Elektrobit

EB GUIDE Studio

User guide

Version 6.8.0.190618155600

Elektrobit Automotive GmbH
Am Wolfsmantel 46
D-91058 Erlangen
GERMANY

Phone: +49 9131 7701-0
Fax: +49 9131 7701-6333
http://www.elektrobit.com

Legal notice

Confidential and proprietary information

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Elektrobit Automotive GmbH.

All brand names, trademarks and registered trademarks are property of their rightful owners and are used only for description.
Copyright 2019, Elektrobit Automotive GmbH.

Table of Contents

1. About this documentation

1.1. Target audience: Modelers

Modelers use EB GUIDE Studio to create a human machine interface (HMI). In EB GUIDE the HMI is called EB GUIDE model. Communication with applications is carried out through determined events using the event mechanism, through datapool items using the datapool and through user-specific EB GUIDE Script functions.

Modelers perform the following tasks:

- Use an architecture of widgets and views to specify graphical elements on the displays
- Communicate with designers and usability experts to optimize user interfaces
- Use state machine functionality to specify when graphical elements are displayed
- Define how elements react to input from devices such as control panels or touch screens
- Define how elements receive information from hardware or software applications that offer services like a navigation unit
- Define interfaces between model elements as well as input and output devices

Modelers have profound knowledge of the following:

- EB GUIDE Studio features
- The UML state machine concept
- The specifications and requirements of the domain
- The interchanged data and the EB GUIDE GTF communication mechanism
- The specifications of 3D graphics, if 3D graphics are used in the project

1.2. Structure of user documentation

The information is structured as follows:

- Background information

  Background information introduce you to a specific topic and important facts. With this information you are able to carry out the related instructions.

- How-to-instruction

The instructions guide you step-by-step through a specific task and show you how to use EB GUIDE. Instructions are recognized by the present participle in the title *(ing)*, for example, *Starting EB GUIDE Studio*.

- Tutorial

  A tutorial is an extended version of a how-to-instruction. It guides you through a complex task. The headline starts with *Tutorial:*, for example *Tutorial: Creating a button*.

- Reference

  References provide detailed technological parameters and tables.

- Demonstration

  Demonstrations give you insight into how an application is written and the sequence of interactions. The demonstrations are part of the EB GUIDE GTF SDK.

1.3. Typography and style conventions

The following pictographs and signal words are used in this documentation to indicate important information.

The signal word *WARNING* indicates information that is vital for the success of the configuration.

WARNING

Source and kind of problem

What can happen to the software?

What are the consequences of the problem?

How does the user avoid the problem?

The signal word *NOTE* indicates important information on a subject.

NOTE

Important information

Gives important information on a subject.

The signal word *TIP* provides helpful hints, tips and shortcuts.

TIP

Helpful hints

Gives helpful hints

Throughout the documentation you will find words and phrases that are displayed in **bold** or in *italic* or monospaced font.

To find out what these conventions mean, see the following examples.

All default text is written in Arial Regular font.

Font	Description	Example
Arial italics	to emphasize new or important terms	The *basic building blocks* of a configuration are module configurations.
Arial boldface	for GUI elements and keyboard keys	1. In the **Project** drop-down list box, select Project_A. 2. Press the **Enter** key.
Monospaced font (Courier)	for file names, directory names and chapter names	Put your script in the `function_name/abcdirectory`.
Monospaced font (Courier)	for user input, code, and file directories	`CC_FILES_TO_BUILD =(PROJECT_PATH)/source/network/can_node.c CC_FILES_TO_BUILD += $(PROJECT_PATH)/source/network/can_config.c` The module calls the `BswM_Dcm_RequestSessionMode()` function. For the project name, enter `Project_Test`.
Square brackets [ ]	to denote optional parameters; for command syntax with optional parameters	insertBefore [<opt>]
Curly brackets {}	to denote mandatory parameters; for command syntax with mandatory parameters	insertBefore {<file>}
Three dots …	to indicate further parameters; for command syntax with multiple parameters	insertBefore [<opt>…]
A vertical bar |	to indicate all available parameters; for command syntax in which you select one of the available parameters	allowinvalidmarkup {on|off}

This is a step-by-step instruction

Whenever you see the bar with step traces, you are looking at step-by-step instructions or how-tos.

Prerequisite:

- This line lists the prerequisites to the instructions.

Step 1
An instruction to complete the task.

Step 2
An instruction to complete the task.

Step 3
An instruction to complete the task.

1.4. Naming conventions

In EB GUIDE documentation the following directory names are used:

- The directory to which you installed EB GUIDE is referred to as `$GUIDE_INSTALL_PATH`.

  For example:

  `C:/Program Files/Elektrobit/EB GUIDE Studio 6.8`

- The directory for your EB GUIDE SDK platform is referred to as `$GTF_INSTALL_PATH`. The name pattern is `$GTF_INSTALL_PATH/platform/<platform name>`.

  For example:

  `C:/Program Files/Elektrobit/EB GUIDE Studio 6.8/platform/win64`

- The directory to which you save EB GUIDE projects is referred to as `$GUIDE_PROJECT_PATH`.

  For example:

  `C:/Users/[user name]/Documents/EB GUIDE 6.8/projects/`

- The directory to which you export your EB GUIDE model is referred to as `$EXPORT_PATH`.

1.5. Path conventions

EB GUIDE Studio supports handling of path names with more than 260 characters in Windows 10. A full path name can have more than 260 characters, however, single file names or directory names in the path still have a limit of 248 characters.

NOTE

Long path names in Windows 7

Windows 7 does not support handling of long path names. To use long path names, run EB GUIDE Studio on Windows 10. For more information on how to enable long path names in Windows 10, see the Windows 10 documentation.

2. Safe and correct use

2.1. Intended use

- EB GUIDE Studio and EB GUIDE GTF are intended to be used in user interface projects for infotainment head units, cluster instruments and selected industry applications.
- Main use cases are mass production, specification and prototyping usage depending on the scope of the license.

2.2. Possible misuse

WARNING

Possible misuse and liability

You may use the software only as in accordance with the intended usage and as permitted in the applicable license terms and agreements. Elektrobit Automotive GmbH assumes no liability and cannot be held responsible for any use of the software that is not in compliance with the applicable license terms and agreements.

- Do not use the EB GUIDE product line as provided by Elektrobit Automotive GmbH to implement human machine interfaces in safety-relevant systems as defined in ISO 26262/A-SIL.
- EB GUIDE product line is not intended to be used in safety-relevant systems that require specific certification such as DO-178B, SIL or A-SIL.

  Usage of EB GUIDE GTF in such environments is not allowed. If you are unsure about your specific application, contact Elektrobit Automotive GmbH for clarification at chapter 3, “Support“.

3. Support

EB GUIDE support is available in the following ways.

- For community edition:

  Find comprehensive information in our articles, blogs, and forums.

- For enterprise edition:

  Contact us according to your support contract.

When you look for support, prepare the version number of your EB GUIDE installation. To find the version number, go to the project center and click **Help**. The version number is located in the lower right corner of the dialog.

4. Introduction to EB GUIDE

EB GUIDE assists users in development process of the human machine interface (HMI). The EB GUIDE product line provides tooling and platform for graphical or speech user interfaces. The EB GUIDE product line is intended to be used in projects for infotainment head units, cluster instruments and selected industry applications. Main use cases are mass production, specification, and prototyping.

4.1. The EB GUIDE product line

The EB GUIDE product line comprises the following software parts:

- EB GUIDE Studio
- EB GUIDE TF

EB GUIDE Studio is the modeling tool on your PC. With EB GUIDE Studio you model the whole HMI functionality as a central control element that provides the user access to functions.

EB GUIDE TF executes an EB GUIDE model created in EB GUIDE Studio. EB GUIDE TF is available for development PCs and for different embedded platforms.

The EB GUIDE model that is created with EB GUIDE Studio and the exported EB GUIDE model that is executed on EB GUIDE TF are completely separated. They interact with each other, but cannot block one another.

4.2. EB GUIDE Studio

4.2.1. Modeling HMI behavior

The dynamic behavior of the EB GUIDE model is specified by placing states and by combining multiple states in state machines.

State machines
: A state machine is a deterministic finite automaton and describes the dynamic behavior of the system. In EB GUIDE Studio different types of state machines are available, for example a haptic state machine. Haptic state machines allow the specification of graphical user interfaces.

States
: States are linked by transitions. Transitions are the connection between states and trigger state changes.

4.2.2. Modeling HMI appearance

In EB GUIDE Studio you define the graphical user interface and the speech user interface of the EB GUIDE model.

Widgets
: To create a graphical user interface EB GUIDE Studio offers widgets. Widgets are model elements that define the look. They are mainly used to display information, for example text labels or images. Widgets also allow users to control system behavior, for example buttons or sliders. Multiple widgets are assembled to a structure, which is called view.

Spidgets
: To create a speech user interface EB GUIDE Studio offers spidgets. Spidgets are used to specify the fundamental parts of a speech dialog. Speech recognition as user input and speech synthesis as system output. A prompt spidget allows the modeling of text that is played through a text-to-speech synthesizer (TTS). A command spidget allows the modeling of grammars that describe what a speech recognizer understands. Related spidgets are grouped together through model elements. This group is called talk.

4.2.3. Handling data

The communication between the HMI and the application is implemented with the datapool and the event system.

Datapool
: The datapool is an embedded database that holds all data to be displayed and further internal information. Datapool items store and exchange data.

Event system
: Events are temporary triggers. Events can be sent to both parties to signal that something specific happens.

Application software can access events and the datapool through the API.

4.2.4. Simulating the EB GUIDE model

With EB GUIDE Studio you can test the functionality of your EB GUIDE model during simulation. You start the simulation with a mouse-click and can immediately experience the look and feel of your EB GUIDE model.

You interact with simulation using input devices like mouse, keyboard, or touch screen.

You can also control your EB GUIDE model with EB GUIDE Monitor and do the following:

- Change the displayed data by changing values of datapool items
- Simulate user input by firing events
- Track all changes in the log
- Start scripts

You can also use EB GUIDE Monitor as a stand-alone application.

4.2.5. Exporting the EB GUIDE model

To use the EB GUIDE model on the target device, you need to export the EB GUIDE model from EB GUIDE Studio and to convert it into a format that the target device understands. During the export, all relevant data is exported as a set of ASCII files.

4.3. EB GUIDE TF

EB GUIDE TF consists of the `GtfStartup` executable file and a set of libraries, which are required to execute an EB GUIDE model.

Depending on the project type selected in EB GUIDE Studio you execute:

- EB GUIDE GTF

  EB GUIDE Graphics Target Framework is the run-time environment executing a graphical HMI.

- EB GUIDE STF

  EB GUIDE Speech Target Framework is the run-time environment executing speech functionality in the HMI.

Most of the program code of EB GUIDE TF is platform-independent. The code can be ported to a new system very easily.

It is possible to exchange the complete HMI, simply by exchanging the EB GUIDE model files. It is not necessary to recompile EB GUIDE TF. The changed EB GUIDE model just needs to be re-exported from EB GUIDE Studio.

EB GUIDE TF uses the following platform abstractions:

- OS abstraction

  Platform dependencies of the operating system (OS) are encapsulated by the Operating System Abstraction Layer (`GtfOSAL`). Functionalities that EB GUIDE TF uses from the operating system are for example the file system or TCP sockets.

- GL abstraction

  Platform dependencies of the graphics subsystem are encapsulated by the renderer. An EB GUIDE model contains element properties such as geometry and lighting. The data contained in the exported EB GUIDE model is passed to the renderer for processing and output to a digital image. The renderer is the abstraction to the real graphic system on your hardware. EB GUIDE TF supports various renderers for different platforms.

- Audio abstraction

  The speech user interface requires access to audio hardware. The audio abstraction provides access to microphones and speakers. EB GUIDE STF implements speech recognition and text-to-speech synthesis. For this purpose EB GUIDE STF incorporates third-party speech engines.

5. Tutorial: Getting started

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

The following section gives you a short overview on HMI modeling with EB GUIDE Studio. It explains you how to start EB GUIDE Studio, how to create a project, how to model the behavior and appearance of an EB GUIDE model, and how to simulate an EB GUIDE model.

Approximate duration: 20 minutes.

5.1. Starting EB GUIDE

Starting EB GUIDE

Prerequisite:

- EB GUIDE is installed.

Step 1
In the Windows **Start** menu, click **All Programs**.

Step 2
In the **Elektrobit** menu, click the version you want to start.

EB GUIDE Studio starts. The project center is displayed.

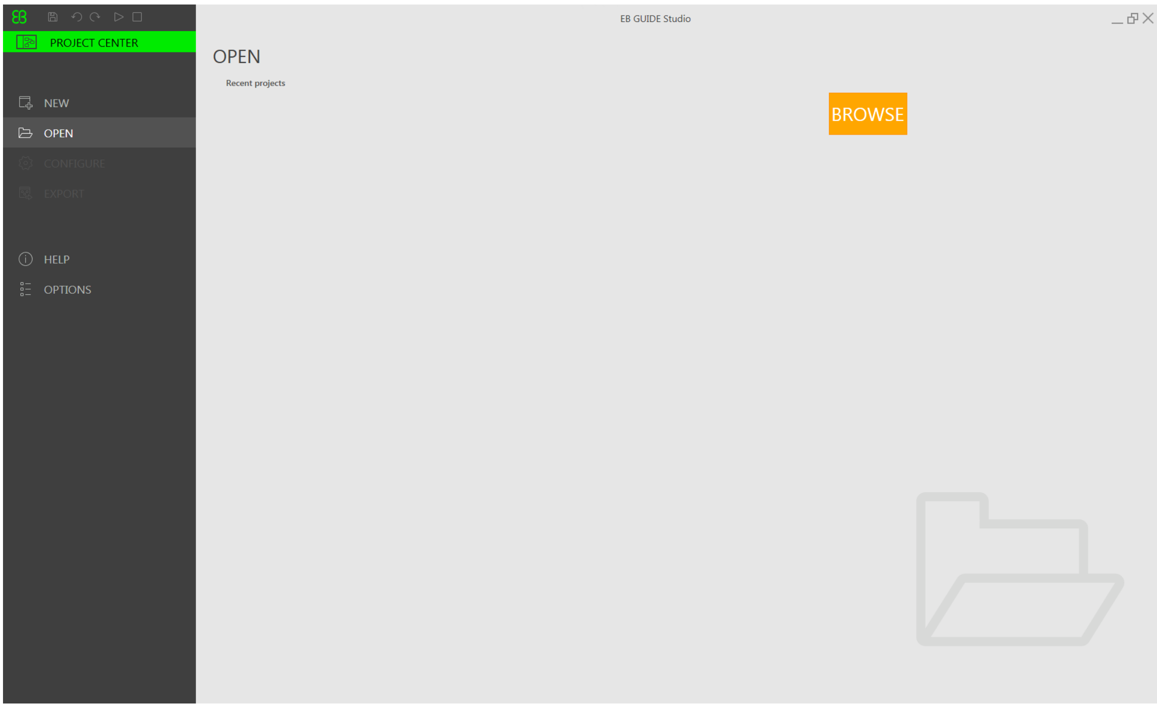


Figure 5.1. Project center

5.2. Creating a project

Creating a project

Prerequisite:

- EB GUIDE Studio is started.
- A directory `C:/temp` is created.

Step 1
In the navigation area of the project center, click **New**.

Step 2
In the content area, select the `C:/temp` directory as **Location**.

Step 3
Enter the project name `MyProject`.

Step 4
Click **Create**.

The project is created. The project editor opens and displays the empty project.

The **Main** state machine is added by default and displayed in the content area.

