRIPTION

The `wb_supervisor_node_get_center_of_mass()` function returns the position of the center of mass of a Solid node expressed in the global (world) coordinate system. The `node` argument must be a `Solid` node (or a derived node), otherwise the function will print a warning message and return 3 `NaN` (Not a Number) values. This function returns a vector containing exactly 3 values. If the `node` argument has a `NULL` `physics` node, the return value is always the zero vector.

The `WEBOTS_HOME/projects/samples/.wbt` project shows how to use this function.

[1] `http://en.wikipedia.org/wiki/Rotation_representation`


*The returned pointer is valid during one time step only as memory will be deallocated at the next time step.*

---

### NAME

wb_supervisor_node_get_contact_point – *get the contact point with given index in the contact point list of the given solid.*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

const double *wb_supervisor_node_get_contact_point (WbNodeRef node, int index);
```

### DESCRIPTION

The `wb_supervisor_node_get_contact_point()` function returns the contact point with given index in the contact point list of the given `Solid`. The function `wb_supervisor_node_get_number_of_contact_points()` allows you to retrieve the length of this list. Contact points are expressed in the global (world) coordinate system. If the index is less than the number of contact points, then the x (resp. y, z) coordinate of the $index$th contact point is the element number $0$ (resp. $1, 2$) in the returned array. Otherwise the function returns a `NaN` (Not a Number) value for each of these numbers. The `node` argument must be a `Solid` node (or a derived node), which moreover has no `Solid` parent, otherwise the function will print a warning message and return `NaN` values on the first 3 array components.

The `WEBOTS_HOME/projects/samples/howto/worlds/cylinder_stack.wbt` project shows how to use this function.


*The returned pointer is valid during one time step only as memory will be deallocated at the next time step.*

---

### NAME

wb_supervisor_node_get_number_of_contact_points – *get the number of contact points of the given solid*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

const double *wb_supervisor_node_get_number_of_contact_points (WbNodeRef node);
```

### DESCRIPTION

The `wb_supervisor_node_get_number_of_contact_points()` function returns the number of contact points of the given `Solid`. The `node` argument must be a `Solid` node (or a derived node), which moreover has no `Solid` parent, otherwise the function will print a warning message and return `-1`.

The `WEBOTS_HOME/projects/samples/howto/worlds/cylinder_stack.wbt` project shows how to use this function.

---

### NAME

wb_supervisor_node_get_static_balance – *return the boolean value of the static balance test based on the support polygon of a solid*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

bool wb_supervisor_node_get_static_balance (WbNodeRef node);
```

### DESCRIPTION

The `wb_supervisor_node_get_static_balance()` function returns the boolean value of the static balance test based on the support polygon of a solid. The `node` argument must be a `Solid` node (or a derived node), which moreover has no `Solid` parent. Otherwise the function will print a warning message and return `false`. The support polygon of a solid is the convex hull of the solid's contact points projected onto a plane that is orthognal to the gravity direction. The test consists in checking whether the projection of the center of mass onto this plane lies inside or outside the support polygon.

---

### NAME

wb_supervisor_node_reset_physics – *stops the inertia of the given solid*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_node_reset_physics (WbNodeRef node);
```

### DESCRIPTION

The `wb_supervisor_node_reset_physics()` function stops the inertia of the given solid. If the specified node is physics-enables, i.e. it contains a `Physics` node, then the linear and angular velocities of the corresonding body are reset to 0, hence the inertia is also zeroed. The `node` argument must be a `Solid` node (or a derived node). This function could be useful for resetting the physics of a solid after changing its translation or rotation. To stop the inertia of all available solids please refer to `wb_supervisor_simulation_reset_physics`.

---

### NAME

wb_supervisor_set_label – *overlay a text label on the 3D scene*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_set_label (int id, const char *text, double x, double
y, double size, int color, double transparency);
```

### DESCRIPTION

The `wb_supervisor_set_label()` function displays a text label overlaying the 3D scene in Webots' main window. The `id` parameter is an identifier for the label; you can choose any value in the range 0 to 65534. The same value may be used later if you want to change that label, or update the text. Id value 65535 is reserved for automatic video caption. The `text` parameter is a text string which should contain only displayable characters in the range 32-127. The `x` and `y` parameters are the coordinates of the upper left corner of the text, relative to the upper left corner of the 3D window. These floating point values are expressed in percent of the 3D window width and height, hence, they should lie in the range 0-1. The `size` parameter defines the size of the font to be used. It is expressed in the same unit as the `y` parameter. Finally, the `color` parameter defines the color of the label. It is expressed as a 3 bytes RGB integer, the most significant byte (leftmost byte in hexadecimal representation) represents the red component, the second most significant byte represents the green component and the third byte represents the blue component. The `transparency` parameter defines the transparency of the label. A transparency level of 0 means no transparency, while a transparency level of 1 means total transparency (the text will be invisible). Intermediate values correspond to semi-transparent levels.

### EXAMPLE

- `wb_supervisor_set_label(0,"hello world",0,0,0.1,0xff0000,0);`

  will display the label "hello world" in red at the upper left corner of the 3D window.

- `wb_supervisor_set_label(1,"hello Webots",0,0.1,0.1,0x00ff00,0.5);`

  will display the label "hello Webots" in semi-transparent green, just below.

- `supervisor_set_label(0,"hello universe",0,0,0.1,0xffff00,0);`

  will change the label "hello world" defined earlier into "hello universe", using a yellow color for the new text.


**language: Matlab**
*In the Matlab version of* `wb_supervisor_set_label()` *the* `color` *argument must be a vector containing the three RGB components:* `[RED GREEN BLUE]`. *Each component must be a value between 0.0 and 1.0. For example the vector* `[1 0 1]` *represents the magenta color.*

---

## NAME

wb_supervisor_simulation_quit – *terminate the simulator and controller processes*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>
void wb_supervisor_simulation_quit (int status);
```

## DESCRIPTION

The `wb_supervisor_simulator_quit()` function quits Webots, as if one was using the menu **File** > **Quit Webots**. This function makes it easier to invoke a Webots simulation from a script because it allows to terminate the simulation automatically, without human intervention. As a result of quitting the simulator process, all controller processes, including the calling supervisor controller, will terminate. The `wb_supervisor_simulator_quit()` sends a request to quit the simulator and immediately returns to the controller process, it does not wait for the effective termination of the simulator. After the call to `wb_supervisor_simulator_quit()`, the controller should call the `wb_robot_cleanup()` function and then exit. The POSIX exit status returned by Webots can be defined by the status `status` parameter. Some typical values for this are the `EXIT_SUCCESS` or `EXIT_FAILURE` macros defined into the `stdlib.h` file. Here is a C example:


**language: C**

```c
#include <webots/robot.h>
#include <webots/supervisor.h>
#include <stdlib.h>

#define TIME_STEP 32

int main(int argc, char *argv[]) {
  wb_robot_init();
  ...
  while (! finished) {
    // your controller code here
    ...
    wb_robot_step(TIME_STEP);
  }
  saveExperimentsData();
  wb_supervisor_simulation_quit(EXIT_SUCCESS); // ask Webots to terminate
  wb_robot_cleanup(); // cleanup resources
  return 0;
}
```

In object-oriented languages, there is no `wb_robot_cleanup()` function, in this case the controller should call its destructor. Here is a C++ example:


**language: C**

```c
#include <webots/Robot.hpp>
#include <webots/Supervisor.hpp>
#include <cstdlib>

using namespace webots;

class MySupervisor : public Supervisor {
public:
  MySupervisor() { ... }
  virtual ~MySupervisor() { ... }
  void run() {
    ...
    while (! finished) {
      // your controller code here
      ...
      step(TIME_STEP);
    }
    simulationQuit(EXIT_SUCCESS);  // ask Webots
       to terminate
  }

int main(int argc, char *argv[]) {
  MySupervisor *controller = new MySupervisor();
  controller->run();
  delete controller; // cleanup resources
  return 0;
}
```

## NAME

wb_supervisor_simulation_revert – *reload the current world*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_simulation_revert ();
```

## DESCRIPTION

The `wb_supervisor_simulator_revert()` function sends a request to the simulator process, asking it to reload the current world immediately. As a result of reloading the current world,

the supervisor process and all the robot processes are terminated and restarted. You may wish to save some data in a file from your supervisor program in order to reload it when the supervisor controller restarts.

---

### NAME

wb_supervisor_simulation_reset_physics – *stop the inertia of all solids in the world and reset the random number generator*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_simulation_reset_physics ();
```

### DESCRIPTION

The `wb_supervisor_simulation_reset_physics()` function sends a request to the simulator process, asking it to stop the movement of all physics-enabled solids in the world. It means that for any `Solid` node containing a `Physics` node, the linear and angular velocities of the corresponding body are reset to 0, hence the inertia is also zeroed. This is actually implemented by calling the ODE `dBodySetLinearVel()` and `dBodySetAngularVel()` functions for all bodies with a zero velocity parameter. This function is especially useful for resetting a robot to its initial position and inertia. To stop the inertia of a single `Solid` node please refer to `wb_supervisor_node_reset_physics`.

Furthermore, this function resets the seed of the random number generator used in Webots, so that noise-based simulations can be be reproduced identically after calling this function.

---

### NAME

wb_supervisor_start_movie,
wb_supervisor_stop_movie,
wb_supervisor_get_movie_status – *export the current simulation into a movie file*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_start_movie (const char *filename, int width, int height, int codec, int quality, int acceleration, bool caption);

void wb_supervisor_stop_movie ();

int wb_supervisor_get_movie_status ();
```

## DESCRIPTION

The `wb_supervisor_start_movie()` function starts saving the current simulation into a movie file. The movie creation process will complete after the `wb_supervisor_stop_movie()` function is called. The movie is saved in the file defined by the `filename` parameter. If the `filename` doesn't end with a `.mp4` extension, the file extension is completed automatically. The `codec` parameter specify the codec used when creating the movie. Currently only MPEG-4/AVC encoding is available and the `codec` value is ignored. The `quality` corresponds to the movie compression factor that affects the movie quality and file size. It should be a value between 1 and 100. Beware, that choosing a too small value may cause the video encoding program to fail because of a too low bitrate. The movie frame rate is automatically computed based on the `basicTimeStep` value of the simulation in order to produce real-time. The `acceleration` specifies the acceleration factor of the created movie with respect to the real simulation time. Default value is 1, i.e. no acceleration. If `caption` parameters is set to true, a default caption is printed on the top right corner of the movie showing the current `acceleration` value.

The `wb_supervisor_get_movie_status()` function return the current status of the movie creation. This function is particularly useful to check if the encoding process is finished and the file has been created by waiting until the returned value is equal to WB_SUPERVISOR_MOVIE_READY.

| value | status |
|---|---|
| WB_SUPERVISOR_MOVIE_RECORDING | recording the movie |
| WB_SUPERVISOR_MOVIE_SAVING | encoding and creating the movie file |
| WB_SUPERVISOR_MOVIE_READY | ready to start movie creation, movie creation completed |
| WB_SUPERVISOR_MOVIE_WRITE_ERROR | problem saving the movie frames or the encoding script |
| WB_SUPERVISOR_MOVIE_ENCODING_ERROR | problem encoding and generating the movie file |
| WB_SUPERVISOR_MOVIE_SIMULATION_ERROR | simulation not started, no movie recorded |

Table 3.13: Return values of the `wb_supervisor_get_movie_status()` function

---

## NAME

wb_supervisor_field_get_type,
wb_supervisor_field_get_type_name,
wb_supervisor_field_get_count – *get a handler and more information on a field in a node*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

WbFieldType wb_supervisor_field_get_type (WbFieldRef field);

const char *wb_supervisor_field_get_type_name (WbFieldRef field);

int wb_supervisor_field_get_count (WbFieldRef field);
```

### DESCRIPTION

The `wb_supervisor_field_get_type()` returns the data type of a field found previously from the `wb_supervisor_node_get_field()` function, as a symbolic value. If the argument is NULL, the function returns 0. Field types are defined in `webots/supervisor.h` and include for example: `WB_SF_FLOAT`, `WB_MF_NODE`, `WB_SF_STRING`, etc.

The `wb_supervisor_field_get_type_name()` returns a text string corresponding to the data type of a field found previously from the `wb_supervisor_node_get_field()` function. Field type names are defined in the VRML'97 specifications and include for example: `"SFFloat"`, `"MFNode"`, `"SFString"`, etc. If the argument is NULL, the function returns NULL.

The `wb_supervisor_field_get_count()` returns the number of items of a multiple field (MF) passed as an argument to this function. If a single field (SF) or NULL is passed as an argument to this function, it returns -1. Hence, this function can also be used to test if a field is MF (like `WB_MF_INT32`) or SF (like `WB_SF_BOOL`).


**language: C++, Java, Python**
*In the oriented-object APIs, the WB_*F_* constants are available as static integers of the* `Field` *class (for example, Field::SF_BOOL). These integers can be directly compared with the output of the* `Field::getType()`

---

### NAME

wb_supervisor_field_get_sf_bool,
wb_supervisor_field_get_sf_int32,
wb_supervisor_field_get_sf_float,
wb_supervisor_field_get_sf_vec2f,
wb_supervisor_field_get_sf_vec3f,
wb_supervisor_field_get_sf_rotation,
wb_supervisor_field_get_sf_color,
wb_supervisor_field_get_sf_string,

wb_supervisor_field_get_sf_node,
wb_supervisor_field_get_mf_int32,
wb_supervisor_field_get_mf_float,
wb_supervisor_field_get_mf_vec2f,
wb_supervisor_field_get_mf_vec3f,
wb_supervisor_field_get_mf_color,
wb_supervisor_field_get_mf_string,
wb_supervisor_field_get_mf_node – *get the value of a field*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

bool wb_supervisor_field_get_sf_bool (WbFieldRef field);

int wb_supervisor_field_get_sf_int32 (WbFieldRef field);

double wb_supervisor_field_get_sf_float (WbFieldRef field);

const double *wb_supervisor_field_get_sf_vec2f (WbFieldRef sf_field);

const double *wb_supervisor_field_get_sf_vec3f (WbFieldRef field);

const double *wb_supervisor_field_get_sf_rotation (WbFieldRef field);

const double *wb_supervisor_field_get_sf_color (WbFieldRef field);

const char *wb_supervisor_field_get_sf_string (WbFieldRef field);

WbNodeRef wb_supervisor_field_get_sf_node (WbFieldRef field);

int wb_supervisor_field_get_mf_in32 (WbFieldRef field, int index);

double wb_supervisor_field_get_mf_float (WbFieldRef field, int index);

const double *wb_supervisor_field_get_mf_vec2f (WbFieldRef field, int in-
dex);

const double *wb_supervisor_field_get_mf_vec3f (WbFieldRef field, int in-
dex);

const double *wb_supervisor_field_get_mf_color (WbFieldRef field, int in-
dex);

const char *wb_supervisor_field_get_mf_string (WbFieldRef field, int index);

WbNodeRef wb_supervisor_field_get_mf_node (WbFieldRef field, int index);
```

### DESCRIPTION

The `wb_supervisor_field_get_sf_*()` functions retrieve the value of a specified single `field` (SF). The type of the field has to match the name of the function used, otherwise the

return value is undefined (and a warning message is displayed). If the `field` parameter is NULL, it has the wrong type, or the `index` is not valid, then a default value is returned. Default values are defined as `0` and `0.0` for integer and double values, `false` in case of boolean values, and NULL for vectors, strings and pointers.

The `wb_supervisor_field_get_mf_*()` functions work the same way as the `wb_supervisor_field_get_sf_*()` functions but with multiple `field` argument. They take an additional `index` argument which refers to the index of the item in the multiple field (MF). The type of the field has to match the name of the function used and the index should be comprised between 0 and the total number of item minus one, otherwise the return value is undefined (and a warning message is displayed).

---

## NAME

wb_supervisor_field_set_sf_bool,
wb_supervisor_field_set_sf_int32,
wb_supervisor_field_set_sf_float,
wb_supervisor_field_set_sf_vec2f,
wb_supervisor_field_set_sf_vec3f,
wb_supervisor_field_set_sf_rotation,
wb_supervisor_field_set_sf_color,
wb_supervisor_field_set_sf_string,
wb_supervisor_field_set_mf_int32,
wb_supervisor_field_set_mf_float,
wb_supervisor_field_set_mf_vec2f,
wb_supervisor_field_set_mf_vec3f,
wb_supervisor_field_set_mf_color,
wb_supervisor_field_set_mf_string – *set the value of a field*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_field_set_sf_bool (WbFieldRef field, bool value);

void wb_supervisor_field_set_sf_int32 (WbFieldRef field, int value);

void wb_supervisor_field_set_sf_float (WbFieldRef field, double value);

void wb_supervisor_field_set_sf_vec2f (WbFieldRef sf_field, const double val-
ues[2]);

void wb_supervisor_field_set_sf_vec3f (WbFieldRef field, const double val-
ues[3]);

void wb_supervisor_field_set_sf_rotation (WbFieldRef field, const double val-
ues[4]);
```

```
void wb_supervisor_field_set_sf_color (WbFieldRef field, const double val-
ues[3]);

void wb_supervisor_field_set_sf_string (WbFieldRef field, const char *value);

void wb_supervisor_field_set_mf_int32 (WbFieldRef field, int index, int value);

void wb_supervisor_field_set_mf_float (WbFieldRef field, int index, double
value);

void wb_supervisor_field_set_mf_vec2f (WbFieldRef field, int index, const
double values[2]);

void wb_supervisor_field_set_mf_vec3f (WbFieldRef field, int index, const
double values[3]);

void wb_supervisor_field_set_mf_color (WbFieldRef field, int index, const
double values[3]);

void wb_supervisor_field_set_mf_string (WbFieldRef field, int index, const
char *value);
```

## DESCRIPTION

The `wb_supervisor_field_set_sf_*()` functions assign a value to a specified single `field` (SF). The type of the field has to match with the name of the function used, otherwise the value of the field remains unchanged (and a warning message is displayed).

The `wb_supervisor_field_set_mf_*()` functions work the same way as the `wb_supervisor_field_set_sf_*()` functions but with a multiple `field` (MF) argument. They take an additional `index` argument which refers to the index of the item in the multiple field. The type of the field has to match with the name of the function used and the index should be comprised between 0 and the total number of item minus one, otherwise the value of the field remains unchanged (and a warning message is displayed).


*Since Webots 7.4.4, the inertia of a solid is no longer automatically reset when changing its translation or rotation using* `wb_supervisor_field_set_sf_vec2f` *and* `wb_supervisor_field_set_sf_rotation` *functions. If needed, the user has to explicitly call* `wb_supervisor_node_reset_physics` *function.*

## EXAMPLES

The `texture_change.wbt` world, located in the `projects/samples/howto/worlds` directory, shows how to change a texture from the supervisor while the simulation is running. The `soccer.wbt` world, located in the `projects/samples/demos/worlds` directory, provides a simple example for getting and setting fields with the above described functions.

## NAME

wb_supervisor_field_import_mf_node – *import a node into an MF_NODE field (typically a "children" field) from a file*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/supervisor.h>

void wb_supervisor_field_import_mf_node (WbFieldRef field, int position, const
char *filename);
```

## DESCRIPTION

The `wb_supervisor_field_import_mf_node()` function imports a Webots node into an MF_NODE. This node should be defined in a `.wbo` file referenced by the `filename` parameter. Such a file can be produced easily from Webots by selecting a node in the scene tree window and using the **Export** button.

The `position` parameter defines the position in the MF_NODE where the new node will be inserted. It can be positive or negative. Here are a few examples for the `position` parameter:

- 0: insert at the beginning of the scene tree.
- 1: insert at the second position.
- 2: insert at the third position.
- -1: insert at the last position.
- -2: insert at the second position from the end of the scene tree.
- -3: insert at the third position from the end.

The `filename` parameter can be specified as an absolute or a relative path. In the later case, it is relative to the location of the supervisor controller.

This function is typically used in order to add a node into a "children" field. Note that a node can be imported into the scene tree by calling this function with the "children" field of the root node.


*Note that this function is still limited in the actual Webots version. For example, a device imported into a Robot node doesn't reset the Robot, so the device cannot be get by using the* `wb_robot_get_device()` *function.*

## 3.66 TextureCoordinate

```
TextureCoordinate {
  MFVec2f   point    []    # (-inf,inf)
}
```

The TextureCoordinate node specifies a set of 2D texture coordinates used by vertex-based Geometry nodes (e.g., IndexedFaceSet) to map textures to vertices. Textures are two-dimensional color functions that, given a coordinate pair *(s,t)*, return a color value *color(s,t)*. Texture map values (ImageTexture) range from 0.0 to 1.0 along the s and t axes. Texture coordinates identify a location (and thus a color value) in the texture map. The horizontal coordinate *s* is specified first, followed by the vertical coordinate *t*.

## 3.67 TextureTransform

```
TextureTransform {
  SFVec2f   center         0 0    # (-inf,inf)
  SFFloat   rotation       0      # (-inf,inf)
  SFVec2f   scale          1 1    # (-inf,inf)
  SFVec2f   translation    0 0    # (-inf,inf)
}
```

The TextureTransform node defines a 2D transformation that is applied to texture coordinates. This node affects the way textures are applied to the surface of a Geometry. The transformation consists of (in order):

- a translation;
- a rotation about the center point;
- a non-uniform scaling operation about the center point.

These parameters support changes in the size, orientation, and position of textures on shapes. Note that these operations appear reversed when viewed on the surface of a geometric node. For example, a scale value of (2 2) will scale the texture coordinates, with the net effect of shrinking the texture size by a factor of 2 (texture coordinates are twice as large and thus cause the texture to repeat). A translation of (0.5 0.0) translates the texture coordinates +0.5 units along the *s* axis, with the net effect of translating the texture -0.5 along the *s* axis on the geometry's surface. A rotation of $\pi/2$ of the texture coordinates results in a $-\pi/2$ rotation of the texture on the geometric node.

The center field specifies a translation offset in texture coordinate space about which the rotation and scale fields are applied. The scale field specifies a scaling factor in *s* and *t* of

$$\mathbf{T}' = CSRC^{-1}T\mathbf{T}$$

$$\mathbf{T}' = \begin{pmatrix} s' \\ t' \\ 0 \end{pmatrix}, \quad \mathbf{T} = \begin{pmatrix} s \\ t \\ 0 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 & 0 & C_s \\ 0 & 1 & C_t \\ 0 & 0 & 1 \end{pmatrix}, S = \begin{pmatrix} S_s & 0 & 0 \\ 0 & S_t & 0 \\ 0 & 0 & 1 \end{pmatrix}, R = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}, T = \begin{pmatrix} 1 & 0 & T_s \\ 0 & 1 & T_t \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3.40: Texture transformation in matrix notation

the texture coordinates about the center point. The `rotation` field specifies a rotation in radians of the texture coordinates about the center point after the scaling operation has been applied. A positive rotation value makes the texture coordinates rotate counterclockwise about the center, thereby rotating the appearance of the texture clockwise. The `translation` field specifies a translation of the texture coordinates.

Given a point **T** with texture coordinates $(s, t)$ and a `TextureTransform` node, **T** is transformed into the point **T'**$= (s', t')$ by the three intermediate transformations described above. Let $C$ be the translation mapping $(0, 0)$ to the point $(C_s, C_t)$, $T$ be the translation of vector $(T_s, T_t)$, $R$ the rotation with center $(0, 0)$ and angle $\theta$ , and $S$ a scaling with scaling factors $S_s, S_t$. In matrix notation, the corresponding `TextureTransform` reads as

where $C^{-1}$ denotes the matrix inverse of $C$.

Note that `TextureTransform` nodes cannot combine or accumulate.

# 3.68 TouchSensor

Derived from `Device`.

