# Webots Reference Manual

release 4.0.27



**`www.cyberbotics.com`**

December 3, 2004

## 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 Jordi Porta, Emanuele Ornella, Yuri Lopez de Meneses, Auke-Jan Ijspeert, Gerald Foliot, Allen Johnson, Michael Kertesz, Aude Billiard, and many others.

Moreover, many thanks are due to Prof. J.-D. Nicoud (LAMI-EPFL) and Dr. F. Mondada for their valuable support.

Finally, thanks to Skye Legon, who proof-read this manual.

# Contents

# Chapter 1

# Introduction

This reference manual contains all the information needed to program robot controllers in Webots. Moreover, it contains reference information on the world description language used in Webots, which is an extension of a subset of the VRML97 3D specification language.

The programming of graphical user interfaces (GUI) is not covered in this manual since Webots 4 can use any GUI library for creating user interfaces for controllers (including GTK+, wxWindows, MFC, etc.). An example of using wxWindows as a GUI for a Webots controller is provided in the `wxgui` controller sample included within the Webots distribution.

# Chapter 2

# Webots Nodes

The nodes listed here are described using the standard VRML description syntax. This information can be also found for each node in the Webots `resources/nodes` directory. A few VRML nodes have been extended to include more fields, like the `WorldInfo` and the `Sphere` node. They are described here as well.

## 2.1 Animation

```
Animation {
  MFFloat          []        key
  MFVec3f          []        translation
  MFRotation       []        rotation
}
```

The `Animation` node should be used only inside the `animation` field of the `Servo` node. Several `Animation` nodes can be inserted in the the `animation` field of a `Servo` node. Each `Animation` node defines an animation for a servo. Please note that such animations do not take into account physics and are mainly intended to perform simple animations rather than physical motions. The `key` field defines a number of time stamps expressed in second at which a specific `translation` and `rotation` is reached by the servo. This is why the sizes of the `translation` and `rotation` arrays should match exactly the size of the `key` array. Please refer to the controller API where the servo functions related to animations are described.

## 2.2 Appearance

```
Appearance {
  SFNode           NULL      material
```

```
  SFNode              NULL      texture
  SFNode              NULL      textureTransform
}
```

The `Appearance` node specifies the visual properties of geometry. 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.

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.

## 2.3 Background

```
Background {
  MFColor             [ 0 0 0 ] skyColor
}
```

The `Background` node defines the background used for rendering the 3D world. The `skyColor` field defines the red green blue components of this color.

## 2.4 Box

```
Box {
  SFVec3f             2 2 2     size
}
```

The `Box` node specifies a rectangular parallelepiped box centred 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 and each component value shall be greater than zero. See illustration on figure 2.1.

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

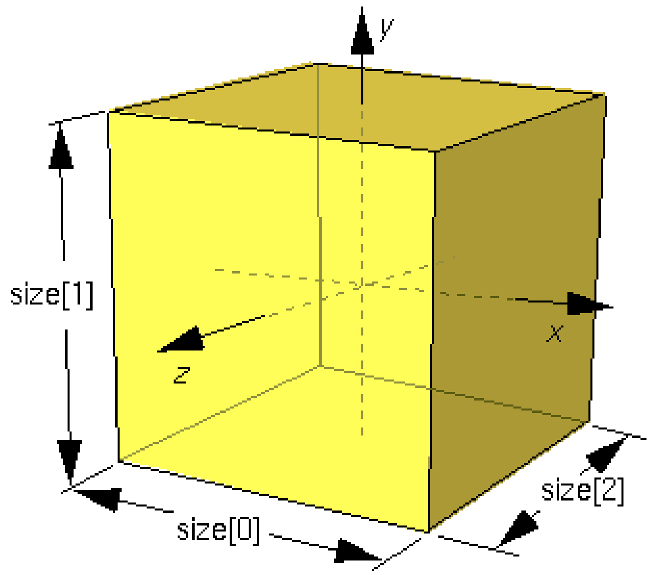


Figure 2.1: The Box node

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.

The `Box` node's geometry requires outside faces only. When viewed from the inside the results are undefined.

## 2.5 Camera

```
Camera {
  scale              1 1 1     SFVec3f
  translation        0 0 0     SFVec3f
  rotation           0 1 0 0   SFRotation
  children           []        MFNode
  name               ""        SFString
  model              ""        SFString
  author             ""        SFString
  constructor        ""        SFString
```

```
  description       ""        SFString
  boundingObject    NULL      SFNode
  physics           NULL      SFNode
  joint             NULL      SFNode
  locked            FALSE     SFBool
  fieldOfView       0.7854    SFFloat
  width             64        SFInt32
  height            64        SFInt32
  type              "color"   SFString
  display           TRUE      SFBool
}
```

The `Camera` node is used to model a robot's on-board camera or range finder. The camera can be either a color camera, a black and white camera, or a range finder device, as defined in the `type` field of the node. It can model a linear camera or range finder (if the `height` field is set to 1). The range finder device rely of the OpenGL depth buffer information. The `Camera` node inherits from the `Solid` node. The fields specific to the `Camera` node are:

- `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”, ”black and white” or ”range-finder”.
- `display`: specify if a camera window should pop up, displaying the image taken by the camera. If such a camera window is used, it should not be iconified or covered by any other window, otherwise the image data might be corrupted. It might be useful to let this field to TRUE for debugging a controller program. However, it is safer to set it to FALSE for extensive experiments.

## 2.6 Charger

```
Charger {
 scale              1 1 1     SFVec3f
 translation        0 0 0     SFVec3f
 rotation           0 1 0 0   SFRotation
 children           []        MFNode
 name               ""        SFString
```

```
 model             ""        SFString
 author            ""        SFString
 constructor       ""        SFString
 description       ""        SFString
 boundingObject    NULL      SFNode
 physics           NULL      SFNode
 joint             NULL      SFNode
 locked            FALSE     SFBool
 battery           []        MFFloat
 radius            0.4       SFFloat
}
```

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 you plug to the power supply. Instead, it is a battery itself: it accumulates energy with time. It could be compared to a solar power plan loading a battery. When the robot comes to get energy, it can't get more than the charger has currently accumulated.

The `Charger` node inherits from the `Solid` node. The fields specific to the `Charger` node are:

- `battery`: this field should contain three values: the current 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 2.2.

## 2.7 Color

```
Color {
  color            []        MFColor
}
```

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 lit geometry. 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 geometric shape will result in the Color node being ignored.

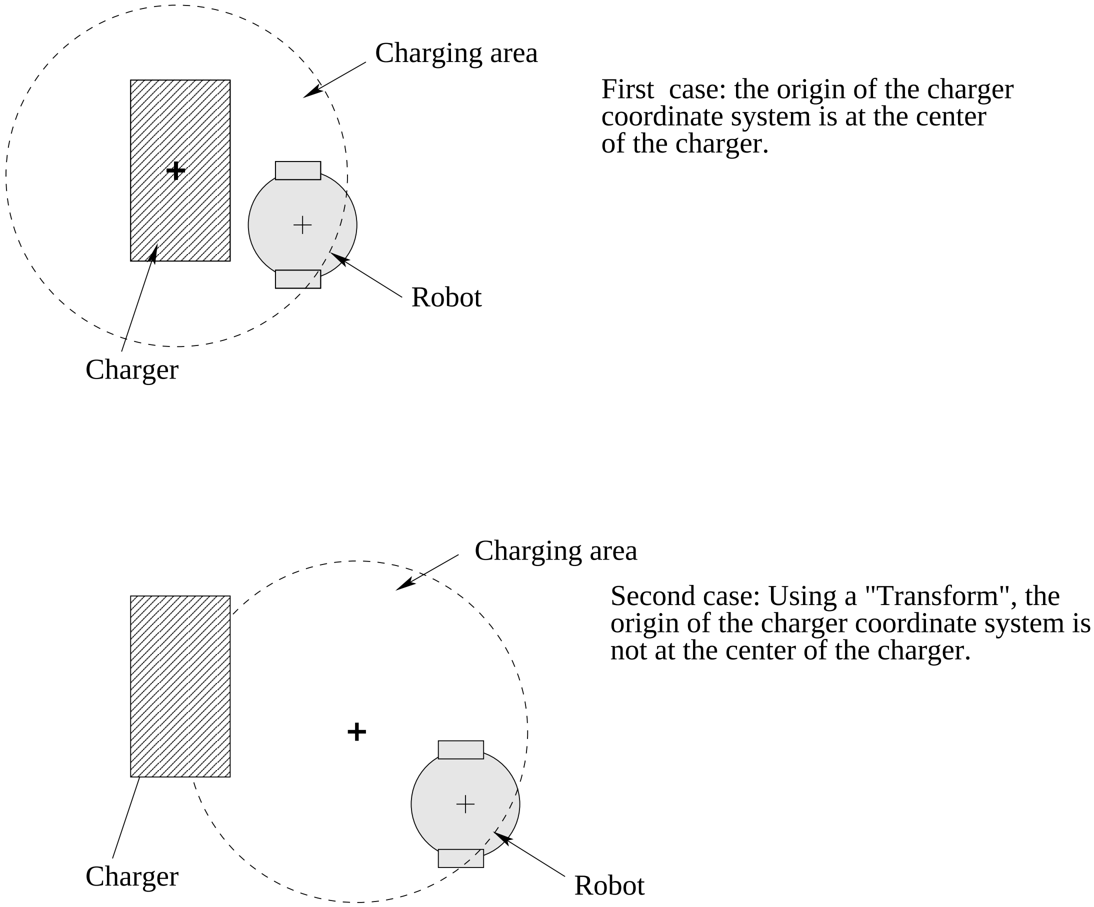


Figure 2.2: The sensitive area of a charger

## 2.8 Cone

```
Cone {
  bottomRadius     1         SFFloat
  height           2         SFFloat
  side             TRUE      SFBool
  bottom           TRUE      SFBool
}
```

The `Cone` node specifies a cone which is centred 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 centre 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. Both `bottomRadius` and `height` shall be greater than zero.

The `side` field specifies whether 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 `Cone` geometry requires outside faces only. When viewed from the inside the results are undefined.

Textures cannot be applied to the `Cone` geometry.

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

## 2.9 Coordinate

```
Coordinate {
  point            []        MFVec3f
}
```

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

## 2.10 Cylinder

```
Cylinder {
  bottom           TRUE      SFBool
  height           2         SFFloat
  radius           1         SFFloat
  side             TRUE      SFBool
  top              TRUE      SFBool
}
```

The `Cylinder` node specifies a cylinder centred at (0,0,0) in the local coordinate system and with a central axis oriented along the local Y-axis. By default, the cylinder is sized at -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. Both radius and height shall be greater than zero. See illustration on figure 2.3.

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.

Cylinders cannot be textured.

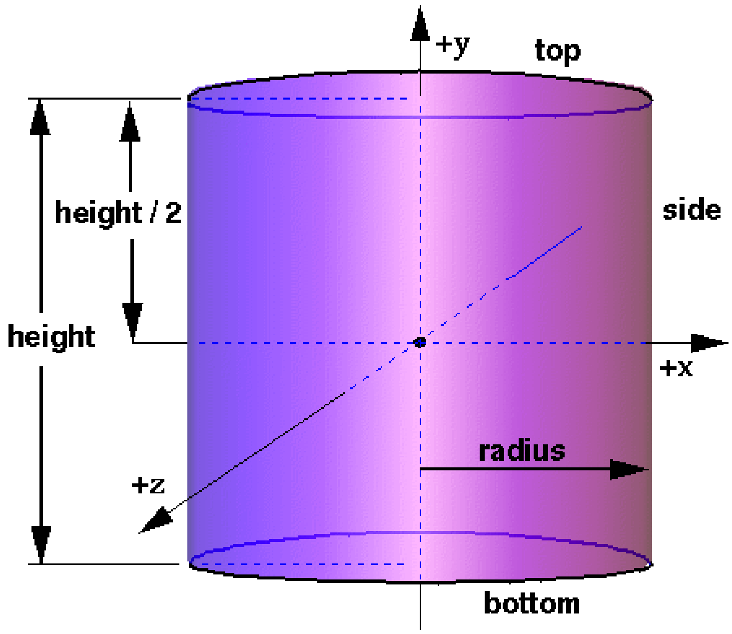


Figure 2.3: The Cylinder node

## 2.11 CustomRobot

```
CustomRobot {
 scale              1 1 1     SFVec3f
 translation        0 0 0     SFVec3f
 rotation           0 1 0 0   SFRotation
 children           []        MFNode
 name               ""        SFString
 model              ""        SFString
 author             ""        SFString
 constructor        ""        SFString
 description        ""        SFString
 boundingObject     NULL      SFNode
 physics            NULL      SFNode
 joint              NULL      SFNode
 locked             FALSE     SFBool
 controller         "void"    SFString
 synchronisation    TRUE      SFBool
 battery            []        MFFloat
 cpuConsumption     0         SFFloat
}
```

## 2.12 DifferentialWheels

```
DifferentialWheels {
 scale              1 1 1     SFVec3f
 translation        0 0 0     SFVec3f
 rotation           0 1 0 0   SFRotation
 children           []        MFNode
 name               ""        SFString
 model              ""        SFString
 author             ""        SFString
 constructor        ""        SFString
 description        ""        SFString
 boundingObject     NULL      SFNode
 physics            NULL      SFNode
 joint              NULL      SFNode
 locked             FALSE     SFBool
 controller         "void"    SFString
 synchronisation    TRUE      SFBool
 battery            []        MFFloat
 cpuConsumption     0         SFFloat
 motorConsumption   0         SFFloat
 axleLength         0.1       SFFloat
 wheelRadius        0.01      SFFloat
 maxSpeed           10        SFFloat
 maxAcceleration    10        SFFloat
 speedUnit          0.1       SFFloat
 slipNoise          0.1       SFFloat
 encoderNoise       -1        SFFloat
}
```

The `DifferentialWheels` node inherits from the `Solid` node. It is used to represent any robot with two-wheel differential steering. The two specific fields which are essential for the simulation are `axleLength` and `wheelRadius`. The value of `axleLength` is the distance (in meters) between the two wheels of the robot, and the value of `wheelRadius` is the radius (in meters) of the wheels.