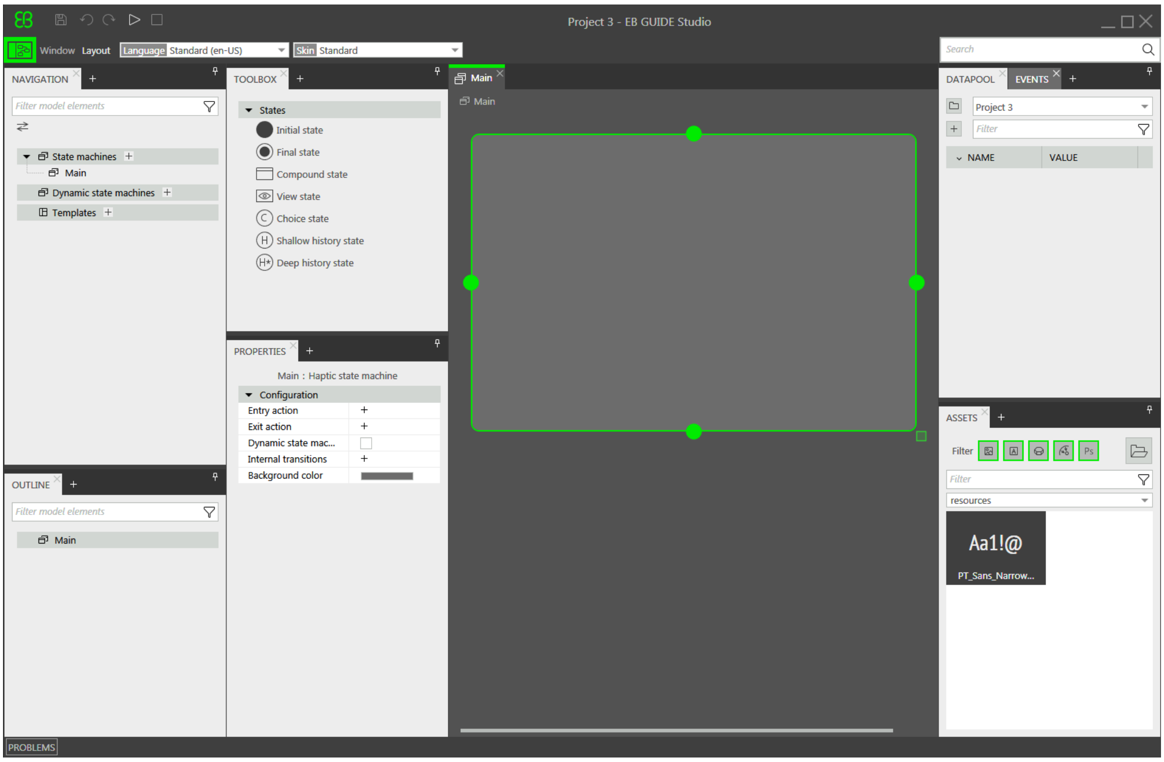


Figure 5.2. Project editor with **Main** state machine

5.3. Modeling HMI behavior

The behavior of your EB GUIDE model is defined by state machines. EB GUIDE uses a syntax similar to UML to do that.

In the following section, you learn how to model a state machine that displays a defined view on start-up and changes to a different view when a button is pressed.

Adding states to the state machine

EB GUIDE offers a variety of states. The following section shows three different states. An initial state defines the starting point of the state machine. A view state displays a view by default. And the final state of the state machine terminates the state machine.

Prerequisite:

- The project `MyProject` is created.

- The content area displays the **Main** state machine.

Step 1
Drag a view state from the **Toolbox** into the state machine.

Along with `View state 1`, a view is added to the EB GUIDE model.

Step 2
Repeat step 1.

`View state 2` is added.

Step 3
Drag an initial state from the **Toolbox** into the state machine.

Step 4
Drag a final state from the **Toolbox** into the state machine.

The four states you added to the **Main** state machine are displayed both in the content area as a state chart and in the **Navigation** component as a hierarchical tree view.

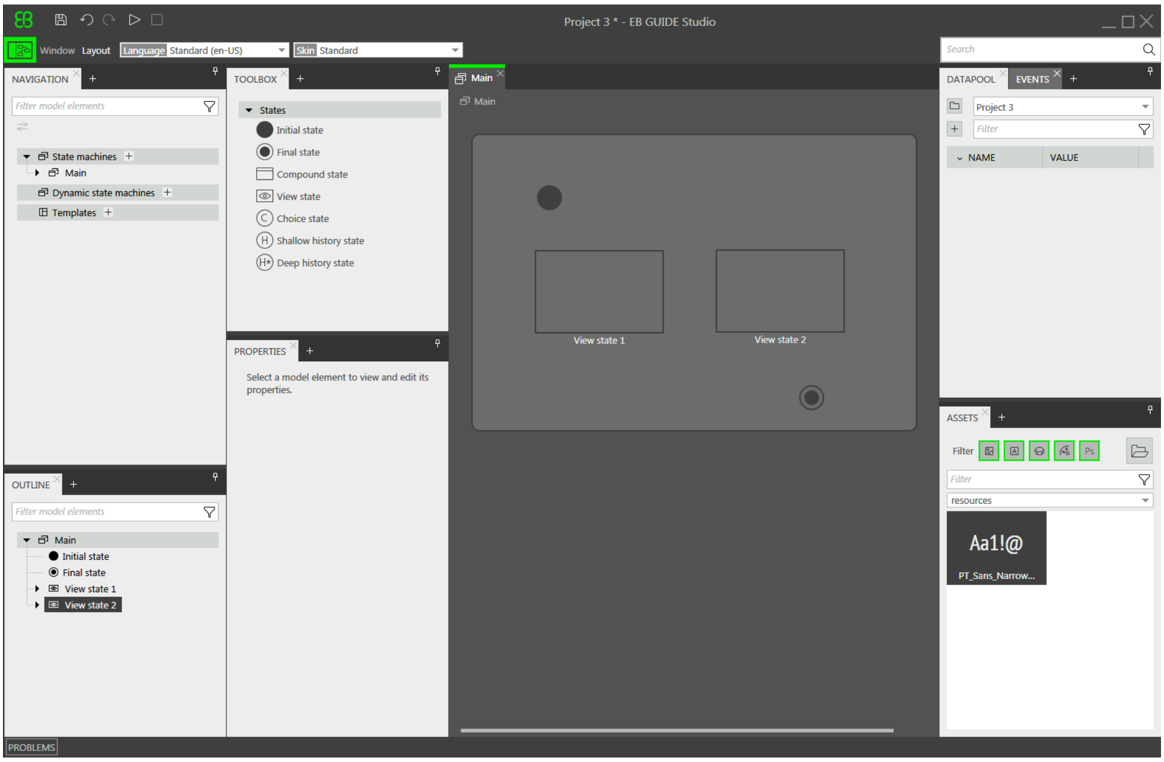


Figure 5.3. Project editor with states

Adding a transition

Transitions are the connection between states and trigger state changes. There are different transition types. The following section shows a default transition and an event-triggered transition.

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state, two view states, and a final state.

Step 1
Select the initial state as a source state for the transition.

Step 2
Click the green drag point and keep the mouse button pressed.

Step 3
Drag the mouse into the target state, `View state 1`.

Step 4
When the target state is highlighted green, release the mouse button.

A transition is created and displayed as a green arrow.

Step 5
Add a transition between `View state 1` and `View state 2`.

Select `View state 1` and repeat steps 2 - 4 for `View state 2` as the target state.

Step 6
Select the transition between `View state 1` and `View state 2`.

As a next step, you associate the transition to an event.

Step 7
Go to the **Properties** component, enter `Event 1` in the **Trigger** combo box and click **Add event**.

An event called `Event 1` is created and added as a transition trigger. Whenever `Event 1` is fired, the transition is executed.

Step 8
Add a transition between `View state 2` and the final state.

Select `View state 2` and repeat steps 2 - 4 for the final state as the target state.

Add a new event `Event 2` as a trigger.

At this point, your state machine resembles the following figure:

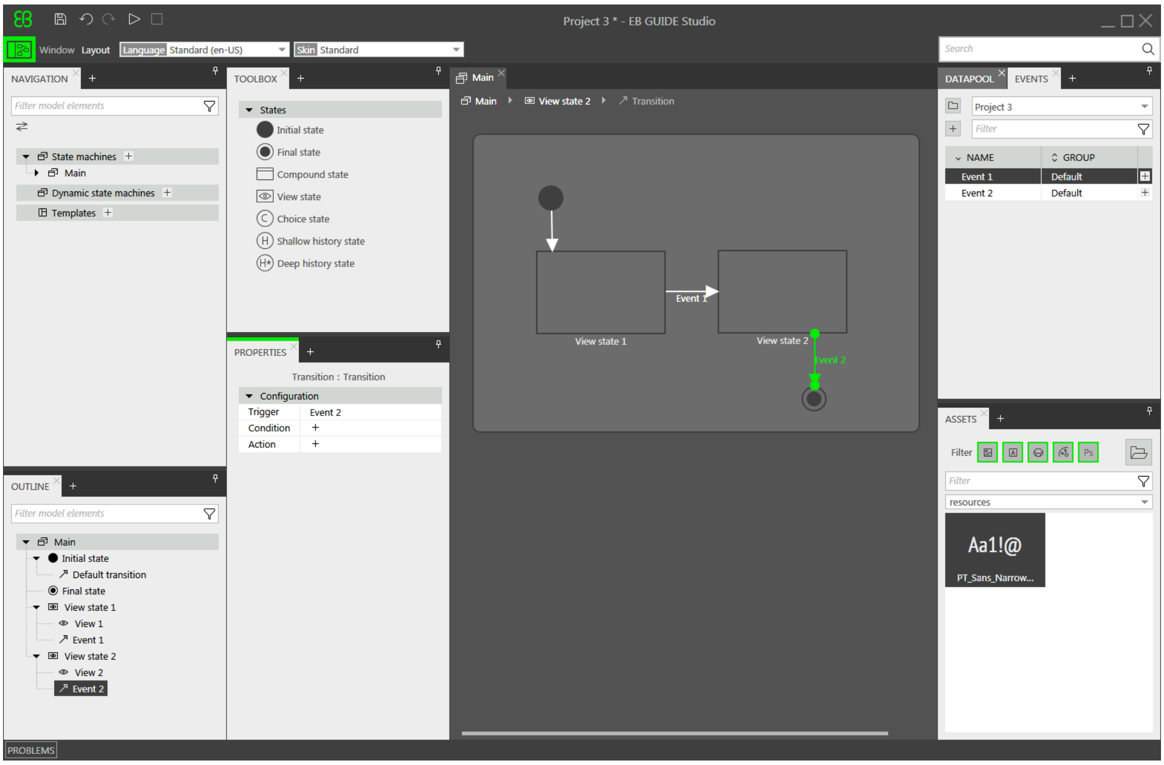


Figure 5.4. States linked by transitions with events

You have defined the behavior of a basic state machine.

5.4. Modeling HMI appearance

The state machine you created in the section above contains two view states. In the following section, you learn how to model a view.

Opening a view

Prerequisite:

- `View state 1` is added to the model.

Step 1
Double-click `View state 1`.

The content area displays `View 1`.

Adding a button to a view

With EB GUIDE Studio you have a variety of options to model the appearance of a view.

To give you one example, the next section shows you how to add a rectangle to a view. The rectangle reacts on user input and thus functions as a button.

Prerequisite:

- The content area displays `View 1`.

Step 1
Drag a rectangle from the **Toolbox** into the view.

Step 2
In the **Properties** component, go to the **Widget feature properties** category, and click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 3
Under **Available widget features**, expand the **Input handling** category, and select **Touch released**.

Click **Accept**.

The related widget feature properties are added to the **Properties** component.

Step 4
In the **Properties** component, from the `touchPolicy` drop-down list box select `Press then react`.

The rectangle reacts on touch input in the simulation mode.

Step 5
Go to the `touchShortReleased` property, and click **Edit**.

Step 6
Enter the following EB GUIDE Script:

```
function(v:touchId::int, v:x::int, v:y::int, v:fingerId::int)
    {
    fire_delayed 500, ev:"Event 1"()
    true
    }
```

If the rectangle is touched in the simulation mode, `Event 1` is fired after 500 milliseconds.

Step 7
Click **Accept**.

Step 8
In the **Properties** component, for the `fillColor` property select red.

Step 9
In the **Navigation** component, double-click `View 2`.

The content area displays `View 2`.

Step 10
Repeat steps 1-5.

Step 11
Enter the following EB GUIDE Script:

```
function(v:touchId::int, v:x::int, v:y::int, v:fingerId::int)
    {
    fire_delayed 500, ev:"Event 2"()
    true
    }
```

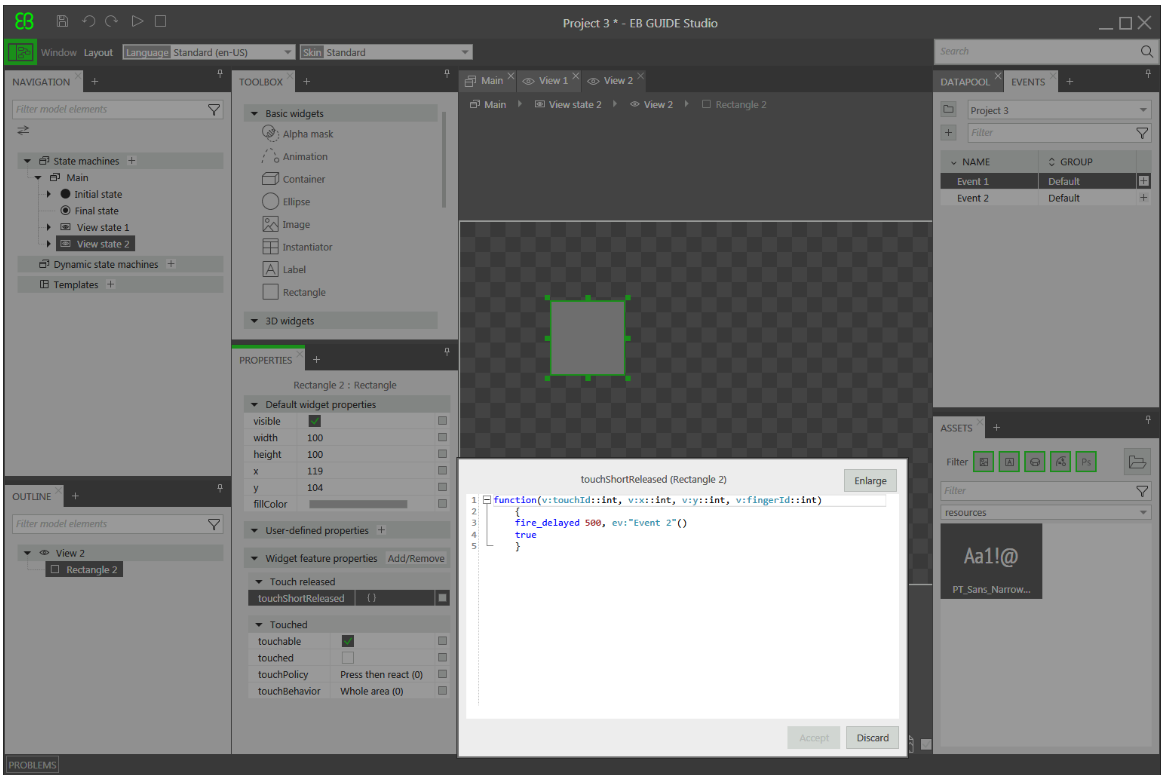


Figure 5.5. Widget property with an EB GUIDE Script

Step 12
Click **Accept**.

If the rectangle is touched in the simulation mode, `Event 2` is fired after 500 milliseconds.

Step 13
In the **Properties** component, for the `fillColor` property select blue.

5.5. Starting the simulation

EB GUIDE allows you to simulate your model on the PC before exporting it to the target device.

Starting the simulation

Step 1

To save the project, click 💾 in the command area.

Step 2

In the command area, click ▷.

The EB GUIDE model starts and shows the behavior and appearance you modeled.

First, `View 1` is displayed. A click on the red rectangle changes the screen to `View 2`. This is because the click fires `Event 1` and `Event 1` executes the transition from `View state 1` to `View state 2`.

Then, `View 2` is displayed. A click on the blue rectangle in `View 2` terminates the state machine. This is because the click fires `Event 2` and `Event 2` executes the transition from `View state 2` to the final state. The simulation window remains open. To stop the simulation, click ☐.

6. Background information

The topics in this chapter are sorted alphabetically.

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

6.1. 3D graphics

EB GUIDE Studio offers the possibility to use 3D graphics in your EB GUIDE project.

6.1.1. Supported 3D graphic formats

Only the renderers for OpenGL ES 2.0 or higher can display 3D graphics. The supported 3D graphic formats are COLLADA (`.dae`) and Filmbox (`.fbx`). For best results, use the Filmbox format.