```
TouchSensor {
  SFString   type          "bumper"
  MFVec3f    lookupTable   [ 0 0 0, 5000 50000 0 ]
  SFFloat    resolution    -1
}
```

## 3.68.1 Description

A `TouchSensor` node is used to model a bumper or a force sensor. The `TouchSensor` comes in three different types. The ”bumper” type simply detects collisions and returns a boolean status. The ”force” type measures the force exerted on the sensor’s body on one axis ($z$-axis). The ”force-3d” type measures the 3d force vector exerted by external object on the sensor’s body.

Examples of using the TouchSensor are provided by the hoap2_sumo.wbt and hoap2_walk.wbt worlds (located in the projects/robots/hoap2/worlds directory of Webots) and by the force_sensor.wbt and bumper.wbt worlds (located in the projects/samples/devices/worlds directory of Webots).

### 3.68.2 Field Summary

- type: allows the user to select the type of sensor: "bumper", "force", or "force-3d", described below.
- lookupTable: similar to the one used by the DistanceSensor node.
- resolution: This field allows to define the resolution of the sensor, the resolution is the smallest change that it is able to measure. Setting this field to -1 (default) means that the sensor has an 'infinite' resolution (it can measure any infinitesimal change). This field is used only if the type is "force" or "force-3d" and accepts any value in the interval (0.0, inf).

### 3.68.3 Description

#### "bumper" Sensors

A "bumper" TouchSensor returns a boolean value that indicates whether or not there is a collision with another object. More precisely, it returns 1.0 if a collision is detected and 0.0 otherwise. A collision is detected when the boundingObject of the TouchSensor intersects the boundingObject of any other Solid object. The lookupTable field of a "bumper" sensor is ignored. The Physics node of a "bumper" sensor is not required.

#### "force" Sensors

A "force" TouchSensor computes the (scalar) amount of force currently exerted on the sensor's body along the $z$-axis. The sensor uses this equation: $r = |f| * cos(\alpha)$, where $r$ is the return value, $f$ is the cumulative force currently exerted on the sensor's body, and $\alpha$ is the angle between $f$ and the sensor's $z$-axis. So the "force" sensor returns the projection of the force on its $z$-axis; a force perpendicular to the $z$-axis yields zero. For this reason, a "force" sensor must be oriented such that its positive $z$-axis points outside of the robot, in the direction where the force needs to me measured. For example if the TouchSensor is used as foot sensor then the $z$-axis should be oriented downwards. The scalar force value must be read using the wb_touch_sensor_get_value() function.

### "force-3d" Sensors

A "force-3d" TouchSensor returns a 3d-vector that represents the cumulative force currently applied to its body. This 3d-vector is expressed in the coordinate system of the TouchSensor. The length of the vector reflects the magnitude of the force. The force vector must be read using the wb_touch_sensor_get_values() function.

| sensor type | "bumper" | "force" | "force-3d" |
|---|---|---|---|
| boundingObject | required | required | required |
| Physics node | not required | required | required |
| lookupTable | ignored | used | used |
| return value | 0 or 1 | scalar | vector |
| API function | wb_touch_sensor_get_-value() | wb_touch_sensor_get_-value() | wb_touch_sensor_get_-values() |

Table 3.14: TouchSensor types

### Lookup Table

A "force" and "force-3d" sensors can optionally specify a lookupTable to simulate the possible non-linearity (and saturation) of the real device. The lookupTable allows the user to map the simulated force measured in Newtons (N) to an output value that will be returned by the wb_touch_sensor_get_value() function. The value returned by the force sensor is first computed by the ODE physics engine, then interpolated using the lookupTable, and finally noise is added (if specified in the lookupTable). Each line of the lookupTable contains three numbers: (1) an input force in Newtons, (2) the corresponding output value, and (3) a noise level between 0.0 and 1.0 (see DistanceSensor for more info). Note that the default lookupTable of the TouchSensor node is:

```
[   0     0 0
 5000 50000 0 ]
```

and hence it maps forces between 0 and 5000 Newtons to output values between 0 and 50000, the output unit being 0.1 Newton. You should empty the lookupTable to have Newtons as output units.

### Collision detection

TouchSensors detect collisions based on the 3D geometry of its boundingObject. So the boundingObject must be specified for every type of TouchSensor. Because the actual 3D intersection of the sensors boundingObjects with other boundingObjects is used in the calculation, it is very important that the sensors' boundingObjects are correctly positioned; they should be able to collide with other objects, otherwise they would be ineffective. For that

reason, the boundingObjects of TouchSensors should always extend beyond the other boundingObjects of the robot in the area where the collision is expected.

For example, let's assume that you want to add a TouchSensor under the foot of a humanoid robot. In this case, it is critical that the boundingObject of this sensor (and not any other boundingObject of the robot) makes the actual contact with the floor. Therefore, it is necessary that the sensor's boundingObject extend below any other boundingObject of the robot (e.g., foot, ankle, etc.).

#### Coordinate System

It is easy to check the orientation of the coordinate system of a TouchSensor: if you select the TouchSensor object in the Scene Tree, then only the bounding object of this TouchSensor should be shown in the 3D window. If you zoom in on this bounding object, you should see the red/green/blue depiction of the TouchSensor's coordinate system (the color coding is: *x/y/z* = red/green/blue). For a "force" sensor, the blue ($z$) component should point in the direction where the collision is expected.

#### Accuracy

The force measured by the ODE physics engine is only a rough approximation of a real physical force. This approximation usually improves as the basicTimeStep (WorldInfo node) decreases.

### 3.68.4 TouchSensor Functions

---

#### NAME

wb_touch_sensor_enable,
wb_touch_sensor_disable,
wb_touch_sensor_get_sampling_period,
wb_touch_sensor_get_value,
wb_touch_sensor_get_values – *enable, disable and read last touch sensor measurements*

#### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/touch_sensor.h>

void wb_touch_sensor_enable (WbDeviceTag tag, int ms);

void wb_touch_sensor_disable (WbDeviceTag tag);
```

```
int wb_touch_sensor_get_sampling_period (WbDeviceTag tag);

double wb_touch_sensor_get_value (WbDeviceTag tag);

const double *wb_touch_sensor_get_values (WbDeviceTag tag);
```

### DESCRIPTION

`wb_touch_sensor_enable()` allows the user to enable a touch sensor measurement every `ms` milliseconds.

`wb_touch_sensor_disable()` turns the touch sensor off, saving computation time.

`wb_touch_sensor_get_value()` returns the last value measured by a ”bumper” or ”force” `TouchSensor`. This function can be used with a sensor of type ”bumper” or ”force”. For a ”force” sensor, the value may be altered by an optional lookup table. For a ”bumper” sensor, the value can be 0.0 or 1.0.

The `wb_touch_sensor_get_sampling_period()` function returns the period given into the `wb_touch_sensor_enable()` function, or 0 if the device is disabled.

`wb_touch_sensor_get_values()` returns the last force vector measured by a ”force-3d” `TouchSensor`. This function can be used with a sensor of type ”force-3d” exclusively.

---

### NAME

wb_touch_sensor_get_type – *get the touch sensor type*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/touch_sensor.h>

int wb_touch_sensor_get_type (WbDeviceTag tag);
```

### DESCRIPTION

This function allows to retrieve the touch sensor type defined by the `type` field. If the value of the `type` field is ”force” then this function returns WB_TOUCH_SENSOR_FORCE, if it is ”force-3d” then it returns WB_TOUCH_SENSOR_FORCE3D and otherwise it returns WB_-TOUCH_SENSOR_BUMPER.

## 3.69 Transform

Derived from `Group`.

| TouchSensor.type | return value |
|---|---|
| ”bumper” | WB_TOUCH_SENSOR_BUMPER |
| ”force” | WB_TOUCH_SENSOR_FORCE |
| ”force-3d” | WB_TOUCH_SENSOR_FORCE3D |

Table 3.15: Return values for the `wb_touch_sensor_get_type()` function

```
Transform {
  SFVec3f      translation   0 0 0      # 3D vector
  SFRotation   rotation      0 1 0 0    # 3D unit vector,angle (rad)
  SFVec3f      scale         1 1 1      # 3 real scaling factors
}
```

Direct derived nodes: `Solid`.

### 3.69.1 Description

The `Transform` node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its parent.

### 3.69.2 Field Summary

- The `translation` field defines the translation from the parent coordinate system to the children's coordinate system.

- The `rotation` field defines an arbitrary rotation of the children's coordinate system with respect to the parent coordinate system. This field contains four floating point values: *rx, ry, rz* and $\alpha$. The first three numbers, *rx ry rz*, define a normalized vector giving the direction of the axis around which the rotation must be carried out. The fourth value, $\alpha$, specifies the rotation angle around the axis in radians. When $\alpha$ is zero, no rotation is carried out. All the values of the rotation field can be positive or negative. Note however that the length of the 3D vector *rx ry rz* must be normalized (i.e. its length is 1.0), otherwise the outcome of the simulation is undefined.

  For example, a rotation of $\pi/2$ radians around the *z*-axis is represented like this:

  ```
  rotation 0 0 1 1.5708
  ```

  A rotation of $\pi$ radians around an axis located exactly between the *x* and *y*-axis is represented like this:

  ```
  rotation 0.7071 0.7071 0 3.1416
  ```

  And finally, note that these two rotations are identical:

```
rotation 0 1 0 -1.5708
rotation 0 -1 0 1.5708
```

- The `scale` field specifies a possibly non-uniform scale. Only positive values are permitted; non-positive values scale are automatically reset to 1. Graphical objects support any positive non-uniform scale whereas physical objects are subjected to restrictions. This is so because scaled geometries must remain admissible for the physics engine collision detection. Restrictions for `Geometries` placed inside `boundingObjects` are as follows: `Sphere`s and `Capsule`s only support uniform scale; the scale coordinates x and z of a `Transform` with a `Cylinder` descendant must be the same. For the remaining `Geometries`, the scale is not restricted. The `scale` fields of a `Solid` node and its derived nodes must be uniform, i.e., of the form *x x x* so as to comply with the physics engine. For such nodes a positive `scale` field initially set to *x y z* is automatically reset to *x x x*. The same holds for a `Transform` placed inside a `boundingObject` and with a `Sphere` or a `Capsule` descendant. In the case of a `Cylinder`, *x y z* will be reset to *x z x*. If some value changes within one of the previous constrained scale fields, the two others are actuated using the new value and the corresponding constraint rule.


*If a* `Transform` *is named using the* `DEF` *keyword and later referenced inside a* `boundingObject` *with a USE statement, the constraint corresponding to its first* `Geometry` *descendant applies to the* `scale` *fields of the defining* `Transform` *and of all its further references.*

## 3.70 Viewpoint

```
Viewpoint {
  SFFloat       fieldOfView     0.785398  # (0,pi)
  SFRotation    orientation     0 0 1 0   # 3D unit vector, angle (rad)
  SFVec3f       position        0 0 0     # 3D vector
  SFString      description     ""
  SFFloat       near            0.05      # [0,inf)
  SFString      follow          ""
}
```

The `Viewpoint` node defines a specific location in the local coordinate system from which the user may view the scene.

The `position` and `orientation` fields of the `Viewpoint` node specify absolute locations in the coordinate system. In the default position and orientation, the viewer is on the *z*-axis, looking down the -*z*-axis toward the origin with +*x* to the right and +*y* straight up.

Navigating in the 3D view by dragging the mouse pointer dynamically changes the `position` and the `orientation` fields of the `Viewpoint` node.

The `fieldOfView` field specifies the viewing angle in radians. A small field of view roughly corresponds to a telephoto lens; a large field of view roughly corresponds to a wide-angle lens.

The `near` field defines the distance from the camera to the near clipping plane. This plane is parallel to the projection plane for the 3D display in the main window. The near field determines the precision of the OpenGL depth buffer. A too small value may cause depth fighting between overlaid polygons, resulting in random polygon overlaps. The far clipping plane is parallel to the near clipping plane and is defined at an infinite distance from the camera. The far clipping plane distance cannot be modified.

The `near` and the `fieldOfView` fields define together the viewing frustum. Any 3D shape outside this frustum won't be rendered. Hence, shapes too close (standing between the camera and the near plane) won't appear.

The `follow` field can be used to specify the name of a robot (or other object) that the viewpoint needs to follow during the simulation. If the string is empty, or if it does not correspond to any object, then the viewpoint will remain fixed. The `follow` field is modified by the **View** > **Follow Object** menu item.

## 3.71 WorldInfo

```
WorldInfo {
  SFString   title                          ""
  MFString   info                           []
  SFVec3f    gravity                        0 -9.81 0
  SFFloat    CFM                            0.00001  # [0,inf)
  SFFloat    ERP                            0.2      # [0,1]
  SFString   fast2d                         ""
  SFString   physics                        ""
  SFString   sound                          ""
  SFFloat    basicTimeStep                  32       # in ms
  SFFloat    FPS                            60
  SFFloat    physicsDisableTime             1        # time after
     which the objects are disabled if they are idle
  SFFloat    physicsDisableLinearThreshold  0.01     # threshold
     determining if an object is idle or not
  SFFloat    physicsDisableAngularThreshold 0.01     # threshold
     determining if an object is idle or not
  SFNode     defaultDamping                 NULL     # default damping
      parameters
  SFFloat    inkEvaporation                 0        # make ground
     textures evaporate
  SFVec3f    northDirection                 1 0 0    # for compass and
      InertialUnit
```

```
  SFFloat    lineScale                      0.1       # control the
     length of every arbitrary-sized lines
  MFNode     contactProperties              []        # see
     ContactProperties node
}
```

The WorldInfo node provides general information on the simulated world:

- The title field should briefly describe the purpose of the world.
- The info field should give additional information, like the author who created the world, the date of creation and a description of the purpose of the world. Several character strings can be used.
- The gravity field defines the gravitational acceleration to be used in physics simulation. The gravity is set by default to the gravity found on earth. You should change it if you want to simulate rover robots on Mars, for example. The gravity vector defines the orientation of the ground plane used by InertialUnits.
- The ERP field defines the *Error Reduction Parameter* use by ODE to manage contacts joints. This applies by default to all contact joints, except those whose contact properties are defined in a ContactProperties node. The ERP specifies what proportion of the contact joint error will be fixed during the next simulation step. If ERP=0 then no correcting force is applied and the bodies will eventually drift apart as the simulation proceeds. If ERP=1 then the simulation will attempt to fix all joint error during the next time step. However, setting ERP=1 is not recommended, as the joint error will not be completely fixed due to various internal approximations. A value of ERP=0.1 to 0.8 is recommended (0.2 is the default).
- The CFM field defines the *Constraint Force Mixing* use by ODE to manage contacts joints. This applies by default to all contact joints, except those whose contact properties are defined in a ContactProperties node. Along with the ERP, the CFM controls the spongyness and springyness of the contact joint. If a simulation includes heavy masses, then decreasing the CFM value for contacts will prevent heavy objects from penetrating the ground. If CFM is set to zero, the constraint will be hard. If CFM is set to a positive value, it will be possible to violate the constraint by *pushing on it* (for example, for contact constraints by forcing the two contacting objects together). In other words the constraint will be soft, and the softness will increase as CFM increases. What is actually happening here is that the constraint is allowed to be violated by an amount proportional to CFM times the restoring force that is needed to enforce the constraint (see ODE documentation for more details).
- The fast2d field allows the user to switch to Fast2d mode. If the fast2d field is not empty, Webots tries to load a Fast2d plugin with the given name. Subsequent kinematics, collision detection, and sensor measurements are computed using the plugin. The objective

is to carry out these calculations using a simple 2D world model that can be computed faster than the 3D equivalent. The Webots distribution comes with a pre-programmed plugin called "enki." In addition, a Webots user can implement his own plugin. However, Fast2d mode is limited to simple world models containing only cylindrical and rectangular shapes. The Webots distribution contains an example of world using Fast2d: `khepera_fast2d.wbt` (located in the `projects/robots/khepera/worlds` directory of Webots). For more information on the Fast2d plugin, please refer to chapter 7.

- The `physics` field refers to a physics plugin which allows the user to program custom physics effects using the ODE API. See chapter 6 for a description on how to set up a physics plugin. This is especially useful for modeling hydrodynamic forces, wind, non-uniform friction, etc.

- The `sound` is an experimental field not effective yet.

- The `basicTimeStep` field defines the duration of the simulation step executed by Webots. It is a floating point value expressed in milliseconds. The minimum value for this field is 0.001, that is, one microsecond. Setting this field to a high value will accelerate the simulation, but will decrease the accuracy and the stability, especially for physics computations and collision detection. It is usually recommended to tune this value in order to find a suitable speed/accuracy trade-off.

- The `FPS` (frames per second) field represents the maximum rate at which the 3D display of the main window is refreshed in `Real-time` and `Run` mode. It is particularly usefull to limit the refresh rate, in order to speed up simulations having a small `basicTimeStep` value.

- The `physicsDisableTime` determines the amount of simulation time (in seconds) before the idle solids are automatically disabled from the physics computation. Set this to zero to disable solids as soon as they become idle. This field matchs directly with the `dBodySetAutoDisableTime` ODE function. This feature can improve significantly the speed of the simulation if the solids are static most of the time. The solids are enabled again after any interaction (collision, movement, ...).

- The `physicsDisableLinearThreshold` determines the solid's linear velocity threshold (in meter/seconds) for automatic disabling. The body's linear velocity magnitude must be less than this threshold for it to be considered idle. This field is only useful if `physicsDisableTime` is bigger or equal to zero. This field matchs directly with the `dBodySetAutoDisableLinearThreshold` ODE function.

- The `physicsDisableAngularThreshold` determines the solid's angular velocity threshold (in radian/seconds) for automatic disabling. The body's angular velocity magnitude must be less than this threshold for it to be considered idle. This field is only useful if `physicsDisableTime` is bigger or equal to zero. This field matchs directly with the `dBodySetAutoDisableAngularThreshold` ODE function.

- The `defaultDamping` field allows to specifiy a `Damping` node that defines the default damping parameters that must be applied to each `Solid` in the simulation.
- If the `inkEvaporation` field is set to a non-null value, the colors of the ground textures will slowly turn to white. This is useful on a white-textured ground in conjunction with a `Pen` device, in order to have the track drawn by the `Pen` device disappear progressively. The `inkEvaporation` field should be a positive floating point value defining the speed of evaporation. This evaporation process is a computationally expensive task, hence the ground textures are updated only every `WorldInfo.basicTimeStep` * `WorldInfo.displayRefresh` milliseconds (even in fast mode). Also, it is recommended to use ground textures with low resolution to speed up this process. As with the pen device, the modified ground textures can be seen only through infra-red distance sensors, and not through cameras (as the ground textures are not updated on the controller side).
- The `northDirection` field is used to indicate the direction of the *virtual north* and is used by `Compass` and `InertialUnit` nodes.
- The `lineScale` field allows the user to control the size of the optionally rendered arbitrary-sized lines or objects such as the connector and the hinge axes, the local coordinate systems and centers of mass of solid nodes, the rays of light sensors, the point light representations, the camera frustums, or the offsets used for drawing bounding objects and the laser beam. Increasing the `lineScale` value can help in case of depth fighting problems between the red spot of a laser beam and the detected object. The value of this field is somehow arbitrary, but setting this value equal to the average size of a robot (expressed in meter) is likely to be a good initial choice.
- The `contactProperties` field allows to specifiy a number of `ContactProperties` nodes that define the behavior when `Solid` nodes collide.

# Chapter 4

# Motion Functions

The `wbu_motion*()` functions provide a facility for reading and playing back `.motion` files. Motion file specify motion sequences that usually involve several motors playing simultaneously, e.g., a walking sequence, a standing up sequence, etc.

The motions files have a user-readable format. They can be edited using the motion editor. More information on how to use the motion editor can be find into the user guide.

## 4.1 Motion

---

### NAME

wbu_motion_new,
wbu_motion_delete – *obtaining and releasing a motion file handle*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/utils/motion.h>

WbMotionRef wbu_motion_new (const char *filename);

void wbu_motion_delete (WbMotionRef motion);
```

### DESCRIPTION

The `wbu_motion_new()` function allows to read a motion file specified by the `filename` parameter. The `filename` can be specified either with an absolute path or a path relative to the controller directory. If the file can be read, if its syntax is correct and if it contains at least one pose and one joint position, then `wbu_motion_new()` returns a `WbMotionRef` that can

be used as parameter in further `wbu_motion_*()` calls. If an error occurred, an error message is printed to Webots' console, and `NULL` is returned. Motions are created in *stopped mode*, `wbu_motion_play()` must be called to start the playback.

The `wbu_motion_delete()` function frees all the memory associated with the `WbMotionRef`. This `WbMotionRef` can no longer be used afterwards.