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

The `DifferentialWheels` node inherits from the `Solid` node. The additional fields are:

- `controller`: name of the program controlling the robot. This program lies in the directory with the same name in the controllers directory; for example, the `void` (or `void.exe`) controller is found in the `webots/controllers/void/` directory . The simulator will use this program to control the robot.

- `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 synchronization.

- `battery`: this field should contain three values: the first one corresponds to the current energy of the robot in Joules(*J*), the second one is the maximum energy the robot can hold in Joules, the third one is the speed of energy recharge in Watts (*[W]=[J]/[s]*). The simulator updates the first value, while the two others remain constant.

- `cpuConsumption`: consumption of the CPU (central processing unit) of the robot in Watts.

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

- `axleLength`: distance between the two wheels in meters.

- `wheelRadius`: radius of the wheels in meters. Both wheels must have the same radius.

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

- `maxAcceleration`: maximum acceleration of the wheels, expressed in *rad/s*$^2$.

- `speedUnit`: defines the unit used in the `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 of +/- 10 percent is added to the command for each simulation step.

- `encoderNoise`: noise added to the incremental encoder. If the value is -1, the encoders are not simulated. If the value is 0, encoders are simulated without noise. Otherwise a noise is added to encoder values. When the robot faces 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 speed is affected by the `slipNoise` parameter. The angular speed is used to compute the amount of 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 multiplicated by the `encoderResolution` (see below) and used to increment the encoder value.

- `encoderResolution`: defines the number of encoder incrementations per radian of the wheel. An `encoderResolution` of *100* will make the encoders increment their value of about *628* each times the wheel makes a complete revolution.

## 2.13 DirectionalLight

```
DirectionalLight {
  ambientIntensity 0                  SFFloat      # [0,1]
  color            1 1 1              SFColor      # [0,1]
  direction        0 0 -1             SFVec3f      # (-,)
  intensity        1                  SFFloat      # [0,1]
  on               TRUE               SFBool
}
```

The `DirectionalLight` node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector. A description of the lighting fields is provided in the VRML97 description of the lighting model.

The `direction` field specifies the direction vector of the illumination emanating from the light source in the local coordinate system. Light is emitted along parallel rays from an infinite distance away. A directional light source illuminates only the objects in its enclosing parent group. The light may illuminate everything within this coordinate system, including all children and descendants of its parent group. The accumulated transformations of the parent nodes affect the light.

`DirectionalLight` nodes do not attenuate with distance.

## 2.14 DistanceSensor

```
DistanceSensor {
 scale             1 1 1              SFVec3f
 translation       0 0 0              SFVec3f
 rotation          0 1 0 0            SFRotation
 children          []                 MFNode
 name              ""                 SFString
 model             ""                 SFString
 author            ""                 SFString
 constructor       ""                 SFString
 description       ""                 SFString
 boundingObject    NULL               SFNode
 physics           NULL               SFNode
 joint             NULL               SFNode
 locked            FALSE              SFBool
 lookupTable       0 0 0,0.1 1000 0   MFVec3f
 type              "infra-red"        SFString
}
```

The `DistanceSensor` node is used to model sonar sensors, infra-red sensors and laser range finders. It uses a ray casting algorithm to detect collision between the sensor ray and `Solid` nodes in the world. The `DistanceSensor` node inherits from the `Solid` node. it includes two additional specific fields:

- `type`: type of sensor: currently only the ”infra-red” type behaves differently from other types (like ”sonar” or ”laser” types). Infra red sensors have a special property: the are color sensitive and will see better light or red obstacles than dark or black ones.
- `lookupTable`: This field is best explained through an example: Let us consider an infra-red sensor. The white noise on the return value is 10 percent. For an obstacle made of a given material and color and for a given ambient light, the response of the sensor is as shown in figure 2.4

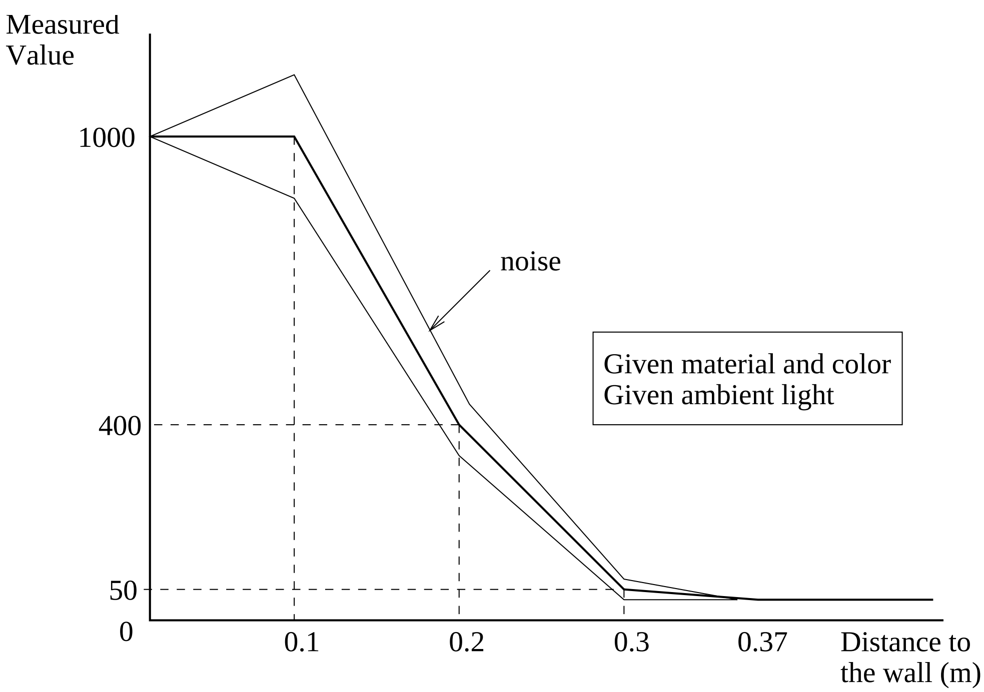


Figure 2.4: Measurements of the light reflected by an obstacle

The values of the `lookupTable` will be:

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

This means that for a distance of 0 meter, 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 10 percent, for a distance value of 0.2 meters, the sensor will return 400 plus or minus 10 percent of noise, etc. For distance values not specified in the lookup table, the simulator will perform a linear interpolation to compute the value returned by the sensor and its associated noise. The first distance value of a lookup table must always be 0.

**Note:** the ray of a sensor can be displayed in the world view by selecting **Display sensor rays** in the **File/Preferences** menu under the **Rendering** panel.

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. This reflection factor is computed as follow: *f = 0.2 + 0.8 * red_level* where *red_level* is the level of red color (`diffuseColor`) of the object hit by the sensor ray. The distance value computed by the simulator is divided by this factor before using the lookup table for computing the output value.

Please note that a primitive support for `DistanceSensor` nodes used for reading the red color level of a textured ground was implemented. This is useful to simulate line following behaviors. This feature is demonstrated in the `rover.wbt` example. In short, the ground texture should lie in a rectangular `IndexedFaceSet` node centered at (0,0,0).

## 2.15 ElevationGrid

```
ElevationGrid {
  color            NULL        SFNode
  height           []          MFFloat
  colorPerVertex   TRUE        SFBool
  xDimension       0           SFInt32
  xSpacing         0.0         SFFloat
  zDimension       0           SFInt32
  zSpacing         0.0         SFFloat
}
```

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 a surface above each point of the grid.

The `xDimension` and `zDimension` fields indicate the number of elements of 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 rectangles 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 x i
P[i,j].y = height[ i + j x xDimension]
P[i,j].z = zSpacing x 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
```

The `color` field specifies per-vertex or per-quadrilateral colours 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

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.

## 2.16 Emitter

```
Emitter {
 scale              1 1 1       SFVec3f
 translation        0 0 0       SFVec3f
 rotation           0 1 0 0     SFRotation
 children           []          MFNode
 name               ""          SFString
 model              ""          SFString
 author             ""          SFString
 constructor        ""          SFString
 description        ""          SFString
 boundingObject     NULL        SFNode
 physics            NULL        SFNode
 joint              NULL        SFNode
```

```
 locked            FALSE       SFBool
 type              "infra-red" SFString
 range             0.5         SFFloat
 channel           0           SFInt32
 baudRate          9600        SFInt32
 byteSize          8           SFInt32
 bufferSize        1024        SFInt32
}
```

The `Emitter` node is used to model an infra-red or radio emitter on-board a robot. You must insert the `Emitter` node into the list of children of the robot. Please note that an emitter can only emit data but it cannot receive any information. In order to enable a bi-directional communication system, a robot needs both an `Emitter` and a `Receiver` node.

The `Emitter` node inherits from the `Solid` node. The fields specific to the `Emitter` node are:

- `type`: type of the emitted signals: ”infra-red” or ”radio”.
- `range`: radius of the emission area in meters. The origin of the coordinate system of a receiver must be in this area to allow this receiver to pick up the signal.
- `channel`: channel of emission. The value is an identification number for an infra-red emitter or a frequency for a radio emitter. The receiver must use the same channel to receive the emitted signals. It can be any positive integer value.
- `baudRate`: the baudRate value is the communication speed expressed in number of bits per second.
- `byteSize`: the byteSize value is the number of bits used to represent one byte (usually 8, but may be more depending on whether control bits are used).
- `bufferSize`: the buffer is a memory area, its size is specified in bytes. The size of the data to be emitted cannot exceed the buffer size, otherwise data is lost. When the emitter emits the data, it flushes the buffer.

## 2.17 Extrusion

```
Extrusion {
  beginCap          TRUE       SFBool
  convex            TRUE       SFBool
  crossSection [ 1 1, 1 -1, -1 -1, -1 1, 1 1] MFVec2f
  endCap             TRUE      SFBool
  spine      [ 0 0 0, 0 1 0 ] MFVec3f # can only change the 1 value
  creaseAngle        0         SFFloat
}
```

The `Extrusion` node specifies geometric shapes based on a two dimensional cross-section extruded along a three dimensional spine in the local coordinate system.

An `Extrusion` node is defined by:

- a 2D `crossSection` piecewise linear curve (described as a series of connected vertices)
- a 3D spine (also described as a series of two connected vertices). Note that the spine is limited to a vector along the Y-axis.

`Extrusion` has three parts: the sides, the `beginCap` (the surface at the initial end of the spine) and the `endCap` (the surface at the final end of the spine). The caps have an associated `SFBool` field that indicates whether each exists (`TRUE`) or doesn't exist (`FALSE`).

When the `beginCap` or `endCap` fields are specified as `TRUE`, planar cap surfaces will be generated regardless of whether the `crossSection` is a closed curve. If `crossSection` is not a closed curve, the caps are generated by adding a final point to `crossSection` that is equal to the initial point. If a field value is `FALSE`, the corresponding cap is not generated.

## 2.18 Fog

```
Fog {
  color            1 1 1       SFColor
  fogType          "LINEAR"    SFString
  visibilityRange  0           SFFloat
}
```

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 outside the `visibilityRange` from the camera are drawn with a constant color of `color`. 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,” an square exponential increase in blending is used, resulting in an even more natural fog appearance (see OpenGL documentation for more details about fog rendering).

## 2.19 GPS

```
GPS {
 scale              1 1 1       SFVec3f
 translation        0 0 0       SFVec3f
 rotation           0 1 0 0     SFRotation
 children           []          MFNode
 name               ""          SFString
 model              ""          SFString
 author             ""          SFString
 constructor        ""          SFString
 description        ""          SFString
 boundingObject     NULL        SFNode
 physics            NULL        SFNode
 joint              NULL        SFNode
 locked             FALSE       SFBool
 type               "satellite" SFString
 resolution         0.001       SFFloat
}
```

The `GPS` node is used to model a Global Positioning Sensor (GPS) which can obtain information about its absolute position and orientation from the controller program. The `GPS` node inherits from the `Solid` node. It includes two additional specific fields:

- `type`: This field defines the type of GPS technology used like ”satellite” or ”laser”, currently ignored.
- `resolution`: This field defines the precision of the GPS, that is the maximal error (expressed in meter) on the absolute position.

## 2.20 Gripper

```
Gripper {
 scale              1 1 1      SFVec3f
 translation        0 0 0      SFVec3f
 rotation           0 1 0 0    SFRotation
 children           []         MFNode
 name               ""         SFString
 model              ""         SFString
 author             ""         SFString
 constructor        ""         SFString
 description        ""         SFString
 boundingObject     NULL       SFNode
```

```
 physics            NULL       SFNode
 joint              NULL       SFNode
 locked             FALSE      SFBool
 position           0          SFFloat
}
```

## 2.21 Group