6.1.2. Settings for 3D graphic files

To make 3D objects appear in a view in EB GUIDE Studio, you need to create the 3D graphic file with the following options:

- A perspective camera
- At least one object containing a mesh and at least one material
- At least one light source

TIP

Gamma correction for scene graphs

The `gamma` property allows to adjust the luminance output of the scene graph to match the luminance response of your monitor or display device for best visual results. The value must be higher than 0.0 and is set to 2.2 per default, which is suitable for most displays.

To create a 3D graphic file, use third-party 3D modeling software.

3D graphic files support a wide variety of additional content, which is listed below:

- 3D objects with positions, normals, binormals, tangents, and one texture channel
- Directional light sources
- Image-based light sources
- Ambient light sources
- Point light sources with constant, linear, quadratic, and cubic attenuation
- Spot light sources with cone angles, constant, linear, quadratic, and cubic attenuation
- Perspective camera support for fields of view, near plane, and far plane
- Textures: Emissive, diffuse, specular, normal map, opacity, reflection cube, and light map

> **TIP**
>
> **Setting up the 3D graphic file**
>
> Be aware that opacity maps need a valid alpha channel.

6.1.3. Import of a 3D graphic file

To add a 3D graphic to a view, you need to import a 3D graphic file using a scene graph. During import EB GUIDE Studio converts the 3D graphic file into a widget tree with scene graph as a parent node. For the content of the 3D graphic file, for example camera, material, meshes, EB GUIDE Studio creates the respective widgets. If the 3D scene of the imported 3D graphic file contains animations, EB GUIDE Studio imports these animations using the linear key value interpolation curve. This curve is applied to animated properties of type float, integer, or color. It is not possible to apply the linear key value interpolation curve in the same way the other animation curves are applied. This curve is only used to import animations of 3D graphics.

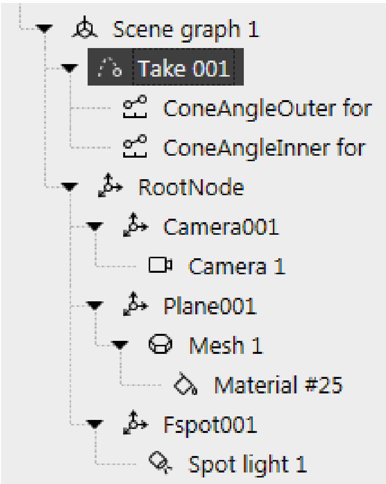


Figure 6.1. Example of a scene graph as displayed in the **Navigation** component

NOTE

Restrictions

Note the following:

- In EB GUIDE Studio only one material per mesh is allowed. If your 3D graphic has more than one material per mesh, during import EB GUIDE Studio creates additional mesh for each additional material.
- During the import of an `.fbx` file only a default material widget is created. If your 3D model has other types of materials, EB GUIDE Studio adds only a default material and its properties are set to default values. In EB GUIDE Studio, you can add other types of materials using PBR Phong material and PBR GGX material widgets.
- If in EB GUIDE Studio a mesh has several materials added, only the topmost material widget is rendered.

After importing a 3D graphic file, a subdirectory is created in the directory `$GUIDE_PROJECT_PATH/<project name>/resources`. The subdirectory is named after the imported `.fbx` file. Additionally date and time of creation are added to the name of the subdirectory.

Example 6.1.
Naming of the import directory

The 3D graphic file is called `car.fbx`. After importing a 3D graphic file in EB GUIDE Studio, in `$GUIDE_PROJECT_PATH/<project name>/resources` you find a subdirectory named `car_-20160102_103029`.

The subdirectory contains the following:

- Meshes as `.ebmesh` files
- Textures as `.png` or `.jpg` files

To use additional textures for your 3D graphics, copy a texture into `$GUIDE_PROJECT_PATH/<project name>/resources`. As texture use `.png` or `.jpg` images.

Import of multiple 3D graphics within one scene graph is possible.

After import, you can add, modify or delete 3D widgets.

For details, see section 6.22, “Widgets”, section 13.12.3, “3D widgets”, and section 13.13.8, “3D”.

For instructions, see section 8.1.3.1, “Adding a scene graph to a view”, and section 12.7, “Tutorial: Working with a 3D graphic”.

6.2. Animations

Animations bring motion and visual effects into your EB GUIDE model. In EB GUIDE, you can use animations for different use cases. You can animate widgets within a view and you can animate the transition from one view to another.

6.2.1. Animations for widgets

Animating a widget means moving a widget along a view. The movement is defined by curves. Therefore, the **Basic widgets** category in the **Toolbox** includes a widget called animation. To every animation you can add a set of curves, for example constant curve, linear interpolation curve, or sinus curve. A curve has a `target` widget property and describes the time-based change of the `target` property.

Each animation has one or more curves associated to it.

Among others, animating a widget can do the following:

- Move a widget within a view
- Change the size of a widget
- Gradually change the color of a widget

An animation is controlled by the EB GUIDE Script functions `f:animation_play`, `f:animation_pause`, `f:animation_cancel`, etc.

TIP

Concurrent animations

In EB GUIDE, animations are concurrent animations and curves are executed in parallel. This means that, if the curves of several animations use the same widget property as a target, the curves overwrite that `target` property's value concurrently.

For animation and curve properties, see section 13.12.2.2, “Animation”.

For instructions, see section 8.1.2.7, “Adding an animation”.

To create your own custom curve, see section 13.12.2.2.6, “Script curve”. For more background information, see section 6.2.3, “Script curve”.

6.2.2. Animations for view transitions

To animate a view transition means to define a moving or fading animation for entering or exiting a view state. A view change triggers such an animation.

You define view transition animations for view states and view templates. Every time you re-use the view template, the instance inherits the view transition animations of the template.

There are various types of view transition animations.

Table 6.1. Animation types

Animation type	Description
Entry animation	The animation is played when the view state with the animation is entered. The animation can only manipulate the widget properties and the widget feature properties of the added view.
Exit animation	Animation is played when the view state with the animation is exited. The animation can only manipulate the widget properties and the widget feature properties of the added view.
Change animation	The animation is played on view state change. The animation can manipulate the properties of source and destination view. The source view is the view state the animation is added to. The destination view can be another view state or view template.
Pop up on animation	Only available for view templates and dynamic state machine views. The animation is played when the respective dynamic state machine is activated (pushed dynamic state machine). The animation can manipulate the properties and the widget properties of the added view.
Pop up off animation	Only available for view templates and dynamic state machine views. The animation is played when the respective dynamic state machine is exited (popped dynamic state machine). The animation can manipulate the properties and the widget properties of the added view.

For animation properties in view templates, see section 13.12.1, “View”.

For instructions, see section 8.6, “Animating a view transition”.

6.2.3. Script curve

The default animation curves already provide plenty of customization options. Additionally EB GUIDE Studio gives you the option to define your own animation curve with the script curve feature. With this feature, you can define your own curve using EB GUIDE Script.

For a better understanding of a script curve animation, get the output of `v:diff` and `v:t_anim` in EB GUIDE Monitor. For instructions, see section 8.1.2.8, “Adding an animation with a script curve”. For a tutorial, see section 12.10, “Tutorial: Using script curves for animations”.

6.3. Application programming interface between application and model

EB GUIDE abstracts all communication data between an application and EB GUIDE TF in an application programming interface (API). An application is for example a media player or a navigation.

The API is defined by datapool items and events. Events are sent between HMI and application.

Example 6.2.
Contents of an API

- Event START_TRACK that is sent to the application and that contains the parameter `track` for the number of the track that should be played
- Event TRACK_STOPPED that is sent from the application to the HMI when the played track has ended
- The dynamic datapool item MEDIA_CURRENT_TRACK that is written by the application
- The dynamic datapool item MEDIA_PLAY_SPEED that defines the speed for playing and is set by the user in the HMI

6.4. Communication context

The communication context describes the environment in which communication occurs. An example for a communication context is a media or a navigation application which communicates with an HMI model. Changes made by one communication context are invisible to other communication contexts until the changes are published by the writer application and updated by the reader application.

A communication context is identified by a unique name and numerical ID (0...255) in the project configuration.

For instructions, see section 9.9, “Establishing external communication”.

6.5. Components of the graphical user interface

The graphical user interface of EB GUIDE Studio is divided into two components: the project center and the project editor. In the project center, you administer your EB GUIDE projects, configure options, and export EB GUIDE models for copying to the target device. In the project editor, you model HMI appearance and behavior.

6.5.1. Project center

The project center is the first screen that is displayed after starting EB GUIDE Studio. All project-related functions are located in the project center. The project center consists of two parts: the navigation area and the content area.

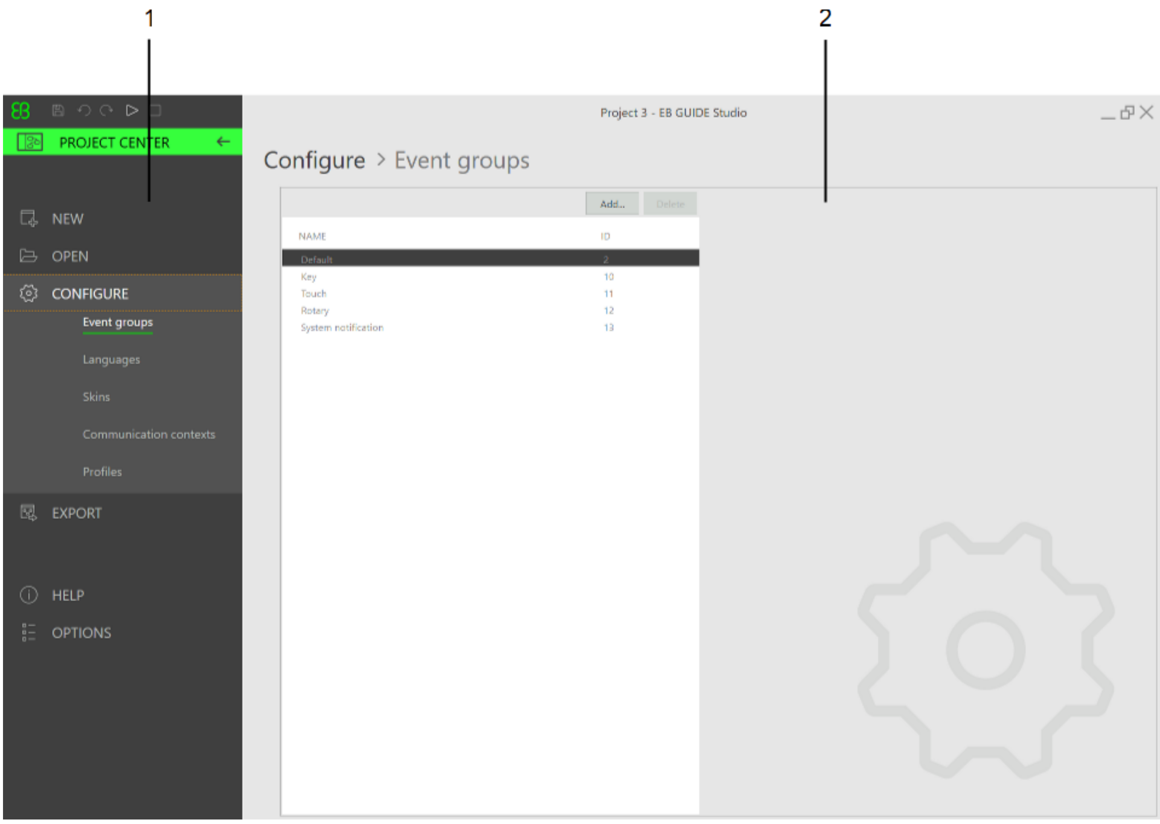


Figure 6.2. Project center with navigation area (1) and content area (2)

6.5.1.1. Navigation area

The navigation area of the project center consists of function tabs such as **Configure** or **Export**. You click a tab in the navigation area and the content area displays the corresponding functions and settings.

6.5.1.2. Content area

The content area of the project center is where project management and configuration takes place. For example, you select a directory to save a project or define the start-up behavior for your EB GUIDE model. The appearance of the content area depends on the tab selected in the navigation area.

6.5.2. Project editor

After creating a project, the project editor is displayed. In the project editor you model the behavior and the appearance of the HMI: you model state machines, create views, and manage events and the datapool. The project editor consists of the following areas and components. All components of the project editor can either be docked or floating and placed at any position of the project editor except the content area.

Figure 6.3. Project editor with its areas and components

1 **Toolbox** component

2 **Properties** component

3 Command area

4 Content area

5 **Datapool** component

6 Search box

7 **Events** component

8 **Assets** component

9 **Problems** component

10 **Outline** component

11 **Navigation** component

12 **Namespaces** component

6.5.2.1. Navigation component

The **Navigation** component displays the model elements such as states, views, animations and transitions of your EB GUIDE model as a hierarchical structure and allows you to navigate to any element. Double-clicking a model element displays the model element in the content area.

The **Navigation** component gives you an overview of all graphical and non-graphical elements of the EB GUIDE model and reflects the state machine hierarchy.

It is also where you add elements to your EB GUIDE model, such as state machines, dynamic state machines, and templates. You can add elements from the **Toolbox** such as widgets and animations using a drag-and-drop operation.

NOTE

Filter box

At the top of the component you find a filter box to search for any element within the component.

Clicking an element in the component and pressing **F3** starts a reference search: It opens the search results window and lists all occurrences of the selected element in the EB GUIDE model.

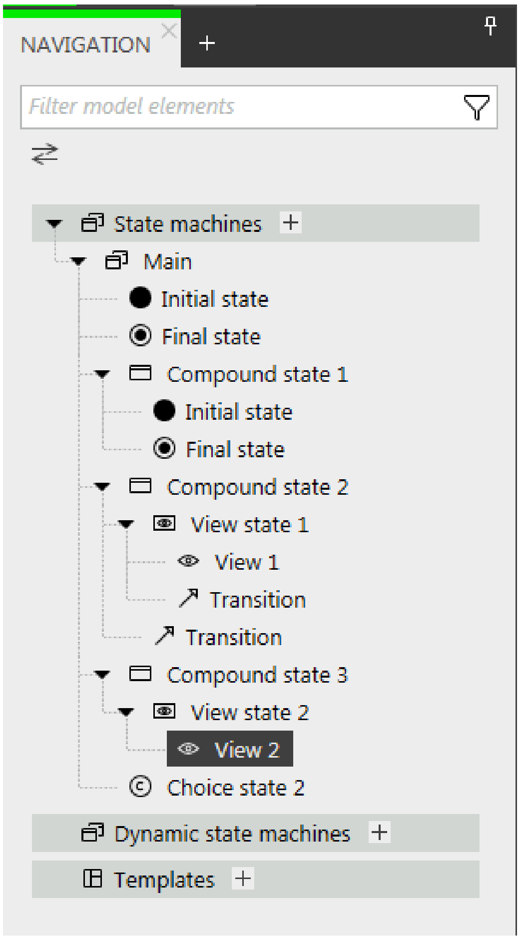


Figure 6.4. **Navigation** component in project editor

6.5.2.2. Outline component

Displays only the structure and model elements contained in the tree part selected in the **Navigation** component or in the editor component currently displayed in the content area.

NOTE

Filter box

At the top of the component you find a filter box to search for any element within the component.

Clicking an element in the component and pressing **F3** starts a reference search: It opens the search results window and lists all occurrences of the selected element in the EB GUIDE model.

6.5.2.3. Toolbox component

All tools you need for modeling are available in the **Toolbox** component, also referred to as **Toolbox**. Depending on the element that is displayed in the content area, the **Toolbox** offers a different set of tools, which can be dragged into the content area or the **Navigation** component. The **Toolbox** can for example contain the following:

- If the content area displays a state machine, the **Toolbox** contains states you can add to the state machine.
- If the content area displays a view, the **Toolbox** contains widgets you can arrange in the view.
- If the content area displays a scripted value property, the **Toolbox** contains EB GUIDE Script functions you can insert.