**language: C++, Java, Python**
*The constructor and destructor of the Motion class are used instead of* `wbu_motion_new()` *and* `wbu_motion_delete()`*. In these languages, an error condition can be detected by calling the* `isValid()` *function after the constructor. If* `isValid()` *yields* `false` *then the* `Motion` *object should be explicitly deleted. See example below.*


**language: C++**

```
1 Motion *walk = new Motion(filename);
2 if (! walk->isValid()) {
3   cerr << "could not load file: " << filename <<
      endl;
4   delete walk;
5 }
```

## SEE ALSO

wbu_motion_play

---

## NAME

wbu_motion_play,
wbu_motion_stop,
wbu_motion_set_loop,
wbu_motion_set_reverse – *Controlling motion files playback*

## SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/utils/motion.h>

void wbu_motion_play (WbMotionRef motion);

void wbu_motion_stop (WbMotionRefmotion);
```

```
void wbu_motion_set_loop (WbMotionRef motion, bool loop);

void wbu_motion_set_reverse (WbMotionRefmotion, bool reverse);
```

## DESCRIPTION

The `wbu_motion_play()` starts the playback of the specified motion. This function registers the motion to the playback system, but the effective playback happens in the background and is activated as a side effect of calling the `wb_robot_step()` function. If you want to play a file and wait for its termination you can do it with this simple function:


**language: C**

```c
void my_motion_play_sync(WbMotionRef motion)
{
  wbu_motion_play(motion);
  do {
    wb_robot_step(TIME_STEP);
  }
  while (! wbu_motion_is_over(motion));
}
```

Several motion files can be played simultaneously by the same robot, however if two motion files have common joints, the behavior is undefined.

Note that the steps of the `wb_robot_step()` function and the pose intervals in the motion file can differ. In this case Webot computes intermediate joint positions by linear interpolation.

The `wbu_motion_stop()` interrupts the playback of the specified motion but preserves the current position. After interruption the playback can be resumed with `wbu_motion_play()`.

The `wbu_motion_set_loop()` sets the *loop mode* of the specified motion. If the *loop mode* is `true`, the motion repeats when it reaches either the end or the beginning (*reverse mode*) of the file. The *loop mode* can be used, for example, to let a robot repeat a series of steps in a walking sequence. Note that the loop mode can be changed while the motion is playing.

The `wbu_motion_set_reverse()` sets the *reverse mode* of the specified motion. If the *reverse mode* is `true`, the motion file plays backwards. For example, by using the *reverse mode*, it may be possible to turn a forwards walking motion into a backwards walking motion. The *reverse mode* can be changed while the motion is playing, in this case, the motion will go back from its current position.

By default, the *loop mode* and *reverse mode* of motions are `false`.

### SEE ALSO

wbu_motion_new

---

### NAME

wbu_motion_is_over,
wbu_motion_get_duration,
wbu_motion_get_time,
wbu_motion_set_time – *controlling the playback position*

### SYNOPSIS [C++] [Java] [Python] [Matlab]

```
#include <webots/utils/motion.h>
bool wbu_motion_is_over (WbMotionRef motion);
int wbu_motion_get_duration (WbMotionRefmotion);
int wbu_motion_get_time (WbMotionRef motion, bool loop);
void wbu_motion_set_time (WbMotionRefmotion, int ms);
```

### DESCRIPTION

The `wbu_motion_is_over()` function returns `true` when the playback position has reached the end of the motion file. That is when the last pose has been sent to the `Motor` nodes using the `wb_motor_set_position()` function. But this does not mean that the motors have yet reached the specified positions; they may be slow or blocked by obstacles, robots, walls, the floor, etc. If the motion is in *loop mode*, this function returns always `false`. Note that `wbu_motion_is_over()` depends on the *reverse mode*. `wbu_motion_is_over()` returns `true` when *reverse mode* is `true` and the playback position is at the beginning of the file or when *reverse mode* is `false` and the playback position is at the end of the file.

The `wbu_motion_get_duration()` function returns the total duration of the motion file in milliseconds.

The `wbu_motion_get_time()` function returns the current playback position in milliseconds.

The `wbu_motion_set_time()` function allows to change the playback position. This enables, for example, to skip forward or backward. Note that, the position can be changed whether the motion is playing or stopped. The minimum value is 0 (beginning of the motion), and the maximum value is the value returned by the `wbu_motion_get_duration()` function (end of the motion).

### SEE ALSO

wbu_motion_play

# Chapter 5

# PROTO

A PROTO defines a new node type in terms of built-in nodes or other PROTO nodes. The PROTO interface defines the fields for the PROTO. Once defined, PROTO nodes may be instantiated in the scene tree exactly like built-in nodes.

## 5.1 PROTO Definition

### 5.1.1 Interface

The PROTO definition defines exactly what the PROTO does in terms of the built-in nodes or of the instances of other PROTO nodes. Here is the syntax for a PROTO definition:

```
PROTO protoName [ protoInterface ] { protoBody }
```

The interface is a sequence of field declarations which specify the types, names and default values for the PROTO's fields. A field declaration has this syntax:

```
field fieldType fieldName defaultValue
```

where `field` is a reserved keyword, `fieldType` is one of: `SFNode`, `SFColor`, `SFFloat`, `SFInt32`, `SFString`, `SFVec2f`, `SFVec3f`, `SFRotation`, `SFBool`, `MFNode`, `MFColor`, `MFFloat`, `MFInt32`, `MFString`, `MFVec2f` and `MFVec3f`. `fieldName` is a freely chosen name for this field and `defaultValue` is a literal default value that depends on `fieldType`.

Here is an example of PROTO definition:

```
PROTO MyProto [
  field SFVec3f    translation   0 0 0
  field SFRotation position      0 1 0 0
  field SFColor    color         0.5 0.5 0.5
  field SFNode     physics       NULL
```

```
  field MFNode     extensionSlot []
]
{
  Solid {
    ...
  }
}
```

The type of the root node in the body of the PROTO definition (a `Solid` node in this example) is called the *base type* of the PROTO. The base type determines where instantiations of the PROTO can be placed in the scene tree. For example, if the base type of a PROTO is `Material`, then instantiations of the PROTO can be used wherever a `Material` mode can be used. A PROTO whose base node is another PROTO is called *derived PROTO*.

### 5.1.2 IS Statements

Nodes in the PROTO definition may have their fields associated with the fields of the PROTO interface. This is accomplished using IS statements in the body of the node. An IS statement consists of the name of a field from a built-in node followed by the keyword IS followed by the name of one of the fields of the PROTO interface:

For example:

```
PROTO Bicycle [
  field SFVec3f    position   0 0 0
  field SFRotation rotation   0 1 0 0
  field SFColor    frameColor 0.5 0.5 0.5
  field SFBool     hasBrakes  TRUE
]
{
  Solid {
    translation IS position
    rotation IS rotation
    ...
    children [
      ...
    ]
    ...
  }
}
```

IS statements may appear inside the PROTO definition wherever fields may appear. IS statements shall refer to fields defined in the PROTO declaration. Multiple IS statements for the same field in the PROTO interface declaration is valid.

It is an error for an IS statement to refer to a non-existent interface field. It is an error if the type of the field being associated does not match the type declared in the PROTO's interface. For example, it is illegal to associate an `SFColor` with an `SFVec3f`. It is also illegal to associate a `SFColor` with a `MFColor` or vice versa. Results are undefined if a field of a node in the PROTO body is associated with more than one field in the PROTO's interface.

## 5.2 PROTO Instantiation

Each PROTO instance can be considered to be a complete copy of the PROTO, with its interface fields and body nodes. PROTO are instantiated using the standard node syntax, for example:

```
Bicycle {
  position   0 0.5 0
  frameColor 0 0.8 0.8
  hasBrakes  FALSE
}
```

When PROTO instances are read from a `.wbt` file, field values for the fields of the PROTO interface may be given. If given, the field values are used for all nodes in the PROTO definition that have IS statements for those fields.

## 5.3 Example

A complete example of PROTO definition and instantiation is provided here. The PROTO is called `TwoColorChair`; it defines a simple chair with four legs and a seating part. For simplicity, this PROTO does not have bounding objects nor `Physics` nodes. A more complete example of this PROTO named `SimpleChair` is provided in Webots distribution.

The `TwoColorChair` PROTO allows to specify two colors: one for the legs and one for the seating surface of the chair. The interface also defines a `translation` field and a `rotation` field that are associated with the equally named fields of the PROTO's `Solid` base node. This allows to store the position and orientation of the PROTO instances.

`TwoColorChair.proto`:

```
# A two-color chair

PROTO TwoColorChair [
  field SFVec3f    translation       0 0.91 0
  field SFRotation rotation          0 1 0 0
  field SFColor    legColor          1 1 0
  field SFColor    seatColor         1 0.65 0
  field SFNode     seatGeometry      NULL
```

```
  field MFNode     seatExtensionSlot []       ]
{
  Solid {
    translation IS translation
    rotation IS rotation
    children [
      Transform {
        translation 0 0 -0.27
        children IS seatExtensionSlot
      }
      Transform {
        translation 0 -0.35 0
        children [
          Shape {
            appearance Appearance {
              material Material { diffuseColor IS seatColor }
            }
            geometry IS seatGeometry
          }
        ]
      }
      Transform {
        translation 0.25 -0.65 -0.23
        children [
          DEF LEG_SHAPE Shape {
            appearance Appearance {
              material Material { diffuseColor IS legColor }
            }
            geometry Box { size 0.075 0.52 0.075 }
          }
        ]
      }
      Transform {
        translation -0.25 -0.65 -0.23
        children [ USE LEG_SHAPE ]
      }
      Transform {
        translation 0.25 -0.65 0.2
        children [ USE LEG_SHAPE ]
      }
      Transform {
        translation -0.25 -0.65 0.2
        children [ USE LEG_SHAPE ]
      }
    ]
```

```
  }
}
```

As you can observe in this example, it is perfectly valid to have several IS statement for one interface field (`seatColor`), as long as the types match. It is also possible to use IS statements inside a defined (DEF) node and then to reuse (USE) that node. This is done here with the `diffuseColor IS legColor` statement placed inside the `DEF LEG_SHAPE Shape` node which is then reused (USE) several times below.

The `ProtoInstantiationExample.wbt` file below exemplifies the instantiation of this PROTO. PROTO nodes are instantiated using the regular node syntax. Fields with the default value can be omitted. Fields which value differ from the default must be specified.

`TwoChairs.wbt`:

```
#VRML_SIM V6.0 utf8

WorldInfo {
}
Viewpoint {
  orientation 0.628082 0.772958 0.089714 5.69177
  position -0.805359 1.75254 2.75772
}
Background {
  skyColor [
    0.4 0.7 1
  ]
}
DirectionalLight {
  direction -0.3 -1 -0.5
  castShadows TRUE
}
TwoColorChair {
  seatGeometry Cylinder {
    height 0.075
    radius 0.38
  }
}
TwoColorChair {
  translation 1.2 0.91 0
  seatColor 0.564706 0.933333 0.564706
  seatGeometry Box { size 0.6 0.075 0.52 }
  seatExtensionSlot [
    Shape {
      appearance Appearance {
        material Material { diffuseColor 0.564706 0.933333 0.564706}
      }
```


Figure 5.1: Two instances of the TwoColorChair PROTO in Webots

```
      geometry Box { size 0.6 0.67 0.0275 }
    }
  ]
}
```

The `TwoChairs.wbt` file once loaded by Webots appears as shown in figure 5.1.

As you can observe in this example, defining MFNode fields in the PROTO interface allows to reuse the same model for slightly different objects or robots. Extenstion slots like `seatExtensionSlot` field could, for example, be used to add additional devices to a base robot without needing to copy the robot definition or creating a new PROTO.

## 5.4 Procedural PROTO nodes

The expressive power of PROTO nodes can be significantly improved by extending them using a scripting language. In this way, the PROTO node may contain constants, mathematic expressions, loops, conditional expressions, randomness, and so on.

### 5.4.1 Scripting language

The used scripting language is Lua[1]. Introducing and learning Lua is outside the scope of this document. Please refer to the Lua documentation[2] for complementary information.

### 5.4.2 Template Engine

A template engine is used to evaluate the PROTO according to the fields values of the PROTO, before being loaded in Webots. The template engine used is slt2[3] (under the MIT license).

### 5.4.3 Programming Facts

- Using the template statements is exclusively allowed inside the content scope of the PROTO (cf. example).
- A template statement is encapsulated inside the "%{" and the "}%" tokens and can be written on several lines.
- Adding an "=" just after the opening token ("%{=") allows to evaluate a statement.
- The fields are accessible into a global Lua dictionary named "fields". The dictionary keys matches the PROTO's fields names. The conversion between the VRML types and the Lua types is detailed in table 5.1.
- As shown in table 5.1, the conversion of a VRML node is a Lua dictionary. This dictionary contains the following keys: "node_name" containing the VRML node name, "fields" which is a dictionary containing the Lua representation of the VRML node fields, and "super" which can contains the super PROTO node (the node above in the hierarchy) if existing. This dictionary is equal to `nil` if the VRML node is not defined (`NULL`). For example, in the SimpleStairs example below, the `fields.appearance.node_name` key contains the `'Appearance'` string.
- The VRML comment ("#") prevails over the Lua statements.
- The following Lua modules are availble directly: base, table, io, os, string, math, debug, package.
- The LUA_PATH environment variable can be modified (before running Webots) to include external Lua modules.

[1] `http://www.lua.org`
[2] `http://www.lua.org/docs.html`
[3] `https://github.com/henix/slt2`

- Lua standard output and error streams are redirected on the Webots console (written respectively in regular and in red colors). This allows developers to use the Lua regular functions to write on these streams.

| VRML type | Lua type |
|---|---|
| SFBool | boolean |
| SFInt32 | number |
| SFFloat | number |
| SFString | string |
| SFVec2f | dictionary (keys = ”x” and ”y”) |
| SFVec3f | dictionary (keys = ”x”, ”y” and ”z”) |
| SFRotation | dictionary (keys = ”x”, ”y”, ”z” and ”a”) |
| SFColor | dictionary (keys = ”r”, ”g” and ”b”) |
| SFNode | dictionary (keys = ”node_name”, ”fields”[, ”super”]) |
| MF* | array (indexes = multiple value positions) |

Table 5.1: VRML type to Lua type conversion

### 5.4.4 Example

```
#VRML_SIM V7.3.1 utf8

PROTO SimpleStairs [
  field SFVec3f    translation 0 0 0
  field SFRotation rotation    0 1 0 0
  field SFInt32    nSteps      10
  field SFVec3f    stepSize    0.2 0.2 0.8
  field SFNode     appearance  NULL
]
{
  # template statements can be used from here
  %{
    -- a template statement can be written on several lines
    if fields.nSteps < 1 then
      print('nSteps should be strictly positive')
    end

    -- print the first texture url of the ImageTexture node
    -- inside the Appearance node
    if fields.appearance and fields.appearance.fields.texture then
      -- The following test is true: fields.appearance.fields.texture.
         node_name == "ImageTexture"
```

```
      print (fields.appearance.fields.texture.url[0])
    end
  }%
  Solid {
    translation IS translation
    rotation IS rotation
    children [
      DEF SIMPLE_STAIRS_GROUP Group {
        children [
        %{ for i = 0, (fields.nSteps - 1) do }%
          %{ x = i * fields.stepSize.x }%
          %{ y = i * fields.stepSize.y + fields.stepSize.y / 2 }%
            Transform {
              translation %{=x}% %{=y}% 0
              children [
                Shape {
                  appearance IS appearance
                  geometry Box {
                    size IS stepSize
                  }
                }
              ]
            }
          %{ end }%
        ]
      }
    ]
    boundingObject USE SIMPLE_STAIRS_GROUP
  }
  # template statements can be used up to there
}
```

## 5.5 Using PROTO nodes with the Scene Tree

Several PROTO examples are packaged with Webots. Instances of these PROTO nodes can be added to the simulation with the Scene Tree buttons. Note that currently the Scene Tree allows the instantiation but not the definition of PROTO nodes. PROTO definitions must be created or modified manually in `.proto` files.

### 5.5.1 PROTO Directories

In order to make a PROTO available to Webots, the complete PROTO definition must be placed in a `.proto` file. Each `.proto` file can contain the definition for only one PROTO, and each

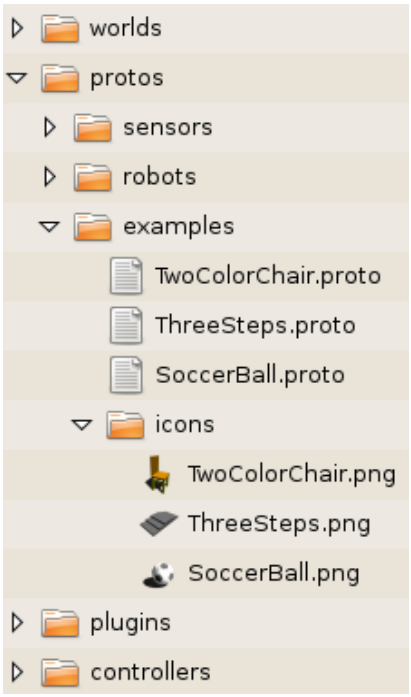


Figure 5.2: PROTO directory in a project directory structure

file must be saved under the name <*PROTOName*>`.proto`, where *PROTOName* is the name of the PROTO as specified after the PROTO keyword (case-sensitive). For example the above `TwoColorChair` PROTO must be saved in a file name `TwoColorChair.proto`.

The `.proto` file should be placed in the `protos` subdirectory of the current project directory. By definition, the current project directory is the parent directory of the `worlds` directory that contains the currently opened `.wbt` file. The figure 5.2 shows where `.proto` files are stored in a project directory.

Note that inside the `protos` directory, the number of subdirectories and their names is free. The user can assign directory names for practical classification reasons; but the names do not influence how PROTO files are searched. The whole subdirectory tree is always searched recursively.

In addition to the current project directory, Webots does also manage a *default* project directory. This directory is structurally similar to the current project directory (see above) but it is located inside Webots distribution. In the default project directory there is also a `protos` subdirectory that provides Webots standard PROTO nodes. These standard PROTO nodes should normally not be modified by the user. Note that `.proto` files will be searched first in the current project directory and then in the default project directory.

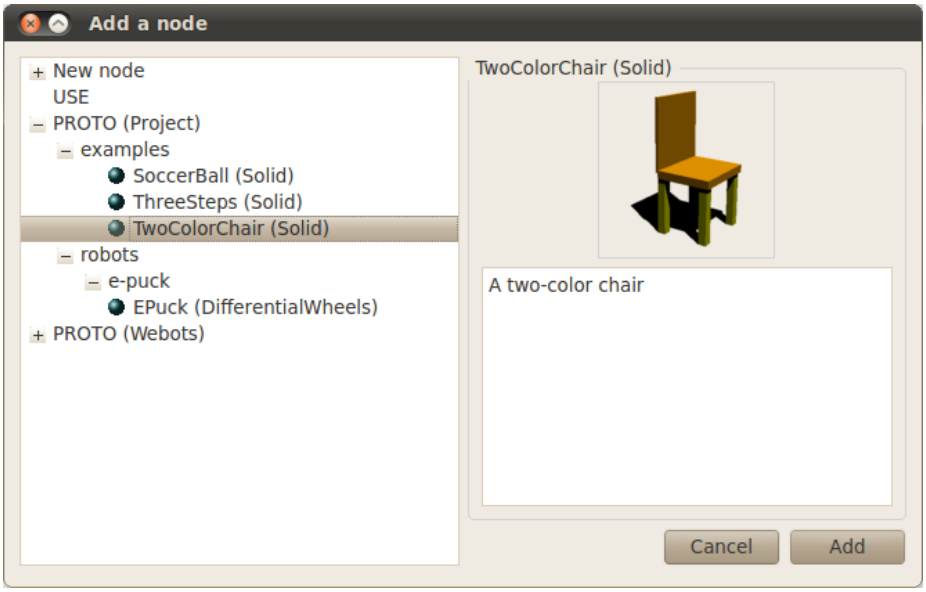


Figure 5.3: Adding an instance of the TwoColorChair PROTO

### 5.5.2 Add a Node Dialog

If a PROTO is saved in a file with proper name and location, it should become visible in the **Add a node** dialog that can be invoked from the **Scene Tree**. In the dialog, the PROTO nodes are organized using the same directory hierarchy found in the project's and Webots's `protos` folders. However this dialog shows a PROTO only if its *base type* is suitable for the chosen insertion point. For example, a PROTO whose base type is `Material` cannot be inserted in a `boundingObject` field. In figure 5.3 you can see how the `TwoColorChair` PROTO appears in the dialog. Note that, the dialog's text pane is automatically filled with any comment placed at the beginning of the `.proto` file.

Icons can be used to better illustrate PROTO nodes. A PROTO icon must be stored in a 128 x 128 pixels `.png` file. The file name must correspond to that of the PROTO plus the `.png` extension and it must be stored in the `icons` subdirectory of the `protos` directory (see figure 5.2). Note that it is possible to create the `.png` files directly with Webots's menu **File** > **Take Screenshot...**. Then the image should be cropped or resized to 128 x 128 pixels using an image editor.

### 5.5.3 Using PROTO Instances

If you hit the **Add** button, the PROTO instance is added to the **Scene Tree**. In the **Scene Tree**, PROTO instances are represented with a different color than built-in nodes (see figure 5.4). PROTO fields can be manipulated exactly like built-in node fields.

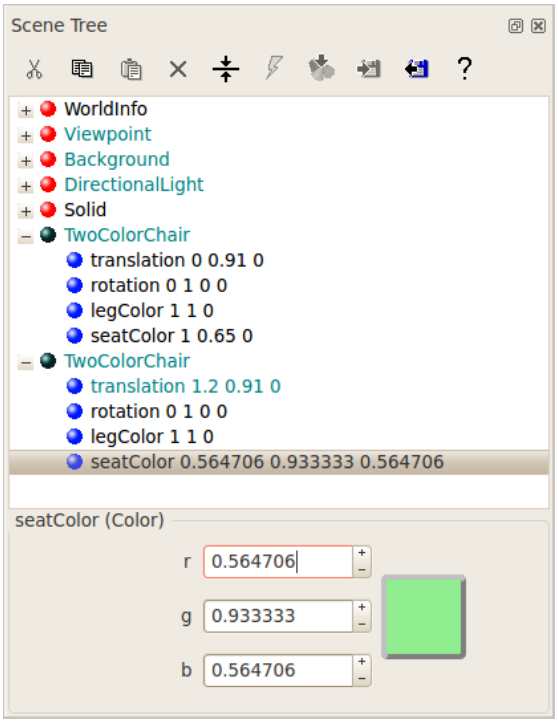


Figure 5.4: Scene Tree with two instances of the TwoColorChair PROTO

## 5.6 PROTO Scoping Rules

PROTO names must be unique: defining a PROTO with the same name as another PROTO or a built-in node type is an error. A `.proto` file can contain only one PROTO definition. A PROTO node can be defined in terms of other PROTO nodes. However, instantiation of a PROTO inside its own definition is not permitted (i.e., recursive PROTO are illegal). An IS statement refers to a field in the interface of the same PROTO, in the same file. Fields declared in the interface can be passed to sub-PROTO nodes using IS statements.

A `.proto` file establishes a DEF/USE name scope separate from the rest of the scene tree and separate from any other PROTO definition. Nodes given a name by a DEF construct inside the PROTO may not be referenced in a USE construct outside of the PROTO's scope. Nodes given a name by a DEF construct outside the PROTO scope may not be referenced in a USE construct inside the PROTO scope.

In case of derived PROTO nodes, it is allowed to declare in the interface a field with the same name as a base PROTO field only if it is not associated with any other field than the homonymous base PROTO field. This means that it is possible to use the derived field in template statements without restrictions, but if it used in a IS statement then the two identifiers before and after the IS keyword have to match. If the derived field has a unique name then no restrictions apply.

## 5.7 PROTO hidden fields

Regular PROTO fields let you change, save and restore, chosen characteristics of your model. In constrast, PROTO encapsulation prevent field values which are not accessible through PROTO fields, but which may change during simulation, from being saved and subsequently restored. Still, this is not true for all field values, since Webots save for you hidden PROTO fields which are bound to change over simulation time. Namely the `translation` and `rotation` fields of `Solid` nodes as well as the `position` fields of `Joint` nodes are saved as hidden PROTO fields in the field scope of every top-level PROTO. In case of `solid merging`, note that hidden `translation` and `rotation` fields are saved only for the `Solid` placed at the top of the solid assembly.

As in the case of non-PROTO objects, initial velocities of physical subparts of a PROTO are saved and can be subsequently restored when reloading your world file. Like the other hidden fields, velocities are saved in the field scope of every top-level PROTO.

Each hidden field appends an index to its name which encodes the location of the `Solid` to which it belongs inside the tree hierarchy rooted at the the PROTO node. This index corresponds is the depth-first pre-order traversal index of the `Solid` in this tree. If a hidden field corresponds to the `position` of `Joint`, an additional index is appended to its name, namely the index of the `Joint` in the list of `Joint` nodes originating from the `Solid` sorted by means of pre-order traversal. As an example, we display below an excerpt of `projects/robots/pioneer/pioneer3at/worlds/pioneer3at.wbt` when saved after one simulation step.