```
Group {
  children        []          SFNode
}
```

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.

## 2.22 ImageTexture

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

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, 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 in figure 2.5.

The texture is read from the file specified by the `url` field. The file can be specified with an absolute or relative path. Supported image formats include JPEG and PNG. The image use must be square. Moreover the image size must be $2^n * 2^n$ pixels (for example 8x8, 16x16, 32x32, 64x64, 128x128 pixels).

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.

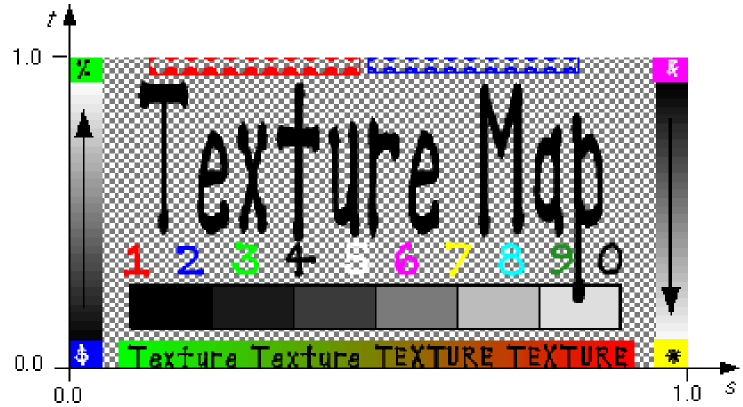


Figure 2.5: Texture map coordinate system

## 2.23 IndexedFaceSet

```
IndexedFaceSet {
 coord              NULL       SFNode
 texCoord           NULL       SFNode
 ccw                TRUE       SFBool
 convex             TRUE       SFBool
 coordIndex         []         MFInt32 # [-1,)
 texCoordIndex      []         MFInt32 # [-1,)
 creaseAngle        0          SFFloat
}
```

The `IndexedFaceSet` node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the `coord` field. 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.

The `IndexedFaceSet` node is specified in the local coordinate system and is affected by the transformations of its ancestors.

Descriptions of the `coord`, `normal`, and `texCoord` fields are provided in the `Coordinate`, `Normal`, and `TextureCoordinate` nodes, respectively.

## 2.24 IndexedLineSet

```
IndexedLineSet {
 coord              NULL       SFNode
 coordIndex         []         MFInt32 # [-1,)
}
```

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 th transformations of its ancestors.

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

Lines are not lit, are not texture-mapped, and do not participate in collision detection.

## 2.25 Joint

```
Joint {
 translation        0 0 0      SFVec3f
}
```

The `Joint` node is used to defined an articulation between two `Solid` nodes. Currently, `Joint` nodes are mostly limited to define an offset value for the location of a joint in a `Servo` node. However, a `Joint` node has to be created for any `Servo` in physics based simulation. It is also mandatory to define a `Joint` node for each `Solid` node representing a wheel in a physics simulation of a `DifferentialWheels` robot.

The `translation` field defines an offset for moving the location of the joint relatively to the origin of its parent node. The parent node should be a solid node, that is a node inheriting from the `Solid` node like `Servo` or `Solid` itself). This is especially useful with `Servo` nodes when you want that a servo rotates around a different point than its local coordinate system.

## 2.26 HyperGate

```
HyperGate {
```

```
 scale              1 1 1      SFVec3f
 translation        0 0 0      SFVec3f
 rotation           0 1 0 0    SFRotation
 children           []         MFNode
 name               ""         SFString
 model              ""         SFString
 author             ""         SFString
 constructor        ""         SFString
 description        ""         SFString
 boundingObject     NULL       SFNode
 physics            NULL       SFNode
 joint              NULL       SFNode
 locked             FALSE      SFBool
 url                ""         SFString
 radius             0.1        SFFloat
 height             0.1        SFFloat
 maxFileSize        65536      SFInt32
}
```

A hypergate is defined as a cylindrical area in the world. When a robot (more precisely the origin of the robot coordinate system) enters it, it disappears and gets transferred to another world specified in the `HyperGate` node.

The `HyperGate` node inherits from the `Solid` node. The fields specific to the `HyperGate` node are:

- `url`: destination URL of the form `"wtp://host.domain.com/file#name"`.
- `radius`: radius of the transfer cylinder.
- `height`: height of the transfer cylinder.
- `maxFileSize`: maximum file size for the `Robot` node accepted by the hypergate.

  For example, an hypergate can look like an arch with the transfer cylinder lying inside the arch. See figure 2.6.

## 2.27 LED

```
LED {
 scale             1 1 1        SFVec3f
 translation       0 0 0        SFVec3f
 rotation          0 1 0 0      SFRotation
 children          []           MFNode
 name              ""           SFString
```

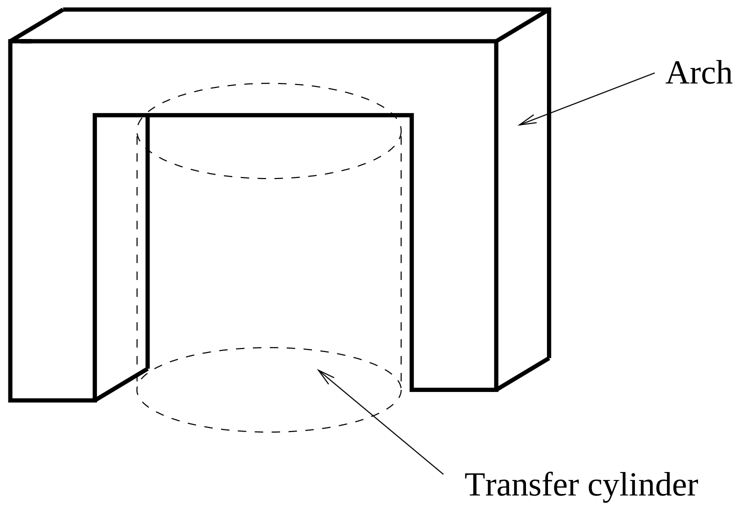


Figure 2.6: An example of an Hypergate

```
 model             ""           SFString
 author            ""           SFString
 constructor       ""           SFString
 description       ""           SFString
 boundingObject    NULL         SFNode
 physics           NULL         SFNode
 joint             NULL         SFNode
 locked            FALSE        SFBool
 color             [ 1 0 0 ]    MFColor # available colors for the LED
}
```

The `LED` node is used to model a light emitting diode (LED). The light produced by a LED can be used for debugging or information purposes. The shape of the light emitting part of the LED device is defined as a `Solid` node in the `children` list of the `LED` node. This `Solid` node should have a `name` field set to "lamp" to be recognized as the light emitting part of the LED device. Upon activation, the `emissiveColor` field of the first `Material` node in this `Solid` node will be changed to the color specified by the `color` field of the `LED` node.

If such a "lamp" Solid node doesn't exist, the color change applies to the first `Shape` node in the `children` list which has a `Material` node defined.

The `LED` node inherits from the `Solid` node. It includes an additional specific field:

- `color`: This defines the colors of the LED device. When off, a led is always black. However, when on it can have deferent colors as specified by the LED programming interface. By default, the `color` defines only one color,which is red, but you can change this and even add extra colors that could be selected from the LED programming interface.

## 2.28 LightSensor

```
LightSensor {
 scale             1 1 1              SFVec3f
 translation       0 0 0              SFVec3f
 rotation          0 1 0 0            SFRotation
 children          []                 MFNode
 name              ""                 SFString
 model             ""                 SFString
 author            ""                 SFString
 constructor       ""                 SFString
 description       ""                 SFString
 boundingObject    NULL               SFNode
 physics           NULL               SFNode
 joint             NULL               SFNode
 locked            FALSE              SFBool
 lookupTable       0 0 0,0.1 1000 0   MFVec3f
}
```

The `LightSensor` node is used to model a phototransistor-like sensor which measure the level of ambient light in a given direction. The light level measured by the `LightSensor` node is computed from each `PointLight` node in the scene, taking into account the distance between the sensor and the light, the orientation of the sensor relatively to the light, the intensity of the light (computed from its ambient intensity, intensity and color). The `LightSensor` node inherits from the `Solid` node. It includes an additional specific field:

- `lookupTable`: similar to the one of the `DistanceSensor` node except that the distance values (first column) are replaced by intensity values. This intensity value results from the sum of intensity values computed for each `PointLight` as follow:

  *distance* is the distance between the `LightSensor` and the `PointLight`.

  *dot* is the dot product between the normalized sensor direction and the normalized vector defined by the `LightSensor` location and the `PointLight` location.

  *att = attenuation.x + attenuation.y * distance + attenuation.z * distance * distance*

  *cf = color.red * color.green * color.blue*

  *intensity_value = (ambientIntensity + intensity) * cf * dot / att*

## 2.29 Material

```
Material {
 ambientIntensity  0.2                SFFloat # [0,1]
```

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

The `Material` node specifies surface material properties for associated geometry nodes and is used by the VRML97 lighting equations during rendering.

All of the fields in the Material node range from 0.0 to 1.0.

The fields in the `Material` node determine how light reflects off an object to create color:

- The `ambientIntensity` field specifies how much ambient light from light sources this surface shall reflect. Ambient light is omnidirectional and depends only on the number of light sources, not their positions with respect to the surface. Ambient colour 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 ”clear” an object is, with 1.0 being completely transparent, and 0.0 completely opaque. If you set the transparency to a positive value, please note that no dynamic alpha sorting is performed in Webots, so that you need to place transparent or semi-transparent objects at the bottom of the scene tree, so that they are rendered at the end and do not interfer with other objects.

## 2.30 Pen

```
Pen {
 scale              1 1 1              SFVec3f
 translation        0 0 0              SFVec3f
```

```
 rotation          0 1 0 0           SFRotation
 children          []                MFNode
 name              ""                SFString
 model             ""                SFString
 author            ""                SFString
 constructor       ""                SFString
 description       ""                SFString
 boundingObject    NULL              SFNode
 physics           NULL              SFNode
 joint             NULL              SFNode
 locked            FALSE             SFBool
 inkColor          0 0 0             SFColor
 inkDensity        0.5               SFFloat
 leadSize          0.002             SFFloat
 write             TRUE              SFBool
}
```

The Pen node is used to model a pen attached to a mobile robot, typically to write down the trajectory of the robot. In order to work, a pen needs to lie over a textured ground. Such a textured ground should be made up of a Solid node containing a Shape with a textured Material in its Appearance. Moreover, its geometry should be a rectangle IndexedFaceSet lying at y=0. An example of appropriate textured ground used with a robot equipped with a pen is given in the hemisson_pen.wbt example world.

**Note:** The drawings performed by a pen can be seen by distance sensors looking towards the ground. Hence, it is possible to implement a robotics experiment where a robot draws a line on the floor witha pen and a second robot performs a line following behavior with the line just drawn by the first robot.

The Pen node inherits from the Solid node. It includes four additional specific fields:

- inkColor: define the color of the ink of the pen. This parameter can be changed from the pen API, using the pen_set_ink_color function.
- inkDensity: define the density of the color of the ink. This parameter can also be changed from the pen API, using the pen_set_ink_color function.
- leadSize: define the size of the lead of the pen. This allows the robot to write a track with a more or less thick width.
- write: this boolean field allows the robot to enable of disable writing for the pen. It is also switchable from the pen API, using the pen_write function.

## 2.31 Physics

```
Physics {
 orientation        0 1 0 0 SFRotation # orientation of inertia matrix
 density            1000    SFFloat    # (kg/m^3) if -1 use mass
 mass               -1      SFFloat    # (kg) ignored if density!=-1
 bounce             0.5     SFFloat    # range between 0 and 1
 bounceVelocity     0.01    SFFloat    # (m/s)
 coulombFriction    0       SFFloat    # ODE Coulomb friction coefficient
 forceDependentSlip 0       SFFloat    # ODE force dependent slip
 inertiaMatrix      []      MFFloat    # 9 float values: inertia matrix
}
```

The `Physics` allows you to define a number of physics parameters to be used by the physics simulation engine. It is useful for example in robot soccer systems, where a robot, or several robots can push a ball which rolls and bounces against the walls. An example of using the `Physics` node is provided in the `soccer.wbt` world. The `Physics` node is also useful when simulating legged robots to define mass repartition and friction parameters, thus allowing the physics engine to simulate a legged robot accurately, making it fall down when necessary. Reading the ODE (Open Dynamics Engine) documentation will help you better understand the parameters of the `Physics` node and their results on the physics simulation.

If the `inertiaMatrix` field is defined, the `orientation` field defines the local orientation of the inertia matrix.

Either the `mass` or `density` parameter can be used to define the total mass of the solid. If the `density` parameter is set different from -1, then it is used regardless of the `mass` parameter to compute the mass of the solid object. Otherwise, the `mass` parameter, which should be set to a positive value, is used.

The `bounce` parameter defines the bouncyness of a solid. This restitution parameter is a floating point value ranging from 0 to 1. 0 means that the surfaces are not bouncy at all, 1 is maximum bouncyness. When two solids hit each other, the resulting bouncyness is the average of the bounce parameter of each solid. If a solid has no `Physics` node, and hence no `bounce` parameter defined, the `bounce` parameter of the other solid is used. The same principle also applies for to `bounceVelocity`, `staticFriction` and `kineticFriction` parameters.

The `bounceVelocity` parameter defines the minimum incoming velocity necessary for bounce. Incoming velocities below this will effectively have a bounce parameter of 0.

The `coulombFriction` parameter defines the friction parameter which applies to the solid regardless of its velocity. Friction approximation in ODE relies on the Coulomb friction model and is documented in the ODE documentation.

The `forceDependentSlip` parameter 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.

The `inertiaMatrix` parameter defines the inertia matrix as specified by ODE. If this parameter is empty or contains less or more than 9 floating point values, it is ignored. Moreover, if the `mass` parameter is -1, the `inertiaMatrix` parameter is ignored. If it contains exactly 9 floating point values, and if the `mass` parameter is different from -1, then it is used as follow: the 9 parameters are the same as the ones used by the `dMassSetParameters` ODE function. The parameters given in the `inertiaMatrix` are: cgx, cgy, cgz, I11, I22, I33, I12, I13, I23, where (cgx,cgy,cgz) is the center of gravity position in the body frame. The Ixx values are the elements of the inertia matrix, expressed in kg.m$^2$:

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

## 2.32 PointLight