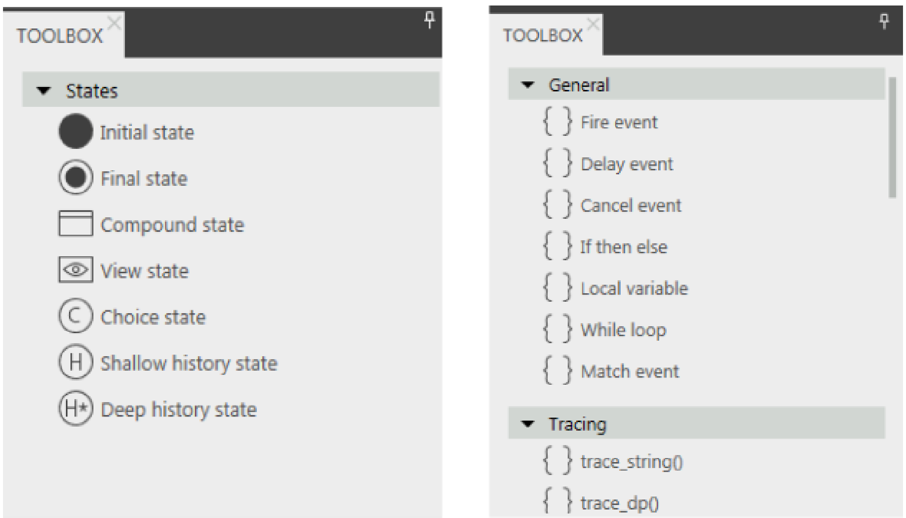


Figure 6.5. Toolbox in project editor

6.5.2.4. Properties component

The **Properties** component displays the properties of the selected model element, for example of a widget or a state. The properties are grouped by categories and can be edited in the **Properties** component.

Clicking a property and pressing **F3** starts a reference search: It opens the search results window and lists all occurrences of the selected property in the EB GUIDE model.

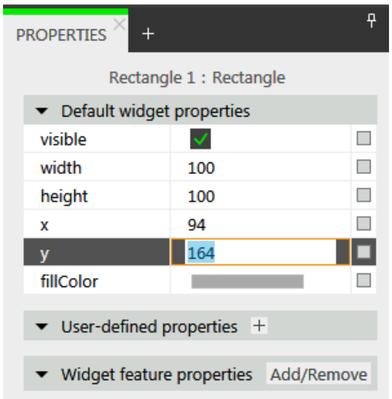


Figure 6.6. **Properties** component displaying properties of a widget

6.5.2.5. Content area

What is displayed in the content area depends on the selection in the **Navigation** component. To edit a model element, you double-click the model element in the **Navigation** component and the content area displays it. For example, you model the states of a state machine, you arrange widgets in a view, or you edit an EB GUIDE Script in the content area.

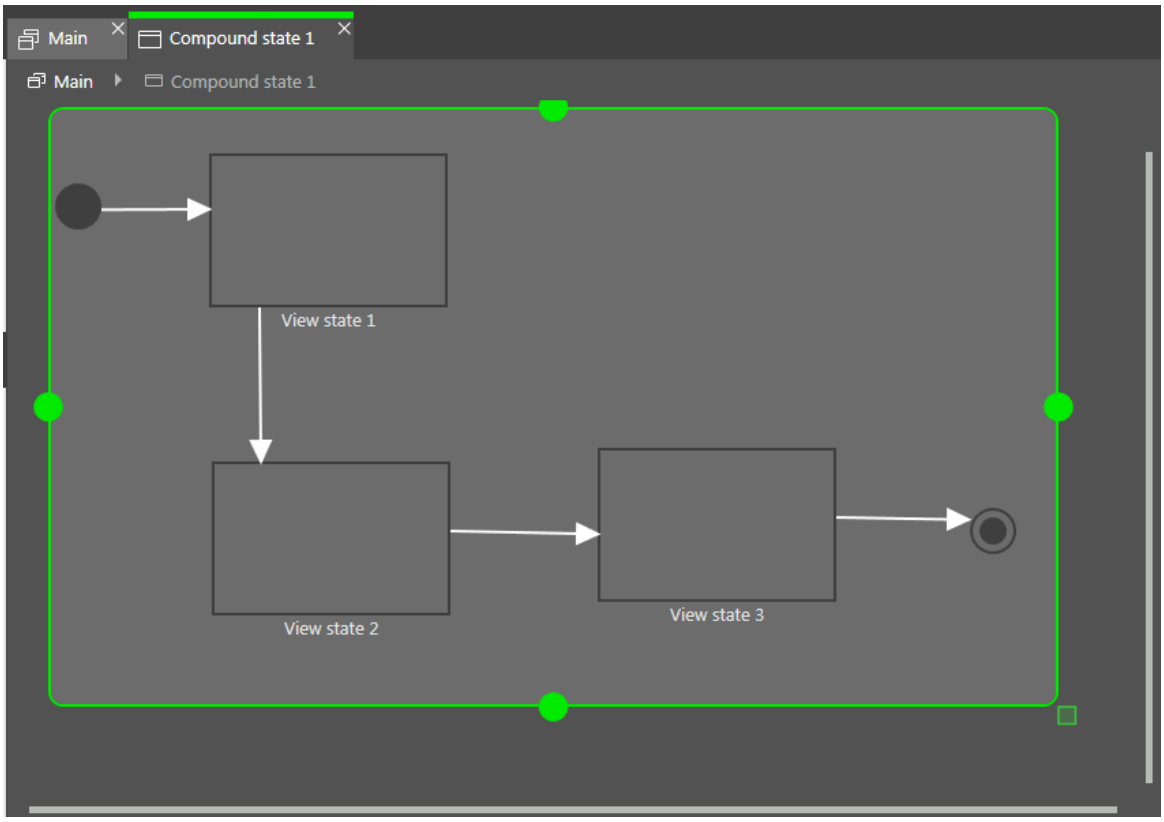


Figure 6.7. Content area in project editor

If in the content area you have an open view and the view contains an animation, the **Animation** editor is opened. In the **Animation** editor you can add curves to widget properties. You can also edit the `delay` and `duration` properties of the curves by moving the handles in the preview.

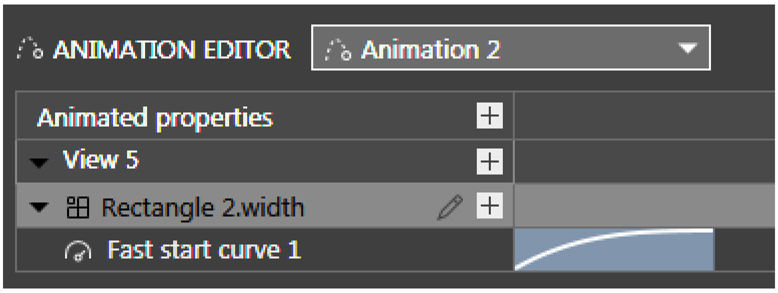


Figure 6.8. Animation editor

To start a reference search, click a state or a widget in the content area and press **F3**. The search results window opens and lists all occurrences of the selected state or widget in the EB GUIDE model.

6.5.2.6. Events component

Here you can add events to a selected namespace and edit the properties such as **Name**, **Group**, **Type** and **Parameter name** in the event table.

NOTE

Filter box

At the top of the component you find a filter box to search for any element within the component.

Clicking an element in the component and pressing **F3** starts a reference search: It opens the search results window and lists all occurrences of the selected element in the EB GUIDE model.

TIP

Multiple selection of model elements

To select multiple elements, hold down **Ctrl** while clicking the elements you want to select. You can also hold down **Shift** while clicking the respective elements or use the **Up arrow** or **Down arrow** keys.

In the **Events** component, you can switch between the following view options:

- Only the events in the selected namespace are shown.
- The events in the selected namespace and its sub-namespaces are shown.
- Events in all namespaces are shown.

6.5.2.7. Datapool component

Here you can add datapool items to a selected namespace and edit the properties such as **Name** and **Value**. You can also add a link to a datapool item, convert a value to script, and add a language and skin support.

NOTE

Filter box

At the top of the component you find a filter box to search for any element within the component.

Clicking an element in the component and pressing **F3** starts a reference search: It opens the search results window and lists all occurrences of the selected element in the EB GUIDE model.

TIP

Multiple selection of model elements

To select multiple elements, hold down **Ctrl** while clicking the elements you want to select. You can also hold down **Shift** while clicking the respective elements or use the **Up arrow** or **Down arrow** keys.

In the **Datapool** component, you can switch between the following view options:

- Only the datapool items in the selected namespace are shown.
- The datapool items in the selected namespace and its sub-namespaces are shown.
- Datapool items in all namespaces are shown.

6.5.2.8. Assets component

Here you can add resources such as images, fonts, `.ebmesh` and `.psd` files. All resource files located in the `$GUIDE_PROJECT_PATH/<project name>/resources` directory and its subdirectories are displayed in the preview area of the component. You can add a resource to the model using a drag-and-drop operation.

NOTE

Filter box

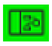
At the top of the component you find a filter box to search for any element within the component.

6.5.2.9. Namespaces component

Here you can create, move and delete namespaces. A root namespace is added by default.

6.5.2.10. Command area

In the command area, you find:

- The  button, which opens the project center
- Search box to search for elements of the model and jump to them
- Further menus

Search box

Model elements can be found with the help of the search box. Use the search box as follows:

- Click the search box or use the **Ctrl+F** shortcut to jump into the search box. Enter the name of the model element to be searched.

  Alternatively, you can select a model element and press **F3**. Search results window opens and search results are shown.
- Jump to a model element by double-clicking it in the hit list.

The left part of the search results window lists the model elements that are found grouped by categories. Use the filter buttons above to show or hide categories. Select a model element to get a preview or to see the properties of the model element in read-only mode.

When closing the search results window the last search term, filter settings and corresponding hit list are saved and shown when the search results window is opened again. When model elements were changed in between, the search needs to be executed again.

The search is not case sensitive.

When using the asterisk `*` for wildcard search the following rules apply:

- Search entry *t* returns all element names containing a t.
- Search entry **t* returns all element names ending with t.
- Search entry *t** returns all element names starting with t.

You can search for the following model element categories.

Table 6.2. Categories in search box

Category	**Description**
States	The hit list also shows the child elements of the states found.
Views	The hit list also shows the child elements of the views found.
Templates	The hit list also shows the child elements of the templates found.
Events	The preview shows the properties of the event.

Category	Description
Datapool items	The preview shows the properties of the datapool item.
Scripts	The preview shows the content of the scripts containing the text. The found text is highlighted.
Properties	The preview shows the widget to which the property belongs.
View transition animations	The preview shows the view to which the view transition animation belongs.

6.5.2.11. Problems component

In the **Problems** component you can check if your model is valid. It displays possible errors and warnings of the currently opened EB GUIDE model. To jump directly to the part where the problems occur, double-click the description.

6.5.2.12. VTA component

In the **VTA** (view transition animations) component you can edit view transition animations of a view state or a view template. You can select different animation types. All relevant animation types such as the first one of each animation type with condition true start at the same time.

The **VTA** component is not shown in the default layout. To open the **VTA** component, select **VTA (view transition animations)** in the **Layout** menu.

6.5.3. Dockable component

You can dock all components of the project as tabs or undock as floating components. You can drag a component as floating component to any part of the project center except the content area.

The arrows of the docking control help you to select a docking location and the live preview shows you how the layout is going to look like.

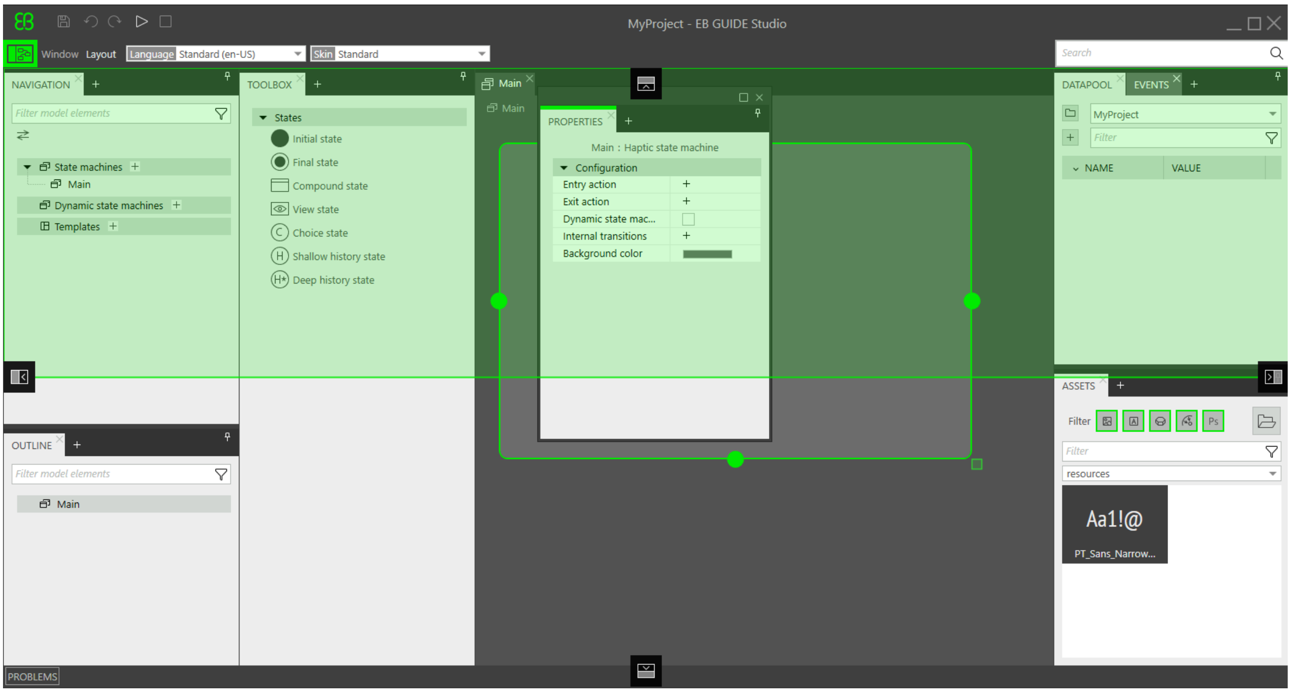


Figure 6.9. Docking control and live preview

NOTE

Default layout

To restore the default layout, go to the command area and select **Layout** > **Reset to default layout**.

NOTE

Auto-hide

To gain more space in the project editor, you can hide components.

- To hide a component or a component group, click the pin symbol.
- To display a hidden component, hover over the tab with the mouse and click the pin symbol again.

6.5.4. EB GUIDE Monitor

EB GUIDE provides the tool EB GUIDE Monitor to observe and control an EB GUIDE model during the simulation. EB GUIDE Monitor includes mechanisms for the communication with datapool, the event system, and the state machines of the EB GUIDE model.

EB GUIDE Monitor is started automatically in EB GUIDE Studio during the EB GUIDE model simulation. You can also use EB GUIDE Monitor as a stand-alone application.