```
DEF PIONEER_3AT Pioneer3at {
 hidden position_0_0 -2.88177e-07
 hidden position_0_1 -4.63679e-07
 hidden position_0_2 -3.16282e-07
 hidden position_0_3 -4.91785e-07
 hidden linearVelocity_0 -0.00425142 -0.0284347 0.0036279
 hidden angularVelocity_0 0.0198558 -9.38102e-07 0.0232653
 hidden translation_2 -0.197 4.04034e-06 0.1331
 hidden rotation_2 -0.013043 0.00500952 0.999902 -2.88177e-07
 hidden linearVelocity_2 -0.00255659 -0.0214607 0.00218164
 hidden angularVelocity_2 0.0198598 -9.84272e-07 0.0232733
 hidden translation_3 0.197 7.85063e-06 0.1331
 hidden rotation_3 -0.00949932 0.00367208 0.999948 -4.63679e-07
 hidden linearVelocity_3 -0.00255694 -0.0330774 0.00218161
 hidden angularVelocity_3 0.0198623 -9.92987e-07 0.0232782
 hidden translation_4 -0.197 4.64107e-06 -0.1331
 hidden rotation_4 -0.011884 0.0045545 0.999919 -3.16282e-07
 hidden linearVelocity_4 -0.00255674 -0.0232922 0.00218172
 hidden angularVelocity_4 0.0198602 -9.84272e-07 0.0232741
 hidden translation_5 0.197 8.45135e-06 -0.1331
 hidden rotation_5 -0.00895643 0.00345587 0.999954 -4.91785e-07
```

```
hidden linearVelocity_5 -0.0025571 -0.0349089 0.00218169
hidden angularVelocity_5 0.0198627 -9.92987e-07 0.023279
translation 2.61431 0.109092 18.5514
rotation -0.000449526 1 0.000227141 -2.66435
controller "obstacle_avoidance_with_lidar"
extensionSlot [
  SickLms291 {
    translation 0 0.24 -0.136
    pixelSize 0
  }
]
```

The names of the first six hidden fields all contain 0 as primary index, which is the index of the `Pioneer3at` PROTO itself. The additional secondary indices for the four hidden `position` fields correspond to the four `HingeJoint` nodes used for the wheels and numbered by means of pre-order traversal. There is no hidden field associated the `Solid` node with index 1, namely the `SickLms291` PROTO, since its relative position and orientation are kept fixed during simulation. The indices ranging from 2 to 5 correspond to the four `Solid` wheels of the `Pioneer3at`.

# Chapter 6

# Physics Plugin

## 6.1 Introduction

This chapter describes Webots capability to add a physics plugin to a simulation. A physics plugin is a user-implemented shared library which is loaded by Webots at run-time, and which gives access to the low-level API of the ODE[1] physics engine. A physics plugin can be used, for example, to gather information about the simulated bodies (position, orientation, linear or angular velocity, etc.), to add forces and torques, to add extra joints, e.g., "ball & socket" or "universal joints" to a simulation. For example with a physics plugin it is possible to design an aerodynamics model for a flying robot, a hydrodynamics model for a swimming robot, etc. Moreover, with a physics plugin you can implement your own collision detection system and define non-uniform friction parameters on some surfaces. Note that physics plugins can be programmed only in C or C++. Webots PRO is necessary to program physics plugins.

## 6.2 Plugin Setup

You can add a new plugin, or edit the existing plugin, by using the menu **Tools** > **Edit Physics Plugin**. After a physics plugin was created it must be associated with the current `.wbt` file. This can be done in the Scene Tree: the `WorldInfo` node has a field called `physics` which indicates the name of the physics plugin associated with the current world. Select the `WorldInfo.physics` field, then hit the **Select...** button. A dialog pops-up and lets you choose one of the plugins available in the current project. Choose a plugin in the dialog and then save the `.wbt` file.

Note that the `WorldInfo.physics` string specifies the name of the plugin source and binary files without extension. The extension will be added by Webots depending on the platform: it will be `.so` (Linux), `.dll` (Windows) or `.dylib` (Mac OS X) for the binary file. For example, this `WorldInfo` node:

[1] `http://www.ode.org`

```
WorldInfo {
  ...
  physics "my_physics"
  ...
}
```

specifies that the plugin binary file is expected to be at the location `my_project/plugins/physics/my_physics/my_physics[.dll|.dylib|.so]` (actual extension depending on the platform) and that the plugin source file should be located in `my_project/plugins/physics/my_physics/my_physics[.c|.cpp]`. If Webots does not find the file there, it will also look in the `WEBOTS_HOME/resources/projects/plugins` and `WEBOTS_MODULES_PATH/projects/default/plugins` directories.

## 6.3 Callback Functions

The plugin code must contain user-implemented functions that will be called by Webots during the simulation. These user-implemented functions and their interfaces are described in this section. The implementation of the `webots_physics_step()` and `webots_physics_cleanup()` functions is mandatory. The implementation of the other callback functions is optional.

Since Webots 7.2.0, the ODE physics library used in Webots is multi-threaded. This allows Webots to run some physics simulation much faster than before on multi-core CPUs. However, it also implies that programming a physics plug-in is slightly more complicated as the `webots_physics_collide()` callback function may be called from different threads. Hence, it should be re-entrant and every call to an ODE API function modifying the current world (contacts, bodies, geoms) should be mutex protected within this callback function. We recommend using POSIX mutexes as exemplified here:

```
static pthread_mutex_t mutex;

void webots_physics_init() {
  pthread_mutex_init(&mutex, NULL);
  ...
}

int webots_physics_collide(dGeomID g1, dGeomID g2) {
...
    dJointGroupID contact_joint_group = dWebotsGetContactJointGroup();
    pthread_mutex_lock(&mutex);
    dJointAttach(dJointCreateContact(world,
                                     contact_joint_group,
                                     &contact[i]),
                                     robot_body,
```

```
                                        NULL);
    pthread_mutex_unlock(&mutex);
...
}

void webots_physics_cleanup() {
  ...
  pthread_mutex_destroy(&mutex);
}
```

### 6.3.1 void webots_physics_init(dWorldID, dSpaceID, dJointGroupID)

This function is called upon initialization of the world. Its arguments are obsolete and should not be used. This function is a good place to call the `dWebotsGetBodyFromDEF()` and `dWebotsGetGeomFromDEF()` functions (see below for details) to get pointers to the objects for which you want to control the physics. Before calling this function, Webots sets the current directory to where the plugin's `.dll`, `.so` or `.dylib` was found. This is useful for reading config files or writing log files in this directory.

The obsolete arguments can be retrieved as follows:

```
void webots_physics_init(dWorldID, dSpaceID, dJointGroupID) {
  // get body of the robot part
  dBodyID body = dWebotsGetBodyFromDEF("MY_ROBOT_PART");
  dGeomID geom = dWebotsGetGeomFromDEF("MY_ROBOT_PART");

  // get the matching world
  dWorldID world = dBodyGetWorld(body);

  // get the matching space
  dSpaceID space = dGeomGetSpace(geom);
}
```

This function is also the preferred place to initialize/reinitialize the random number generator (via `srand()`). Reinitializing the generator with a constant seed allows Webots to run reproducible (deterministic) simulations. If you don't need deterministic behavior you should initialize srand() with the current time: `srand(time(NULL))`. Webots itself does not invoke `srand()`; however, it uses `rand()`, for example to add noise to sensor measurements. In order to have reproducible simulations, it is also required that all controllers run in *synchronous* mode. That means that the `synchronization` field of every `Robot`, `DifferentialWheels` or `Supervisor` must be set to TRUE. Finally, note that ODE uses its own random number generator that you might also want to reinitialize separately via the `dRandSetSeed()` function.

### 6.3.2 int webots_physics_collide(dGeomID, dGeomID)

This function is called whenever a collision occurs between two geoms. It may be called several times (or not at all) during a single simulation step, depending on the number of collisions. Generally, you should test whether the two colliding geoms passed as arguments correspond to objects for which you want to control the collision. If you don't wish to handle a particular collision you should return 0 to inform Webots that the default collision handling code must be used.

Otherwise you should use ODE's `dCollide()` function to find the contact points between the colliding objects and then you can create contact joints using ODE's `dJointCreateContact()` function. Normally the contact joints should be created within the contact joint group given by the `dWebotsGetContactJointGroup()` function. Note that this contact joint group is automatically emptied after each simulation step, see here. Then the contact joints should be attached to the corresponding bodies in order to prevent them from inter-penetrating. Finally, the `webots_physics_collide()` function should return either 1 or 2 to inform Webots that this collision was handled. If the value 2 is returned, Webots will moreover notify graphically that a collision occurred by changing the color of the corresponding boundingObject Geometry in the 3D view.

Since Webots 7.2.0, a multi-threaded version of ODE is used. Therefore, this function may be called from different threads. You should ensure it is re-entrant and that every ODE function call modifying the ODE world is protected by mutexes as explained earlier.

### 6.3.3 void webots_physics_step()

This function is called before every physics simulation step (call to the ODE `dWorldStep()` function). For example it can contain code to read the position and orientation of bodies or add forces and torques to bodies.

### 6.3.4 void webots_physics_step_end()

This function is called right after every physics simulation step (call to the ODE `dWorldStep()` function). It can be used to read values out of `dJointFeedback` structures. ODE's `dJointFeedback` structures are used to know how much torque and force is added by a specific joint to the joined bodies (see ODE User Guide for more information). For example, if the plugin has registered `dJointFeedback` structures (using ODE's function `dJointSetFeedback()`), then the structures will be filled during `dWorldStep()` and the result can be read straight afterwards in `webots_physics_step_end()`.

### 6.3.5 void webots_physics_cleanup()

This function is the counterpart to the `webots_physics_init()` function. It is called once, when the world is destroyed, and can be used to perform cleanup operations, such as closing files and freeing the objects that have been created in the plugin.

### 6.3.6 void webots_physics_draw(int pass, const char *view)

This function is used to add user-specified OpenGL graphics to the 3D view and/or to the cameras. For example, this can be used to draw robots trajectories, force vectors, etc. The function should normally contain OpenGL function calls. This function is called 2 times (2 passes): one right before and one right after the regular OpenGL rendering. The first pass may be useful for drawing solid objects visible through transparent or semi-transparent objects in the world, but generally only the second is used. The `pass` argument allows to distinguish these 2 passes (`pass` = 0 for the pass before the OpenGL rendering, `pass` = 1 for the pass after the OpenGL rendering) The `view` argument allows to determine if the function is called when rendering the 3D view (`view` == NULL) or when rendering a robot camera (`view` == Robot::name). Here is an implementation example:

```
void webots_physics_draw(int pass, const char *view) {
  if (pass == 1 && view == NULL) {
    /* This code is reached only during the second pass of the 3D view
       */

    /* modify OpenGL context */
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glLineWidth(2.0);

    /* draw 1 meter yellow line */
    glBegin(GL_LINES);
    glColor3f(1, 1, 0);
    glVertex3f(0, 0, 0);
    glVertex3f(0, 1, 0);
    glEnd();
  }
```

The above example will draw a meter high yellow line in the center of the world. Note that Webots loads the *world* (global) coordinates matrix right before calling this function. Therefore the arguments passed to `glVertex()` are expected to be specified in *world* coordinates. Note that the default OpenGL states should be restored before leaving this function otherwise the rendering in Webots 3D view may be altered.

## 6.4 Utility Functions

This section describes utility functions that are available to the physics plugin. They are not callback functions, but functions that you can call from your callback functions.

### 6.4.1 dWebotsGetBodyFromDEF()

This function looks for a `Solid` node with the specified name and returns the corresponding dBodyID. The returned dBodyID is an ODE object that represent a rigid body with properties such as mass, velocity, inertia, etc. The dBodyID object can then be used with all the available ODE `dBody*()` functions (see ODE documentation). For example it is possible to add a force to the body with `dBodyAddForce()`, etc. The prototype of this function is:

```
dBodyID dWebotsGetBodyFromDEF(const char *DEF);
```

where DEF is the DEF name of the requested `Solid` node.

It is possible to use dots (.) as scoping operator in the DEF parameter. Dots can be used when looking for a specific node path in the node hierarchy. For example:

```
dBodyID head_pitch_body = dWebotsGetBodyFromDEF("BLUE_PLAYER_1.HeadYaw
    .HeadPitch");
```

means that we are searching for a `Solid` node named "HeadPitch" inside a node named "HeadYaw", inside a node named "BLUE_PLAYER_1". Note that each dot (.) can be substituted by any number of named or unnamed nodes, so in other words it is not necessary to fully specify the path.

This function returns NULL if there is no `Solid` (or derived) node with the specified DEF name. It will also return NULL if the `physics` field of the `Solid` node is undefined (NULL) or if the `Solid` have been *merged* with an ancestor. Solid merging happens between rigidly linked solids with non NULL `physics` fields, see `Physics`'s "Implicit solid merging and joints" for more details. This function searches the Scene Tree recursively, therefore it is recommended to store the result rather than calling it at each step. It is highly recommended to test for NULL returned values, because passing a NULL dBodyID to an ODE function is illegal and will crash the plugin and Webots.

### 6.4.2 dWebotsGetGeomFromDEF()

This function looks for a `Solid` node with the specified name and returns the corresponding dGeomID. A dGeomID is an ODE object that represents a geometrical shape such as a sphere, a cylinder, a box, etc., or a coordinate system transformation. The dGeomID returned by Webots corresponds to the boundingObject of the `Solid`. The dGeomID object can then be used with all the available ODE `dGeom*()` functions (see ODE documentation). The prototype of this function is:

```
dGeomID dWebotsGetGeomFromDEF(const char *DEF);
```

where DEF is the DEF name of the requested `Solid` node.

It is possible to use dots (.) as scoping operator in the DEF parameter, see above. This function returns NULL if there is no `Solid` (or derived) node with the specified DEF name. It will also return NULL if the `boundingObject` field of the `Solid` node is undefined (NULL). This function searches the Scene Tree recursively therefore it is recommended to store the result rather than calling it at each step. It is highly recommended to test for NULL returned values, because passing a NULL dGeomID to an ODE function is illegal and will crash the plugin and Webots.

Using the returned dGeomID, it is also possible to obtain the corresponding dBodyID object using ODE's `dGeomGetBody()` function. This is an alternative to calling the `dWebotsGetGeomFromDEF()` function described above.

Note that this function returns only the top level dGeomID of the boundingObject, but the boundingObject can be made of a whole hierarchy of dGeomIDs. Therefore it is risky to make assumptions about the type of the returned dGeomID. It is safer to use ODE functions to query the actual type. For example this function may return a "transform geom" (dGeomTransformClass) or a "space geom" (dSimpleSpaceClass) if this is required to represent the structure of the boundingObject.

### 6.4.3 dWebotsGetContactJointGroup()

This function allows you to retrieve the contact joint group where to create the contacts. It is typically called inside the `webots_physics_collide()` function. Remark that this group may change during the time and should be retrieved at each `webots_physics_collide()` call.

### 6.4.4 dGeomSetDynamicFlag(dGeomID geom)

This function switches on the dynamic flag of a given ODE geometry (given by the `geom` argument).

By default the ODE geometries linked with an ODE body are dynamic, meaning that they can pass from one to another cluster in the case of the multi-threaded version of ODE. On the other hand, an ODE geometry without any ODE body is static, meaning it is available in every cluster.

There are some cases where one wants to have a dynamic ODE geometry even if it is not linked with an ODE body. This is the puropose of this function. Typically the ODE rays (which don't have bodies) are more efficient if defined as dynamic geometries.

### 6.4.5 dWebotsSend() and dWebotsReceive()

It is often useful to communicate information between your physics plugin and your robot (or Supervisor) controllers. This is especially useful if your physics plugin implements some sensors (like accelerometers, force feedback sensors, etc.) and needs to send the sensor measurement to the robot controller. It is also useful if your physics plugin implements some actuators (like an Akermann drive model), and needs to receive motor commands from a robot controller.

The physics plugin API provides the `dWebotsSend()` function to send messages to robot controllers and the `dWebotsReceive()` function to receive messages from robot controllers. In order to receive messages from the physics plugin, a robot has to contain a `Receiver` node set to an appropriate channel (see Reference Manual) and with a `baudRate` set to -1 (for infinite communication speed). Messages are sent from the physics plugin using the `dWebotsSend()` function, and received through the receiver API as if they were sent by an `Emitter` node with an infinite range and baud rate. Similarly, in order to send messages to the physics plugin, a robot has to contain an `Emitter` node set to `channel` 0 (as the physics plugin only receives data sent on this channel). The `range` and `baudRate` fields of the `Emitter` node should be set to -1 (infinite). Messages are sent to the physics plugin using the standard `Emitter` API functions. They are received by the physics plugin through the `dWebotsReceive()` function.

```
void dWebotsSend(int channel,const void *buffer,int size);
void *dWebotsReceive(int *size);
```

The `dWebotsSend()` function sends `size` bytes of data contained in `buffer` over the specified communication `channel`.

The `dWebotsReceive()` function receives any data sent on channel 0. If no data was sent, it returns NULL; otherwise it returns a pointer to a buffer containing the received data. If `size` is non-NULL, it is set to the number of bytes of data available in the returned buffer. If multiple messages are sent to the physics plugin at the same time step, then they will be concatenated.


*Pay attention when managing multiple concatenated string messages, because every message will terminate with the null character* `\0` *preventing the correct copy and display of the returned data. The following example shows how to split concatenated string messages:*

```
1  int dataSize;
2  char *data = (char *)dWebotsReceive(&dataSize);
3  if (dataSize > 0) {
4    char *msg = new char[dataSize];
5    int count = 1, i = 0, j = 0;
6    for ( ; i < dataSize; ++i) {
7      char c = data[i];
8      if (c == '\0') {
9        msg[j] = c;
10       // process message
11       dWebotsConsolePrintf("Received message %d:
             %s\n", count, msg);
12       // reset for next string
13       ++count;
14       j = 0;
15     } else {
16       msg[j] = c;
17       ++j;
18     }
19   }
20 }
```

### 6.4.6 dWebotsGetTime()

This function returns the current simulation time in milliseconds [ms] as a double precision floating point value. This corresponds to the time displayed in the bottom right corner of the main Webots window.

```
double dWebotsGetTime(void);
```

### 6.4.7 dWebotsConsolePrintf()

This function prints a line of formatted text to the Webots console. The format argument is the same as the standard C `printf()` function, i.e., the format string may contain format characters defining conversion specifiers, and optional extra arguments should match these conversion specifiers. A prefix and a '\n' (new line) character will automatically be added to each line. A '\f' (form feed) character can optionally be used for clearing up the console.

```
void dWebotsConsolePrintf(const char *format, ...);
```

## 6.5 Structure of ODE objects

This table shows how common `.wbt` constructs are mapped to ODE objects. This information shall be useful for implementing physics plugins.

| Webots construct | ODE construct |
|---|---|
| Solid { physics Physics {...} } | dBodyID |
| Solid { boundingObject ... } | dGeomID |
| Solid { boundingObject Box {...} } | dGeomID (dBoxClass) |
| Solid { boundingObject Sphere {...} } | dGeomID (dSphereClass) |
| Solid { boundingObject Capsule {...} } | dGeomID (dGeomTransformClass + dCapsuleClass) |
| Solid { boundingObject Cylinder {...} } | dGeomID (dGeomTransformClass + dCylinderClass) |
| Solid { boundingObject Plane {...} } | dGeomID (dPlaneClass) |
| Solid { boundingObject IndexedFaceSet {...} } | dGeomID (dTriMeshClass) |
| Solid { boundingObject ElevationGrid {...} } | dGeomID (dHeightfieldClass) |
| Solid { boundingObject Transform {...} } | dGeomID (dGeomTransformClass) |
| Solid { boundingObject Group {...} } | dSpaceID (dSimpleSpaceClass) |
| BallJoint { } | dJointID (dJointTypeBall) |
| HingeJoint { } | dJointID (dJointTypeHinge) |
| Hinge2Joint { } | dJointID (dJointTypeHinge2) |
| SliderJoint { } | dJointID (dJointTypeSlider) |

Table 6.1: Mapping between common Webots constructs and ODE objects.


*Although a physics plugin grants you access to the dGeomIDs created and managed by Webots, you should never attempt to set a user-defined data pointer by means of dGeomSetData() for these dGeomIDs as Webots stores its own data pointer in them. Using dGeomSetData() on a dGeomID defined by Webots will almost surely result into a Webots crash.*

## 6.6 Compiling the Physics Plugin

When a plugin is created using the menu **Wizard** > **New Physics Plugin**, Webots will automatically add a suitable `.c` or `.cpp` source file and a Makefile to the plugin's directory. Your plugin can be compiled with Webots text editor or manually by using `gcc` and `make` commands in a terminal. On Windows, you can also use Visual C++ to compile the plugin. In this case, please note that

the plugin should be dynamically linked to the ODE library. The Webots `lib` directory contains the gcc (`libode.a`) and Visual C++ (`ode.lib`) import libraries. Under Linux, you don't need to link the shared library with anything.

## 6.7 Examples

Webots comes with several examples of physics plugin. When opening an example, the code of the physics plugin should appear in Webots text editor. If it does not appear automatically, then you can always use the menu: **Tools** > **Edit Physics Plugin**.

A simple example is the `WEBOTS_HOME/projects/samples/howto/worlds/physics.wbt` world. In this example, the plugin is used to add forces to make the robot fly, to communicate with the Webots model, to detect objects using a `Ray` object, to display this object using OpenGL and to define a frictionless collision between the robot and the floor.

The `WEBOTS_HOME/projects/samples/howto/worlds/contact_points.wbt` example shows how to detect collision of an arbitrary object with the floor, draw the collision contact points in the 3D window, set up contact joints to define the collison behavior, and determines the forces and torques involved in the collision. This example can be helpful if you need a detailed feedback about the contact points and forces involved in the locomotion of a legged robot.

The `WEBOTS_HOME/projects/samples/demos/worlds/blimp_lis.wbt` shows how to suppress gravity, and apply a thrust force (propeller) for a blimp model.

The `WEBOTS_HOME/projects/samples/demos/worlds/salamander.wbt` shows how to simulate Archimedes'buoyant force and hydrodynamic drag forces.

## 6.8 ODE improvments

In order to extend ODE possibilities and correct some bugs, the version of ODE bundled with Webots was improved. New functions were added and some existing functions were modified.

### 6.8.1 Hinge joint

It is possible to set and get the suspension axis thanks to the following two functions:

```
void dJointSetHingeSuspensionAxis (dJointID, dReal x, dReal y, dReal z
   );
void dJointGetHingeSuspensionAxis (dJointID, dVector3 result);
```

Furthermore, the `dJointSetHingeParam()` and `dJointGetHingeParam()` functions support `dParamSuspensionERP` and `dParamSuspensionCFM` parameters.

### 6.8.2 Hinge 2 joint

By default in ODE, the suspension is along one of the axes of the joint, in the ODE version of Webots, the suspension has been improved in order to use any arbitrary axis. It is possible to set and get this axis thanks to the following two functions:

```
void dJointSetHinge2SuspensionAxis (dJointID, dReal x, dReal y, dReal
   z);
void dJointGetHinge2SuspensionAxis (dJointID, dVector3 result);
```

## 6.9 Troubleshooting

Unlike the controller code, the physics plugin code is executed in the same process and memory space as the Webots application. Therefore, a segmentation fault in the physics plugin code will cause the termination of the Webots application. Webots termination is often misinterpreted by users who believe that Webots is unstable, while the error is actually in the user's plugin code. For that reason, it is important to precisely locate the crash before reporting a bug to Cyberbotics Ltd.

The following are some debugging hints that should help you find the exact location of a crash using `gdb` (the GNU Debugger). The first step is to recompile the physics plugin with the *-g* flag, in order to add debugging information to the compiled plugin. This can be achieved by adding this line to the plugin's `Makefile`:

```
CFLAGS=-g
```

Then you must rebuild the plugin using Webots Text Editor or using these commands in a terminal:

```
$ make clean
$ make
```

Make sure that the *-g* flag appears in the compilation line. Once you have rebuilt the plugin, you can quit Webots, and restart it using `gdb` in a terminal, like this:

```
$ cd /usr/local/webots
$ export LD_LIBRARY_PATH=/usr/local/webots/lib:$LD_LIBRARY_PATH
$ gdb ./webots-bin
(gdb) run
```

Note that the above path corresponds to a default Webots installation on Linux: the actual path might be different depending on your specific system or installation. The LD_LIBRARY_PATH environment variable indicates where to find the shared libraries that will be required by Webots.

When Webots window appears, run the simulation until it crashes, or make it crash by some manipulations if necessary. If the plugin crahes due to a segmentation fault, `gdb` should print an error message similar to this:

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1208154400 (LWP 30524)]
0x001f5c7e in webots_physics_init (w=0xa6f8060, s=0xa6f80e0, j=0
   xa6f5c00)
at my_physics.c:50
50            float pos = position[0] + position[1] + position[2];
...
```

This indicates precisely the file name and line number where the problem occurred. If the indicated file name corresponds to one of the plugin source files, then the error is located in the plugin code. You can examine the call stack more precisely by using the `where` or the `bt` command of `gdb`. For example:

```
(gdb) where
#0  0x001f5c7e in webots_physics_init (w=0xa6f8060, s=0xa6f80e0, j=0
   xa6f5c00)
at my_physics.c:50
#1  0x081a96b3 in A_PhysicsPlugin::init ()
#2  0x081e304b in A_World::preprocess ()
#3  0x081db3a6 in A_View::render ()
#4  0x081db3f3 in A_View::onPaint ()
#5  0x084de679 in wxEvtHandler::ProcessEventIfMatches ()
#6  0x084de8be in wxEventHashTable::HandleEvent ()
#7  0x084def90 in wxEvtHandler::ProcessEvent ()
#8  0x084ea393 in wxGLContext::SetCurrent ()
...
```

In this example you see that the error is located in the plugin's `webots_physics_init()` function. If the error is reported in an unknown function (and if the line number and file name are not displayed), then the crash may have occurred in Webots, or possibly in a library used by your plugin.