```
PointLight {
 ambientIntensity  0          SFFloat # [0,1]
 attenuation       1 0 0      SFVec3f # [0,)
 color             1 1 1      SFColor # [0,1]
 intensity         1          SFFloat # [0,1]
 location          0 0 0      SFVec3f # (-,)
 on                TRUE       SFBool
}
```

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; that is, it is omnidirectional. `PointLight` nodes are specified in the local coordinate system and are affected by ancestor transformations. Hence it is possible to embed a `PointLight` onboard a mobile robot to create lights moving with the robot.

A `PointLight` node illuminates geometry from its `location`. The `location` is affected by ancestors' transformations.

`PointLight` node's illumination falls off with distance as specified by three `attenuation` coefficients. The attenuation factor is 1/max(attenuation[0] + attenuation[1] x r + attenuation[2] x r2, 1), where r is the distance from 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 shall be greater than or equal to zero.

## 2.33 Receiver

```
Receiver {
 scale              1 1 1       SFVec3f
 translation        0 0 0       SFVec3f
 rotation           0 1 0 0     SFRotation
 children           []          MFNode
 name               ""          SFString
 model              ""          SFString
 author             ""          SFString
 constructor        ""          SFString
 description        ""          SFString
 boundingObject     NULL        SFNode
 physics            NULL        SFNode
 joint              NULL        SFNode
 locked             FALSE       SFBool
 type               "infra-red" SFString
 channel            0           SFInt32
 baudRate           9600        SFInt32
 byteSize           8           SFInt32
 bufferSize         1024        SFInt32
}
```

The `Receiver` node is used to model an infra-red or radio receiver. A receiver, just like an emitter, is usually on-board a robot. Please note that a receiver can only receive data but it cannot emit any information. In order to enable a bi-directional communication system, a robot needs both an `Emitter` and a `Receiver` node.

The fields and values of the `Receiver` node are nearly the same as those of the `Emitter` node. As the `Emitter` node, the `Receiver` node inherits from the `Solid` node. The fields specific to the `Receiver` node are:

- `type`: type of the received signals: ”infra-red” or ”radio”.
- `channel`: channel of reception. The value is an identification number for an infra-red receiver or a frequency for a radio receiver. The emitter must use the same channel to detect the emitted signals.
- `baudRate`: the baudRate value is the communication speed expressed in bits per second. It must be the same as the speed of the emitter.
- `byteSize`: the byteSize value is the number of bits used to represent one byte (usually 8, but may be more if control bits are used). It must be the same size as the emitter buffer.
- `bufferSize`: the buffer is a memory area, its size is specified in bytes. The size of the received data can’t exceed the buffer size, otherwise data is lost. When the receiver reads

the data, it flushes the buffer. If the old data has not been read when the new data is received, the former is lost.

## 2.34 Servo

```
Servo {
 scale              1 1 1      SFVec3f
 translation        0 0 0      SFVec3f
 rotation           0 1 0 0    SFRotation
 children           []         MFNode
 name               ""         SFString
 model              ""         SFString
 author             ""         SFString
 constructor        ""         SFString
 description        ""         SFString
 boundingObject     NULL       SFNode
 physics            NULL       SFNode
 joint              NULL       SFNode
 locked             FALSE      SFBool
 maxVelocity        10         SFFloat
 maxForce           10         SFFloat
 controlP           10         SFFloat
 acceleration       -1         SFFloat
 maxPosition        0          SFFloat # should be positive or 0
 minPosition        0          SFFloat # should be negative or 0
 animation          []         MFNode
}
```

The `Servo` node models a servo motor. It inherits from the `Solid` node.

A servo can be controlled in position only through the servo API, using the `servo_set_position` function.

However, it is possible to control it in torque and in velocity. To control a servo in torque, set `maxForce` to the desired target torque with `servo_set_force`, set a big enough target position with `servo_set_position` and a big enough `maxVelocity` with `servo_set_velocity`.

Similarly, to control a servo in velocity, set the maximum torque, set the desired velocity as `maxVelocity` and a big enough target position.

Please note that the `maxForce` and `maxVelocity` parameters should always be positive.

The `controlP` parameter controls the proportional PID parameter used to compute the target speed from the requested position. A too small value yields to a long time needed to reach the target position while a too big value yields to unstabilities reaching the target position.

The `acceleration` field defines the acceleration used by the position controller. This acceleration should be expressed in rad/$s^2$ and should be set to a value smaller than `maxForce` to achieve smooth and slow movements. Please note that this parameter doesn't specify the actual force of the servo, but rather the acceleration used by the position controller. Hence, to achieve a slow and smooth movement, it is better to set a small value to the `acceleration` field rather than to the `maxForce` field. A small `maxForce` parameter may indeed result in a servo unable to move or to maintain a desired position because of the weight it has to support. If the `acceleration` field is set to -1 (default value), then it is ignored and the maximum force is used to achieve the target position.

The `minPosition` and `maxPosition` fields define the limits of the `Servo` position expressed in radians. The initial position of the servo should always be 0 and this position should lie between `minPosition` and `maxPosition`. Hence `minPosition` should be negative or 0 and `maxPosition` should be positive or 0. If you don't want to set limits for a `Servo` node, set `minPosition` and `maxPosition` both to the same value, 0 for example. This is the default value.

The `animation` field refers to an `Animation` node used for animating the servo in a non-realistic simulation.

## 2.35 Solid

```
Solid {
 scale              1 1 1     SFVec3f
 translation        0 0 0     SFVec3f
 rotation           0 1 0 0   SFRotation
 children           []        MFNode
 name               ""        SFString
 model              ""        SFString
 author             ""        SFString
 constructor        ""        SFString
 description        ""        SFString
 boundingObject     NULL      SFNode
 physics            NULL      SFNode
 joint              NULL      SFNode
 locked             FALSE     SFBool
}
```

A solid is a group of shapes that you can drag and drop in the world, using the mouse. Moreover, the sensors of the robots and the collision detector of the simulator are able to detect solids. The `Solid` node represents this group of shapes in the scene tree.

A description of the fields of the `Solid` node is given below.

The `Solid` node inherits from the VRML `Transform` node. However, the `scale` field of a `Solid` node should always be set to `1 1 1` to avoid problems with the bounding objects that ignore the `scale` parameter. The additional fields are:

- `name`: individual name of the solid (e.g.: ”my blue chair”).
- `model`: generic name of the solid (e.g.: ”chair”).
- `author`: name of the author of the simulation model of the solid.
- `constructor`: name of the company or individual who made the real solid.
- `description`: short description (1 line) of the solid.
- `boundingObject`: the `boundingObject` of a `Solid` should contain either: (1) a `Box` node, (2) a `Cylinder` node (a flat end cylinder, not a capped cylinder), (3) a `Sphere` node, (4) an `IndexedFaceSet` node, (5) a `Shape` node containing one of the above nodes as a `geometry`, (6) a `Transform` node with a single `children` node being one of above nodes and the `scale` field set to 1 1 1, or (7) a `Group` node with several `children`, each being one of the above mentioned nodes.

  In the case of an `IndexedFaceSet`, two different options are possible: The first option is an indexed face set with a single quadrilateral face which defines a plane. This plane is considered as infinite by the collision detection engine. This option should be used to model a flat floor as in `boebot.wbt`. The second option is an indexed face set of triangles defining a triangle mesh (or trimesh). Such indexed face sets can be easily exported from most 3D modelling software after performing a conversion to a triangle mesh. This option should be used to model rough terrain as in `aibo_ers210_rough.wbt` or to model complex 3D objects.

  The bounding object defines the shape used for collision detection and to automatically compute the inertia matrix of a `Solid` from its `physics` field. Please note however that the center of mass of the `Solid` node always remains the same as the origin of the node (defined by the `translation` and `rotation` fields) regardless of what is defined in the bounding object. If this field is left to NULL, no collision detection and no physics computation is performed.
- `physics`: this field is used when it is necessary to model a minimum of physics for a `Solid` object. In this case, it contains a `Physics` object which defines a number of physical properties for the solid. This is especially useful when implementing a robot pushing an object like a ball. In this case, both the robot and the ball should have a `Physics` node in their `physics` field.
- `joint`: if set to a `Joint` node, implement a rotational joint for a `Servo` node.
- `locked`: if `TRUE`, the solid object cannot be moved using the mouse. This is useful to prevent moving an object by error.

## 2.36 Shape

```
Shape {
 appearance         NULL      SFNode
 geometry           NULL      SFNode
}
```

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. The specified geometry node is rendered with the specified appearance nodes applied.

## 2.37 Sphere

```
Sphere {
 radius             1         SFFloat
 subdivision        1         SFInt32
}
```

The `Sphere` node specifies a sphere centred at (0,0,0) in the local coordinate system. The `radius` field specifies the radius of the sphere and shall be greater than zero. See illustration on figure 2.7.

The `Sphere` node's geometry requires outside faces only. When viewed from the inside the results are undefined.

The `Sphere` node cannot be textured.

The VRML97 `Sphere` node was extended to include a `subdivision` field which controls the shape of the rendered sphere. Spheres are rendered as icosaedrons 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, which makes 80 faces. With a subdivision parameter 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 entering in a very long rendering process. A value of 10 will turn the sphere appearance into a black and white soccer ball.

## 2.38 Supervisor

```
Supervisor {
 scale              1 1 1     SFVec3f
```

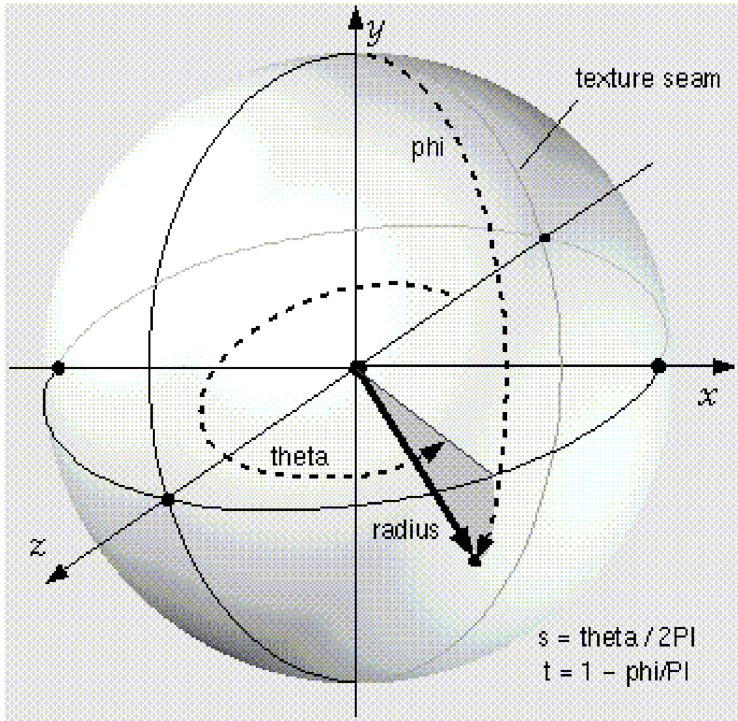


Figure 2.7: The Sphere node

```
 translation        0 0 0     SFVec3f
 rotation           0 1 0 0   SFRotation
 children           []        MFNode
 name               ""        SFString
 model              ""        SFString
 author             ""        SFString
 constructor        ""        SFString
 description        ""        SFString
 boundingObject     NULL      SFNode
 physics            NULL      SFNode
 joint              NULL      SFNode
 locked             FALSE     SFBool
 controller         "void"    SFString
 synchronisation    TRUE      SFBool
 battery            []        MFFloat
 cpuConsumption     0         SFFloat
}
```

A supervisor is a program which controls a world and its robots. For convenience it is represented as a robot without any wheels, driven by a controller with extended capabilities which supervises the whole world. A world cannot have more than one supervisor.

The `Supervisor` node inherits from the `Solid` node. Its other fields include some of the `DifferentialWheels` node fields:

- `controller`
- `synchronization`
- `battery`: usually meaningless for a `Supervisor` node.
- `cpuConsumption`: usually meaningless for a `Supervisor` node.

## 2.39 TextureCoordinate

```
TextureCoordinate {
 point              []                MFVec2f
}
```

The `TextureCoordinate` node specifies a set of 2D texture coordinates used by vertex-based geometry nodes (e.g., `IndexedFaceSet` and `ElevationGrid`) to map textures to vertices. Textures are two dimensional color functions that, given an (s,t) coordinate, return a color value colour(s,t). Texture map values (`ImageTexture`) range from [0.0,1.0] along the S-axis and T-axis. 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.

## 2.40 TextureTransform

```
TextureTransform {
 center             0 0               SFVec2f
 rotation           0                 SFFloat
 scale              1 1               SFVec2f
 translation        0 0               SFVec2f
}
```

The `TextureTransform` node defines a 2D transformation that is applied to texture coordinates. This node affects the way textures coordinates are applied to the geometric surface. The transformation consists of (in order):

- a translation;
- a rotation about the centre point;
- a non-uniform scale about the centre point.

These parameters support changes to the size, orientation, and position of textures on shapes. Note that these operations appear reversed when viewed on the surface of geometry. For example, a `scale` value of (2 2) will scale the texture coordinates and have 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 +.5 units along the S-axis and has 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 geometry.

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 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 scale has been applied. A positive rotation value makes the texture coordinates rotate counterclockwise about the centre, thereby rotating the appearance of the texture itself clockwise. The `translation` field specifies a translation of the texture coordinates.

## 2.41 TouchSensor