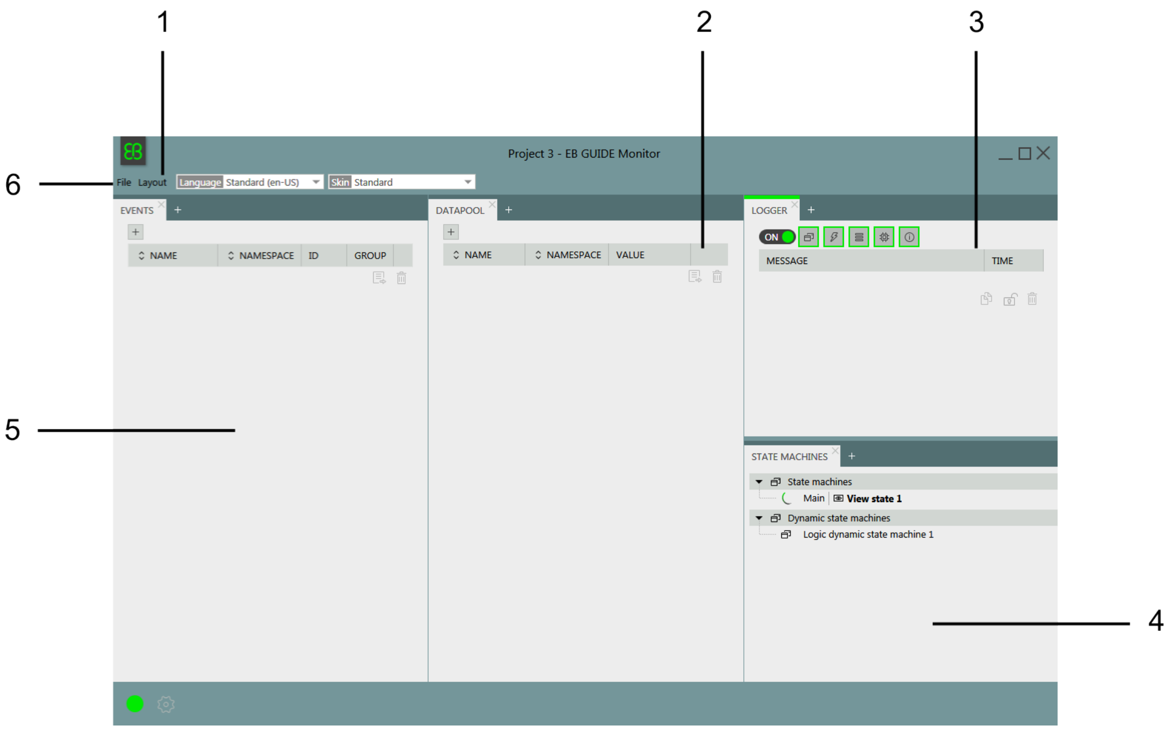


Figure 6.10. EB GUIDE Monitor with default layout

1 **Layout** menu

2 **Datapool** component

3 **Logger** component

4 **State machines** component

5 **Events** component

6 **File** menu

EB GUIDE Monitor contains the following components:

- In the **Events** component you can search and fire events. If an event has parameters, you can change the parameters and then fire this event.
- In the **Datapool** component you can search for datapool items and change their values.
- In the **Logger** component all changes, information messages, errors, and warnings are tracked. At the top of the component you find filter buttons to filter entries within the component. To enable or disable the auto-scrolling functionality, check or clear the auto-scrolling check box at the bottom of the component.

NOTE

Editing the number of log messages

In the `$INSTALL_PATH\tools\monitor\Monitor.exe.config` configuration file, you can define the behavior of EB GUIDE Monitor when the logger reaches its memory limit. For this change the following parameters:

- `limit` defines the number of entries that are displayed in the **Logger** component.
- `removeCount` defines the number of entries that are going to be deleted when the limit is reached.

- In the **State machines** component the currently active state and state machine are shown.
- In the **Scripting** component you can start scripts and see the output script messages. Note that the **Scripting** component is not in the default layout. To add the component, click **Layout** > **Scripting**.

You can rearrange components and add new components according to your project's needs. You can also dock and undock components within the EB GUIDE Monitor window.

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Monitor window to default layout by selecting **Layout** > **Reset to default layout**.

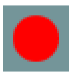
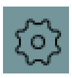
TIP

Resetting the size of the EB GUIDE Monitor window

The size of the EB GUIDE Monitor window and also its position on the screen are stored for each individual EB GUIDE project separately in `C:\<user>\AppData\Local\Temp\eb_guide_simulation_export\<project>`.

To reset the size and the position to the default values, delete `monitor_layout.xml` and `monitor_settings.xml` in this folder. During the next simulation, the new `monitor_layout.xml` and `monitor_settings.xml` files are created with the default size and position values.

In the left bottom corner of the EB GUIDE Monitor window you find the following buttons for the connection status.

Button	Status
	EB GUIDE Monitor is connected. If you click the button, EB GUIDE Monitor disconnects.
	EB GUIDE Monitor is disconnected. If you click the button, EB GUIDE Monitor connects.
	EB GUIDE Monitor is disconnected.

Button	Status
	If you click the button, you can configure the connection settings of EB GUIDE Monitor.

It is also possible to change the language and the skin using the drop-down boxes in the command area.

For instructions, see section 10.9, “Working with EB GUIDE Monitor”.

For the EB GUIDE Monitor API, see `$GUIDE_INSTALL_PATH/doc/monitor/monitor_api.chm`.

6.6. Datapool

6.6.1. Concept

During the execution, a model communicates with different applications. To enable the communication, your EB GUIDE model has to provide an interface. The datapool is an interface which allows access to datapool items to exchange data. Datapool items store values and communicate between HMI and applications. Datapool items are defined in the EB GUIDE model.

6.6.2. Datapool items

Datapool items are model elements that are used to do the following:

- Send data from the applications to the HMI
- Send data from the HMI to the applications
- Store data which is only used in either HMI or applications

For instructions, see section 9.6, “Adding a datapool item”.

To channel communication, you use writer and reader applications.

Internal communication is used to store data. Using two different applications establishes external communication.

For instructions, see section 9.9, “Establishing external communication”.

6.6.3. Windowed lists

The EB GUIDE product line supports the concept of windowed lists. The windowed list operating mode is often used to reduce memory consumption for the display of large lists, for example all MP3 titles in a directory. Those lists are typically provided by one application, for example media application, and are only partially displayed by another application, for example HMI.

The writer application defines a virtual list length and a number of windows, which possibly contain only parts of the list. The reader application reads data only from locations that are covered by windows. Reading from other locations fails. In such a use case, the reader application has to inform the writer application about the currently required parts of the list. For example, HMI can make application calls that provide the current cursor position within the complete list.

**Example 6.3.
Windowed list**

The MP3 title list of an audio player device has 1,000,000 elements. The HMI has to display this list on three different displays in parallel: head unit display, cluster instrument display, and head-up display.

Each display is controlled separately, has a different number of display lines and has a different cursor position within the complete list.

Whenever one of the three cursors moves, the HMI sends the new position asynchronously to the media application through an event. The media application provides a list with three windows. Each of the three windows is associated to one of the three displays. Window updates delay a little bit after the cursor moves. Therefore it is advisable to use window positions and window sizes which cover an extended range around the lines that are shown by the specific display.

6.7. EB GUIDE model and EB GUIDE project

An EB GUIDE model is the sum of all elements that describe the look and behavior of an HMI. It is built entirely in EB GUIDE Studio. You can simulate the EB GUIDE model on your PC.

To execute an EB GUIDE model on a target device, you export the EB GUIDE model and copy the resulting binary files to the target device.

An EB GUIDE project consists of an EB GUIDE model and settings that are needed for modeling. It includes project-specific options, extensions, resources, and, for graphical projects, the description of a haptic dialog.

An EB GUIDE project contains objects that are configured and linked within an EB GUIDE model. These objects are called EB GUIDE model elements. Examples for EB GUIDE model elements are as follows:

- Datapool item
- Event
- State
- State machine

- Widget
- Resource
- Language

6.7.1. Storage format

The EB GUIDE project is stored in multiple files of the EB GUIDE-proprietary file format. The file format is represented by two file extensions:

- `.ebguide` for the EB GUIDE project file
- `.gdata` for all other project files

The EB GUIDE Studio storage format is defined by the following pseudo-EBNF syntax:

```
INT = [0-9]+ ;

HEXINT = '0' ( 'x' | 'X' ) [a-fA-F0-9]+

FLOAT = <as represented in the C# specification> ;

STRING = " " ; //escape characters are supported as specified in MSDN

SUFFIX = [a-zA-Z_-][a-zA-Z0-9_-]* ;

COLOR = [a-fA-F0-9]{8} ;

IDENTIFIER = ( '_' | [a-zA-Z] ) ( [a-zA-Z] | [0-9] | '_' | '$' | '.' )* ;

file = header object ;

header = 'EBGUIDE' INT '.' INT '.' INT '.' INT SUFFIX ';' ;

object = type '(' objectId ')' '{' propertyList '}' ;

type = identifier [ '<' type { ',' type } '>' ] ;

property = identifier ':' value ;

value = bool
      | int
      | float
      | string
      | color
      | object
```

```
      | externalObject
      | nullObject
      | objectReference
      | propertyReference
      | list ;

string : STRING { '\' STRING }  ;

int = [ '+' | '-' ] INT
    | HEXINT ;

color = '#' COLOR ;

float = [ '+' | '-' ] FLOAT ;

bool = 'true' | 'false' ;

externalObject = '(' objectId ')' ;

nullObject = type '(' 'none' ')' ;

objectReference = '@' objectId '(' type ')' ;

propertyReference = identifier '@' objectId '(' type ')' ;

list = type '[' [ value { ',' value } ] ']' ;

identifier = IDENTIFIER | STRING ;

objectId = GUID ; //encoded as hex digits in
                  //the XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX format
```

The EB GUIDE project directory `$GUIDE_PROJECT_PATH/<project name>/` contains the following:

- Directories with files that are relevant for configuration, namespaces, and state machines
- The `resource` directory with the project-specific resources. For more information, see section 6.15, “Resource management”.
- The `.gdata` files with data for contexts, datapool, event system, languages, and skins
- The `.txt` files with information about the loading errors, migration or import messages for `.xliff` files, `.psd` files, and `.fbx` files

6.7.2. Validation criteria for EB GUIDE project

EB GUIDE Studio performs two types of validation check for an EB GUIDE project.

6.7.2.1. Validation while opening an EB GUIDE project

When you open your EB GUIDE project, EB GUIDE Studio performs several structural verifications, for example as follows:

- If the `.ebguide` project file does not exist, or several `.ebguide` files are located in the same folder
- If the object IDs are duplicated
- If child objects are missing within the EB GUIDE project
- If there are duplicate property names
- If values of list items are inconsistent
- If the EB GUIDE Studio version number in the `.gdata` files does not correspond with the the EB GUIDE Studio version number in the `.ebguide` file
- If an unknown type is referenced

If one of these criteria is met, the EB GUIDE project cannot be opened and a log file with the type of error and the position of this error inside the project files is created in `$GUIDE_PROJECT_PATH/<project name>/<project name>_LoadingErrorLog.txt`.

For instructions on how to open an EB GUIDE project, see section 10.2, “Opening a project”.

6.7.2.2. Validation using the Problems component

When the EB GUIDE project is already opened, you can validate the EB GUIDE model in the **Problems** component. The errors are, for example, as follows:

- If the EB GUIDE Script usage is not valid
- If a default transition is missing
- If a target of a linked item is missing

If any errors are found, you cannot simulate and export the EB GUIDE model.

For instructions on how to validate an EB GUIDE model, see section 10.4, “Validating and simulating an EB GUIDE model”.

6.8. Event handling

6.8.1. Event system

The event system is an asynchronous mechanism for communication within or between applications.

The EB GUIDE event system delivers all events exactly in the order they were sent. There is no pre-defined order for delivering an event to different subscribers.

6.8.2. Events

An event in EB GUIDE is a model element that has a unique event ID and belongs to an event group. The event ID is used by EB GUIDE GTF to send and receive the event.

Example 6.4.
Usage of events

In an HMI that has a rotary button or a set of hard keys, for example left, right, up, down, enter, it may not be clear with which element a user should interact next. Therefore, these systems typically highlight the currently active display element. For example, a `YES` button is marked as active with a colored border. In EB GUIDE Studio you model this highlighting feature using the **Focused** widget feature. The element that is currently focused, i.e. the `focused` property is set to `true`, is the element that is also active. Also the parents of this element, which form the focus path, are also active. If the focused element cannot handle a key or rotary input, the input is processed along the focus path in backwards direction, i.e. towards the root element. If one of the elements of the focus path handles the input, the focus is considered as processed.

In an HMI with touch input, the interaction is done with an element at a certain position. For example, when you press a `YES` button on a touch screen, the input is not ambiguous. The reason for this is that based on the location on the display which has been pressed, the system knows that the interaction was done with the `YES` button.

Event group IDs between 0 and 65535 are reserved for the internal use within the EB GUIDE product line. Exceptions to that are the event groups that are listed in the following table.

Table 6.3. Allowed event groups and IDs

<table>
<tr><th>Event group</th><th>ID</th><th>Details</th></tr>
<tr><td>Default</td><td>2</td><td>Internal group, i.e. only the core where the scene is running receives the event</td></tr>
<tr><td>Key input events</td><td>10</td><td>You can configure the following parameters:
<table>
<tr><th>Parameter</th><th>Type</th><th>Details</th></tr>
<tr><td><code>display</code></td><td>integer</td><td>The scene ID that should receive the input event</td></tr>
<tr><td><code>status</code></td><td>integer</td><td>0: key press
1: key release</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Event group</th><th>ID</th><th>Details</th></tr>
<tr><td></td><td></td><td><table><tr><th>Parameter</th><th>Type</th><th>Details</th></tr><tr><td></td><td></td><td>2: key unicode</td></tr></table></td></tr>
<tr><td>Touch input events</td><td>11</td><td>You can configure the following parameters:<table><tr><th>Parameter</th><th>Type</th><th>Details</th></tr><tr><td><code>display</code></td><td>integer</td><td>The scene ID that should receive the input event</td></tr><tr><td><code>status</code></td><td>integer</td><td>0: touch press
1: touch move
2: touch release
3: proximity move
4: touch new
5: touch gone</td></tr><tr><td><code>x</code></td><td>integer</td><td>The x-coordinate of the touch event</td></tr><tr><td><code>y</code></td><td>integer</td><td>The y-coordinate of the touch event</td></tr><tr><td><code>fingerId</code></td><td>integer</td><td>The number to track multiple parallel touch positions for multi-touch support</td></tr></table></td></tr>
<tr><td>Rotary input events</td><td>12</td><td>You can configure the following parameters:<table><tr><th>Parameter</th><th>Type</th><th>Details</th></tr><tr><td><code>display</code></td><td>integer</td><td>The scene ID that should receive the input event</td></tr><tr><td><code>increment</code></td><td>integer</td><td>The increment value</td></tr></table></td></tr>
<tr><td>System notification events</td><td>13</td><td>Used for system events like screen rotation, e.g. on Android, or lifecycle management, e.g. shutdown</td></tr>
</table>

The following figure shows how you can model the touch, key, and rotary events in EB GUIDE Studio.

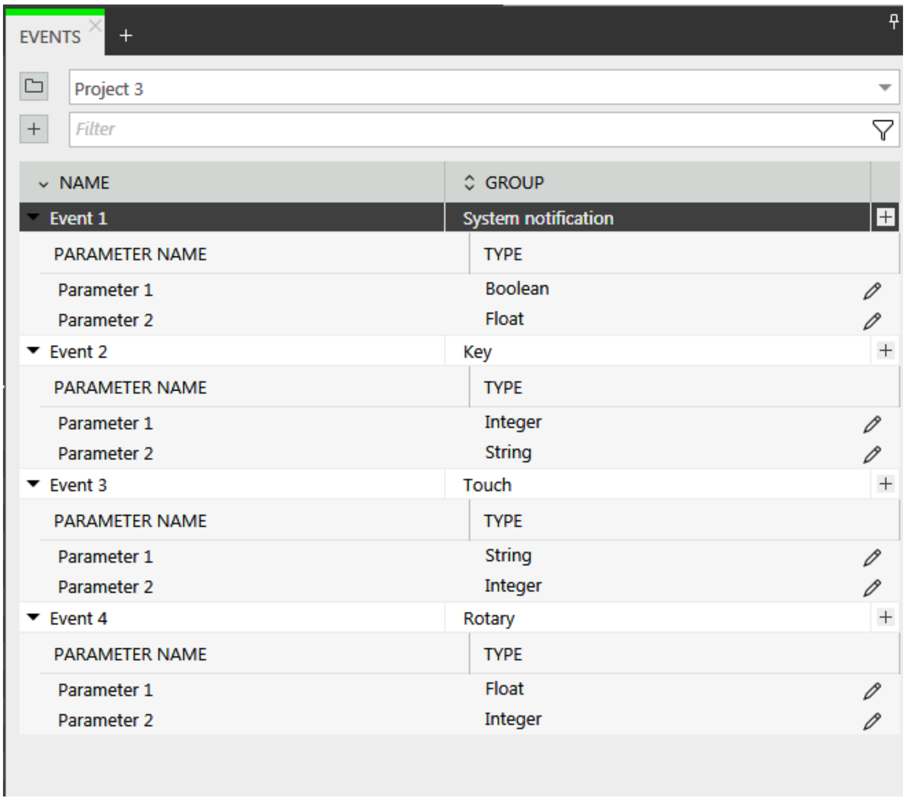


Figure 6.11. Example of event groups and event IDs

The remaining range of group IDs is available for customer-specific applications.

For instructions, see the following:

- section 9.1, “Adding an event”
- section 9.3, “Addressing an event”

For references, see section 13.5, “Events”.