## 6.10 Execution Scheme

The following diagram illustrates the sequence of execution of the plugin callback functions. In addition, the principal interactions of Webots with the ODE functions are indicated.

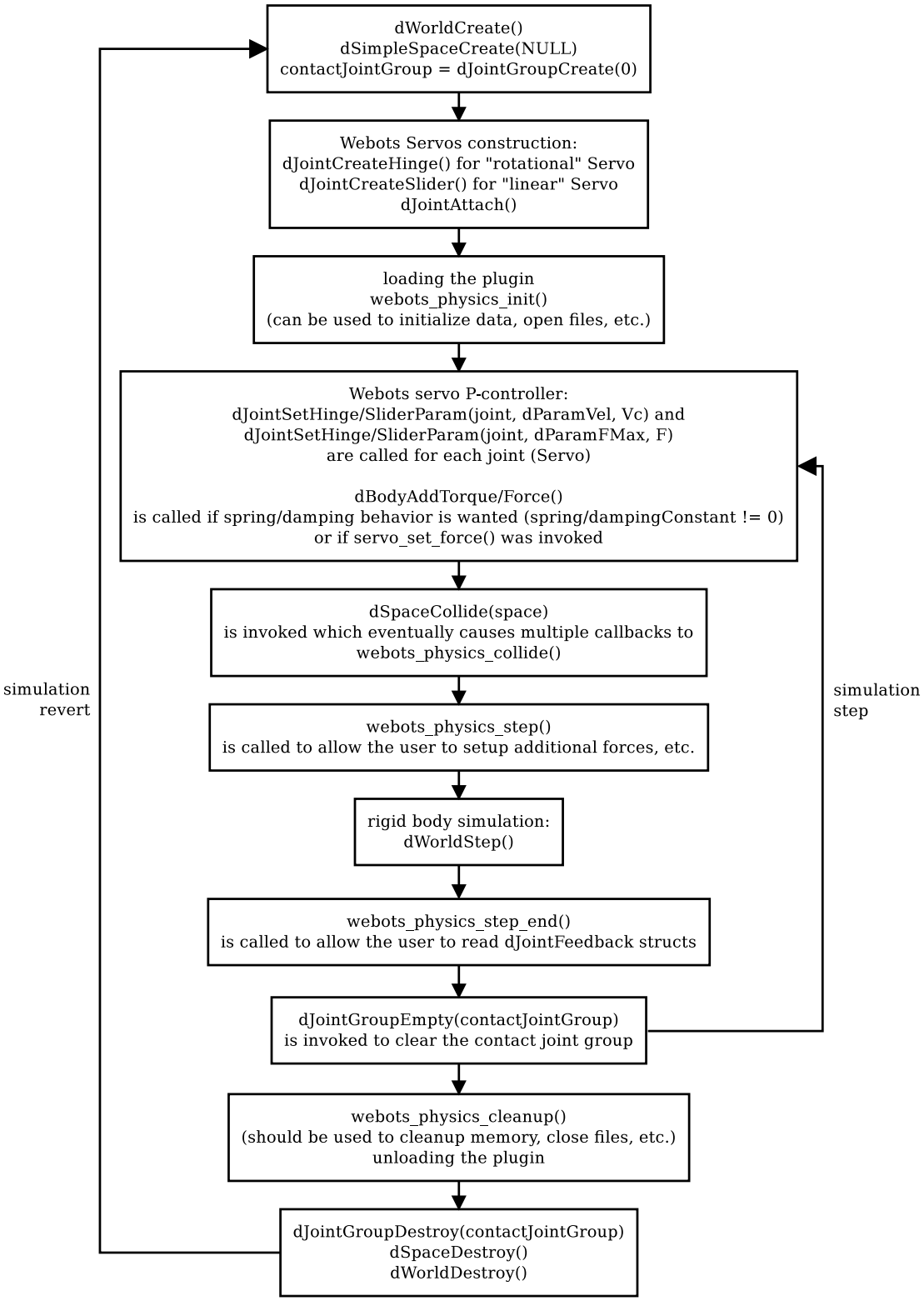


Figure 6.1: Physics Plugin Execution Scheme

# Chapter 7

# Fast2D Plugin

## 7.1 Introduction

In addition to the usual 3D and physics-based simulation modes, Webots offers a 2D simulation mode called Fast2D. The Fast2D mode enables very fast simulation for worlds that require only two-dimensional (2D) computations. Many simulations are carried out on a 2D area using wheeled robots such as Alice™or Khepera™; in such simulations the height and elevation of the objects are generally irrelevant, therefore the overhead of 3D computations can be avoided by using Fast2D. The Fast2D plugin is designed for situations where the speed of a simulation is more important than its realism, as in evolutionary robotics or swarm intelligence, for example.

## 7.2 Plugin Architecture

### 7.2.1 Overview

The Webots' Fast2D mode is built on a *plugin* architecture. The Fast2D plugin is a dynamically linked library that provides the functions necessary for the 2D simulation. These functions are responsible for the simulation of:

- Differential wheels robots (kinematics, friction model, collision detection)
- Obstacles (collision detection)
- Distance sensors (distance measurement)

The plugin architecture makes it possible to use different plugins for different worlds (`.wbt` files) and allows Webots users to design their own custom plugins.

### 7.2.2 Dynamically Linked Libraries

The Fast2D plugin is loaded by Webots when the user loads a world (`.wbt` file) that requires Fast2D simulation mode. The `WorldInfo` node of the world has a field called `fast2d` which specifies the name of the dynamically linked library to be used as plugin for this world. For example:

```
WorldInfo {
  fast2d "enki"
}
```

An empty `fast2d` field means that no plugin is required and that the simulation must be carried out in 3D mode. When the `fast2d` field in not empty, Webots looks for the corresponding plugin in the `plugins/fast2d` directory located at the same directory level as the `worlds` and `controllers` directories. More precisely, Webots looks for the plugin file $(pluginname)/$(pluginname).$(extension) at these two locations:

1. `$(projectdir)/plugins/fast2d/`
2. `$(webotdir)/resources/projects/plugins/fast2d/`

Where $(projectdir) represents a Webots project directory, $(pluginname) is the plugin name as specified in the `fast2d` field of the `WorldInfo` node, $(extension) is an operating system dependent filename extension such as `so` (Linux) or `dll` (Windows) and $(webotdir) is the path specified by the WEBOTS_HOME environment variable. If WEBOTS_HOME is undefined then $(webotdir) is the path from which the Webots executable was started. If the required plugin is not found, Webots attempts to run the simulation using the built-in 3D simulator. According to the ”enki” example above, and assuming that the current project directory $(projectdir) is `/home/user/webots` and that `WEBOTS_HOME=/usr/local/webots`, then the Linux version of Webots looks for the plugin in:

1. `/home/user/webots/plugins/fast2d/enki/enki.so`
2. `/usr/local/webots/resources/projects/plugins/fast2d/enki/enki.so`

Since the plugin name is referred to by the `WorldInfo` node of a world (`.wbt` file), it is possible to have a different plugin for each world.

### 7.2.3 Enki Plugin

The Linux and Windows distributions of Webots come with a pre-installed Fast2D plugin called the *Enki plugin*. The Enki plugin is based on the Enki simulator, which is a fast open source 2D robot simulator developed at the Laboratory of Intelligent Systems, at the EPFL in Lausanne,

Switzerland, by Stephane Magnenat, Markus Waibel and Antoine Beyeler. You can find more information about Enki at the Enki website[1].

## 7.3 How to Design a Fast2D Simulation

Webots' scene tree allows a large choice of 3D objects to be assembled in complex 3D worlds. Because Fast2D is designed to run simulations exclusively in 2D, the 3D worlds must be simplified before the Fast2D simulation can handle them properly.

### 7.3.1 3D to 2D

The most important simplification is to remove one dimension from the 3D worlds; this is carried out by Webots automatically. In 3D mode, the *xz*-plane is traditionally used to represent the ground, while the positive *y*-axis represents the "up" direction. In Fast2D mode Webots projects 3D objects onto the *xz*-plane simply by removing the *y*-dimension. Therefore, Fast2D mode ignores the *y*-axis and carries out simulations in the xz-plane exclusively. However, the naming convention in Fast2D changes, using the *y*-axis to represent the 3D *z*-axis. See table 7.1.

| 3D | -> | Fast2D |
|---|---|---|
| $x$ | -> | $x$ |
| $y$ | -> | none |
| $z$ | -> | $y$ |
| $\alpha$ (rotation angle) | -> | -$\alpha$ (orientation angle) |

Table 7.1: Conversion from 3D to Fast2D coordinate systems.

In short, the 3D *y*-axis does not matter with Fast2D. The objects' heights and elevations are ignored, and the worlds intended for Fast2D simulation must be designed with this in mind. Furthermore, Fast2D worlds must be designed such that the *y*-axes of all its Solid and DifferentialWheels nodes are aligned with the world's *y*-axis. In other words, the `rotation` field of `Solid` and `DifferentialWheels` nodes must be:

```
Solid {
  rotation 0 1 0 <alpha>
  ...
```

This leaves the rotation angle `alpha` as the only parameter that you can tune. If a Fast2D world does not fulfill this requirement, the result of the simulation is undefined. Note also that Fast2D rotation angles are equal to the negative of the 3D rotation angles. See table 7.1.

[1] `http://home.gna.org/enki`

### 7.3.2 Scene Tree Simplification

In Fast2D mode, Webots takes only the top level objects of the scene tree into account. Each `Solid` or `DifferentialWheels` node defined at the root level will be used in the Fast2D simulation, but other `Solid` or `DifferentialWheels` nodes will be ignored. It is possible to use a `Solid` as a child of another `Solid` or as a child of a `DifferentialWheels` node, but be aware that in this case, although the child `Solid` does appear graphically, it is not taken into account by the simulation.

### 7.3.3 Bounding Objects

In Fast2D, just as in 3D simulation, only bounding objects are used in collision detection. Although Webots allows a full choice of bounding objects, in Fast2D mode, it is only possible to use a single Cylinder or a single Box as a bounding object. Furthermore, Fast2D mode requires that the coordinate systems of an object and of its corresponding bounding object must be the same. In other words, any Transform of the bounding object will be ignored in Fast2D mode.

## 7.4 Developing Your Own Fast2D Plugin

The Enki-based Fast2D plugin that comes with Webots is highly optimized, and should be suitable for most 2D simulations. However, in some cases you might want to use your own implementation of kinematics and collision detection. In this case you will have to develop your own Fast2D plugin; this section explains how to proceed.

### 7.4.1 Header File

The data types and interfaces required to compile your own Fast2D plugin are defined in the `fast2d.h` header file. This file is located in Webots installation directory, in the `include/plugins/fast2d` subdirectory. It can be included like this:

```
#include <plugins/fast2d/fast2d.h>
...
```

The `fast2d.h` file contains C types and function declarations; it can be compiled with either a C or C++ compiler.

### 7.4.2 Fast2D Plugin Types

Four basic types are defined in `fast2d.h`: `ObjectRef`, `SolidRef`, `RobotRef` and `SensorRef`. In order to enforce a minimal amount of type-checking and type-awareness, these

basic types are declared as non-interchangeable pointer types. They are only dummy types, not designed to be used as-is, but rather to be placeholders for the real data types that the plugin programmer is responsible for implementing. We suggest that you declare your own four data types as C structs or C++ classes. Then in your implementation of the Fast2D functions, you should cast the addresses of your data instances to the Fast2D types, as in the example below, where `MyRobotClass` and `MySensorClass` are user-defined types:

```
RobotRef webots_fast2d_create_robot() {
  return (RobotRef) new MyRobotClass();
}

void webots_fast2d_robot_add_sensor(RobotRef robotRef,
          SensorRef sensorRef, double x, double y, double angle) {

  MyRobotClass *robot = (MyRobotClass*) robotRef;
  MySensorClass *sensor = (MySensorClass*) sensorRef;
  robot->addSensor(sensor, x, y, angle);
  ...
}
```

In this example, Webots calls `webots_fast2d_create_robot()` when it requires a new robot object; this function instantiates the object and casts it to a Fast2D type before returning it. Webots will then pass back this pointer as an argument to every subsequent plugin call that involves the same object. Apart from storing its address and passing it back, Webots does nothing with the object; it is completely safe for you to cast to any pointer type. However, the simplest and most effective method is to directly cast the addresses of your data instances. You are however free to do otherwise, provided that you assign a unique reference to each object.

Your data types should contain certain attributes in order for the Fast2D functions to be able to operate on them. The UML diagram in in figure 7.1 shows the types and attributes that make sense according to the Fast2D functionality. This diagram is an implementation guideline for your own type declarations. We recommended implementing four data types in order to match exactly the four Fast2D basic types; we also suggest that in the implementation of these types you use similar attributes as those indicated in the diagram.

- `ObjectRef`: Reference to a solid or a robot object. ObjectRef is used in the Fast2D API to indicate that both SolidRef and RobotRef are suitable parameters. ObjectRef can be considered as a base class for a solid object or a robot because it groups the attributes common to both objects. These attributes are the object's position (xpos and ypos) and orientation (angle), the object's mass, the object's bounding radius (for circular objects) and the object's bounding rectangle (for rectangular objects). The object's position and angle are defined with respect to the world's coordinate system.

- `SolidRef`: Reference for a solid object. A SolidRef has the same physical properties as ObjectRef, but it is used to implement a wall or another obstacle.

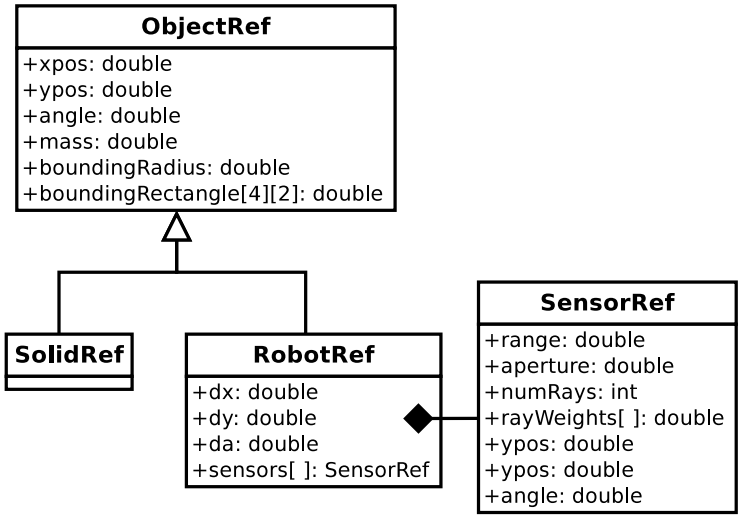


Figure 7.1: Fast2D Plugin Entity Relationship

- `RobotRef`: Reference for a robot object. A RobotRef has the same physical properties as an ObjectRef, but additionally contains linear speed (dx and dy) and angular speed (da). It is used to implement a differential wheeled robot.

- `SensorRef`: Reference for a distance sensor object. A SensorRef represents a distance sensor that must be associated with a robot (RobotRef). SensorRef attributes are: the sensor's maximal range (range), the sensor's aperture angle in radians (aperture), the number of rays of the sensor (numRays), the weight of the individual rays (rayWeights), and the position (xpos and ypos) and orientation (angle) of the sensor. The sensor's position and angle are defined with respect to the coordinate system of the corresponding robot.

### 7.4.3 Fast2D Plugin Functions

In order for your plugin to be operational, it has to implement *all* of the Fast2D functions. Once the plugin is loaded, Webots checks that every function is present; if a function is missing, Webots will attempt to run the simulation using the built-in 3D routines instead of the Fast2D plugin.

The Fast2D API uses two types of coordinates: *global* and *local*. The *global* coordinate system is the world's coordinate system, as described in table 7.1. Positions and angles of an `ObjectRef` (including `RobotRef` and `SolidRef`) are expressed in the *global* coordinate system. On the other hand, the position and angle of `SensorRef` and the coordinates of bounding rectangles are expressed in the *local* coordinate system of the object they belong to. For example, the position and angle of a sensor is expressed with respect to the local coordinate system of the robot which contains the sensor. As in 3D, an angle of zero in the Fast2D coordinate system matches up with a direction parallel to the $x$-axis.

#### void webots_fast2d_init()

The `webots_fast2d_init()` function is called by Webots to initialize the plugin. This function is called before any other Fast2D function: its purpose is to allocate and initialize the plugin's global data structures. Note than when the **Revert** button is pressed or whenever something changes in the scene tree, Webots reinitializes the plugin by first calling `webots_fast2d_cleanup()` and then `webots_fast2d_init()`. See also figure 7.2.

#### void webots_fast2d_cleanup()

This function must be implemented to free all the data structures used by the plugin. After a call to this function, no further Fast2D calls will be made by Webots, with the exception of `webots_fast2d_init()`. A subsequent call to `webots_fast2d_init()` will indicate that the plugin must be reinitialized because the world is being re-created. The plugin is responsible for allocating and freeing all of the Fast2D objects. If `webots_fast2d_cleanup()` fails to free all the memory that was allocated by the plugin, this will result in memory leaks in Webots.

#### void webots_fast2d_step(double dt)

This function must perform a simulation step of `dt` seconds. It is invoked by Webots once for each simulation step (basic simulation step) when the simulation is running, or once each time the **Step** button is pressed. The `dt` parameter corresponds to the world's basic time step (set in the `WorldInfo` node) converted to seconds (i.e., divided by 1000). The job of this function is to compute the new position and angle (as returned by `webots_fast2d_object_get_transform()`) of every simulated object (`ObjectRef`) according to your implementation of kinematics and collision handling. This function usually requires the largest amount of implementation work on the user's part.

#### RobotRef webots_fast2d_create_robot()

Requests the creation of a robot by the plugin. This function must return a valid robot reference (`RobotRef`) to Webots. The exact properties of the robot will be specified in subsequent Fast2D calls.

#### SolidRef webots_fast2d_create_solid()

Requests the creation of a solid object by the plugin. This function must return a valid solid reference (`SolidRef`) to Webots. The exact properties of the solid object will be specified in subsequent Fast2D calls.

**void webots_fast2d_add_object(ObjectRef object)**

Requests the insertion of an object (robot or solid) into the 2D world model. This function is called by Webots after an object's properties have been set and before executing the first simulation step (`webots_fast2d_step()`).

**SensorRef webots_fast2d_create_irsensor(RobotRef robot, double xpos, double ypos, double angle, double range, double aperture, int numRays, const double rayWeights[])**

Requests the creation of an infra-red sensor. This function must return a valid sensor reference (`SensorRef`) to Webots. The `robot` parameter is a robot reference previously created through `webots_fast2d_create_robot()`. The `xpos`, `ypos` and `angle` parameters indicate the desired position and orientation of the sensor in the the local coordinate system of the robot. The `range` parameter indicates the maximum range of the sensor. It is determined by the `lookupTable` of the corresponding `DistanceSensor` in the Webots scene tree. The `aperture` parameter corresponds to the value of the `aperture` field of the `DistanceSensor`. The `numRays` parameter indicates the value of the `numberOfRays` field of the `DistanceSensor`. The `rayWeights` parameter is an array of `numRays` double-precision floats which specifies the individual weights that must be associated with each sensor ray. The sum of the ray weights provided by Webots is always exactly 1.0, and it is always left/right symmetrical. For more information on the sensor weights, please refer to the description of the `DistanceSensor` node in the Webots Reference Manual. In order to be consistent with the Webots graphical representation, the plugin's implementation of the sensors requires that:

- All the rays have the same length (the specified sensor range)
- The rays are distributed uniformly (equal angles from each other)
- The angle between the first and the last ray be exactly equal to the specified aperture

**double webots_fast2d_sensor_get_activation(SensorRef sensor)**

Requests the current distance measured by a sensor. The `sensor` parameter is a sensor reference that was created through `webots_fast2d_create_irsensor()`. This function must return the average of the weighted distances measured by the sensor rays. The distances must be weighted using the `rayWeights` values that were passed to `webots_fast2d_create_irsensor()`. Note that this function is responsible only for calculating the *weighted average distance* measured by the sensor. It is Webots responsibility to compute the *final activation value* (the value that will finally be returned to the controller) from the average distance and according to the `DistanceSensor`'s lookup table.

**void webots_fast2d_object_set_bounding_rectangle(ObjectRef object, const double x[4], const double y[4])**

Defines an object as rectangular and sets the object's bounding rectangle. The `object` parameter is a solid or robot reference. The `x` and `y` arrays specify the coordinates of the four corners of the bounding rectangle in the object's coordinate system. The sequence (x[0], y[0]), (x[1], y[1]), (x[2], y[2]), (x[3], y[3]) is specified counter-clockwise.

**void webots_fast2d_object_set_bounding_radius(ObjectRef object, double radius)**

Defines an object as circular and sets the objects's bounding radius. The `object` parameter is a solid or robot reference. In the Fast2D plugin, an object can be either rectangular or circular; Webots indicates this by calling either `webots_fast2d_object_set_bounding_rectangle()` or `webots_fast2d_object_set_bounding_radius()`.

**void webots_fast2d_object_set_mass(ObjectRef object, double mass)**

Request to set the mass of an object. The `object` parameter is a solid or robot reference. The `mass` parameter is the object's required mass. According to your custom implementation, the mass of an object can be involved in the calculation of a robot's acceleration and ability to push other objects. The implementation of this function is optional. Note that Webots calls this function only if the corresponding object has a `Physics` node. In this case the `mass` parameter equals the `mass` field of the `Physics` node. A negative mass must be considered infinite. If your model does not support the concept of mass, you should implement an empty `webots_fast2d_object_set_mass()` function.

**void webots_fast2d_object_set_position(ObjectRef object, double xpos, double ypos)**

Request to set the position of an object. The `object` parameter is a solid or robot reference. The `xpos` and `ypos` parameters represent the required position specified in the global coordinate system. This function is called by Webots during the construction of the world model. Afterwards, the object positions are only modified by the `webots_fast2d_step()` function. See also figure 7.2.

**void webots_fast2d_object_set_angle(ObjectRef object, double angle)**

Request to set the angle of an object. The `object` parameter is a solid or robot reference. The `angle` parameter is the requested object angle specified in the global coordinate system. This function is called by Webots during the construction of the world model. Afterwards, the object angles are only modified by the `webots_fast2d_step()` function. See also figure 7.2.

**void webots_fast2d_robot_set_speed(RobotRef robot, double dx, double dy)**

Request to change the speed of a robot. The `robot` parameter is a robot reference. The `dx` and `dy` parameters are the two vector components of the robot's speed in the global coordinate system. This corresponds to change per second in the position of the robot (xpos and ypos). More precisely: $dx = v * sin(\alpha)$ and $dy = v * cos(\alpha)$, where $\alpha$ is the robot's orientation angle and where $v$ is the robot's absolute speed which is calculated according to the wheels' radius and rotation speed. For more information, see the description of the `DifferentialWheels` node and the `differential_wheels_set_speed()` function in the Webots Reference Manual.

**void webots_fast2d_robot_set_angular_speed(RobotRef object, double da)**

Request to change the angular speed of a robot. The `robot` parameter is a robot reference. The `da` parameter indicates the requested angular speed. A robot's angular speed is the speed of its rotation around its center in radians per second.