```
TouchSensor {
 scale              1 1 1               SFVec3f
 translation        0 0 0               SFVec3f
 rotation           0 1 0 0             SFRotation
 children           []                  MFNode
 name               ""                  SFString
 model              ""                  SFString
 author             ""                  SFString
 constructor        ""                  SFString
 description        ""                  SFString
 boundingObject     NULL                SFNode
 physics            NULL                SFNode
 joint              NULL                SFNode
 locked             FALSE               SFBool
 type               "bumper"            SFString
 lookupTable        0 0 0,0.1 1 0       MFVec3f
}
```

The `TouchSensor` node is used to model bumper sensors. A bumper sensor will detect the collision with any `Solid` object in the world, including other `DifferentialWheels` nodes. Collision detection is based upon the `boundingObject` field of the `TouchSensor` node and the `boundingObject` field of other `Solid` nodes. The `TouchSensor` node inherits from the `Solid` node. It includes two additional specific fields:

- `lookupTable`: similar to the one of the `DistanceSensor` node.
- `type`: type of sensor: ”bumper”.

**Note:** only the ”bumper” type is currently supported, but other types, including ”button”, ”force” or ”whisker” are likely to be implemented in a forthcoming version of Webots.

## 2.42 Transform

```
Transform {
 translation       0 0 0            SFVec3f
 rotation          0 1 0 0          SFRotation
 scale             1 1 1            SFVec3f
 children          []               MFNode
}
```

The `Transform` node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors.

The `translation`, `rotation`, `scale`, define a geometric 3D transformation consisting of (in order):

- a (possibly) non-uniform scale;
- a rotation;
- a translation.

## 2.43 Viewpoint

```
Viewpoint {
 fieldOfView       0.785398         SFFloat
 orientation       0 0 1 0          SFRotation
 position          0 0 10           SFVec3f
 near              0.05             SFFloat
 far               50               SFFloat
}
```

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 changes dynamically 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` and `far` parameters define the distance from the camera to the near and far clipping planes. These planes are parallel to the projection plane for the 3D display in the main window. Along with the `fieldOfView` parameter, they define the viewing frustum. Any 3D shape outside this frustum won't be rendered. Hence, shapes too far away (below the far plane) won't appear. Similarly, shapes too close (standing before the near parameter) won't appear either.

## 2.44 WorldInfo

```
WorldInfo {
 title                ""                   SFString
 info                 []                   MFString
 gravity              0 -9.81 0            SFVec3f
 CFM                  0.00001              SFFloat
 ERP                  0.2                  SFFloat
 physics              ""                   SFString
 fast2d               FALSE                SFBool
}
```

The `WorldInfo` node provides general information on the simulated world:

- The `title` field should describe shortly 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 gravity 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 rovers robots on Mars.
- The `CFM` and `ERP` fields correspond to the physics simulation world parameters used by ODE. See ODE documentation for more details about these parameters.

- The `physics` field refers to a shared library allowing the user to define custom physics properties using the ODE library. See Webots user guide for a description on how to set up custom physics properties. This is especially useful for modelling hydrodynamic forces, wind, non-uniform friction, etc.
- The `fast2d` field allows the user to switch to a fast 2D mode. This mode works only with simple worlds with cylindrical robots, distance sensors and without physics such as `khepera_fast2d.wbt`. It runs faster than the normal 3D mode because the collision detections and distance sensor measurements are performed in 2D only.

# Chapter 3

# Controller API

## 3.1 Robot

### robot_battery_sensor_enable

### robot_battery_sensor_disable

### robot_battery_sensor_get_value

#### NAME

`robot_battery_sensor_enable`,
`robot_battery_sensor_disable`,
`robot_battery_sensor_get_value` – *battery sensor function*

#### SYNOPSIS

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

void robot_battery_sensor_enable(unsigned short ms);

void robot_battery_sensor_disable();

float robot_battery_sensor_get_value();
```

#### DESCRIPTION

These functions allow you to measure the current level of the robot battery. First, it is necessary to enable the battery sensor measurement by calling the `robot_battery_sensor_enable` function. The `ms` parameter is expressed in milliseconds and defines how frequently measurements are performed. After being enabled a value can be read from the battery sensor by calling the `robot_battery_sensor_get_value` function. The returned value corresponds to the current level of the battery of the robot. The `robot_battery_sensor_disable` function should be used to stop the battery sensor measurements.

# robot_die

### NAME

`robot_die` – *declare an exit function*

### SYNOPSIS

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

void robot_die(void (*exit_function)(void));
```

### DESCRIPTION

This function declares an exit function to be used whenever a controller quits. A controller can quit for the following reasons: the simulator quits, or the robot quits the simulator by entering an HyperGate to be transfered to another simulation server. In the latter case, it might be useful for the robot to save important data (like an acquired behavior) before it quits, so that this data can be transfered to the target simulator corresponding to the HyperGate. Hence, when the robot restarts on the other side of the HyperGate, it can retrieve its data in its reset function before it starts running again.

The amount of time allocated to the die function is however limited to one second. After one second, if the controller has not quitted (i.e., returned from the die function), the controller will be forced to quit, even if the die method has not completed. This prevents the simulator to hang in case a controller never terminates or crashes.

### SEE ALSO

`robot_live`

# robot_get_device

### NAME

`robot_get_device` – *get a pointer to a device*

### SYNOPSIS

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

```
DeviceTag robot_get_device(const char *name);
```

### DESCRIPTION

This function returns a pointer to a device corresponding to a specified DEF name. For example, if the robot contains a `DistanceSensor` node which `DEF name` is ”ds1”, the function will return a pointer to that device. This `DeviceTag` pointer will be used subsequently for enabling, sending command to, or reading data from this device.

### SEE ALSO

`robot_live`

## robot_get_mode

### NAME

`robot_get_mode` – *get operation mode, simulation or real robot*

### SYNOPSIS

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

```
int robot_get_mode();
```

### DESCRIPTION

This function returns an integer value determining the current operation mode for the controller:

- 0: simulation in Webots.
- 1: cross-compiled version running natively on real robot.
- 2: remote controlled robot from Webots.

# robot_keyboard_enable

# robot_keyboard_disable

# robot_keyboard_get_key

### NAME

robot_keyboard_enable,
robot_keyboard_disable,
robot_keyboard_get_key – *keyboard reading function*

### SYNOPSIS

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

void robot_keyboard_enable(unsigned short ms);

void robot_keyboard_disable();

int robot_keyboard_get_key();
```

### DESCRIPTION

These functions allow you to read the key pressed on the computer keyboard from a controller program while the 3D simulation window of Webots is selected and the simulation is running. First, it is necessary to enable the keyboard readings by calling the `robot_keyboard_enable` function. The `ms` parameter is expressed in milliseconds and defines how frequently readings are updated. After being enabled values can be read by calling the `robot_keyboard_get_key` function repeatly until this function returns 0. The returned value, if non null, is a key code corresponding to a key currently pressed. If no key is currently pressed, the function will return 0. Calling the `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 key pressed. The `robot_keyboard_disable` function should be used to stop the keyboard readings.

# robot_live

### NAME

`robot_live` – *initialize a robot controller*

### SYNOPSIS

`#include <device/robot.h>`

`void robot_live(void (*reset_function)(void));`

### DESCRIPTION

This function must be called before any other controller API function. It is necessary to initialize the robot controller and optionally to provide a reset function to the controller. This reset function is useful to perform some initializations, so that the controller knows which sensors and actuators are available. The reset function should be a void function without any argument. It is called once at the beginning of the simulation and may be called again if the simulator needs to reset the robot. However, this rarely happens in practice.

### EXAMPLE

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

static DeviceTag my_sensor, my_actuator;

void my_reset_function() { /* called at init. */
  printf("hello!\n");
  my_sensor = robot_get_device("my_sensor");
  my_actuator = robot_get_device("my_actuator");
}

void my_exit_function() { /* called before quitting  */
  printf("bye bye!\n");
}

int my_run_function(int ms) {
  /* read the sensors and write to the actuators */
  ...

  return 64;
}

int main() {
  robot_live(my_reset_function); /* called when robot starts */
  robot_die(my_exit_function); /* called when robot quits */
  robot_run(my_run_function); /* called repeatly */
```

```
  return 0; /* this statement will never be reached */
}
```

### SEE ALSO

robot_get_device

robot_die

robot_run

## robot_run

### NAME

robot_run – *start the control loop*

### SYNOPSIS

`#include <device/robot.h>`

`void robot_run(int (*run)(int));`

### DESCRIPTION

The `robot_run` function starts the control loop for a robot. It declares a `run` function to be called repeatedly to control the robot . This `robot_run` never return. Hence, subsequent statements are never reached.

The `run` function receive an integer as an argument. It is 0 the first time the function is called and it take a possibly different value on subsequent calls. In synchronous simulation mode, this value is always 0. In asynchronous mode (and with some real robots), this value may be different from 0. Actually the `ms` integer value returned by the `run` function is the requested time step expressed in milliseconds. This time step define the duration of an iteration of the control loop. It starts at the beginning of a control loop iteration (call to the `run` function) and ends at the beginning of the next iteration. If the simulator (or real robot) can respect this requested time step, the `dt` parameter passed to `run` function is 0. Otherwise, this parameter has a non zero value. Let `controller_date` be the current time of the controller, the `dt` parameter be interpreted as follow:

- if `dt` = 0, then, the behavior is equivalent to the one of synchronous mode (request respected, no delay).

- if 0 <= `dt` > `ms`, then the actuator values were set at `controller_date` + `dt` and the sensor values where measured at `controller_date` + `ms`, as requested. It means that the step actually lasted the requested number of milliseconds, but the actuators command could not be executed on time.

- if `dt` > `ms`, then the actuators values were set at `controller_date` + `dt` and the sensors values where measured also at `controller_date` + `dt`. It means that the requested step duration could not be respected.

### SEE ALSO

`robot_live`

## robot_step

### NAME

`robot_step` – *execute a simulation step*

### SYNOPSIS

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

```
unsigned int robot_step(unsigned int ms);
```

### DESCRIPTION

This function is now obsolete. You should use the `robot_run` function instead. The `robot_step` function requests the simulator to perform a simulation step of `ms` milliseconds, that is to advance in the simulated time of this amount of time. In synchronous simulation mode, the request is always fulfilled and the function always return 0. In asynchronous mode, the request may not be fulfilled. In this case, the return value `dt`, representing the delay, may not be 0. Let `controller_date` be the current time of the controller, the return value be interpreted as follow:

- if `dt` = 0, then, the behavior is equivalent to the one of synchronous mode.

- if 0 <= `dt` > `ms`, then the actuator values were set at `controller_date` + `dt` and the sensor values where measured at `controller_date` + `ms`, as requested. It means that the step actually lasted the requested number of milliseconds, but the actuators command could not be executed on time.

- if `dt` $>$ `ms`, then the actuators values were set at `controller_date` + `dt` and the sensors values where measured also at `controller_date` + `dt`. It means that the requested step duration could not be respected.

### SEE ALSO

`robot_live`

## robot_task_new

### NAME

`robot_task_new` – *start a new thread of execution*

### SYNOPSIS

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

```
void 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

`robot_mutex_new`

## robot_mutex_new

### NAME

`robot_mutex_new`,
`robot_mutex_delete`,

robot_mutex_lock,
robot_mutex_unlock – *mutex functions*

### SYNOPSIS

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

MutexRef robot_mutex_new();

void robot_mutex_delete(MutexRef mutex);

void robot_mutex_lock(MutexRef mutex);

void robot_mutex_unlock(MutexRef mutex);
```

### DESCRIPTION

The `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 `robot_mutex_delete` function deletes the specified `mutex`. This function should be used when a mutex is no longer in use.

The `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 it locks it. This functions returns only after it locked the specified `mutex`.

The `robot_mutex_unlock` function unlocks the specified `mutex`, allowing other threads to lock it.

### SEE ALSO

`robot_task_new`

You should read some documentation on multi-thread programming techniques using mutexes if you are not familiar with this technology.

## 3.2 CustomRobot

# custom_robot_move

**NAME**

`custom_robot_move` – *control the position of the robot*

**SYNOPSIS**

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

```
void custom_robot_move(float tx,float ty,float tz,float rx,float ry,float
tz,float alpha);
```

**DESCRIPTION**

This function allows the user to modify the position and orientation of a custom robot. The move will be performed at the beginning of the next simulation step. If the collision detection system detects a collision between the `CustomRobot` node and any another `Solid` object, the move will not be performed and the custom robot position and orientation will remain unchanged. The `tx`, `ty` and `tz` values represent the requested translation relative to the current translation value of the robot. The `rx`, `ry`, `rz` and `alpha` values represent the offsets to be added to the current rotation vector and angle of the robot.

# custom_robot_set_rel_force_and_torque

**NAME**

`custom_robot_set_abs_force_and_torque`,
`custom_robot_set_rel_force_and_torque` – *apply a force and a torque to the robot body in absolute world coordinates or relative robot coordinates*

**SYNOPSIS**

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

```
void custom_robot_set_abs_force_and_torque(float fx,float fy,float fz,float
tx,float ty,float tz);
```

```
void custom_robot_set_rel_force_and_torque(float fx,float fy,float fz,float
tx,float ty,float tz);
```

**DESCRIPTION**

These functions apply only to a `CustomRobot` node which include a `Physics` node in its `physics` field. They allow the user to set a arbitrary force and a torque to the body of the custom robot. Typically, these force and torque result of the action of one or several actuators on the robot, like a propeller in a plane or a boat. Absolute force and torque would rather result from an external action, like wind or fluid friction. The force and torques are applied to the center of mass of the body of the robot (origin of the robot). With `custom_robot_set_abs_force_and_torque`, both the force and torque are specified in the world global coordinate system (absolute coordinates). With `custom_robot_set_rel_force_and_torque`, both the force and torque are specified in the robot local coordinate system (relative coordinates). The force components are specified by the `fx`, `fy` and `fz` parameters, expressed in Newton (N). The torque components are specified by the `tx`, `ty` and `tz` parameters, expressed in Joule (J).

Is is possible to use at the same time and on the same robot both `custom_robot_set_abs_force_and_torque` and `custom_robot_set_rel_force_and_torque` functions. The resulting forces and torques will be added.

The force and torque defined by a call to either `custom_robot_set_abs_force_and_torque` or `custom_robot_set_rel_force_and_torque` are applied continuously to the custom robot until a different force and torque are specified with the same function. To reset it to no force and no torque, you should use: `custom_robot_set_abs_force_and_torque(0,0,0,0,0,0)` or `custom_robot_set_rel_force_and_torque(0,0,0,0,0,0)`.

## 3.3 DifferentialWheels

## differential_wheels_set_speed

### NAME

`differential_wheels_set_speed` – *control the speed of the robot*

### SYNOPSIS

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

void differential_wheels_set_speed(short left,short right);
```

### DESCRIPTION

This function allows the user to specify a speed for the differentially wheeled robot. This speed will be send 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 0.1 radian per seconds. Hence a speed value of 20 will make the wheel rotate at a speed of 2 radian 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 on the command. Both the wheel radius and the noise on the command are documented in the `DifferentialWheels` node.