6.9. Extensions

6.9.1. EB GUIDE Studio extension

An EB GUIDE Studio extension is a supplement to EB GUIDE Studio and is valid for all EB GUIDE models. The EB GUIDE Studio extension does not concern EB GUIDE GTF.

For more information, contact chapter 3, “Support“.

6.9.2. EB GUIDE GTF extension

An EB GUIDE GTF extension is a supplement to EB GUIDE GTF which provides additional features in EB GUIDE Studio, but is only valid for one EB GUIDE model. The EB GUIDE GTF extension is based on the EB GUIDE GTF.

Typical EB GUIDE GTF extensions are:

- New widget features
- New EB GUIDE Script functions

EB GUIDE GTF extensions are dynamic link library (`.dll`) or shared object (`.so`) files.

Place the EB GUIDE GTF extension, including their third party libraries in the following directory:

```
$GUIDE_PROJECT_PATH/<project name>/resources/target
```

For more information and instructions, see EB GUIDE GTF user guide.

For examples of EB GUIDE GTF extensions, see EB GUIDE SDK examples. Download the `EB_GUIDE_-Examples.zip` archive with all EB GUIDE SDK examples from https://www.elektrobit.com/ebguide/learn/resources/. For instructions on how to work with the EB GUIDE SDK examples, see the `EB GUIDE Studio Howto Using examples in EB GUIDE Studio.pdf` file enclosed in the `.zip` archive.

For more information on classes and interfaces, see EB GUIDE GTF API documentation.

6.9.3. EB GUIDE Monitor extensions

A EB GUIDE Monitor extension provides additional functionalities to EB GUIDE Monitor.

Typical EB GUIDE Monitor extensions are:

- Additional components for EB GUIDE Monitor
- Extensions to create screenshots during the simulation

For examples of EB GUIDE Monitor extensions, see EB GUIDE SDK examples. Download the `EB_GUIDE_-Examples.zip` archive with all EB GUIDE SDK examples from https://www.elektrobit.com/ebguide/learn/resources/. For instructions on how to work with the EB GUIDE SDK examples, see the `EB GUIDE Studio Howto Using examples in EB GUIDE Studio.pdf` file enclosed in the `.zip` archive.

For more information on classes and interfaces, see EB GUIDE Monitor API documentation.

6.10. Gamma-correct rendering

6.10.1. Concepts

Gamma correction plays an important role in the rendering pipeline. Gamma correction affects color reproduction on the screen as well as image color storage. Gamma expresses the relationship between color values and the perceived intensities on the screen, for an example of gamma see figure 6.12, “Example of gamma ”.

The human visual system (HVS) exhibits a similar behavior. It is more sensitive to luminance differences in dark image regions than to luminance differences in bright image regions. Common 8-bit image formats (JPEG, PNG) exploit this fact and store colors in the sRGB color space that uses a non-linear transfer function in order to increase the precision in image dark regions. This affects 3D lighting computations as well as textured alpha-blending, because they rely on linear texture color input. For this reason EB GUIDE employs gamma-correct rendering to counter these effects as shown in figure 6.13, “Example of an sRGB textures”.

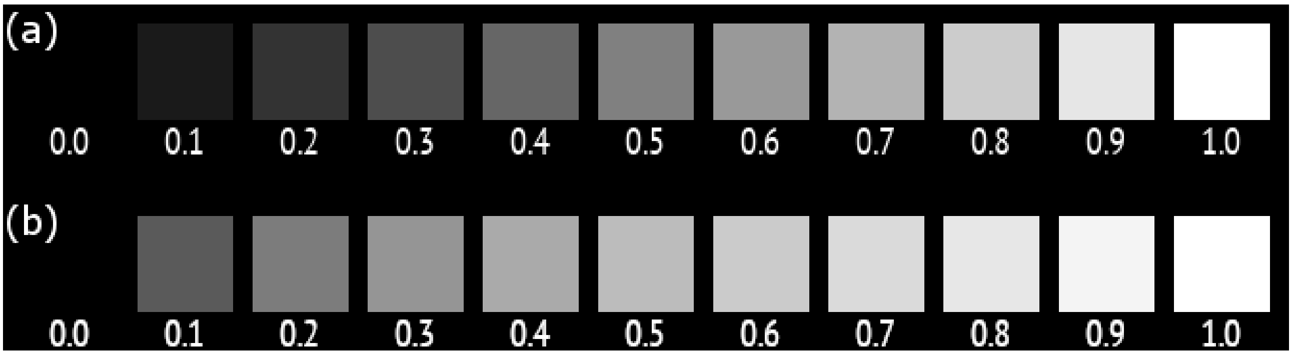


Figure 6.12. Example of gamma

(a) Values below the colored squares denote gray levels. Due to the non-linear relationship between color values and displayed luminance, 50% brightness is approximately reached at a 0.7 gray level for a correctly calibrated monitor. Note that despite the non-linear luminance response of the screen, the relative differences are perceptually uniform. (b) Color values that have been gamma encoded prior to display. Because the encoding gamma cancels out the display’s gamma, 50% brightness is reached at a 0.5 gray level.

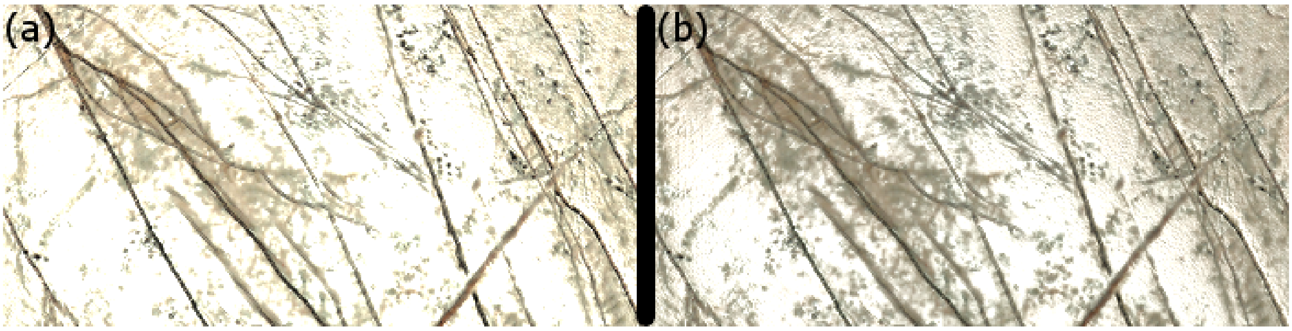


Figure 6.13. Example of an sRGB textures

(a) sRGB texture remains uncorrected in lighting computations and output is not gamma corrected. The lighting is oversaturated and details are washed out. (b) sRGB textures are linearized before lighting and the result is gamma corrected. Details and surface structure become visible.

6.10.2. Gamma correction in EB GUIDE Studio

In EB GUIDE Studio gamma-correct rendering requires from you to configure the rendering pipeline as follows:

- The output configuration controls gamma encoding for the display itself.
- The input configuration tells EB GUIDE Studio which image and texture resources shall be treated as sRGB images in order for the rendering pipeline to properly linearize them for rendering operations.

To configure the input encoding, it has to be configured for each image or texture that is used. Note that the image format itself does not provide the information about sRGB encoding. You need to acquire this information beforehand. For instructions, see section 12.8, “Tutorial: Rendering gamma correctly”

6.11. Image-based lighting

Image-based lighting (IBL) is a technique that makes it possible to use an image as light for 3D objects. In EB GUIDE Studio, IBL is applied with the image-based light widget. You can apply this widget to scene graph nodes. It is not possible to have more than one image-based light per scene graph. If you add more than one, only the first image-based light in the hierarchy is used in the scene.

Figure 6.14. Example of image based lighting. Left: A teapot with a ceramic PBR GGX material lit by three-point-light sources. Center: Using image-based lighting (IBL), the teapot is lit by a virtual environment and the ceramic PBR GGX material appears realistic. Right: Additionally, textures are used to spatially vary the material parameters.

For more information on the ibl datatype, see section 13.3.7, “Ibl”. For more information on the image-based light widget, see section 13.12.3.4, “Image-based light”.

6.11.1. IBLGenerator, file formats and importing

Storing lighting information requires an image format that supports high dynamic range image data. EB GUIDE Studio supports two IBL formats:

- Portable float map (.pfm)
- RGBE (.hdr)

For the RGBE format, EB GUIDE does not support the XYZ color space. Only the -Y +X orientation is supported.

To use these IBL files in EB GUIDE Studio, you need to transform them into the `.ebibl` format. Do this with IBLGenerator. IBLGenerator is controlled through the command line and it is included in your installation in `$GUIDE_INSTALL_PATH\tools`. For instructions, see section 8.1.6, “Importing IBL files”.

IBL files can contain images in either cube, sphere, or latitude-longitude parameterization. In IBLGenerator you can choose the parameterization type. To see all of the options, that IBLGenerator provides, type the following in the command line: `IBLGenerator.exe -h`

6.11.2. Limitations to IBL with an OpenGL renderer

An OpenGL 3 renderer always supports IBL. But if you use the OpenGL renderer, your OpenGL ES 2.0 driver must support the following OpenGL extensions. If one of the following extensions is not supported, the image-based light widget is ignored:

- `GL_EXT_shader_texture_lod`
- `GL_EXT_texture_rg`
- `GL_OES_texture_float`
- `GL_OES_texture_half_float`

6.12. Languages

6.12.1. Display languages in EB GUIDE Studio

EB GUIDE Studio offers different display languages for the graphical user interface. You select the display language in the project center, in the tab **Options**.

For instructions, see section 10.6, “Changing the display language of EB GUIDE Studio”.

6.12.2. Languages in the EB GUIDE model

Most human machine interfaces offer the possibility to display texts in the user's preferred language. Such language management is also provided by EB GUIDE Studio.

In the project configuration you add a language for your EB GUIDE model. You can then export the texts, send them to your localization service provider and import the translated texts back into your EB GUIDE model.

It is possible to add language support to all datapool item types and thus enable your EB GUIDE model to display texts in different languages. A datapool item defines a value for each language. To support languages, select the **Language support** property for the respective datapool item. The language of the exported EB GUIDE model can be changed during run-time.

NOTE

No skin support available

When you have defined a language support for a datapool item, it is not possible to add a skin support to the same item.

For more information, see section 8.4.1, “Adding a language to the EB GUIDE model” and section 6.12.3, “Export and import of language-dependent texts”.

Example 6.5.
Language-dependent texts for a multilingual user interface

In the project configuration three languages are added: English, German, and French. You can now model a label that changes, when the language of the user interface is changed. For this, link the label's `text` property to a datapool item with the value `Welcome` in English and the values `Willkommen` in German and `Bienvenue` in French.

For instructions, see section 12.6, “Tutorial: Adding a language-dependent text to a datapool item”.

6.12.3. Export and import of language-dependent texts

Use the export and import functionality in EB GUIDE Studio to export, edit, translate, and import all language-dependent texts. The texts are exported to an `.xliff` file. `.xliff` (XML Localization Interchange File Format) is an XML-based format to store extracted text and carry the data from one step to another in the localization process. The `.xliff` can be sent to your localization service provider and understood by any translation tool.

After translation, you import the translated `.xliff` file back into your EB GUIDE model.

For instructions, see section 10.8, “Exporting and importing language-dependent texts”.

The `.xliff` file is structured as follows:

- The header contains metadata about the source and target language:
  - The `source-language` and the `target-language` tag follow the ISO 639 standards for representing language names and ISO 3166-1 standards for representing country codes.
  - A unique alphanumeric `sourcelanguageid` and `targetlanguageid` is created for every project and language pair. These IDs prevent unintentional import of an `.xliff` file from another project or target language.

- The `trans-unit` elements contain the localizable data. Each `trans-unit` element holds a `source` element to store the source text, and a `target` element to store the translated text. When a new language is added to the EB GUIDE model, the `target` elements are filled with the source language. Therefore, when exporting an `.xliff` file, all the `target` elements that were not translated yet show the source language.

6.13. Namespaces

In EB GUIDE Studio, with namespaces you create groups of model elements like datapool items and events. These groups have usually a defined functionality. Each namespace creates a naming scope for model elements so that model elements in different namespaces can have the same name.

Each model element belongs exactly to one namespace.

The root namespace is the default namespace and can neither be deleted nor renamed. The root namespace has the same name as the EB GUIDE project. All other namespaces are derived from this namespace. Model elements are always added to the default namespace in the following cases:

- If from the context menu you select **Add link to a datapool item** and create a new datapool item
- If you create an event in the **Trigger** combo box for a transition or an internal transition

You can move model elements between the namespaces.

NOTE

Moving model elements
If you move model elements from one namespace to another and the target namespace already contains an element with the same name, the move operation is not successful and an error message is shown.

Example 6.6.
Namespace tree

In figure 6.15, “Example of a namespace tree”, an example for a namespace tree is shown. The `myProject` namespace is the default namespace and also the name of the EB GUIDE project. Some namespaces are nested inside other namespaces.

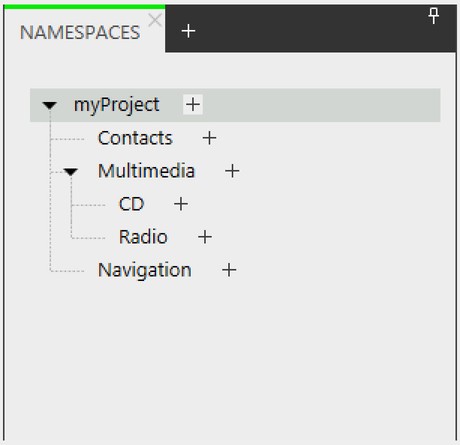


Figure 6.15. Example of a namespace tree

For more information, see section 6.5, “Components of the graphical user interface”.

For instructions, see section 9.12, “Working with namespaces”.

6.14. Photoshop file format support

EB GUIDE Studio supports the `.psd` file format. The supported color spaces are 8-bit, 16-bit, and 32-bit RGB as well as CMYK. You can import a `.psd` file directly or you can extract the images from the `.psd` file.

Importing

Elements from the `.psd` file are put directly into your model and a widget tree is created. The widget tree consists of containers, images, and labels derived from the layers of the `.psd` file. For instructions, see section 8.1.4, “Importing a `.psd` file to a view”. Note the following:

- If a layer in the `.psd` file is set to invisible, the check box next to the `visible` property of the corresponding container or image is cleared.
- Text layers in `.psd` files are imported as labels. You can see the according labels in the widget tree after the import.
- Image layers in `.psd` files are imported as images.
- Group layers in `.psd` files are imported as containers. Containers are named after the group layers. Containers can contain images, labels, or other containers.

Extracting

A subdirectory is created that contains the images from the `.psd` file but the EB GUIDE model that you are working on is not changed. For instructions, see section 8.1.5, “Extracting images from a `.psd` file”.

Limitations

EB GUIDE Studio does not support the following features of the Photoshop file format:

- Layer effects, filters, and textures
- Color models other than RGB or CMYK
- Masks
- Multiple masks applied to a layer (layer mask and vector mask)
- Text styling and fonts
- Only color channels are used

6.15. Resource management

Resources are content that is not created within EB GUIDE but is required by your projects. Locate all resources of an EB GUIDE project in the resources directory.

The resources directory is located at `$GUIDE_PROJECT_PATH/<project name>/resources`.

EB GUIDE supports the following types of resource files:

1. Fonts
2. `.ebibl` file format for 3D graphics
3. Images
4. Meshes for 3D graphics
5. `.psd` file format

To use resources in the project, add the resource files to the directory `$GUIDE_PROJECT_PATH/<project name>/resources`.

6.15.1. Fonts

To use a font in the project, add the font to the directory `$GUIDE_PROJECT_PATH/<project name>/resources`.

Supported font types are TrueType fonts (`*.ttf`, `*.ttc`), OpenType fonts (`*.otf`), and bitmap fonts (`*.fnt`).

For instructions, see section 8.1.2.4.1, “Changing the font of a label”.

6.15.1.1. Bitmap fonts

EB GUIDE Studio supports the `*.fnt` bitmap fonts from Angelcode in version 3.0. To create a bitmap font, use a third-party font generator, for example Angelcode Bitmap Font Generator. For more information, see http://www.angelcode.com.