**void webots_fast2d_object_get_transform(ObjectRef object, double *xpos, double *ypos, double *angle)**

Reads the current position and angle of an object. The `object` parameter is a robot or solid reference. The `xpos`, `ypos` and `angle` parameters are the pointers to `double` values where this function should write the values. These parameters are specified according to the global coordinate system.

### 7.4.4 Fast2D Plugin Execution Scheme

This section describes the sequence used by Webots for calling the plugin functions. Please refer to the diagram in figure 7.2.

1. The plugin is loaded. Go to step 2.
2. The `webots_fast2d_init()` function is called. Go to step 3 or 5.
3. The world model is created. This is achieved through a sequence of calls to the functions `webots_fast2d_create_*()`, `webots_fast2d_set_*()` and `webots_fast2d_add_*()`. Question marks are used to represent a choice among several functions names. Although the exact sequence is unspecified, for each object it is guarantied that: the corresponding `webots_fast2d_create_*()` function is called first, the corresponding `webots_fast2d_set_*()` functions are called next and that the corresponding `webots_ fast2d_add_*()` function is called last. Go to step 4 or 5.

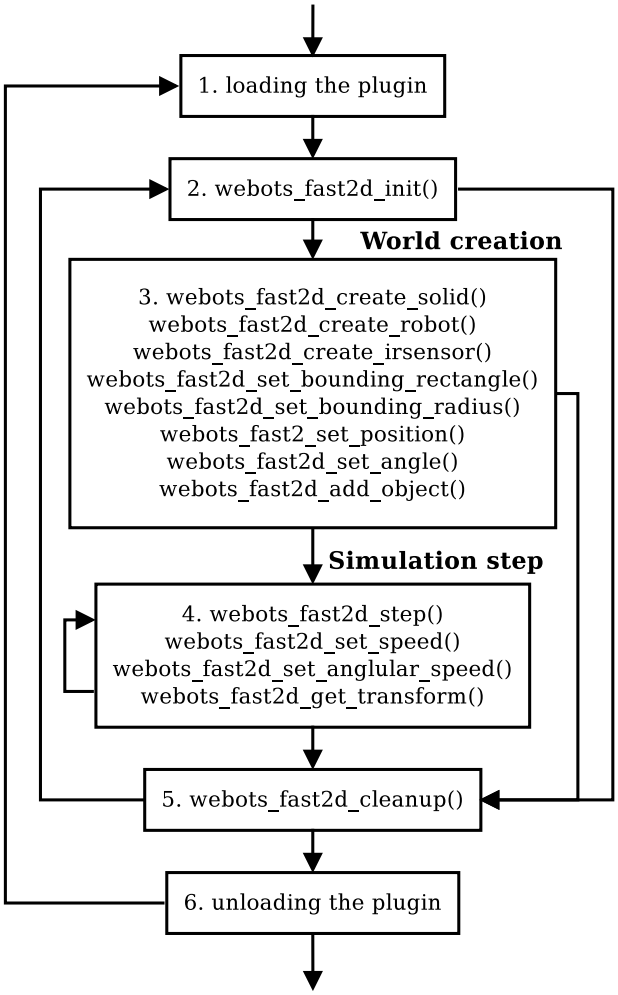


Figure 7.2: Fast2D Plugin Execution Scheme

4. A simulation step is carried out. This is achieved through an unspecified sequence of calls to `webots_fast2d_step()`, `webots_fast2d_set_speed`, `webots_fast2d_set_angular_speed()` and `webots_fast2d_get_transform()`. Go to step 4 or 5.
5. The `webots_fast2d_cleanup()` function is called. Go to step 2 or 6.
6. The plugin is unloaded. Go to step 1.

### 7.4.5 Fast2D Execution Example

This section shows an example of a Webots scene tree and the corresponding Fast2D calls that are carried out when the world is interpreted using Fast2D. Ellipses represent omitted code or parameters. Examine this example carefully. In keeping with what was explained earlier, you will notice that, when transformed from 3D to Fast2D:

- The objects rotation angles are negated
- The objects' $y$-coordinates (height and elevation) are ignored
- The 3D $z$-axis becomes the Fast2D y-axis

```
Solid {
  translation 0.177532 0.03 0.209856
  rotation 0 1 0 0.785398
  ...
  boundingObject Box {
    size 0.2 0.06 0.2
  }
}

DifferentialWheels {
  translation -0.150197 0 0.01018
  rotation 0 1 0 -4.85101
  children [
    ...
    DistanceSensor {
      translation -0.0245 0.0145 -0.012
      rotation 0 1 0 3.0543
      ...
      lookupTable [
        0 1023 0
        0.05 0 0.01
      ]
      aperture 0.5
    }
```

```
  ]
  ...
  boundingObject Transform {
    translation 0 0.011 0
    children [
      Cylinder {
        height 0.022
        radius 0.0285
      }
    ]
  }
...
}

webots_fast2d_init()
webots_fast2d_create_solid()
webots_fast2d_object_set_bounding_polygon(...)
webots_fast2d_object_set_position(..., xpos=0.177532, ypos=0.209856)
webots_fast2d_object_set_angle(..., angle=-0.785398)
webots_fast2d_add_object()

webots_fast2d_create_robot()
webots_fast2d_object_set_bounding_radius(..., radius=0.0285)
webots_fast2d_object_set_position(..., xpos=-0.150197, ypos=0.01018)
webots_fast2d_object_set_angle(..., angle=4.85101)
webots_fast2d_add_object()

webots_create_irsensor(..., xpos=-0.0245, ypos=-0.012, angle=-3.0543,
  range=0.05, aperture=0.5, numRays=1, ...)
```

Finally, note that the largest input value of the `DistanceSensor`'s lookup table (0.05) becomes the sensor's range in Fast2D.

You will find further information about the `DifferentialWheels` and `DistanceSensor` nodes and controller API in the Webots Reference Manual.

# Chapter 8

# Webots World Files

## 8.1 Generalities

Webots world files must use the `.wbt` file name extension. The first line of a `.wbt` file uses this header:

```
#VRML_SIM V6.0 utf8
```

where the version *6.0* specifies that the file can be open with *Webots 6* and *Webots 7*. Although the header specifies *utf8*, at the moment only ascii is supported.

The comments placed just below the header store the window configuration associated with this world.

One (and only one) instance of each of the `WorldInfo, ViewPoint` and `Background` nodes must be present in every `.wbt` file. For example:

```
#VRML_SIM V6.0 utf8

WorldInfo {
  info [
    "Description"
    "Author: first name last name <e-mail>"
    "Date: DD MMM YYYY"
  ]
}
Viewpoint {
  orientation 1 0 0 -0.8
  position 0.25 0.708035 0.894691
}
Background {
  skyColor [
    0.4 0.7 1
```

```
  ]
}
PointLight {
  ambientIntensity 0.54
  intensity 0.5
  location 0 1 0
}
```

## 8.2 Nodes and Keywords

### 8.2.1 VRML97 nodes

Webots implements only a subset of the nodes and fields specified by the VRML97 standard. In the other hand, Webots also adds many nodes, which are not part of the VRML97 standard, but are specialized to model robotic experiments.

The following VRML97 nodes are supported by Webots:

`Appearance, Background, Box, Color, Cone, Coordinate, Cylinder, DirectionalLight, ElevationGrid, Fog, Group, ImageTexture, IndexedFaceSet, IndexedLineSet, Material, PointLight, Shape, Sphere, SpotLight, TextureCoordinate, TextureTransform, Transform, Viewpoint` and `WorldInfo`.

Please refer to chapter 3 for a detailed description of Webots nodes and fields. It specifies which fields are actually used. For a comprehensive description of the VRML97 nodes, you can also refer to the VRML97 documentation.

The exact features of VRML97 are subject to a standard managed by the International Standards Organization (ISO/IEC 14772-1:1997). You can find the complete specification of VRML97 on the Web3D Web site[1].

### 8.2.2 Webots specific nodes

In order to describe more precisely robotic simulations, Webots supports additional nodes that are not specified by the VRML97 standard. These nodes are principally used to model commonly used robot devices. Here are Webots additional nodes:

`Accelerometer, BallJoint, BallJointParameters, Camera, CameraZoom, Capsule, Charger, Compass, Connector, ContactProperties, Damping, DifferentialWheels, DistanceSensor, Display, Emitter, GPS, Gyro, HingeJoint, HingeJointParameters, Hinge2Joint, Hinge2JointParameters,`

[1] `http://www.web3d.org`

InertialUnit, JointParameters, LED, LightSensor, LinearMotor, Pen, Physics, Plane, PositionSensor, Receiver, Robot, RotationalMotor, Servo, SliderJoint, Solid, SolidReference, Supervisor, and TouchSensor.

Please refer to chapter 3 for a detailed description of Webots nodes and fields.

### 8.2.3 Reserved keywords

These reserved keywords cannot be used in DEF or PROTO names:

DEF, USE, PROTO, IS, TRUE, FALSE, NULL, field, vrmlField, SFNode, SFColor, SFFloat, SFInt32, SFString, SFVec2f, SFVec3f, SFRotation, SFBool, MFNode, MFColor, MFFloat, MFInt32, MFString, MFVec2f and MFVec3f.

## 8.3 DEF and USE

A node which is named using the DEF keyword can be referenced later by its name in the same file with USE statements. The DEF and USE keywords can be used to reduce redundancy in `.wbt` and `.proto` files. DEF name are limited in scope to a single `.wbt` or `.proto` file. If multiple nodes are given the same DEF name, each USE statement refers to the closest node with the given DEF name preceding it in the `.wbt` or `.proto` file.

```
[DEF defName] nodeName { nodeBody }

USE defName
```


*Although it is permitted to name any node using the DEF keyword, USE statements are not allowed for* Solid, Joint, JointParameters, *and* BallJointParameters *nodes and their derived nodes. Indeed, the ability for identical solids or joints to occupy the same position is useless, if not hazardous, in a physics simulation. To safely duplicate one of these nodes, you can design a* PROTO *model for this node and then add different PROTO instances to your world.*

# Chapter 9

# Other APIs

Webots allows to program controllers in some other languages than C. This chapter describes the API of these other languages. Each section corresponds to one language and each subsection to a device. This chapter should be used with the chapter 3 of this document which describes the C functions. Generally speaking, each C function has one and only one counterpart for in a specific language.

## 9.1 C++ API

The following tables describe the C++ classes and their methods.

| #include <webots/Accelerometer.hpp> |
|---|
| class `Accelerometer` : public `Device` { |
| virtual void `enable`(int ms); |
| virtual void `disable`(); |
| int `getSamplingPeriod`(); |
| const double *`getValues`() const; |
| }; |

| #include <webots/Brake.hpp> |
|---|
| class `Brake` : public `Device` { |
| void `setDampingConstant`(double dampingConstant) const; |
| int `getType`() const; |
| }; |

```
#include <webots/Camera.hpp>
class Camera : public Device {
  enum {COLOR, RANGE_FINDER, BOTH};
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  double getFov() const;
  virtual void setFov(double fov);
  int getWidth() const;
  int getHeight() const;
  double getNear() const;
  double getMaxRange() const;
  int getType() const;
  const unsigned char *getImage() const;
  static unsigned char imageGetRed(const unsigned char *image,
    int width, int x, int y);
  static unsigned char imageGetGreen(const unsigned char *image,
    int width, int x, int y);
  static unsigned char imageGetBlue(const unsigned char *image,
    int width, int x, int y);
  static unsigned char imageGetGrey(const unsigned char *image,
    int width, int x, int y);
  const float *getRangeImage() const;
  static float rangeImageGetDepth(const float *image,
    int width, int x, int y);
  int saveImage(const std::string &filename, int quality) const;
};
```

```
#include <webots/Compass.hpp>
class Compass : public Device {
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  const double *getValues() const;
};
```

```
#include <webots/Connector.hpp>
class Connector : public Device {
  virtual void enablePresence(int ms);
  virtual void disablePresence();
  int getPresence() const;
  virtual void lock();
  virtual void unlock();
};
```

```
#include <webots/Device.hpp>
class Device {
  const std::string &getName() const;
  int getNodeType() const;
};
```

```
#include <webots/DifferentialWheels.hpp>
class DifferentialWheels : public Robot {
  DifferentialWheels();
  virtual ~DifferentialWheels();
  virtual void setSpeed(double left, double right);
  double getLeftSpeed() const;
  double getRightSpeed() const;
  virtual void enableEncoders(int ms);
  virtual void disableEncoders();
  int getEncodersSamplingPeriod();
  double getLeftEncoder() const;
  double getRightEncoder() const;
  virtual void setEncoders(double left, double right);
  double getMaxSpeed() const;
  double getSpeedUnit() const;
};
```

```
#include <webots/Display.hpp>
class Display : public Device {
  enum {RGB, RGBA, ARGB, BGRA};
  int getWidth() const;
  int getHeight() const;
  virtual void setColor(int color);
  virtual void setAlpha(double alpha);
  virtual void setOpacity(double opacity);
  virtual void drawPixel(int x1, int y1);
  virtual void drawLine(int x1, int y1, int x2, int y2);
  virtual void drawRectangle(int x, int y, int width, int height);
  virtual void drawOval(int cx, int cy, int a, int b);
  virtual void drawPolygon(const int *x, const int *y, int size);
  virtual void drawText(const std::string &txt, int x, int y);
  virtual void fillRectangle(int x, int y, int width, int height);
  virtual void fillOval(int cx, int cy, int a, int b);
  virtual void fillPolygon(const int *x, const int *y, int size);
  ImageRef *imageCopy(int x, int y, int width, int height) const;
  virtual void imagePaste(ImageRef *ir, int x, int y);
  ImageRef *imageLoad(const std::string &filename) const;
  ImageRef *imageNew(int width, int height, const void *data, int format) const;
  void imageSave(ImageRef *ir, const std::string &filename) const;
  void imageDelete(ImageRef *ir) const;
};
```

```
#include <webots/DistanceSensor.hpp>
class DistanceSensor : public Device {
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  double getValue() const;
};
```

```
#include <webots/Emitter.hpp>
class Emitter : public Device {
  enum {CHANNEL_BROADCAST};
  virtual int send(const void *data, int size);
  int getChannel() const;
  virtual void setChannel(int channel);
  double getRange() const;
  virtual void setRange(double range);
  int getBufferSize() const;
};
```

```
#include <webots/Field.hpp>
class Field {
  enum { SF_BOOL, SF_INT32, SF_FLOAT, SF_VEC2F, SF_VEC3F, SF_ROTATION,
SF_COLOR, SF_STRING, SF_NODE, MF, MF_INT32, MF_FLOAT, MF_VEC2F,
MF_VEC3F, MF_COLOR, MF_STRING, MF_NODE };
  int getType() const;
  std::string getTypeName() const;
  int getCount() const;
  bool getSFBool() const;
  int getSFInt32() const;
  double getSFFloat() const;
  const double *getSFVec2f() const;
  const double *getSFVec3f() const;
  const double *getSFRotation() const;
  const double *getSFColor() const;
  std::string getSFString() const;
  Node *getSFNode() const;
  int getMFInt32(int index) const;
  double getMFFloat(int index) const;
  const double *getMFVec2f(int index) const;
  const double *getMFVec3f(int index) const;
  const double *getMFColor(int index) const;
  std::string getMFString(int index) const;
  Node *getMFNode(int index) const;
  void setSFBool(bool value);
  void setSFInt32(int value);
  void setSFFloat(double value);
  void setSFVec2f(const double values[2]);
  void setSFVec3f(const double values[3]);
  void setSFRotation(const double values[4]);
  void setSFColor(const double values[3]);
  void setSFString(const std::string &value);
  void setMFInt32(int index, int value);
  void setMFFloat(int index, double value);
  void setMFVec2f(int index, const double values[2]);
  void setMFVec3f(int index, const double values[3]);
  void setMFColor(int index, const double values[3]);
  void setMFString(int index, const std::string &value);
  void importMFNode(int position, const std::string &filename);
};
```

```
#include <webots/GPS.hpp>
class GPS : public Device {
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  const double *getValues() const;
};
```

```
#include <webots/Gyro.hpp>
class Gyro : public Device {
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  const double *getValues() const;
};
```

```
#include <webots/ImageRef.hpp>
class ImageRef {
};
```

```
#include <webots/InertialUnit.hpp>
class InertialUnit : public Device {
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  const double *getRollPitchYaw() const;
};
```

```
#include <webots/LED.hpp>
class LED : public Device {
  virtual void set(int value);
  int set() const;
};
```

```
#include <webots/LightSensor.hpp>
class LightSensor : public Device {
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  double getValue() const;
};
```

```
#include <webots/utils/Motion.hpp>
class Motion {
  Motion(const std::string &fileName);
  virtual ~Motion();
  bool isValid() const;
  virtual void play();
  virtual void stop();
  virtual void setLoop(bool loop);
  virtual void setReverse(bool reverse);
  bool isOver() const;
  int getDuration() const;
  int getTime() const;
  virtual void setTime(int time);
};
```

```
#include <webots/Motor.hpp>
class Motor : public Device {
  enum {ROTATIONAL, LINEAR};
  virtual void setPosition(double position);
  double getTargetPosition(double position) const;
  virtual void setVelocity(double vel);
  virtual void setAcceleration(double force);
  virtual void setAvailableForce(double motor_force);
  virtual void setAvailableTorque(double motor_torque);
  virtual void setControlPID(double p, double i, double d);
  double getMinPosition() const;
  double getMaxPosition() const;
  virtual void enableForceFeedback(int ms);
  virtual void disableForceFeedback();
  int getForceFeedbackSamplingPeriod();
  double getForceFeedback() const;
  virtual void setForce(double force);
  virtual void enableTorqueFeedback(int ms);
  virtual void disableTorqueFeedback();
  int getTorqueFeedbackSamplingPeriod();
  double getTorqueFeedback() const;
  virtual void setTorque(double torque);
  int getType() const;
};
```

```
#include <webots/Node.hpp>
class Node {
  enum { NO_NODE, APPEARANCE, BACKGROUND, BOX, COLOR, CONE,
  COORDINATE, CYLINDER, DIRECTIONAL_LIGHT, ELEVATION_GRID,
  EXTRUSION, FOG, GROUP, IMAGE_TEXTURE, INDEXED_FACE_SET,
  INDEXED_LINE_SET, MATERIAL, POINT_LIGHT, SHAPE, SPHERE,
  SPOT_LIGHT, SWITCH, TEXTURE_COORDINATE, TEXTURE_TRANSFORM,
  TRANSFORM, VIEWPOINT, WORLD_INFO, CAPSULE, PLANE, ROBOT,
  SUPERVISOR, DIFFERENTIAL_WHEELS, SOLID, PHYSICS, CAMERA_ZOOM,
  CHARGER, DAMPING, CONTACT_PROPERTIES, ACCELEROMETER, BRAKE,
  CAMERA, COMPASS, CONNECTOR, DISPLAY, DISTANCE_SENSOR,
  EMITTER, GPS,GYRO, LED, LIGHT_SENSOR, MICROPHONE, MOTOR, PEN,
  POSITION_SENSOR, RADIO, RECEIVER, SERVO, SPEAKER,
  TOUCH_SENSOR };
  int getType() const;
  std::string getTypeName() const;
  Field *getField(const std::string &fieldName) const;
  const double *getPosition() const;
  const double *getOrientation() const;
  const double *getCenterOfMass() const;
  const double *getContactPoint(int index) const;
  int getNumberOfContactPoints() const;
  bool getStaticBalance() const;
  void resetPhysics();
};
```

```
#include <webots/Pen.hpp>
class Pen : public Device {
  virtual void write(bool write);
  virtual void setInkColor(int color, double density);
};
```

```
#include <webots/PositionSensor.hpp>
class PositionSensor : public Device {
  enum {ANGULAR, LINEAR};
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  double getValue() const;
  int getType() const;
};
```

```
#include <webots/Receiver.hpp>
class Receiver : public Device {
  enum {CHANNEL_BROADCAST};
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  int getQueueLength() const;
  virtual void nextPacket();
  const void *getData() const;
  int getDataSize() const;
  double getSignalStrength() const;
  const double *getEmitterDirection() const;
  virtual void setChannel(int channel);
  int getChannel() const;
};
```

```
#include <webots/Robot.hpp>
class Robot {
  enum {MODE_SIMULATION, MODE_CROSS_COMPILATION,
  MODE_REMOTE_CONTROL};
  enum {KEYBOARD_END, KEYBOARD_HOME, KEYBOARD_LEFT,
  KEYBOARD_UP, KEYBOARD_RIGHT, KEYBOARD_DOWN,
  KEYBOARD_PAGEUP, KEYBOARD_PAGEDOWN,
  KEYBOARD_NUMPAD_HOME, KEYBOARD_NUMPAD_LEFT,
  KEYBOARD_NUMPAD_UP, KEYBOARD_NUMPAD_RIGHT,
  KEYBOARD_NUMPAD_DOWN, KEYBOARD_NUMPAD_END,
  KEYBOARD_KEY, KEYBOARD_SHIFT, KEYBOARD_CONTROL,
  KEYBOARD_ALT};
  Robot();
  virtual ~Robot();
  virtual int step(int ms);
  Accelerometer *getAccelerometer(const std::string &name);
  Brake *getBrake(const std::string &name);
  Camera *getCamera(const std::string &name);
  Compass *getCompass(const std::string &name);
  Connector *getConnector(const std::string &name);
  Display *getDisplay(const std::string &name);
  DistanceSensor *getDistanceSensor(const std::string &name);
  Emitter *getEmitter(const std::string &name);
  GPS *getGPS(const std::string &name);
  Gyro *getGyro(const std::string &name);
  InertialUnit *getInertialUnit(const std::string &name);
  LED *getLED(const std::string &name);
  LightSensor *getLightSensor(const std::string &name);
  Motor *getMotor(const std::string &name);
  Pen *getPen(const std::string &name);
  PositionSensor *getPositionSensor(const std::string &name);
  Receiver *getReceiver(const std::string &name);
  Servo *getServo(const std::string &name);
  TouchSensor *getTouchSensor(const std::string &name);
  int getNumberOfDevices();
  Device *getDeviceByIndex(int index);
  virtual void batterySensorEnable(int ms);
  virtual void batterySensorDisable();
  int batterySensorGetSamplingPeriod();
```

```
  double batterySensorGetValue() const;
  double getBasicTimeStep() const;
  int getMode() const;
  std::string getModel() const;
  std::string getData() const;
  void setData(const std::string &data);
  std::string getName() const;
  std::string getControllerName() const;
  std::string getControllerArguments() const;
  std::string getProjectPath() const;
  bool getSynchronization() const;
  double getTime() const;
  virtual void keyboardEnable(int ms);
  virtual void keyboardDisable();
  int keyboardGetKey() const;
  int getType() const;
};
```

```
#include <webots/Servo.hpp>
class Servo : public Device {
  enum {ROTATIONAL, LINEAR};
  virtual void setPosition(double position);
  double getTargetPosition(double position) const;
  virtual void setVelocity(double vel);
  virtual void setAcceleration(double force);
  virtual void setMotorForce(double motor_force);
  virtual void setControlP(double p);
  double getMinPosition() const;
  double getMaxPosition() const;
  virtual void enablePosition(int ms);
  virtual void disablePosition();
  int getPositionSamplingPeriod();
  double getPosition() const;
  virtual void enableMotorForceFeedback(int ms);
  virtual void disableMotorForceFeedback();
  int getMotorForceFeedbackSamplingPeriod();
  double getMotorForceFeedback() const;
  virtual void setForce(double force);
  int getType() const;
};
```

```
#include <webots/Supervisor.hpp>
class Supervisor : public Robot {
  enum {MOVIE_READY, MOVIE_RECORDING, MOVIE_SAVING, MOVIE_WRITE_ERROR, MOVIE_ENCODING_ERROR, MOVIE_SIMULATION_ERROR};
  Supervisor();
  virtual ~Supervisor();
  void exportImage(const std::string &file, int quality) const;
  Node *getRoot();
  Node *getSelf();
  Node *getFromDef(const std::string &name);
  virtual void setLabel(int id, const std::string &label, double xpos, double ypos,
    double size, int color, double transparency);
  virtual void simulationQuit(int status);
  virtual void simulationRevert();
  virtual void simulationResetPhysics();
  void startMovie(const std::string &file, int width, int height, int codec, int quality,
  int acceleration, bool caption) const;
  void stopMovie() const;
  int getMovieStatus() const;
};
```

```
#include <webots/TouchSensor.hpp>
class TouchSensor : public Device {
  enum {BUMPER, FORCE, FORCE3D};
  virtual void enable(int ms);
  virtual void disable();
  int getSamplingPeriod();
  double getValue() const;
  const double *getValues() const;
  int getType() const;
};
```