# differential_wheels_enable_encoders

### NAME

differential_wheels_enable_encoders,
differential_wheels_disable_encoders – *enable or disable the incremental encoders of the robot wheels*

### SYNOPSIS

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

void differential_wheels_enable_encoders(unsigned short ms);

void differential_wheels_disable_encoders (void);
```

### DESCRIPTION

These functions allow the user to enable or disable the incremental wheel encoders for both wheels of the DifferentialWheels robot. Incremental encoder are counters that incremented each time a wheel turns. The amount added to incremental encoder is computed from the angle the wheel rotated and from the encoderResolution paramter 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 when the DifferentialWheels robot faces 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.

# differential_wheels_get_left_encoder

### NAME

differential_wheels_get_left_encoder,
differential_wheels_get_right_encoder,
differential_wheels_set_encoders – *read or set the encoders of the robot wheels*

### SYNOPSIS

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

int differential_wheels_get_left_encoder (void);
```

```
int differential_wheels_get_right_encoder (void);
void differential_wheels_set_encoders (int left,int right);
```

#### DESCRIPTION

These functions are used to read or set the values of the left and right encoders. The encoders have to be enabled with `differential_wheels_enable_encoders`, so that the functions can read correct values. Moreover, the `encoderNoise` of the corresponding `DifferentialWheels` node should be positive. Setting encoders value will not make the wheels rotate to reach the specified value, instead, it will simply reset the encoders with the specified value.

## 3.4 DistanceSensor

### distance_sensor_enable

#### NAME

`distance_sensor_enable`,
`distance_sensor_disable` – *enable and disable the distance sensor measurements*

#### SYNOPSIS

```
#include <device/distance_sensor.h>
void distance_sensor_enable (DeviceTag sensor,unsigned short ms);
void distance_sensor_disable (DeviceTag sensor);
```

#### DESCRIPTION

`distance_sensor_enable` allows the user to enable a distance sensor measurement each `ms` milliseconds.

`distance_sensor_disable` turns the distance sensor off, saving computation time.

### distance_sensor_get_value

#### NAME

`distance_sensor_get_value` – *get the distance sensor measure*

#### SYNOPSIS

```
#include <device/distance_sensor.h>
unsigned short distance_sensor_get_value (DeviceTag sensor);
```

#### DESCRIPTION

`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 value range for the return value is defined by this lookup table.

## 3.5 Camera

### camera_enable

#### NAME

`camera_enable`,
`camera_disable` – *enable and disable the camera measurements*

#### SYNOPSIS

```
#include <device/camera.h>
void camera_enable (DeviceTag camera,unsigned short ms);
void camera_disable (DeviceTag camera);
```

#### DESCRIPTION

`camera_enable` allows the user to enable a camera measurement each `ms` milliseconds.

`camera_disable` turns the camera off, saving computation time.

### camera_get_fov

#### NAME

`camera_get_fov`,
`camera_set_fov` – *get and set field of view for a camera*

#### SYNOPSIS

```
#include <device/camera.h>
float camera_get_fov (DeviceTag camera);
void camera_set_fov (DeviceTag camera,float 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 however, that changing the field of view using `camera_set_fov` will not change the value of the `fieldOfView` field on the simulator side. It will only affect the controller side, making new rendered images use the specified field of view for the specified camera.

## camera_get_width

#### NAME

`camera_get_width`,
`camera_get_height` – *get the size of the camera image*

#### SYNOPSIS

```
#include <device/camera.h>
unsigned short camera_get_width (DeviceTag camera);
unsigned short camera_get_height (DeviceTag camera);
```

#### DESCRIPTION

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

## camera_get_type

#### NAME

`camera_get_type` – *get the type of the camera*

### SYNOPSIS

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

```
char camera_get_type (DeviceTag camera);
```

### DESCRIPTION

This function returns the type of a camera as defined in the corresponding `Camera` node. If the type is ”black and white” or ”grey”, then the return value is ’g’, if the type is ”color”, the return value is ’c’. Finally, if the type is ”range-finder”, the return value is ’r’.

## camera_get_image

### NAME

`camera_get_image`,
`camera_image_get_red`,
`camera_image_get_green`,
`camera_image_get_blue`,
`camera_image_get_grey` – *get the image data from a camera*

### SYNOPSIS

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

```
unsigned char *camera_get_image (DeviceTag camera);
```

```
unsigned char camera_image_get_red (image,width,x,y);
```

```
unsigned char camera_image_get_green (image,width,x,y);
```

```
unsigned char camera_image_get_blue (image,width,x,y);
```

```
unsigned char camera_image_get_grey (image,width,x,y);
```

### DESCRIPTION

The `camera_get_image` function allows you to read the contents of the last image grabbed by the camera. The image is coded as a series of three bytes coding for the red, green and blue levels of a pixel. Pixels are stored in lines ranging from the bottom left hand side of the image up to top right hand side. The memory chunk returned by this function doesn’t need to be released, as it is handled by the camera itself. The size in bytes of this memory chunk can be computed as follow:

`size` = `camera_width` * `camera_height` * 3

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

The `camera_image_get_` C macros are useful helpers for accessing directly the pixel colors from the pixel coordinates. They are not available in the Java programming interface (see below for details). The `camera_image_get_grey` macros is useful only for black and white cameras. These macros are defined as follow:

```
#define camera_image_get_red(image,width,x,y) \
                                        (image[3*((y)*(width)+(x))])

#define camera_image_get_green(image,width,x,y) \
                                        (image[3*((y)*(width)+(x))+1])

#define camera_image_get_blue(image,width,x,y) \
                                        (image[3*((y)*(width)+(x))+2])

#define camera_image_get_grey(image,width,x,y) \
                                        (image[3*((y)*(width)+(x))])
```

The Java version of this function returns an array of `int`. The size of this array is the number of pixels in the image, that is the width of the image multiplicated by the height of the image. Each `int` value represents one pixel coded using the RGB color model with 8 bits of red, green and blue data. For example red is `0xff0000`, yellow is `0xffff00`, etc. A black and white camera would return identical values for the red, blue and green components, like `0x4d4d4d`, hence the grey level in the 0-255 range can be retrived from a bitwise and with 0xff:

```
int [] image;
int [] grey_level = new int[64]; // K213 example 64x1 pixel B&W camera
...
image = camera_get_image(camera);
for(int i=0;i<64;i++) int grey_level[i] = image[i] & 0xff;
```

# camera_get_range_image

### NAME

`camera_get_range_image`,
`camera_range_image_get_value` – *get the range image and range data from a range-finder camera*

### SYNOPSIS

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

```
float *camera_get_range_image (DeviceTag camera);
```

```
float camera_range_image_get_value (range_image,width,x,y);
```

### DESCRIPTION

The `camera_get_range_image` macro allows you to read the contents of the last range image grabbed by a range-finder camera. The range image corresponds to the depth buffer produced by the OpenGL rendering. For each pixel, it provides the distance from the object to the camera. However, it is necessary to use the `camera_range_image_get_value` macro to obtain a linear distance information expressed in meters. Otherwise, the raw value in the buffer is non-linear, corresponding to the raw OpenGL depth buffer. The range image is coded as an array floating point value corresponding to the range value of each pixel of the image. Pixels are stored in lines ranging from the bottom left hand side of the image up to top right hand side. The memory chunk returned by this function doesn't need to be released, as it is handled by the camera itself. The size in bytes of this memory chunk can be computed as follow:

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

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

The `camera_range_image_get_value` macro is a useful helper for accessing directly the pixel range value from the pixel coordinates. This macro transfroms the distance value, so that it is linear and expressed in meters.

## camera_save_image

### NAME

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

### SYNOPSIS

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

```
int camera_save_image (DeviceTag camera,const char *file,int q);
```

### DESCRIPTION

The `camera_save_image` function allows you to save a `camera` image which was previouly obtained with the `camera_get_image` function. The image is saved in a file in either PNG or

JPEG format. The image format is specified by the `file` parameter. If `file` is terminated by `.png`, the image format is PNG. If the `file` is terminated by `.jpg` or `.jpeg`, the image format is JPEG. Other image formats are not supported. The `q` parameter is useful only for JPEG image. It defines the JPEG quality of the saved image. The `q` should be in the range 1 (worst quality) - 100 (best quality). Low quality JPEG files will use little disk space. For PNG images, the `q` parameter is ignored.

## 3.6 Emitter

# emitter_get_buffer

### NAME

`emitter_get_buffer`,
`emitter_get_buffer_size` – *get information on the emitter buffer*

### SYNOPSIS

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

void *emitter_get_buffer (DeviceTag emitter);

int emitter_get_buffer_size (DeviceTag emitter);
```

### DESCRIPTION

The `emitter_get_buffer` function returns a pointer to the buffer used by the emitter to send data. The `emitter_get_buffer_size` function returns the size of this buffer, expressed in bytes.

# emitter_send

### NAME

`emitter_send` – *send a message through the emitter*

### SYNOPSIS

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

```
void emitter_send (DeviceTag emitter,unsigned int size);
```

### DESCRIPTION

The `emitter_send` function sends `size` bytes of data contained in the beginning of the emitter buffer.

## emitter_get_channel

### NAME

`emitter_get_channel`,
`emitter_set_channel` – *get or set channel information for an emitter.*

### SYNOPSIS

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

int emitter_get_channel (DeviceTag emitter);

void emitter_set_channel (DeviceTag emitter,int channel);
```

### DESCRIPTION

The `emitter_get_channel` function returns the channel value of the `Emitter` node. Only receivers set to the same channel of the emitter can receive message from this emitter.

The `emitter_set_channel` function allows the controller to change the emission channel, so that different receivers may receive the messages of the emitter. Calling this function will change the channel field of the `Emitter` node.

# 3.7 LED

## led_set

### NAME

`led_set` – *turn on or off a LED*

### SYNOPSIS

```
#include <device/led.h>
void led_set (DeviceTag device,unsigned char value);
```

### DESCRIPTION

`led_set` switches on or off a LED. If the `value` parameter is 0, the LED is turned off. 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 using the second color specified in the `color` field of the `LED` node. And so on. The `value` parameter should not be bigger than the size of the `color` field of the corresponding `LED` node.

## 3.8 LightSensor

# light_sensor_enable

### NAME

`light_sensor_enable`,
`light_sensor_disable` – *enable and disable the light sensor measurements*

### SYNOPSIS

```
#include <device/light_sensor.h>
void light_sensor_enable (DeviceTag sensor,unsigned short ms);
void light_sensor_disable (DeviceTag sensor);
```

### DESCRIPTION

`light_sensor_enable` allows the user to enable a light sensor measurement each `ms` milliseconds.

`light_sensor_disable` turns the light sensor off, saving computation time.

# light_sensor_get_value

### NAME

`light_sensor_get_value` – *get the light sensor measure*

### SYNOPSIS

`#include <device/light_sensor.h>`

`unsigned short light_sensor_get_value (DeviceTag sensor);`

### DESCRIPTION

`light_sensor_get_value` returns the last value measured by the specified light sensor. This value is computed by the simulator according to the lookup table of the `LightSensor` node. Hence, the value range for the return value is defined by this lookup table.

## 3.9 Pen

# pen_write

### NAME

`pen_write` – *enable or disable pen writing*

### SYNOPSIS

`#include <device/pen.h>`

`void pen_write (DeviceTag pen,gboolean write);`

### DESCRIPTION

`pen_write` allows to switch up or down a pen device to disable or enable writing. If the `write` parameter is TRUE, the specified `pen` device will write, whereas if `write` is FALSE, it won't write.

# pen_set_ink_color

### NAME

`pen_set_ink_color` – *change the color of the ink of a pen*

### SYNOPSIS