Make sure that the generated font has the following settings:

- The desired font size is defined.
- The character set is Unicode.
- The font descriptor is binary.
- The textures are provided as 8-bit `.png` files.

Note the following:

- In EB GUIDE Studio you are not able to change the font size of a bitmap font using the `font` property of a label. That means that you need to define the size when you generate your `.fnt` font.
- The **Stroke** widget feature does not apply to bitmap fonts. If you need a specific outline for your font, define it when you generate your `.fnt` font.
- In the `$GUIDE_PROJECT_PATH/resources` directory, create a subdirectory for your `.fnt` bitmap font and `.png` texture files that you generated with a third-party tool. EB GUIDE Studio expects to find the `.png` files in same directory as the `.fnt` file.

  If you have several bitmap fonts, create a subdirectory for each of these fonts.

6.15.1.2. Multifont support

In EB GUIDE Studio you can create your own font combinations using the multifont support. This feature is useful, for example, if the font that you selected does not provide all necessary characters. In this case you can replace missing characters with characters from a different font.

The multifont support can be added to the following model elements:

- Properties of type font and entries of font list
- User-defined properties of type font and entries of font list
- Datapool items of type font or entries of font list

You can define the Unicode character range, for which you want to use the specified font, as follows:

- With a single Unicode character, for example `0000`.
- With several Unicode characters separated by comma, for example `0000, 0001`.
- With a range of Unicode characters, for example `0000-FFFF`.
- With several ranges separated by comma, for example `0000-0022, 0045-0055`.

The characters are specified using the hexadecimal number format.

The fonts should be available in `$GUIDE_PROJECT_PATH/resources`.

When you add multifont support, automatically a default multifont value is added. You cannot delete the default multifont value nor edit its priority and range. However, you can edit size and font for the default value.

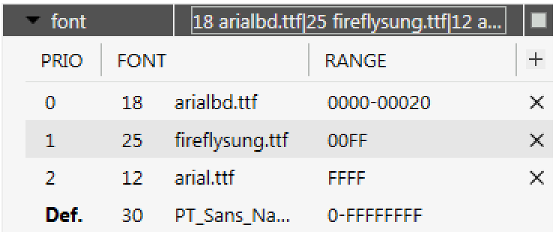


Figure 6.16. Example of a property of type font with added multifont support

For instructions on how to use the multifont support, see section 8.2.8, “Managing multifont support”.

6.15.2. Image-based lighting for 3D graphics

It is possible to use image-based lighting in EB GUIDE Studio. The external command line tool IBLGenerator takes a `.pfm` or `.hdr` file as input data and creates an `.ebibl` file which represents an IBL resource. The IBL resource is used by the `ibl` property of the image-based light widget.

For instruction, about how to get an `.ebibl` file, see section 8.1.6, “Importing IBL files”.

For background information, see section 6.11, “Image-based lighting”.

6.15.3. Images

To use an image in the project, add the image to the directory `$GUIDE_PROJECT_PATH/<project name>/resources`. If you select an image from a different directory, the image is copied to the directory .

The supported image formats are Portable Network Graphic (`*.png`), JPEG (`*.jpg`) and 9-patch images (`*.9.png`).

For instructions, see section 8.1.2.3, “Adding an image”.

6.15.3.1. 9-patch images

EB GUIDE Studio supports images with additional meta information according to the 9-patch image approach. 9-patch images are stretchable `.png` images. 9-patch images contain two black markers, one at the top and one at the left side of the image. Areas that are not marked are not scaled. Marked areas are scaled. Markers are not displayed in EB GUIDE Studio.

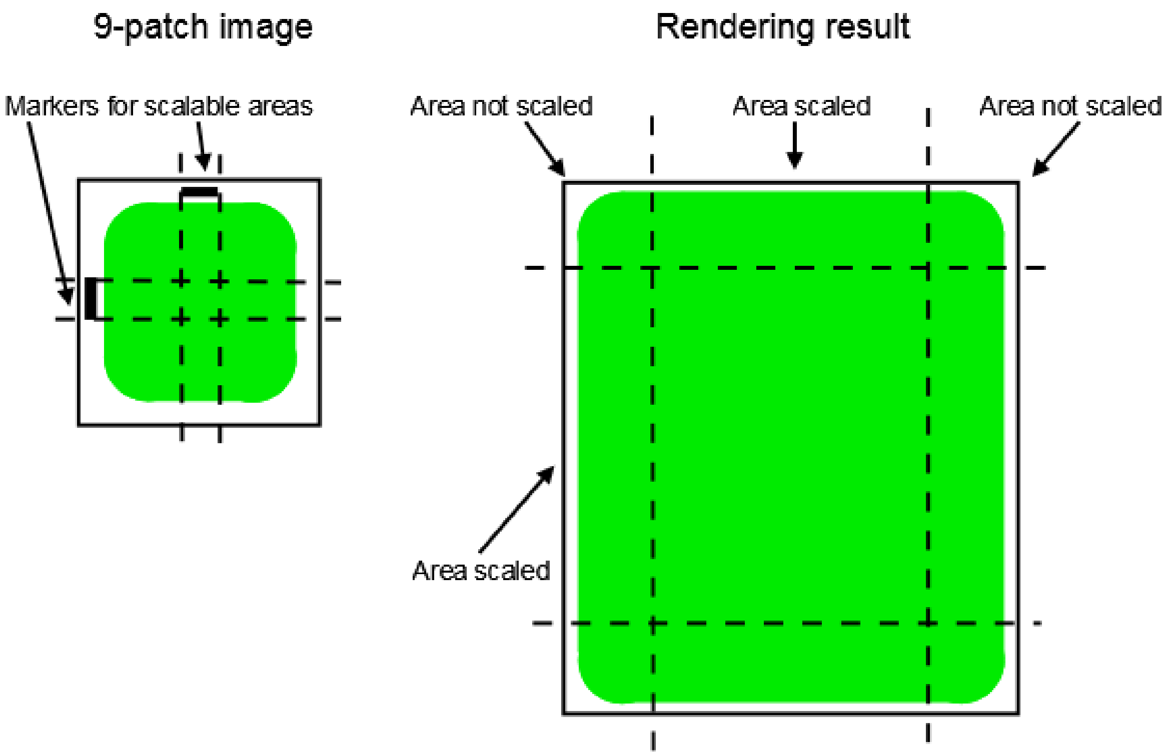


Figure 6.17. 9-patch example

When you work with 9-patch images, consider the following:

- 9-patch processing works only with the renderers for OpenGL ES 2.0 or higher.
- 9-patch processing works with `.png` images only.
- The `*.9.png` extension is mandatory for 9-patch images.
- It is possible to specify none, one, or more than one marker at the top and the left side. The 9-patch definition also includes markers for text areas at the right side and at the bottom of the image. These markers are not evaluated in EB GUIDE Studio.

For instructions, see section 8.1.2.3, “Adding an image”.

6.15.4. Meshes for 3D graphics

It is possible to import 3D graphic files in EB GUIDE Studio. After you have imported a 3D graphic file in EB GUIDE Studio, in `$GUIDE_PROJECT_PATH/<project name>/resources`, you find a subdirectory. Meshes as defined in the 3D graphic file are imported as `.ebmesh` files. For details, see section 6.1.3, “Import of a 3D graphic file”.

For instructions, see section 8.1.3.1, “Adding a scene graph to a view”.

6.16. Scripting language EB GUIDE Script

EB GUIDE Script is the built-in scripting language of EB GUIDE. This chapter describes EB GUIDE Script language features, syntax, and usage.

6.16.1. Capabilities and areas of application

You can use EB GUIDE Script in a variety of places in a project, for example:

- In a widget property
- In the state machine as part of a transition or state
- In a datapool item

Not all features of EB GUIDE Script are available in all cases. For example access to local widget properties is only allowed when the script is part of a widget. Access to the datapool, on the other hand, is always allowed.

With EB GUIDE Script you can directly manipulate model elements, for example to do the following:

- Fire events
- Write datapool items
- Modify widget properties

6.16.2. Prefixes and identifiers

In EB GUIDE, it is possible to give identical names to different kinds of objects. For example, you can name both an event and a datapool item `Napoleon`. To make this possible, every identifier, i.e. name of an object, in EB GUIDE Script must have a prefix. The prefix defines the type of an object, followed by a colon.

The set of prefixes is fixed in EB GUIDE Script, you cannot introduce new prefixes. The following prefixes exist:

- `ev`: events
- `dp`: datapool items
- `f`: user-defined actions (foreign functions)
- `v`: local variables

For example, `ev:Napoleon` specifies the event named `Napoleon` while `dp:Napoleon` specifies the datapool item named `Napoleon`.

Identifiers without a prefix are string constants.

Identifiers in EB GUIDE contain many characters including spaces and punctuation. Thus it can be necessary to quote identifiers in EB GUIDE Script. If an identifier does not contain special characters, for example a valid C identifier consisting only of letters, numbers and underscores, it does not have to be quoted.

Example 6.7.
Identifiers in EB GUIDE Script

```
dp:some_text = foo; // foo is a string here
dp:some_text = "foo"; // this statement is identical to the one above
dp:some_text = v:foo; // foo is the name of a local variable
// of course you can quote identifiers, even if it is not strictly necessary
dp:some_text = v:"foo";
// again, a string constant
dp:some_text = "string with spaces, and -- punctuation!";
// identifiers can also contain special characters, but you have to quote them
dp:some_text = v:"identifier % $ with spaces @ and punctuation!";
```

6.16.3. Comments

EB GUIDE Script has two kinds of comment: C style block comments and C++ style line comments. Block comments must not be nested.

Example 6.8.
Comments in EB GUIDE Script

```
/* this is a C style block comment */
// this is a C++ style line comment
```

For every EB GUIDE Script comment that contains a string "todo", EB GUIDE Studio shows a warning in the **Problems** component when you validate a project. Use this feature to mark all your open tasks and display them at a glance.

NOTE

Default comment for conditional scripts

By default, a datapool item or a property of type `Conditional script` contains a comment `// todo: auto generated return value, please adapt`. To eliminate the warning, delete the `todo` string from the comment once you entered the required EB GUIDE Script code.

6.16.4. Types

EB GUIDE Script is a strongly-typed and statically-typed programming language. Every expression has a well defined type. Supplying an unexpected type results in an error.

EB GUIDE Script supports the following types:

- Integer
- Unicode strings (string)
- Objects with reference counting
- Type definitions to the above listed types and to the following:
  - Color (integer for 32-bit RGBA value)
  - Boolean
  - IDs of different model elements: datapool items, views, state machines, pop-ups (all of integer type)
- Void, also known as the unit type. This type has a role as in functional programming, for example Haskell.
- Widget and event references. These are record types, the fields of which you may access by using the `dot` notation, as known in C or Java. You cannot directly create new objects of these kinds, they are created automatically where appropriate.

All types and type definitions are incompatible with each other and there are no typecasts. This feature ensures type safety once a script is successfully compiled.

6.16.5. Expressions

EB GUIDE Script is expression-based. Every language construct is an expression. You form larger expressions by combining smaller expressions with operators.

To evaluate an expression means to replace it by its value.

Example 6.9.
Evaluation of an integer value

```
1 + 2 // when this expression is evaluated, it yields the integer 3
```

6.16.6. Constants and references

The basic expressions are integer, color, boolean, and string constants and references to model elements.

The void type also has a value constant that can be written in two different but semantically equivalent ways:

- With the opening curly brace followed by the closing curly brace `{}`
- With the keyword `unit`

Example 6.10.

Usage of constants

```
"hello world"  // a string constant
true           // one of the two boolean constants
ev:back        // the event named "back" of type event_id
dp:scrollIndex // the datapool item named "scrollIndex",
               // the type is whichever type the dp item has
5              // integer constants have a dummy type "integer constant"
5::int     // typecast your constants to a concrete type!
color:255,255,255,255 // the color constant for white in RGBA format


 // the following are two ways to express the same
                        if( true )
{
}
else
{
}

if( true )
    unit
else
    unit
```

6.16.7. Arithmetic and logic expressions

EB GUIDE Script supports the following arithmetic expressions:

- Addition `(+)`, subtraction `(-)`, multiplication `(*)`, division `(/)`, and modulo `(%)` can be applied to expressions of type integer.
- The logical operators or `(||)`, and `(&&)`, not `(!)` can be applied to expressions of type boolean.
- Integers and strings can be compared with the comparison operators greater-than `(>)`, less-than `(<)`, greater-than-or-equal `(>=)`, less-than-or-equal `(<=)`.
- Data types can be compared with the equality operators: equal to `(==)` and not equal to `(!=)`.

  Strings can be compared without case sensitivity with the equality operator `(=Aa=)`.

  NOTE **Availability of equality operators**
  Events and resource data types, for example 3D graphics, fonts and images, do not support the equality operators `(==)` and `(!=)`.

- Strings can be concatenated with the `(+)` operator.

Example 6.11.
Arithmetic and logic expressions

```
10::int + 15::int // arithmetic expression of type int
dp:scrollIndex % 2        // arithmetic expression of type int,
                          // the concrete type depends on the type
                          // of dp:scrollIndex
"Morning Star" == "Evening Star" // type bool and value false (wait, what?)
"name" =Aa= "NAME" // type bool and value true
!true              // type bool, value false
!(0 == 1)          // type bool, value true
// as usual, parenthesis can be used to group expressions
((10 + dp:scrollIndex) >= 50) && (!dp:buttonClicked)
// string concatenation
"Napoleon thinks that " + "the moon is made of green cheese"
f:int2string(dp:speed) + " km/h" // another string concatenation
```

6.16.8. L-values and r-values

There are two kinds of expressions in EB GUIDE Script: `l-values` and `r-values`. L-values have an address and can occur on the left hand side of an assignment. R-values do not have an address and may never occur on the left hand side of an assignment.

- L-values are datapool references, local widget properties, and local variables.
- R-values are event parameters and constant expressions such as string or integer constants.

6.16.9. Local variables

The `let` expression introduces local variables. It consists of a list of variable declarations and the `in` expression, in which the variables are visible. Variables are l-values, you can use them on the left hand side of assignments. Variables have the prefix `v:`. The syntax of the `let` expression is as follows:

```
let v:<identifier> = <expression> ;
    [ v:<identifier> = <expression> ; ]...
in
    <expression>
```

The type and value of the `let` expression are equal to the type and value of the `in` expression.

`let` expressions may be nested, variables of the outer `let` expressions are also visible in the inner expressions.

Example 6.12.
Usage of the `let` expression

```
// assign 5 to the datapool item "Napoleon"
let v:x = 5 in dp:Napoleon = v:x;

// define several variables at once
let v:morning_star = "Venus";
    v:evening_star = "Venus";
in
    v:morning_star == v:evening_star; // Aha!

let v:x = 5;
    v:y = 20 * dp:foo;
in
{
    // Of course you may have a sequence as the in expression,
    // but parenthesis or braces are required then.
    v:x = v:y * 10;
    dp:foo = v:x;
}
// Because let expression also have types and values, we can have them
// at the right hand side of assignments.
dp:x = let v:sum = dp:x + dp:y + dp:z
       in v:sum; // this is the result
                 // of the let expression

// A nested let expression
let v:x = dp:x + dp:y;
v:a = 5;
in
{
    let v:z = v:x + v:a;
    in
    {
        dp:x = v:z;
    }
}
```

6.16.10. While loops

`while` loops in EB GUIDE Script have a syntax similar to that in C or Java, they consist of a condition expression and a do expression. The syntax is as follows:

```
while (<condition expression> ) <do expression>
```

The do expression is evaluated repeatedly until the condition expression yields `false`. The `condition expression` must be of type boolean, the do expression must be of type `void`. The `while` expression is of type `void` and must not occur at the left or right hand side of an assignment.

Example 6.13.
Usage of the `while` loop

```
// Assume dp:whaleInSight is of type bool
while( ! dp:whaleInSight )
{
    dp:whaleInSight = f:lookAtHorizon();
}
```

6.16.11. If-then-else

`if-then-else` in EB GUIDE Script behaves like the ternary conditional operator `(?:)` in C and Java.

The `if-then-else` expression consists of the following sub-expressions:

- condition expression
- then expression
- `else` expression

The syntax is as follows:

```
if ( < condition expression> ) <then expression> else <else expression>
```

`if-then-else` is processed as follows:

1. First, the condition expression is evaluated. It must be of type boolean.
2. If the condition is true, the then expression is evaluated.
3. If the condition is false, the `else` expression is evaluated.