## 9.2 Java API

The following tables describe the Java classes and their methods.

| import com.cyberbotics.webots.controller.Accelerometer; |
|---|
| public class `Accelerometer` extends `Device` { |
| public void `enable`(int ms); |
| public void `disable`(); |
| int `getSamplingPeriod`(); |
| public double[] `getValues`(); |
| } |

| import com.cyberbotics.webots.controller.Brake; |
|---|
| public class `Brake` extends `Device` { |
| public void `setDampingConstant`(double dampingConstant); |
| public int `getType`(); |
| } |

```
import com.cyberbotics.webots.controller.Camera;
public class Camera extends Device {
  public final static int COLOR, RANGE_FINDER, BOTH;
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double getFov();
  public void setFov(double fov);
  public int getWidth();
  public int getHeight();
  public double getNear();
  public double getMaxRange();
  public int getType();
  public int[] getImage();
  public static int imageGetRed(int[] image, int width, int x, int y);
  public static int imageGetGreen(int[] image, int width, int x, int y);
  public static int imageGetBlue(int[] image, int width, int x, int y);
  public static int imageGetGrey(int[] image, int width, int x, int y);
  public static int pixelGetRed(int pixel);
  public static int pixelGetGreen(int pixel);
  public static int pixelGetBlue(int pixel);
  public static int pixelGetGrey(int pixel);
  public float[] getRangeImage();
  public static float rangeImageGetDepth(float[] image,
    int width, int x, int y);
  public int saveImage(String filename, int quality);
}
```

```
import com.cyberbotics.webots.controller.Compass;
public class Compass extends Device {
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double[] getValues();
}
```

```
import com.cyberbotics.webots.controller.Connector;
public class Connector extends Device {
  public void enablePresence(int ms);
  public void disablePresence();
  public int getPresence();
  public void lock();
  public void unlock();
}
```

```
import com.cyberbotics.webots.controller.Device;
public class Device {
  public String getName();
  public int getNodeType();
}
```

```
import com.cyberbotics.webots.controller.DifferentialWheels;
public class DifferentialWheels extends Robot {
  public DifferentialWheels();
  protected void finalize();
  public void setSpeed(double left, double right);
  public double getLeftSpeed();
  public double getRightSpeed();
  public void enableEncoders(int ms);
  public void disableEncoders();
  public int getEncodersSamplingPeriod();
  public double getLeftEncoder();
  public double getRightEncoder();
  public void setEncoders(double left, double right);
  public double getMaxSpeed();
  public double getSpeedUnit();
}
```

```
import com.cyberbotics.webots.controller.Display;
public class Display extends Device {
  public final static int RGB, RGBA, ARGB, BGRA;
  public int getWidth();
  public int getHeight();
  public void setColor(int color);
  public void setAlpha(double alpha);
  public void setOpacity(double opacity);
  public void drawPixel(int x1, int y1);
  public void drawLine(int x1, int y1, int x2, int y2);
  public void drawRectangle(int x, int y, int width, int height);
  public void drawOval(int cx, int cy, int a, int b);
  public void drawPolygon(int[] x, int[] y);
  public void drawText(String txt, int x, int y);
  public void fillRectangle(int x, int y, int width, int height);
  public void fillOval(int cx, int cy, int a, int b);
  public void fillPolygon(int[] x, int[] y);
  public ImageRef imageCopy(int x, int y, int width, int height);
  public void imagePaste(ImageRef ir, int x, int y);
  public ImageRef imageLoad(String filename);
  public ImageRef imageNew(int width, int height, int[] data, int format);
  public void imageSave(ImageRef ir, String filename);
  public void imageDelete(ImageRef ir);
}
```

```
import com.cyberbotics.webots.controller.DistanceSensor;
public class DistanceSensor extends Device {
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double getValue();
}
```

```
import com.cyberbotics.webots.controller.Emitter;
public class Emitter extends Device {
  public final static int CHANNEL_BROADCAST;
  public int send(byte[] data);
  public int getChannel();
  public void setChannel(int channel);
  public double getRange();
  public void setRange(double range);
  public int getBufferSize();
}
```

```
import com.cyberbotics.webots.controller.Field;
public class Field {
  public final static int SF_BOOL, SF_INT32, SF_FLOAT,
  SF_VEC2F, SF_VEC3F, SF_ROTATION, SF_COLOR, SF_STRING,
  SF_NODE, MF, MF_INT32, MF_FLOAT, MF_VEC2F, MF_VEC3F,
  MF_COLOR, MF_STRING, MF_NODE;
  public int getType();
  public String getTypeName();
  public int getCount();
  public bool getSFBool();
  public int getSFInt32();
  public double getSFFloat();
  public double[] getSFVec2f();
  public double[] getSFVec3f();
  public double[] getSFRotation();
  public double[] getSFColor();
  public String getSFString();
  public Node getSFNode();
  public int getMFInt32(int index);
  public double getMFFloat(int index);
  public double[] getMFVec2f(int index);
  public double[] getMFVec3f(int index);
  public double[] getMFColor(int index);
  public String getMFString(int index);
  public Node getMFNode(int index);
  public void setSFBool(bool value);
  public void setSFInt32(int value);
  public void setSFFloat(double value);
  public void setSFVec2f(double values[2]);
  public void setSFVec3f(double values[3]);
  public void setSFRotation(double values[4]);
  public void setSFColor(double values[3]);
  public void setSFString(String value);
  public void setMFInt32(int index, int value);
  public void setMFFloat(int index, double value);
  public void setMFVec2f(int index, double values[2]);
  public void setMFVec3f(int index, double values[3]);
  public void setMFColor(int index, double values[3]);
  public void setMFString(int index, String value);
```

```
  public void importMFNode(int position, String filename);
}
```

```
import com.cyberbotics.webots.controller.GPS;
public class GPS extends Device {
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double[] getValues();
}
```

```
import com.cyberbotics.webots.controller.Gyro;
public class Gyro extends Device {
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double[] getValues();
}
```

```
import com.cyberbotics.webots.controller.ImageRef;
public class ImageRef {
}
```

```
import com.cyberbotics.webots.controller.InertialUnit;
public class InertialUnit extends Device {
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double[] getRollPitchYaw();
}
```

```
import com.cyberbotics.webots.controller.LED;
public class LED extends Device {
  public void set(int state);
  public int get();
}
```

```
import com.cyberbotics.webots.controller.LightSensor;
public class LightSensor extends Device {
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double getValue();
}
```

```
import com.cyberbotics.webots.controller.Motion;
public class Motion {
  public Motion(String fileName);
  protected void finalize();
  public bool isValid();
  public void play();
  public void stop();
  public void setLoop(bool loop);
  public void setReverse(bool reverse);
  public bool isOver();
  public int getDuration();
  public int getTime();
  public void setTime(int time);
}
```

```
import com.cyberbotics.webots.controller.Motor;
public class Motor extends Device {
  public final static int ROTATIONAL, LINEAR;
  public void setPosition(double position);
  public double getTargetPosition();
  public void setVelocity(double vel);
  public void setAcceleration(double force);
  public void setAvailableForce(double motor_force);
  public void setAvailableTorque(double motor_torque);
  public void setControlPID(double p, double i, double d);
  public double getMinPosition();
  public double getMaxPosition();
  public void enableForceFeedback(int ms);
  public void disableForceFeedback();
  public int getForceFeedbackSamplingPeriod();
  public double getForceFeedback();
  public void setForce(double force);
  public void enableTorqueFeedback(int ms);
  public void disableTorqueFeedback();
  public int getTorqueFeedbackSamplingPeriod();
  public double getTorqueFeedback();
  public void setTorque(double torque);
  public int getType();
}
```

```
import com.cyberbotics.webots.controller.Node;
public class Node {
  public final static int NO_NODE, APPEARANCE, BACKGROUND,
  BOX, COLOR, CONE, COORDINATE, CYLINDER, DIRECTIONAL_LIGHT,
  ELEVATION_GRID, EXTRUSION, FOG, GROUP, IMAGE_TEXTURE,
  INDEXED_FACE_SET, INDEXED_LINE_SET, MATERIAL, POINT_LIGHT,
  SHAPE, SPHERE, SPOT_LIGHT, SWITCH, TEXTURE_COORDINATE,
  TEXTURE_TRANSFORM, TRANSFORM, VIEWPOINT, WORLD_INFO,
  CAPSULE, PLANE, ROBOT, SUPERVISOR, DIFFERENTIAL_WHEELS, SOLID,
  PHYSICS, CAMER_ZOOM, CHARGER, DAMPING,
  CONTACT_PROPERTIES, ACCELEROMETER, BRAKE, CAMERA, COMPASS,
  CONNECTOR, DISPLAY, DISTANCE_SENSOR, EMITTER, GPS, GYRO, LED,
  LIGHT_SENSOR, MICROPHONE, MOTOR, PEN, POSITION_SENSOR, RADIO,
  RECEIVER, SERVO, SPEAKER, TOUCH_SENSOR;
  public int getType();
  public String getTypeName();
  public Field getField(String fieldName);
  public double[] getPosition();
  public double[] getOrientation();
  public double[] getCenterOfMass();
  public double[] getContactPoint(int index);
  public int getNumberOfContactPoints();
  public bool getStaticBalance();
  public void resetPhysics();
}
```

```
import com.cyberbotics.webots.controller.Pen;
public class Pen extends Device {
  public void write(bool write);
  public void setInkColor(int color, double density);
}
```

```
import com.cyberbotics.webots.controller.PositionSensor;
public class PositionSensor extends Device {
  public final static int ANGULAR, LINEAR;
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double getValue();
  public int getType();
}
```

```
import com.cyberbotics.webots.controller.Receiver;
public class Receiver extends Device {
  public final static int CHANNEL_BROADCAST;
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public int getQueueLength();
  public void nextPacket();
  public byte[] getData();
  public int getDataSize();
  public double getSignalStrength();
  public double[] getEmitterDirection();
  public void setChannel(int channel);
  public int getChannel();
}
```

```
import com.cyberbotics.webots.controller.Robot;
public class Robot {
  public final static int MODE_SIMULATION,
  MODE_CROSS_COMPILATION, MODE_REMOTE_CONTROL;
  public final static int KEYBOARD_END, KEYBOARD_HOME,
  KEYBOARD_LEFT, KEYBOARD_UP, KEYBOARD_RIGHT,
  KEYBOARD_DOWN, KEYBOARD_PAGEUP, KEYBOARD_PAGEDOWN,
  KEYBOARD_NUMPAD_HOME, KEYBOARD_NUMPAD_LEFT,
  KEYBOARD_NUMPAD_UP, KEYBOARD_NUMPAD_RIGHT,
  KEYBOARD_NUMPAD_DOWN, KEYBOARD_NUMPAD_END,
  KEYBOARD_KEY, KEYBOARD_SHIFT,
  KEYBOARD_CONTROL, KEYBOARD_ALT;
  public Robot();
  protected void finalize();
  public int step(int ms);
  public Accelerometer getAccelerometer(String name);
  public Brake getBrake(String name);
  public Camera getCamera(String name);
  public Compass getCompass(String name);
  public Connector getConnector(String name);
  public Display getDisplay(String name);
  public DistanceSensor getDistanceSensor(String name);
  public Emitter getEmitter(String name);
  public GPS getGPS(String name);
  public Gyro getGyro(String name);
  public InertialUnit getInertialUnit(String name);
  public LED getLED(String name);
  public LightSensor getLightSensor(String name);
  public Motor getMotor(String name);
  public Pen getPen(String name);
  public PositionSensor getPositionSensor(String name);
  public Receiver getReceiver(String name);
  public Servo getServo(String name);
  public TouchSensor getTouchSensor(String name);
  public int getNumberOfDevices();
  public Device getDeviceByIndex(int index);
  public void batterySensorEnable(int ms);
  public void batterySensorDisable();
  public int batterySensorGetSamplingPeriod();
```

```
  public double batterySensorGetValue();
  public double getBasicTimeStep();
  public int getMode();
  public String getModel();
  public String getData();
  public setData(String data);
  public String getName();
  public String getControllerName();
  public String getControllerArguments();
  public String getProjectPath();
  public bool getSynchronization();
  public double getTime();
  public void keyboardEnable(int ms);
  public void keyboardDisable();
  public int keyboardGetKey();
  public int getType();
}
```

```
import com.cyberbotics.webots.controller.Servo;
public class Servo extends Device {
  public final static int ROTATIONAL, LINEAR;
  public void setPosition(double position);
  public double getTargetPosition();
  public void setVelocity(double vel);
  public void setAcceleration(double force);
  public void setMotorForce(double motor_force);
  public void setControlP(double p);
  public double getMinPosition();
  public double getMaxPosition();
  public void enablePosition(int ms);
  public void disablePosition();
  public int getPositionSamplingPeriod();
  public double getPosition();
  public void enableMotorForceFeedback(int ms);
  public void disableMotorForceFeedback();
  public int getMotorForceFeedbackSamplingPeriod();
  public double getMotorForceFeedback();
  public void setForce(double force);
  public int getType();
}
```

```
import com.cyberbotics.webots.controller.Supervisor;
public class Supervisor extends Robot {
  public final static int MOVIE_READY, MOVIE_RECORDING, MOVIE_SAVING,
MOVIE_WRITE_ERROR, MOVIE_ENCODING_ERROR, MOVIE_SIMULATION_-
ERROR
  public Supervisor();
  protected void finalize();
  public void exportImage(String file, int quality);
  public Node getRoot();
  public Node getSelf();
  public Node getFromDef(String name);
  public void setLabel(int id, String label, double xpos, double ypos,
    double size, int color, double transparency);
  public void simulationQuit(int status);
  public void simulationRevert();
  public void simulationResetPhysics();
  public void startMovie(String file, int width, int height, int codec, int quality,
    int acceleration, boolean caption);
  public void stopMovie();
  public int getMovieStatus();
}
```

```
import com.cyberbotics.webots.controller.TouchSensor;
public class TouchSensor extends Device {
  public final static int BUMPER, FORCE, FORCE3D;
  public void enable(int ms);
  public void disable();
  public int getSamplingPeriod();
  public double getValue();
  public double[] getValues();
  public int getType();
}
```

## 9.3 Python API

The following tables describe the Python classes and their methods.

| from controller import Accelerometer |
|---|
| class `Accelerometer` (`Device`) : |
| def `enable`(self, ms) |
| def `disable`(self) |
| def `getSamplingPeriod`(self) |
| def `getValues`(self) |

| from controller import Brake |
|---|
| class `Brake` (`Device`) : |
| def `setDampingConstant`(self, dampingConstant) |
| def `getType`(self) |

| from controller import Camera |
|---|
| class Camera (Device) : |
| COLOR, RANGE_FINDER, BOTH |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getFov(self) |
| def setFov(self, fov) |
| def getWidth(self) |
| def getHeight(self) |
| def getNear(self) |
| def getMaxRange(self) |
| def getType(self) |
| def getImage(self) |
| def getImageArray(self) |
| def imageGetRed(image, width, x, y) |
| imageGetRed = staticmethod(imageGetRed) |
| def imageGetGreen(image, width, x, y) |
| imageGetGreen = staticmethod(imageGetGreen) |
| def imageGetBlue(image, width, x, y) |
| imageGetBlue = staticmethod(imageGetBlue) |
| def imageGetGrey(image, width, x, y) |
| imageGetGrey = staticmethod(imageGetGrey) |
| def getRangeImage(self) |
| def getRangeImageArray(self) |
| def rangeImageGetDepth(image, width, x, y) |
| rangeImageGetDepth = staticmethod(rangeImageGetDepth) |
| def saveImage(self, filename, quality) |

| from controller import Compass |
|---|
| class Compass (Device) : |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValues(self) |

| from controller import Connector |
|---|
| class Connector (Device) : |
| def enablePresence(self, ms) |
| def disablePresence(self) |
| def getPresence(self) |
| def lock(self) |
| def unlock(self) |

| from controller import Device |
|---|
| class Device : |
| def getName(self) |
| def getNodeType(self) |

| from controller import DifferentialWheels |
|---|
| class DifferentialWheels (Robot) : |
| def __init__(self) |
| def __del__(self) |
| def setSpeed(self, left, right) |
| def getLeftSpeed(self) |
| def getRightSpeed(self) |
| def enableEncoders(self, ms) |
| def disableEncoders(self) |
| def getEncodersSamplingPeriod(self) |
| def getLeftEncoder(self) |
| def getRightEncoder(self) |
| def setEncoders(self, left, right) |
| def getMaxSpeed(self) |
| def getSpeedUnit(self) |

| from controller import Display |
|---|
| class Display (Device) : |
| RGB, RGBA, ARGB, BGRA |
| def getWidth(self) |
| def getHeight(self) |
| def setColor(self, color) |
| def setAlpha(self, alpha) |
| def setOpacity(self, opacity) |
| def drawPixel(self, x1, y1) |
| def drawLine(self, x1, y1, x2, y2) |
| def drawRectangle(self, x, y, width, height) |
| def drawOval(self, cx, cy, a, b) |
| def drawPolygon(self, x, y) |
| def drawText(self, txt, x, y) |
| def fillRectangle(self, x, y, width, height) |
| def fillOval(self, cx, cy, a, b) |
| def fillPolygon(self, x, y) |
| def imageCopy(self, x, y, width, height) |
| def imagePaste(self, ir, x, y) |
| def imageLoad(self, filename) |
| def imageNew(self, data, format) |
| def imageSave(self, ir, filename) |
| def imageDelete(self, ir) |

| from controller import DistanceSensor |
|---|
| class DistanceSensor (Device) : |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValue(self) |

| from controller import Emitter |
|---|
| class Emitter (Device) : |
| CHANNEL_BROADCAST |
| def send(self, data) |
| def getChannel(self) |
| def setChannel(self, channel) |
| def getRange(self) |
| def setRange(self, range) |
| def getBufferSize(self) |

| from controller import Field |
| --- |
| class Field : |
| SF_BOOL, SF_INT32, SF_FLOAT, SF_VEC2F, SF_VEC3F, |
| SF_ROTATION, SF_COLOR, SF_STRING, SF_NODE, MF, |
| MF_INT32, MF_FLOAT, MF_VEC2F, MF_VEC3F, MF_COLOR, |
| MF_STRING, MF_NODE |
| def `getType`(self) |
| def `getTypeName`(self) |
| def `getCount`(self) |
| def `getSFBool`(self) |
| def `getSFInt32`(self) |
| def `getSFFloat`(self) |
| def `getSFVec2f`(self) |
| def `getSFVec3f`(self) |
| def `getSFRotation`(self) |
| def `getSFColor`(self) |
| def `getSFString`(self) |
| def `getSFNode`(self) |
| def `getMFInt32`(self, index) |
| def `getMFFloat`(self, index) |
| def `getMFVec2f`(self, index) |
| def `getMFVec3f`(self, index) |
| def `getMFColor`(self, index) |
| def `getMFString`(self, index) |
| def `getMFNode`(self, index) |
| def `setSFBool`(self, value) |
| def `setSFInt32`(self, value) |
| def `setSFFloat`(self, value) |
| def `setSFVec2f`(self, values) |
| def `setSFVec3f`(self, values) |
| def `setSFRotation`(self, values) |
| def `setSFColor`(self, values) |
| def `setSFString`(self, value) |
| def `setMFInt32`(self, index, value) |
| def `setMFFloat`(self, index, value) |
| def `setMFVec2f`(self, index, values) |
| def `setMFVec3f`(self, index, values) |
| def `setMFColor`(self, index, values) |
| def `setMFString`(self, index, value) |

| def importMFNode(self, position, filename) |
|---|

| from controller import GPS |
|---|
| class GPS (Device) : |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValues(self) |

| from controller import Gyro |
|---|
| class Gyro (Device) : |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValues(self) |

| from controller import ImageRef |
| --- |
| class ImageRef : |

| from controller import InertialUnit |
| --- |
| class InertialUnit (Device) : |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getRollPitchYaw(self) |

| from controller import LED |
| --- |
| class LED (Device) : |
| def set(self, state) |
| def get(self) |

| from controller import LightSensor |
| --- |
| class LightSensor (Device) : |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValue(self) |

| from controller import Motion |
| --- |
| class Motion : |
| def __init__(self, fileName) |
| def __del__(self) |
| def isValid(self) |
| def play(self) |
| def stop(self) |
| def setLoop(self, loop) |
| def setReverse(self, reverse) |
| def isOver(self) |
| def getDuration(self) |
| def getTime(self) |
| def setTime(self, time) |

| from controller import Motor |
|---|
| class Motor (Device) : |
| ROTATIONAL, LINEAR |
| def setPosition(self, position) |
| def getTargetPosition(self) |
| def setVelocity(self, vel) |
| def setAcceleration(self, force) |
| def setAvailableForce(self, motor_force) |
| def setAvailableTorque(self, motor_torque) |
| def setControlPID(self, p, i, d) |
| def getMinPosition(self) |
| def getMaxPosition(self) |
| def enableForceFeedback(self, ms) |
| def disableForceFeedback(self) |
| def getForceFeedbackSamplingPeriod(self) |
| def getForceFeedback(self) |
| def setForce(self, torque) |
| def enableTorqueFeedback(self, ms) |
| def disableTorqueFeedback(self) |
| def getTorqueFeedbackSamplingPeriod(self) |
| def getTorqueFeedback(self) |
| def setTorque(self, torque) |
| def getType(self) |

| from controller import Node |
|---|
| class Node : |
| NO_NODE, APPEARANCE, BACKGROUND, BOX, COLOR, CONE, |
| COORDINATE, CYLINDER, DIRECTIONAL_LIGHT, ELEVATION_GRID, |
| EXTRUSION, FOG, GROUP, IMAGE_TEXTURE, INDEXED_FACE_SET, |
| INDEXED_LINE_SET, MATERIAL, POINT_LIGHT, SHAPE, SPHERE, |
| SPOT_LIGHT, SWITCH, TEXTURE_COORDINATE, TEXTURE_TRANSFORM, |
| TRANSFORM, VIEWPOINT, WORLD_INFO, CAPSULE, PLANE, ROBOT, |
| SUPERVISOR, DIFFERENTIAL_WHEELS, SOLID, PHYSICS, CAMERA_ZOOM, |
| CHARGER, DAMPING, CONTACT_PROPERTIES, ACCELEROMETER, BRAKE, |
| CAMERA, COMPASS, CONNECTOR, DISPLAY, DISTANCE_SENSOR, |
| EMITTER, GPS, GYRO, LED, LIGHT_SENSOR, MICROPHONE, MOTOR, PEN, |
| POSITION_SENSOR, RADIO, RECEIVER, SERVO, SPEAKER, |
| TOUCH_SENSOR |
| def getType(self) |
| def getTypeName(self) |
| def getField(self, fieldName) |
| def getPosition(self) |
| def getOrientation(self) |
| def getCenterOfMass(self) |
| def getContactPoint(self, index) |
| def getNumberOfContactPoints(self) |
| def getStaticBalance(self) |
| def resetPhysics(self) |