```
#include <device/pen.h>
void pen_set_ink_color (DeviceTag pen,float r,float g,float b,float d);
```

### DESCRIPTION

`pen_set_ink_color` changes the current ink color of the specified `pen` device. The `r`, `g`, `b` and `d` parameters are floating point values ranging between 0 and 1 and defining the new color of the ink. The `d` parameter defines the ink density, 0 meaning transparent ink and 1 meaning opaque ink.

### EXAMPLE

```
pen_set_ink_color(pen,0.9,0.2,0.2.0.9);
```

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

## 3.10 GPS

## `gps_enable`

### NAME

`gps_enable`,
`gps_disable` – *enable and disable the GPS measurements*

### SYNOPSIS

```
#include <device/gps.h>
void gps_enable (DeviceTag sensor,unsigned short ms);
void gps_disable (DeviceTag sensor);
```

### DESCRIPTION

gps_enable allows the user to enable a GPS measurement each ms milliseconds.

gps_disable turns the GPS off, saving computation time.

# gps_get_matrix

### NAME

gps_get_matrix,
gps_position_x,
gps_position_y,
gps_position_z,
gps_euler – *get the GPS measurement represented as a 4x4 matrix*

### SYNOPSIS

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

const float *gps_get_matrix (DeviceTag sensor);

float gps_position_x (float *matrix);

float gps_position_y (float *matrix);

float gps_position_z (float *matrix);

void gps_euler (const float *matrix,float *euler);
```

### DESCRIPTION

gps_get_matrix returns the last value measured by the specified GPS sensor. The value returned is an array of 16 floating point numbers representing the standard OpenGL 4x4 matrix corresponding to the absolute position, orientation and scale of the GPS node.

gps_position_x, gps_position_y and gps_position_z are helper macros used to retrive the x, y and z coordinate of the GPS sensor from the matrix data. They are defined as follow:

```
#define gps_position_x(matrix) ((matrix)[12]/(matrix)[15])
#define gps_position_y(matrix) ((matrix)[13]/(matrix)[15])
#define gps_position_z(matrix) ((matrix)[14]/(matrix)[15])
```

The gps_euler is also a helper function that returns the three local Euler angles from the GPS matrix. The matrix parameter is a pointer to the OpenGL 4x4 matrix returned by the gps_get_matrix function. The euler parameter should point to an array of three floating point numbers that will receive the Euler angles. The first and last Euler angles can be interpreted as inclinometer angle values along the local X and Z axis. The second Euler angle can be interpreted as a compass angle value. These angle values are expressed in radians.

## 3.11 Gripper

# gripper_set_position

### NAME

gripper_set_position – *open or close the gripper*

### SYNOPSIS

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

```
void gripper_set_position (DeviceTag gripper,float position);
```

### DESCRIPTION

The gripper_set_position function allows the user to close or open the gripper depending on the specified position value which represents the aperture of the gripper device, expressed in meters. Hence a value of 0 will close the gripper and a value of 0.04 will open the gripper 4 cm wide.

# gripper_enable_position

### NAME

gripper_enable_position,
gripper_enable_resistivity,
gripper_disable_position,
gripper_disable_resistivity – *enable or disable the position and resistivity sensors on a gripper*

### SYNOPSIS

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

```
void gripper_enable_position (DeviceTag gripper,unsigned short ms);
```

```
void gripper_enable_resistivity (DeviceTag gripper,unsigned short ms);
```

```
void gripper_disable_position (DeviceTag gripper);
```

```
void gripper_disable_resistivity (DeviceTag gripper);
```

### DESCRIPTION

These functions enable each `ms` milliseconds or disable the gripper position and resistivity measurement.

## gripper_get_position

### NAME

`gripper_get_position`,
`gripper_get_resistivity` – *return the position and resistivity values measured on the gripper*

### SYNOPSIS

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

float gripper_get_position (DeviceTag gripper);

float gripper_get_resistivity (DeviceTag gripper);
```

### DESCRIPTION

The `gripper_get_position` function returns the position measurement performed on the specified gripper device. The position is expressed in meters and corresponds to the aperture of the gripper as with the `gripper_set_position` function. However, it returns the current position of the gripper and not the target position specified with `gripper_set_position` (which may be the same value when the target position is reached). This function may be useful to measure the size of a gripped object.

The `gripper_get_resistivity` function returns the resistivity measurement performed on the specified gripper device. This value is expressed in ohm. In this first version, we assume that any object has a resistivity of one ohm. It will return *Inf* when no object is gripped and 1.0 when an object is gripped.

# 3.12 MTN

## mtn_new

### NAME

`mtn_new`,
`mtn_get_error`,
`mtn_fprint`,
`mtn_delete` – *handle a MTN motion file*

### SYNOPSIS

```
#include <device/mtn.h> #include <stdio.h>

MTN *mtn_new (const char *filename);

const char *mtn_get_error ();

void mtn_fprint (FILE *fd,MTN *mtn);

void mtn_delete (MTN *mtn);
```

### DESCRIPTION

The MTN functions are a facility for reading and playing back motions running simultaneously on several servo devices. The file format used for these motions is compatible with the Sony MTN file format used with the Sony Aibo robots. A motion file may contain all the information necessary for a walking gait.

`mtn_new` allows the user to open a MTN motion file specified by the `filename` parameter.

If an error occurs, the `mtn_get_error` will return a text description of the last error, otherwise it returns null.

`mtn_fprint` prints out the `mtn` structure passed as an argument into the specified `fd` file descriptor. The `fd` parameter may be a file opened with `fopen` with write access, or a standard C output, like `stdout`.

`mtn_delete` deletes the `mtn` structure passed as an argument. This `mtn` parameter should not be used any more after calling `mtn_delete`.

### SEE ALSO

`mtn_play`

## `mtn_play`

### NAME

`mtn_play`,
`mtn_is_over`,
`mtn_get_length`,
`mtn_get_time` – *control the execution of a MTN motion file*

### SYNOPSIS

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

void mtn_play (MTN *mtn);

int mtn_get_length (MTN *mtn);

int mtn_get_time (MTN *mtn);

int mtn_is_over (MTN *mtn);
```

### DESCRIPTION

`mtn_play` starts the execution of a `mtn` motion passed as an argument for controlling several servo simultaneously. The control will start at the next iteration step (each time the `run` function returns) by issuing automatically a number of `servo_set_position` function calls corresponding to the execution of the specified motion.

`mtn_get_length` returns the length expressed in milliseconds of the specified `mtn` motion.

`mtn_get_time` returns the current time of execution of the specified `mtn` motion. This time value is expressed in millisecond. The minimum value is 0 (beginning of the motion) and the maximum value is the value returned by the `mtn_get_length` function (end of the motion).

`mtn_is_over` returns 1 if the specified `mtn` motion has completed and 0 otherwise. It is useful to test when a motion is finished.

### SEE ALSO

`mtn_new` `servo_set_position`

## 3.13 Receiver

# `receiver_enable`

### NAME

`receiver_enable`,
`receiver_disable` – *enable and disable the receiver measurements*

### SYNOPSIS

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

```
void receiver_enable (DeviceTag receiver,unsigned short ms);
```

```
void receiver_disable (DeviceTag receiver);
```

### DESCRIPTION

`receiver_enable` allows the user to enable a receiver measurement each `ms` milliseconds.

`receiver_disable` turns the receiver off, saving computation time.

## receiver_get_buffer

### NAME

`receiver_get_buffer`,
`receiver_get_buffer_size` – *get information on the receiver buffer*

### SYNOPSIS

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

```
void *receiver_get_buffer (DeviceTag receiver);
```

```
int receiver_get_buffer_size (DeviceTag receiver);
```

### DESCRIPTION

The `receiver_get_buffer` function returns a pointer to the buffer used by the receiver to store received data. This function needs to be called each time new data arrives in the receiver because the address of the buffer changes when new data arrives. The returned memory chunk doesn't need to be released. Memory management is done by the receiver. Moreover calling `receiver_get_buffer` will cause the data to be flushed out of the receiver, hence calling `receiver_get_buffer_size` immediately after will return 0;

The `receiver_get_buffer_size` function returns the size of this buffer, expressed in bytes, that is the number of bytes received and stored in the buffer. It has to be called before the `receiver_get_buffer` function, otherwise, it returns always 0.

## 3.14 Servo

# servo_enable_position

### NAME

servo_enable_position,
servo_disable_position,
servo_get_position – *get the actual position of a servo*

### SYNOPSIS

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

void servo_enable_position (DeviceTag servo,unsigned short ms);

void servo_disable_position (DeviceTag servo);

float servo_get_position (DeviceTag servo);
```

### DESCRIPTION

The servo_enable_position function activates the position measurement for the specified servo. A new position measurement will be performed each ms millisecond and can be obtained from the servo_get_position function. The returned value is the last measurement of the servo position. If the servo is a rotation servo, the unit of the returned value is radian, otherwise, it is meter. The servo_get_position returned value is valid only if the corresponding servo was previously enabled.

The servo_disable_position function desactivates the position measurement for the specified servo. The servo_get_position should not be used any more after a servo position measurement was disabled, as it will return outdated or erroneous values.

# servo_get_feedback

### NAME

servo_get_feedback – *get feedback on the absolute position, orientation, linear velocity and angular velocity of a servo*

### SYNOPSIS

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

```
float *servo_get_feedback (DeviceTag servo,unsigned short ms);
```

### DESCRIPTION

The `servo_get_feedback` function activates the servo feedback measurement for the specified `servo`. A new feedback measurement will be performed each `ms` millisecond and stored in the float array returned by the function. This array contains a number of floating point values which should be accessed using the following macros:

The `servo_feedback_position` macro returns the absolute position of the servo, as a pointer to three floating point values (see the `dBodyGetPosition` function in the ODE documentation for more details).

The `servo_feedback_quaternion` macro returns the orientation quaternion of the servo, as a pointer to four floating point values, respecting ODE convention (see the `dBodyGetQuaternion` function in the ODE documentation for more details).

The `servo_feedback_linear_vel` macro returns the linear velocity of the servo, as a pointer to three floating point values (see the `dBodyGetLinearVel` function in the ODE documentation for more details).

The `servo_feedback_angular_vel` macro returns the angular velocity of the servo, as a pointer to three floating point values (see the `dBodyGetAngularVel` function in the ODE documentation for more details).

All these four macros take the return value of the `servo_get_feedback` function as a unique argument.

To desactivate the feedback measurement for a servo, call `servo_get_feedback` with a `ms` parameter set to 0.

## servo_set_position

### NAME

`servo_set_position`,
`servo_set_velocity`,
`servo_set_acceleration`,
`servo_set_force` – *set servo parameters*

### SYNOPSIS

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

```
void servo_set_position (DeviceTag servo,float position);
void servo_set_velocity (DeviceTag servo,float vel);
void servo_set_acceleration (DeviceTag servo,float acc);
void servo_set_force (DeviceTag servo,float force);
```

### DESCRIPTION

The `servo_set_position` function gives a new target position the servo will try to reach. If the servo is a rotation servo, the unit of the `position` parameter is radian, otherwise, it is meter. If the servo has no maximum and minimum position and if the value passed as `position` is `SERVO_INFINITY`, the servo will turn endlessly in the positive direction. Setting it to `-SERVO_INFINITY` will make the servo turn endlessly in the negative direction.

The `servo_set_velocity` function gives the target speed the servo will try to reach in order to achieve the given position. If the servo is a rotation servo, the unit of the `vel` parameter is rad/s, otherwise, it is m/s.

The `servo_set_acceleration` function changes the acceleration value used by the position controller. If the servo is a rotation servo, the unit of the `acc` parameter is rad/$s^2$, otherwise, it is m/$s^2$.

The `servo_set_force` function gives the maximum torque or force the servo will have. If the servo is a rotation servo, the unit of the `force` parameter is Joule (J) as it is a torque, otherwise, it is Newton (N) as it is a force.

### SEE ALSO

`mtn_new` `mtn_play`

## servo_motor_off

### NAME

`servo_motor_off` – *turn off the servo motor*

### SYNOPSIS

```
#include <device/servo.h>
void servo_motor_off (DeviceTag servo);
```

### DESCRIPTION

The servo_motor_off function turns the motor of the specified servo off. This means that no control is performed any more on the servo motor. Moreover, the servo will become soft and will probably move to a new position due to the action of external forces and torques on it. To turn a servo motor on after it was turned off, simply use the servo_set_position function with the desired new position. Note that when a servo motor is off, its position sensor may still be enabled, hence the servo_get_position function still returns the actual position of the servo.

# servo_set_rel_force_and_torque

### NAME

servo_set_abs_force_and_torque,
servo_set_rel_force_and_torque – *apply a force and a torque to a servo in absolute world coordinates or relative robot coordinates*

### SYNOPSIS

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

void servo_set_abs_force_and_torque(DeviceTag servo,float fx,float fy,float fz,float tx,float ty,float tz);

void servo_set_rel_force_and_torque(DeviceTag servo,float fx,float fy,float fz,float tx,float ty,float tz);
```

### DESCRIPTION

These functions apply only to a Servo node which include a Physics node in its physics field. Moreover, this Servo node should have its forceAndTorque field set to TRUE. Otherwise, the functions will simply be ignored.

These functions allows the user to set a arbitrary force and a torque to servo. Typically, relative force and torque result of the action of one or several actuators on the robot, like a propeller in a plane or a boat. Absolute force and torque would rather result from an external action, like wind or fluid friction. The force and torque are applied to the local origin the servo. With servo_set_abs_force_and_torque, both the force and torque are specified in the world global coordinate system (absolute coordinates). With servo_set_rel_force_and_torque, both the force and torque are specified in the robot local coordinate system (relative coordinates). The force components are specified by the fx, fy and fz parameters, expressed in Newton (N). The torque components are specified by the tx, ty and tz parameters, expressed in Joule (J).

Is is possible to use at the same time on the same servo both servo_set_abs_force_and_torque and servo_set_rel_force_and_torque functions. These forces and torques will be added.

The force and the torque defined by a call to either `servo_set_abs_force_and_torque` or `servo_set_rel_force_and_torque` are applied continuously to the custom robot until a different force and torque are specified with the same function. To reset it to no force and no torque, you should use:

`servo_set_abs_force_and_torque(0,0,0,0,0,0)` or

`servo_set_rel_force_and_torque(0,0,0,0,0,0)`.

# servo_run_animation

### NAME

`servo_run_animation`,
`servo_get_animation_number`,
`servo_get_animation_range` – *servo animation functions*

### SYNOPSIS

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

void servo_run_animation (DeviceTag servo,int anim);

int servo_get_animation_number (DeviceTag servo);

float servo_get_animation_range (DeviceTag servo,int anim);
```

### DESCRIPTION

These functions are useful to perform non-robot-realistic animations. They do not refer to a real servo device, and permit to change dynamically the translation and rotation field of the Servo node. This results in more life-like animations, but should not be used in realistic simulations of real servo devices.

The `servo_run_animation` function starts the animation specified by `anim` which corresponds to the index of the `Animation` node in the `Servo` `animation` field. 0 is the first `Animation` node of the `MFNode` list. The animation is also started recursively in all the children `Servo` of the `Servo` specified by the `servo` parameter. Passing `-1` as `anim` will stop the animation in the specified `servo` and recursively in its subsequent `Servo` children.

The `servo_get_animation_number` function returns the number of `Animation` nodes present in the `animation` field of the specified `servo`.

The `servo_get_animation_range` function returns the range of the animation, that is the last value of the `key` field of the `Animation` node. The `Animation` node is specified by its `anim` index like with the `servo_run_animation`. The range value corresponds to the length of the animation cycle expressed in seconds.

## 3.15 Supervisor

The supervisor controller is a particular case of a robot controller, hence the `robot_live`, `robot_run`, `robot_get_device`, etc. functions also apply to supervisor controllers. Moreover, as long as the supervisor contains sensors and actuators in its list of children, the corresponding sensor and actuator functions can be used (except for the `differential_wheels_*` functions that are specific to differential wheels robots).

This section covers the supervisor specific functions, allowing the supervisor controller to track the position and orientation of `Solid` nodes in the scene, to move them, to take a snapshot of the scene, etc.

# supervisor_export_image

### NAME

`supervisor_export_image` – *save the current 3D image of the simulator into a JPEG file, suitable for building a webcam system*

### SYNOPSIS

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

```
void supervisor_export_image (char *filename,unsigned char quality);
```

### DESCRIPTION

The `supervisor_export_image` function saves the current 3D image of the simulator window into a `jpeg` file as specified in the `filename` parameter. The `quality` parameter defines the `jpeg` quality (in the range 0 - 100). The `filename` parameter should specify a `jpeg` file (as an absolute or relative path), i.e., `"my_image.jpeg"` or `"/var/www/html/images/shot.jpg"`. Indeed, 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

A simple example of using the `supervisor_export_image` is provided in the `photographer` directory of the `controllers` directory.

An example of a webcam system using `supervisor_export_image` is provided in the `webcam` directory of the `controllers` directory.

# supervisor_import_node

### NAME

supervisor_import_node – *import a node into the scene*

### SYNOPSIS

```
#include <device/supervisor.h>
void supervisor_import_node (char *filename,int position);
```

### DESCRIPTION

The `supervisor_import_node` function imports a Webots node into the scene. This node should be defined in a Webots file referenced to 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 Object** button.

The `position` parameter defines the position in the scene tree where the new node is going to 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.
- etc.
- -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.
- etc.

As in `supervisor_export_image`, the `filename` parameter can be specified with an absolute or a relative path.

# supervisor_node_get_from_def

### NAME

`supervisor_node_get_from_def`,
`supervisor_node_was_found` – *get a pointer to a node of the scene from its DEF name and check if that node exists.*

### SYNOPSIS

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

```
NodeRef supervisor_node_get_from_def (char *defname);
```

```
gboolean supervisor_node_was_found (NodeRef node);
```

### DESCRIPTION

The `supervisor_node_get_from_def` function retrieves a pointer to a node of the scene from its DEF name. The return value can be used for subsequent calls to functions referring to a node of the scene. Note that this function always return a non NULL value, even if the node does not exist in the scene.

The `supervisor_node_was_found` checks whether the node referred to by `node` really exists in the scene. It returns `TRUE` if the node exists and `FALSE` otherwise.

## supervisor_set_label

### NAME

`supervisor_set_label` – *display a text label over the 3D scene*

### SYNOPSIS

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

```
NodeRef supervisor_set_label (unsigned short id,char *text,float x,float
y,float size,unsigned int color);
```

### DESCRIPTION

The `supervisor_set_label` function displays a text label over the 3D scene in Webots' main window. The `id` parameter is a an identifier for the label, you can choose any value in the range 0 - 65536. It will be used later on when you want to change that label, like updating the text. 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 with the same unit as the `y` parameter. Finally, the `color` parameter defines the color for the label. It is expressed as 32 bits RGB integer value, where the first byte defines the transparency level, the second byte represents the red component, the third byte represents the green component and the last byte represents the blue component. A transparency level of 0 means no transparency while a transparency level of `0xFF` means total transparency. Intermediate values correspond to semi-transparency levels.

### EXAMPLE

- `supervisor_set_label(0,"hello world",0,0,0.1,0x00ff0000);`

  will display the label ”hello world” in red at the upper left corner of the 3D window.
- `supervisor_set_label(1,"hello dad",0,0.1,0.1,0x8000ff00);`

  will display the label ”hello dad” in semi-transparent green, just below.
- `supervisor_set_label(0,"hello universe",0,0,0.1,0xffff00);`

  will change the label ”hello world” defined earlier into ”hello universe”, setting a yellow color to the new text.

## supervisor_simulation_quit

### NAME

`supervisor_simulation_quit` – *terminate the simulator and controller processes*

### SYNOPSIS

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

```
void supervisor_simulation_quit ();
```

### DESCRIPTION

The `supervisor_simulator_quit` function sends a request to the simulator process, asking to terminate and quit immediately. As a result of terminating the simulator process, all the controller processes, including the calling supervisor controller process will terminate.

# supervisor_simulation_revert

### NAME

supervisor_simulation_revert – *reload the current scene*

### SYNOPSIS

```
#include <device/supervisor.h>
void supervisor_simulation_revert ();
```

### DESCRIPTION

The `supervisor_simulator_revert` function sends a request to the simulator process, asking 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 might want to save some data in a file from you supervisor program to be able to reload it when the supervisor controller restarts.

# supervisor_simulation_physics_reset

### NAME

supervisor_simulation_physics_reset – *stop the inertia of all solids in the world*

### SYNOPSIS

```
#include <device/supervisor.h>
void supervisor_simulation_physics_reset ();
```

### DESCRIPTION

The `supervisor_simulator_physics_reset` function sends a request to the simulator process, asking 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 is reset to 0, hence the inertia is stopped. This is actually implemented by calling the ODE `dBodySetLinearVel` and `dBodySetAngularVel` functions for all bodies with a nul velocity parameter. This function is especially useful when resetting a robot at an initial position from which no initial inertia is required.

# supervisor_robot_set_controller

### NAME

supervisor_robot_set_controller – *change the controller of a specified robot*

### SYNOPSIS

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

```
void supervisor_robot_set_controller (NodeRef robot,const char * ctr);
```

### DESCRIPTION

The supervisor_robot_set_controller function sends a request to the simulator, asking to change the controller of the specified robot to the one defined by the ctr parameter. The current robot controller process is then immediately terminated and the requested controller process is launched instead to control the robot. This function can be used for both robot and supervisor controllers.

# supervisor_start_animation

### NAME

supervisor_start_animation,
supervisor_stop_animation – *save the current simulation into a Webview animation file*

### SYNOPSIS

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

```
void supervisor_start_animation (char *filename);
```

```
void supervisor_start_animation ();
```

### DESCRIPTION

The supervisor_start_animation function starts saving the current simulation in a Webview animation file. Saving the animation will complete after the supervisor_stop_animation function is called. The filename parameter should refer to a file with a WVA extension.

Webview is the Webots animation viewer. It is freely available as a stand alone application or a plugin for Mozilla, Netscape and Internet Explorer. It allows you to demonstrate your simulations as 3D animations in which the users can navigate to observe the behavior of the robots. Webview is available free of charge from Cyberbotics web site.

# supervisor_field_get, supervisor_field_set

### NAME

`supervisor_field_get`,
`supervisor_field_set` – *get and set the contents of the field of a node in the scene*

### SYNOPSIS

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

```
void supervisor_field_get (NodeRef node,field_type type,void *data,unsigned short ms);
```

```
void supervisor_field_set (NodeRef node,field_type type,void *data);
```

### DESCRIPTION

The `supervisor_field_get` function allows the supervisor controller to track the evolution of some fields of a node. Currently only a few fields are trackable, as described in the following list of field types. Each `ms` milliseconds, the new value of the field (if any) is stored at `data` with a specific data type (usually an array of `float`). The type parameter should be a combination of the following primitive constants, as defined in the `supervisor.h` header file:

```
For any solid node (incl. Solid, DifferentialWheels and CustomRobot):
SUPERVISOR_FIELD_TRANSLATION_X
SUPERVISOR_FIELD_TRANSLATION_Y
SUPERVISOR_FIELD_TRANSLATION_Z
SUPERVISOR_FIELD_ROTATION_X
SUPERVISOR_FIELD_ROTATION_Y
SUPERVISOR_FIELD_ROTATION_Z
SUPERVISOR_FIELD_ROTATION_ANGLE

For any robot node (incl. DifferentialWheels and CustomRobot):
SUPERVISOR_FIELD_BATTERY_CURRENT

For any light node (incl. PointLight and DirectionalLight):
SUPERVISOR_FIELD_LIGHT_INTENSITY
```

Some predefined combinations include:

```
SUPERVISOR_FIELD_TRANSLATION = SUPERVISOR_FIELD_TRANSLATION_X+
SUPERVISOR_FIELD_TRANSLATION_Y+SUPERVISOR_FIELD_TRANSLATION_Z

SUPERVISOR_FIELD_ROTATION = SUPERVISOR_FIELD_ROTATION_X+
SUPERVISOR_FIELD_ROTATION_Y+SUPERVISOR_FIELD_ROTATION_Z+
SUPERVISOR_FIELD_ROTATION_ANGLE

SUPERVISOR_FIELD_TRANSLATION_AND_ROTATION =
SUPERVISOR_FIELD_TRANSLATION+SUPERVISOR_FIELD_ROTATION
```

The rotation angle requested by SUPERVISOR_FIELD_ROTATION_ANGLE is expressed in radian. Its minimum value is 0. Its maximum value is 2 PI.

It is necessary that the `data` parameter be a pointer towards a large enough array of `float`, able to contain all the requested values. One `float` is necessary for each primitive value. Please note that this `data` pointer should point to a valid memory chunk at the time of the `run` control function. Hence, it should not be stored on the heap of a local function. Instead, it has to be dynamically allocated, or declared as a local or global static variable.

The values pointed by the `data` parameter are updated by the simulator every `ms` simulated millisecond. This update is performed if necessary before calling the `run` control function.

In order to disable the tracking of a field, call the `supervisor_field_get` function with a `ms` parameter set to 0.

There should be only one call to `supervisor_field_get` for a node. Requested values are updated at regular time steps. A common error is to call `supervisor_field_get` to retrive the translation field of a node and then to call again `supervisor_field_get` to retrieve the orientation field of the same node. The result is that the translation will never be retrieved. Instead you should call once `supervisor_field_get` and ask for both the translation and rotation information (using the predefined combinations described earlier or using the + or OR operators).

The `supervisor_field_set` function works the same way as `supervisor_field_get`, except that it changes the value of the requested `field` instead of reading it.

### EXAMPLE

An simple example of using field tracking is given in the `supervisor` controller.

## 3.16 TouchSensor

# touch_sensor_enable

### NAME

`touch_sensor_enable`,
`touch_sensor_disable` – *enable and disable the touch sensor measurements*

### SYNOPSIS

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

void touch_sensor_enable (DeviceTag sensor,unsigned short ms);

void touch_sensor_disable (DeviceTag sensor);
```

### DESCRIPTION

`touch_sensor_enable` allows the user to enable a touch sensor measurement each `ms` milliseconds.

`touch_sensor_disable` turns the touch sensor off, saving computation time.

# touch_sensor_get_value

### NAME

`touch_sensor_get_value` – *get the touch sensor measure*

### SYNOPSIS

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

unsigned short touch_sensor_get_value (DeviceTag sensor);
```

### DESCRIPTION

`touch_sensor_get_value` returns the last value measured by the specified touch sensor. This value is computed by the simulator according to the lookup table of the `TouchSensor` node. Hence, the value range for the return value is defined by this lookup table.

# Chapter 4

# Webots File Format

## 4.1 File Structure

Webots files must begin with the characters:

```
#VRML_SIM V4.0 utf8
```

and the following nodes have to appear:

```
  WorldInfo
  Viewpoint
  Background
```

### 4.1.1 Example

```
#VRML_SIM V4.0 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
}
```

The file extension is `.wbt` (standing for WeBoTs).