| from controller import Pen |
|---|
| class Pen (Device) : |
| def write(self, write) |
| def setInkColor(self, color, density) |

| from controller import PositionSensor |
|---|
| class PositionSensor (Device) : |
| ANGULAR, LINEAR |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValue(self) |
| def getType(self) |

| from controller import Receiver |
| --- |
| class Receiver (Device) : |
| CHANNEL_BROADCAST |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getQueueLength(self) |
| def nextPacket(self) |
| def getData(self) |
| def getDataSize(self) |
| def getSignalStrength(self) |
| def getEmitterDirection(self) |
| def setChannel(self, channel) |
| def getChannel(self) |

| from controller import Robot |
| --- |
| class Robot : |
| MODE_SIMULATION, MODE_CROSS_COMPILATION, |
| MODE_REMOTE_CONTROL |
| KEYBOARD_END, KEYBOARD_HOME, KEYBOARD_LEFT, KEYBOARD_UP, |
| KEYBOARD_RIGHT, KEYBOARD_DOWN, KEYBOARD_PAGEUP, |
| KEYBOARD_PAGEDOWN, KEYBOARD_NUMPAD_HOME, |
| KEYBOARD_NUMPAD_LEFT, KEYBOARD_NUMPAD_UP, |
| KEYBOARD_NUMPAD_RIGHT, KEYBOARD_NUMPAD_DOWN, |
| KEYBOARD_NUMPAD_END, KEYBOARD_KEY, KEYBOARD_SHIFT, |
| KEYBOARD_CONTROL, KEYBOARD_ALT |
| def __init__(self) |
| def __del__(self) |
| def step(self, ms) |
| def getAccelerometer(self, name) |
| def getBrake(self, name) |
| def getCamera(self, name) |
| def getCompass(self, name) |
| def getConnector(self, name) |
| def getDisplay(self, name) |
| def getDistanceSensor(self, name) |
| def getEmitter(self, name) |
| def getGPS(self, name) |
| def getGyro(self, name) |
| def getInertialUnit(self, name) |
| def getLED(self, name) |
| def getLightSensor(self, name) |
| def getMotor(self, name) |
| def getPen(self, name) |
| def getPositionSensor(self, name) |
| def getReceiver(self, name) |
| def getServo(self, name) |
| def getTouchSensor(self, name) |
| def getNumberOfDevices(self) |
| def getDeviceByIndex(self, index) |
| def batterySensorEnable(self, ms) |
| def batterySensorDisable(self) |
| def batterySensorGetSamplingPeriod(self) |
| def batterySensorGetValue(self) |

| |
|---|
| def getBasicTimeStep(self) |
| def getMode(self) |
| def getModel(self) |
| def getData(self) |
| def setData(self, data) |
| def getName(self) |
| def getControllerName(self) |
| def getControllerArguments(self) |
| def getProjectPath(self) |
| def getSynchronization(self) |
| def getTime(self) |
| def keyboardEnable(self, ms) |
| def keyboardDisable(self) |
| def keyboardGetKey(self) |
| def getType(self) |

| from controller import Servo |
| --- |
| class Servo (Device) : |
| ROTATIONAL, LINEAR |
| def setPosition(self, position) |
| def getTargetPosition(self) |
| def setVelocity(self, vel) |
| def setAcceleration(self, force) |
| def setMotorForce(self, motor_force) |
| def setControlP(self, p) |
| def getMinPosition(self) |
| def getMaxPosition(self) |
| def enablePosition(self, ms) |
| def disablePosition(self) |
| def getPositionSamplingPeriod(self) |
| def getPosition(self) |
| def enableMotorForceFeedback(self, ms) |
| def disableMotorForceFeedback(self) |
| def getMotorForceFeedbackSamplingPeriod(self) |
| def getMotorForceFeedback(self) |
| def setForce(self, force) |
| def getType(self) |

| from controller import Supervisor |
| --- |
| class Supervisor (Robot) : |
| MOVIE_READY, MOVIE_RECORDING, MOVIE_SAVING, MOVIE_WRITE_ERROR, MOVIE_ENCODING_ERROR, MOVIE_SIMULATION_ERROR |
| def __init__(self) |
| def __del__(self) |
| def exportImage(self, file, quality) |
| def getRoot(self) |
| def getSelf(self) |
| def getFromDef(self, name) |
| def setLabel(self, id, label, xpos, ypos, size, color, transparency) |
| def simulationQuit(self, status) |
| def simulationRevert(self) |
| def simulationResetPhysics(self) |
| def startMovie(self, file, width, height, codec, quality, acceleration, caption) |
| def stopMovie(self) |
| def getMovieStatus(self) |

| from controller import TouchSensor |
|---|
| class TouchSensor (Device) : |
| BUMPER, FORCE, FORCE3D |
| def enable(self, ms) |
| def disable(self) |
| def getSamplingPeriod(self) |
| def getValue(self) |
| def getValues(self) |
| def getType(self) |

## 9.4 Matlab API

The following tables describe the Matlab functions.

| % Accelerometer : |
|---|
| wb_accelerometer_enable(tag, ms) |
| wb_accelerometer_disable(tag) |
| period = wb_accelerometer_get_sampling_period(tag) |
| [x y z] = wb_accelerometer_get_values(tag) |

| % Brake : |
|---|
| wb_brake_set_damping_constant(tag, dampingConstant) |
| type = wb_brake_get_type(tag) |

| % Camera : |
|---|
| WB_CAMERA_COLOR |
| WB_CAMERA_RANGE_FINDER |
| WB_CAMERA_BOTH |
| wb_camera_enable(tag, ms) |
| wb_camera_disable(tag) |
| period = wb_camera_get_sampling_period(tag) |
| fov = wb_camera_get_fov(tag) |
| wb_camera_set_fov(tag, fov) |
| width = wb_camera_get_width(tag) |
| height = wb_camera_get_height(tag) |
| near = wb_camera_get_near(tag) |
| type = wb_camera_get_type(tag) |
| image = wb_camera_get_image(tag) |
| image = wb_camera_get_range_image(tag) |
| max_range = wb_camera_get_max_range(tag) |
| wb_camera_save_image(tag, 'filename', quality) |

| % Compass : |
|---|
| wb_compass_enable(tag, ms) |
| wb_compass_disable(tag) |
| period = wb_compass_get_sampling_period(tag) |
| [x y z] = wb_compass_get_values(tag) |

| % Connector : |
|---|
| wb_connector_enable_presence(tag, ms) |
| wb_connector_disable_presence(tag) |
| presence = wb_connector_get_presence(tag) |
| wb_connector_lock(tag) |
| wb_connector_unlock(tag) |

| % Device : |
|---|
| name = wb_device_get_name(tag) |
| type = wb_device_get_node_type(tag) |

| % DifferentialWheels : |
|---|
| wb_differential_wheels_set_speed(left, right) |
| left = wb_differential_wheels_get_left_speed() |
| right = wb_differential_wheels_get_right_speed() |
| wb_differential_wheels_enable_encoders(ms) |
| wb_differential_wheels_disable_encoders() |
| period = wb_differential_wheels_get_encoders_sampling_period() |
| left = wb_differential_wheels_get_left_encoder() |
| right = wb_differential_wheels_get_right_encoder() |
| wb_differential_wheels_set_encoders(left, right) |
| max = wb_differential_wheels_get_max_speed() |
| unit = wb_differential_wheels_get_speed_unit() |

| % Display : |
|---|
| RGB |
| RGBA |
| ARGB |
| BGRA |
| width = wb_display_get_width(tag) |
| height = wb_display_get_height(tag) |
| wb_display_set_color(tag, [r g b]) |
| wb_display_set_alpha(tag, alpha) |
| wb_display_set_opacity(tag, opacity) |
| wb_display_draw_pixel(tag, x, y) |
| wb_display_draw_line(tag, x1, y1, x2, y2) |
| wb_display_draw_rectangle(tag, x, y, width, height) |
| wb_display_draw_oval(tag, cx, cy, a, b) |
| wb_display_draw_polygon(tag, [x1 x2 ... xn], [y1 y2 ... yn]) |
| wb_display_draw_text(tag, 'txt', x, y) |
| wb_display_fill_rectangle(tag, x, y, width, height) |
| wb_display_fill_oval(tag, cx, cy, a, b) |
| wb_display_fill_polygon(tag, [x1 x2 ... xn], [y1 y2 ... yn]) |
| image = wb_display_image_copy(tag, x, y, width, height) |
| wb_display_image_paste(tag, image, x, y) |
| image = wb_display_image_load(tag, 'filename') |
| image = wb_display_image_new(tag, width, height, data ,format) |
| wb_display_image_save(tag, image, 'filename') |
| wb_display_image_delete(tag, image) |

| % DistanceSensor : |
|---|
| wb_distance_sensor_enable(tag, ms) |
| wb_distance_sensor_disable(tag) |
| period = wb_distance_sensor_get_sampling_period(tag) |
| value = wb_distance_sensor_get_value(tag) |

| % Emitter : |
|---|
| WB_CHANNEL_BROADCAST |
| wb_emitter_send(tag, data) |
| wb_emitter_set_channel(tag, channel) |
| channel = wb_emitter_get_channel(tag) |
| range = wb_emitter_get_range(tag) |
| wb_emitter_set_range(tag, range) |
| size = wb_emitter_get_buffer_size(tag) |

| % GPS : |
|---|
| wb_gps_enable(tag, ms) |
| wb_gps_disable(tag) |
| period = wb_gps_get_sampling_period(tag) |
| [x y z] = wb_gps_get_values(tag) |

| % Gyro : |
|---|
| wb_gyro_enable(tag, ms) |
| wb_gyro_disable(tag) |
| period = wb_gyro_get_sampling_period(tag) |
| [x y z] = wb_gyro_get_values(tag) |

| % InertialUnit : |
|---|
| wb_inertial_unit_enable(tag, ms) |
| wb_inertial_unit_disable(tag) |
| period = wb_inertial_unit_get_sampling_period(tag) |
| [roll pitch yaw] = wb_inertial_unit_get_roll_pitch_yaw(tag) |

| % LED : |
|---|
| wb_led_set(tag, state) |
| state = wb_led_get(tag) |

| % LightSensor : |
|---|
| wb_light_sensor_enable(tag, ms) |
| wb_light_sensor_disable(tag) |
| period = wb_light_sensor_get_sampling_period(tag) |
| value = wb_light_sensor_get_value(tag) |

| % Motion : |
|---|
| motion = wbu_motion_new('filename') |
| wbu_motion_delete(motion) |
| wbu_motion_play(motion) |
| wbu_motion_stop(motion) |
| wbu_motion_set_loop(motion, loop) |
| wbu_motion_set_reverse(motion, reverse) |
| over = wbu_motion_is_over(motion) |
| duration = wbu_motion_get_duration(motion) |
| time = wbu_motion_get_time(motion) |
| wbu_motion_set_time(motion, time) |

| % Motor : |
| --- |
| WB_MOTOR_ROTATIONAL, WB_MOTOR_LINEAR |
| wb_motor_set_position(tag, position) |
| target = wb_motor_get_target_position(tag) |
| wb_motor_set_velocity(tag, vel) |
| wb_motor_set_acceleration(tag, acc) |
| wb_motor_set_available_force(tag, force) |
| wb_motor_set_available_torque(tag, torque) |
| wb_motor_set_control_pid(tag, p, i, d) |
| min = wb_motor_get_min_position(tag) |
| max = wb_motor_get_max_position(tag) |
| wb_motor_enable_force_feedback(tag, ms) |
| wb_motor_disable_force_feedback(tag) |
| period = wb_motor_get_force_feedback_sampling_period(tag) |
| force = wb_motor_get_force_feedback(tag) |
| wb_motor_set_force(tag, force) |
| wb_motor_enable_torque_feedback(tag, ms) |
| wb_motor_disable_torque_feedback(tag) |
| period = wb_motor_get_torque_feedback_sampling_period(tag) |
| force = wb_motor_get_torque_feedback(tag) |
| wb_motor_set_torque(tag, torque) |
| type = wb_motor_get_type(tag) |

| Node: |
|---|
| WB_NODE_NO_NODE, WB_NODE_APPEARANCE, WB_NODE_BACKGROUND, |
| WB_NODE_BOX, WB_NODE_COLOR, WB_NODE_CONE, |
| WB_NODE_COORDINATE, WB_NODE_CYLINDER, |
| WB_NODE_DIRECTIONAL_LIGHT, WB_NODE_ELEVATION_GRID, |
| WB_NODE_EXTRUSION, WB_NODE_FOG, WB_NODE_GROUP, |
| WB_NODE_IMAGE_TEXTURE, WB_NODE_INDEXED_FACE_SET, |
| WB_NODE_INDEXED_LINE_SET, WB_NODE_MATERIAL, |
| WB_NODE_POINT_LIGHT, WB_NODE_SHAPE, WB_NODE_SPHERE, |
| WB_NODE_SPOT_LIGHT, WB_NODE_SWITCH, |
| WB_NODE_TEXTURE_COORDINATE, WB_NODE_TEXTURE_TRANSFORM, |
| WB_NODE_TRANSFORM, WB_NODE_VIEWPOINT, WB_NODE_WORLD_INFO, |
| WB_NODE_CAPSULE, WB_NODE_PLANE, WB_NODE_ROBOT, |
| WB_NODE_SUPERVISOR, WB_NODE_DIFFERENTIAL_WHEELS, |
| WB_NODE_SOLID, WB_NODE_PHYSICS, WB_NODE_CAMERA_ZOOM, |
| WB_NODE_CHARGER, WB_NODE_DAMPING, |
| WB_NODE_CONTACT_PROPERTIES, WB_NODE_ACCELEROMETER, WB_-NODE_BRAKE, |
| WB_NODE_CAMERA, WB_NODE_COMPASS, WB_NODE_CONNECTOR, |
| WB_NODE_DISPLAY, WB_NODE_DISTANCE_SENSOR, WB_NODE_EMITTER, |
| WB_NODE_GPS, WB_NODE_GYRO, WB_NODE_LED, |
| WB_NODE_LIGHT_SENSOR, WB_NODE_MICROPHONE, WB_NODE_MOTOR, |
| WB_NODE_PEN, WB_NODE_POSITION_SENSOR, WB_NODE_RADIO, |
| WB_NODE_RECEIVER, WB_NODE_SERVO, WB_NODE_SPEAKER, |
| WB_NODE_TOUCH_SENSOR |

| % Pen : |
|---|
| wb_pen_write(tag, write) |
| wb_pen_set_ink_color(tag, [r g b], density) |

| % PositionSensor : |
|---|
| WB_ANGULAR, WB_LINEAR |
| wb_position_sensor_enable(tag, ms) |
| wb_position_sensor_disable(tag) |
| period = wb_position_sensor_get_sampling_period(tag) |
| value = wb_position_sensor_get_value(tag) |
| type = wb_position_sensor_get_type(tag) |

| % Receiver : |
|---|
| WB_CHANNEL_BROADCAST |
| wb_receiver_enable(tag, ms) |
| wb_receiver_disable(tag) |
| period = wb_receiver_get_sampling_period(tag) |
| length = wb_receiver_get_queue_length(tag) |
| wb_receiver_next_packet(tag) |
| size = wb_receiver_get_data_size(tag) |
| data = wb_receiver_get_data(tag) |
| strength = wb_receiver_get_signal_strength(tag) |
| [x y z] = wb_receiver_get_emitter_direction(tag) |
| wb_receiver_set_channel(tag, channel) |
| channel = wb_receiver_get_channel(tag) |

| % Robot : |
| --- |
| WB_MODE_SIMULATION, |
| WB_MODE_CROSS_COMPILATION, |
| WB_MODE_REMOTE_CONTROL |
| WB_ROBOT_KEYBOARD_END |
| WB_ROBOT_KEYBOARD_HOME |
| WB_ROBOT_KEYBOARD_LEFT |
| WB_ROBOT_KEYBOARD_UP |
| WB_ROBOT_KEYBOARD_RIGHT |
| WB_ROBOT_KEYBOARD_DOWN |
| WB_ROBOT_KEYBOARD_PAGEUP |
| WB_ROBOT_KEYBOARD_PAGEDOWN |
| WB_ROBOT_KEYBOARD_NUMPAD_HOME |
| WB_ROBOT_KEYBOARD_NUMPAD_LEFT |
| WB_ROBOT_KEYBOARD_NUMPAD_UP |
| WB_ROBOT_KEYBOARD_NUMPAD_RIGHT |
| WB_ROBOT_KEYBOARD_NUMPAD_DOWN |
| WB_ROBOT_KEYBOARD_NUMPAD_END |
| WB_ROBOT_KEYBOARD_KEY |
| WB_ROBOT_KEYBOARD_SHIFT |
| WB_ROBOT_KEYBOARD_CONTROL |
| WB_ROBOT_KEYBOARD_ALT |
| wb_robot_step(ms) |
| tag = wb_robot_get_device('name') |
| size = wb_robot_get_number_of_devices() |
| tag = wb_robot_get_device_by_index(index) |
| wb_robot_battery_sensor_enable(ms) |
| wb_robot_battery_sensor_disable() |
| period = wb_robot_battery_sensor_get_sampling_period() |
| value = wb_robot_battery_sensor_get_value() |
| step = wb_robot_get_basic_time_step() |
| mode = wb_robot_get_mode() |
| model = wb_robot_get_model() |
| data = getData() |
| setData('data') |
| name = wb_robot_get_name() |
| name = wb_robot_get_controller_name() |
| name = wb_robot_get_controller_arguments() |
| path = wb_robot_get_project_path() |

| |
|---|
| sync = wb_robot_get_synchronization() |
| time = wb_robot_get_time() |
| wb_robot_keyboard_enable(ms) |
| wb_robot_keyboard_disable() |
| key = wb_robot_keyboard_get_key() |
| type = wb_robot_get_type() |

| % Servo : |
|---|
| WB_SERVO_ROTATIONAL, WB_SERVO_LINEAR |
| wb_servo_set_position(tag, position) |
| target = wb_servo_get_target_position(tag) |
| wb_servo_set_velocity(tag, vel) |
| wb_servo_set_acceleration(tag, acc) |
| wb_servo_set_motor_force(tag, force) |
| wb_servo_set_control_p(tag, p) |
| min = wb_servo_get_min_position(tag) |
| max = wb_servo_get_max_position(tag) |
| wb_servo_enable_position(tag, ms) |
| wb_servo_disable_position(tag) |
| period = wb_servo_get_position_sampling_period(tag) |
| position = wb_servo_get_position(tag) |
| wb_servo_enable_motor_force_feedback(tag, ms) |
| wb_servo_disable_motor_force_feedback(tag) |
| period = wb_servo_get_motor_force_feedback_sampling_period(tag) |
| force = wb_servo_get_motor_force_feedback(tag) |
| wb_servo_set_force(tag, force) |
| type = wb_servo_get_type(tag) |

| % Supervisor : |
| --- |
| WB_SF_BOOL, WB_SF_INT32, WB_SF_FLOAT, WB_SF_VEC2F, |
| WB_SF_VEC3F, WB_SF_ROTATION, WB_SF_COLOR, WB_SF_STRING, |
| WB_SF_NODE, WB_MF, WB_MF_INT32, WB_MF_FLOAT, B_MF_VEC2F, |
| WB_MF_VEC3F, WB_MF_COLOR, WB_MF_STRING, WB_MF_NODE |
| WB_SUPERVISOR_MOVIE_READY, WB_SUPERVISOR_MOVIE_RECORDING, WB_SUPERVISOR_MOVIE_SAVING, WB_SUPERVISOR_MOVIE_WRITE_ERROR, WB_SUPERVISOR_MOVIE_ENCODING_ERROR, WB_SUPERVISOR_MOVIE_SIMULATION_ERROR |
| wb_supervisor_export_image('filename', quality) |
| node = wb_supervisor_node_get_root() |
| node = wb_supervisor_node_get_self() |
| node = wb_supervisor_node_get_from_def('def') |
| wb_supervisor_set_label(id, 'text', x, y, size, [r g b], transparency) |
| wb_supervisor_simulation_quit(status) |
| wb_supervisor_simulation_revert() |
| wb_supervisor_simulation_reset_physics() |
| wb_supervisor_start_movie('filename', width, height, codec, quality, |
| acceleration, caption) |
| wb_supervisor_stop_movie() |
| status = wb_supervisor_get_movie_status() |
| type = wb_supervisor_field_get_type(field) |
| name = wb_supervisor_field_get_type_name(field) |
| count = wb_supervisor_field_get_count(field) |
| b = wb_supervisor_field_get_sf_bool(field) |
| i = wb_supervisor_field_get_sf_int32(field) |
| f = wb_supervisor_field_get_sf_float(field) |
| [x y] = wb_supervisor_field_get_sf_vec2f(field) |
| [x y z] = wb_supervisor_field_get_sf_vec3f(field) |
| [x y z alpha] = wb_supervisor_field_get_sf_rotation(field) |
| [r g b] = wb_supervisor_field_get_sf_color(field) |
| s = wb_supervisor_field_get_sf_string(field) |
| node = wb_supervisor_field_get_sf_node(field) |
| i = wb_supervisor_field_get_mf_int32(field, index) |
| f = wb_supervisor_field_get_mf_float(field, index) |
| [x y] = wb_supervisor_field_get_mf_vec2f(field, index) |
| [x y z] = wb_supervisor_field_get_mf_vec3f(field, index) |
| [r g b] = wb_supervisor_field_get_mf_color(field, index) |
| s = wb_supervisor_field_get_mf_string(field, index) |
| node = wb_supervisor_field_get_mf_node(field, index) |
| wb_supervisor_field_set_sf_bool(field, value) |
| wb_supervisor_field_set_sf_int32(field, value) |

| |
|---|
| wb_supervisor_field_set_sf_float(field, value) |
| wb_supervisor_field_set_sf_vec2f(field, [x y]) |
| wb_supervisor_field_set_sf_vec3f(field, [x y z]) |
| wb_supervisor_field_set_sf_rotation(field, [x y z alpha]) |
| wb_supervisor_field_set_sf_color(field, [r g b]) |
| wb_supervisor_field_set_sf_string(field, 'value') |
| wb_supervisor_field_set_mf_int32(field, index, value) |
| wb_supervisor_field_set_mf_float(field, index, value) |
| wb_supervisor_field_set_mf_vec2f(field, index, [x y]) |
| wb_supervisor_field_set_mf_vec3f(field, index, [x y z]) |
| wb_supervisor_field_set_mf_color(field, index, [r g b]) |
| wb_supervisor_field_set_mf_string(field, index, 'value') |
| wb_supervisor_field_import_mf_node(field, position, 'filename') |
| type = wb_supervisor_node_get_type(node) |
| name = wb_supervisor_node_get_type_name(node) |
| field = wb_supervisor_node_get_field(node, 'field_name') |
| position = wb_supervisor_node_get_position(node) |
| orientation = wb_supervisor_node_get_orientation(node) |
| com = wb_supervisor_node_get_center_of_mass(node) |
| contact_point = wb_supervisor_node_get_contact_point(node, index) |
| number_of_contacts = wb_supervisor_node_get_number_of_contact_points(index) |
| balance = wb_supervisor_node_get_static_balance(node) |
| wb_supervisor_node_reset_physics(node) |

| % TouchSensor : |
| --- |
| WB_TOUCH_SENSOR_BUMPER, WB_TOUCH_SENSOR_FORCE, |
| WB_TOUCH_SENSOR_FORCE3D |
| wb_touch_sensor_enable(tag, ms) |
| wb_touch_sensor_disable(tag) |
| period = wb_touch_sensor_get_sampling_period(tag) |
| value = wb_touch_sensor_get_value(tag) |
| [x y z] = wb_touch_sensor_get_values(tag) |
| type = wb_touch_sensor_get_type(tag) |