`if-then-else` itself is an expression. The type of the whole expression is the type of the then expression and the `else` expression, which must be identical. The value of `if-then-else` expressions is either the value of the then expression, or the value of the `else` expression, in accordance with the rules above.

There is a special form of `if-then-else`, in which you may omit the `else` branch. This special form is of type `void` and cannot be used to return values from scripts.

Example 6.14.

Usage of `if-then-else`

```
// Assume dp:whaleInSight is of type bool
// and dp:user is of type string.
if( dp:whaleInSight && dp:user == "Captain Ahab" )
{
    dp:mode = "insane";
}
else
{
    dp:mode = "normal";
}

// Because if-then-else is also an expression,
// we may simplify the previous example:
dp:mode = if( dp:whaleInSight && dp:user == "Captain Ahab" )
            "insane"
          else
            "normal"

if ( <expression> ) <expression> // This is the reduced way of
                writing if-then-else
            //It is an alternative to the following
            if( <expression> ) { <expression> ; {} } else {}
```

6.16.12. Foreign function calls

You can extend EB GUIDE Script with functions written in C, so-called foreign functions.

An identifier prefixed by `f:` is the name of a foreign function. Foreign functions have an argument list and a return value, as they do in C. The syntax of foreign function calls is as follows:

```
f:<identifier> ( <expression> [ , <expression> ] ... )
```

Example 6.15.
Calling foreign functions

```
// write some text to the connection log
f:trace_string("hello world");
// display dp:some_index as the text of a label
v:this.text = f:int2string(dp:some_index);

// passing different parameters of matching type
f:int2string(v:this.x)
```

```
f:int2string(4)
f:int2string(dp:myInt)
f:int2string(v:myVar)


//passing parameters of different types
// starts an animation (parameter type GtfTypeRecord) from a script
// located in its parent widget
f:animation_play(v:this->Animation);

// checks the number of child widgets of a widget (parameter type widget)
f:widgetGetChildCount(v:this);

// traces debugging information about a datapool item (parameter type dp_id)
// to the connection log; uses the address of the datapool item as parameter
f:trace_dp(&dp:myFlag);
```

6.16.13. Datapool access

Scripts written in EB GUIDE Script can read and write datapool items. An identifier with the prefix `dp:` is called datapool item expression. Its type is `datapool item of type X`, where X is the type of the datapool entry it refers to. Identifier may include only the name of the datapool item, or, if the datapool item is not in the default namespace, the name of the namespace followed by the name of the datapool item.

If a datapool item of type X occurs on the left hand side of an assignment, and an expression of type X occurs on the right hand side of the assignment, the value of the datapool item is written.

If a datapool item occurs somewhere in a program but not on the left hand side of an assignment, the value of the datapool item is read.

Example 6.16.
Assignment of datapool values

```
// Assume intA to be of type int. Assign 10 to it.
dp:intA = 10;
// Assume strA to be of type string. Assign the string "blah" to it.
dp:strA = blah; // Yes, we can omit the quotes, remember?
dp:strA = 42; // Error: integer cannot be assigned to string

// Assign the value of the datapool item intB to intA.
// Both datapool items must have the same type.
dp:intA = dp:intB;
// Multiply the value of intB by two and assign it to intA.
dp:intA = 2 * dp:intB;
// Use the value of a datapool item in an if-clause.
```

```
if( dp:speed > 100 )
{
    // ...
}
```

The following operators can be applied to the datapool items:

- The reference operator (&) can be applied to datapool items. It refers to the address of a datapool item rather than to its value. The reference operator is used in foreign function calls to pass parameters of type `dp_id`.
- The redirect-link operator (=>) changes the link target of a datapool item. Link source can only be a datapool item that was already linked.

6.16.14. Widget properties

If a script is part of a widget, it can access the properties of that widget. EB GUIDE Script creates a variable called `v:this` to access the properties using the dot notation.

A script is part of a widget if it is attached to a widget property, for example as an input reaction such as click or button press.

Example 6.17.
Setting widget properties

```
// assume this script is part of a widget
v:this.x = 10; // if the widget has an x-coordinate

v:this.text = "hello world"; // if the widget is a label and has a text property
// assume testEvent has one integer parameter
fire ev:testEvent(v:this.x);
```

If a script is part of a widget, it can also access properties of other widgets in the widget tree.

The go-to operator (`->`) is used to refer to other widgets within the widget tree. The syntax is as follows:

```
<expression> -> <expression>
```

The expression on the left hand side must refer to a widget and the expression on the right hand side must be a string, the name of a child widget. To navigate to the parent widget, use the symbol `^` on the right hand side. The whole go-to expression refers to a widget.

Navigating the widget tree might affect run-time performance. Widgets are assigned to variables for the efficient manipulation of multiple properties.

Example 6.18.
Accessing widget properties

```
v:this.x       // access the properties of the current widget
v:this->^.x    // access the x property of the parent widget
v:this->^->caption.text // access the text property of a label called caption,
                        // read: "go-to parent, go-to caption, text"

// Modify several properties of the caption.
// This way, the navigation to the caption is only performed once.
let v:cap = v:this->^->caption
in
{
  v:cap.textColor = color:0,0,0,255;
  v:cap.x += 1;
  v:cap.y += 1;
}
```

6.16.15. Lists

Datapool items and widget properties can hold lists. The subscript operator (`[]`) accesses list elements. The syntax is as follows:

```
<expression> [ <expression> ]
```

The first expression must evaluate to a list type, the second expression must evaluate to an integer value. If the list is of type `list A`, the whole list subscript expression must be of type `A`.

If the list subscript expression occurs at the left hand side of an assignment, the value of the referred list element is written.

The `length` keyword returns the number of elements of a list. If it is put in front of a list expression, the whole expression must be of type integer.

Example 6.19.
Lists

```
// Assume this widget is a label and dp:textList is a list of strings
v:this.text = dp:textList[3];

dp:textList[1] = v:this.text; // writing the value of the list element


v:this.width = length dp:textList;// checking the length of the list
dp:textList[length dp:textList - 1] = "the end is here";
```

Adding elements to and removing elements from lists is currently not supported in EB GUIDE Script.

Trying to access list elements beyond the end of a list stops the execution of the script immediately. Make sure that all your list accesses are in range.

6.16.16. Events

EB GUIDE Script offers the following expressions to handle events:

- The `fire` expression sends events. The syntax is as follows:

  ```
  fire ev:<identifier> ( <parameter list> )
  ```

  Events can, but do not need to have parameters. The parameter list of the `fire` expression must match the parameters of the fired event. If an event has no parameters, the parentheses must be empty.

  Example 6.20.
  Using the `fire` expression

  ```
  fire ev:toggleView(); // the event "toggleView" has no parameters
  fire ev:mouseClick(10, 20); // "mouseClick" has two integer parameters
  fire ev:userNameEntered("Ishmael"); // string event parameter
  ```

- The `fire_delayed` expression sends events after a specified time delay. The syntax is as follows:

  ```
  fire_delayed <time> , ev:<identifier> ( <parameter list> )
  ```

  The `time` parameter is an integer value that specifies the delay in milliseconds.

  Example 6.21.
  Using the `fire_delayed` expression

  ```
  fire_delayed 3000, ev:mouseClick(10, 20); // send the event "mouseClick"
                  //in 3 seconds.
  ```

- The `cancel_fire` expression cancels the delayed event. The syntax is as follows:

  ```
  cancel_fire ev:<identifier>
  ```

- The `match_event` expression checks whether the execution of a script has been triggered by an event. The syntax is as follows:

  ```
  match_event v:<identifier> = ev:<identifier>
  in
      <expression>
  else
      <expression>
  ```

The type of the `match_event` expression is the type of the `in` expression and the `else` expression, which must be identical.

There is a special form of the `match_event` expression, in which you can omit the `else` branch. This special form is of type `void` and cannot be used to return values from scripts.

Example 6.22.
Using the `match_event` expression

```
match_event v:theEvent = ev:toggleView in
{
    // this code will be executed when the "toggleView" event
    // has triggered the script
    dp:infoText = "the view has been changed";
}
else {}

match_event ( <expression> ) in <expression> //special form
                   //without an else branch
               //The special form is an alternative way to express the following
               match_event ( <expression> ) in { <expression> ; {} } else {}
```

Identifier may include only the name of the event, or, if the event is not in the default namespace, the name of the namespace followed by the name of the event.

If an EB GUIDE Script has been triggered by an event with parameters, the parameters are accessible in the `in` expression of a `match_event` expression. Read parameters using the dot notation, as you would access fields of a structure in C. Event parameters are not available in the `else` expression.

Example 6.23.
Event parameters

```
// assume that "mouseClick" has two parameters: x and y
match_event v:event = ev:mouseClick in
{
    dp:rectX = v:event.x;
    dp:rectY = v:event.y;
}
```

6.16.17. String formatting

String formatting in EB GUIDE Script is done using the concatenation operator (`+`) on strings in combination with various data-to-string conversion functions. The EB GUIDE Script standard library comes with the `int2string` function for simple integer-to-string conversion.

Example 6.24.
String formatting

```
// Assume this widget is a label and has a text property.
// Further assume that the datapool item dp:time_hour and
// dp:time_minute hold the current time.
v:this.text = "the current time is: " + f:int2string(dp:time_hour)
    + ":" + f:int2string(dp:time_minute);
```

6.16.18. The standard library

EB GUIDE Script comes with a standard library that consists of a set of foreign functions for example as follows:

- String formatting
- Language management
- Tracing
- Time and date
- Random number generation

For details, see section 13.4.3, “EB GUIDE Script standard library”.

6.17. Scripted values

A scripted value is an alternative notation for the value of a widget property or a datapool item. Such properties of widgets or datapool items use other model elements to evaluate their own value or to react on events or property updates. Scripted values are written in the EB GUIDE Script scripting language.

A property in EB GUIDE can be converted to a scripted value and back to its plain value.

For instructions, see section 9.8, “Converting a property to a scripted value”.

For editing a scripted value, EB GUIDE Studio contains an EB GUIDE Script editor which is divided into different categories.

Figure 6.18. EB GUIDE Script editor in EB GUIDE Studio

- The **Read** script is called when the scripted value property is read. If the property is of a list type, the parameters include the list index.

  The return value of the **Read** script represents the current value of the property.

- The **Write** script is called when the scripted value property is written.

  The new property value is a parameter of the **Write** script. If the property is of a list type, the parameters includes the list index.

  The return value of the **Write** script controls change notifications for the property.

- true: trigger a change notification
- false: do not trigger a change notification

- The **Trigger** script contains a list of events, datapool items and widget properties that trigger the execution of the **On trigger** script.

  Clicking on **Add all available triggers to trigger script** will add all triggers highlighted in the corresponding script to the trigger script.

- The **On trigger** script is called on initialization, after an event trigger or after a property update.

  The parameter of the **On trigger** script indicates the cause for the execution of the script. Execution can be caused by initialization or by one of the triggers in the **Trigger** list.

  The return value of the **On trigger** script controls change notifications for the property.

  - true: trigger a change notification
  - false: do not trigger a change notification

- The **Length** script is only available for properties of a list type.

  The return value of the **Length** script represents the current length of the list.

6.18. Shortcuts, buttons and icons

6.18.1. Shortcuts

The following table lists shortcuts available in EB GUIDE Studio and explains their meaning.

Table 6.4. Shortcuts

Shortcut	Description
Ctrl+C	Copy the selection
Ctrl+F	Jump into search box
Ctrl+S	Save
Ctrl+V	Paste the copied selection
Ctrl+Y	Redo
Ctrl+Z	Undo
Ctrl+Enter	In the trigger filter box, add a new event.

Shortcut	Description
Ctrl+Shift+Insert	In the **Namespaces** component, add a new namespace as a child to an existing namespace.
Alt+F4	Close the active window
Shift+F1	Open user documentation for EB GUIDE TF
F1	Open user documentation for EB GUIDE Studio
Shift+F2	Rename the selected element in the **Datapool** or **Events** component and in all locations where the selected element is used, e.g. in EB GUIDE Script. Applicable to datapool items and events.
F2	Rename the selected element
F3	Find all occurrences of the selected element in the EB GUIDE model
F5	Start simulation
F6	Validate
Del	Delete the selected element from the **Content** area or the component
-	Collapse the selected element in the **Navigation** or **Outline** component
***** and **+**	Expand the selected model element in the **Navigation** or **Outline** component
Up/Down/Left/Right	Move the selected state or widget in the content area one pixel up, down, left, or right
Ctrl and click the left mouse button **Shift** and click the left mouse button or **Up arrow** or **Down arrow** keys	Select multiple elements in the **Datapool** and **Events** components.

6.18.2. Command line options

6.18.2.1. Command line options for `Studio.Console.exe`

The following table lists command line options available in EB GUIDE Studio for `Studio.Console.exe` and explains their meaning. Undefined command line options will be ignored.

The general syntax of a command line is as follows:

```
Studio.Console.exe <option> "project_name.ebguide"
```

Table 6.5. Command line options for `Studio.Console.exe`

Option	Description
`-c <logfile dir>`	Validates an EB GUIDE model and writes an logfile to the as `logfile dir` specified directory
`-e <destination dir>`	Exports an EB GUIDE model to the destination directory `destination dir` Use with the command line option `-p`, see an example below.
`-h`	Shows the help message
`-l <language file>`	Imports one language file that is saved as `language file` (`.xliff`) into an EB GUIDE model and creates a logfile
`-m`	Allows the migration of the project
`-o`	Opens the project file
`-p <profile>`	Uses the as `profile` specified profile during export

Example 6.25.
Command line options

The command line `Studio.Console.exe -e "C:/temp/exported_project" -p "target_profile" -o "project_name.ebguide"` exports `project_name.ebguide` by using the profile `target_profile` to the specified destination directory `C:/temp/exported_project`.

For instructions, see the following:

- section 10.4.1.2, “Validating an EB GUIDE model using command line”
- section 10.5.2, “Exporting an EB GUIDE model using command line”
- section 10.8.2.2, “Importing language-dependent texts using command line”

6.18.2.2. Command line options for `Monitor.Console.exe`

The following table lists command line options available in EB GUIDE Monitor for `Monitor.Console.exe` and explains their meaning. Undefined command line options will be ignored.

The general syntax of a command line is as follows:

```
Monitor.Console.exe <option> "monitor.cfg"
```

Table 6.6. Command line options for `Monitor.Console.exe`

Option	Description
`-c <host:port>`	Connects an EB GUIDE model to a running EB GUIDE GTF process

Option	Description
`-h`	Shows the help message
`-l <language>`	Sets the language of EB GUIDE Monitor to one of the following: `en` for English, `ja` for Japanese, `ko` for Korean, `zh-cn` for Chinese.
`-o`	Opens the configuration file `monitor.cfg`
`-s`	Executes all methods in a defined script

Example 6.26.
Command line options

The command line `Monitor.Console.exe -l ko` sets the language of EB GUIDE Monitor to Korean.

For instructions on how to use EB GUIDE Monitor, see section 10.9, “Working with EB GUIDE Monitor”.

6.18.3. Buttons

The following table lists buttons that are used in EB GUIDE Studio and EB GUIDE Monitor and explains their meaning.

Table 6.7. Buttons in EB GUIDE Studio

Button	Description
	Undo
	Redo
	Save
	Validate the project
	Start the simulation
	Stop the simulation
	Open the project center
{ }	Open the EB GUIDE Script editor
	Synchronize content area and **Navigation** component
+	Add an event, a datapool item, or a state machine
	Open a property-related context menu.

Button	Description
	The button's colors listed below indicate the following: Property is local. Property is linked to another property. Property is linked to a datapool item. Property value is equal to template value.
	Fire an event

6.18.4. Icons

The following table lists icons that are used in EB GUIDE Studio and explains their meaning.

Table 6.8. Icons in EB GUIDE Studio

Icon	Description
	Indicates an exit animation of a view state or a view template
	Indicates an entry animation of a view state or a view template
	Indicates a change animation of a view state or a view template
	Indicates an entry action of a state machine or state
	Indicates an exit action of a state machine or state
	Opens a context menu to delete an entry or exit action
	Indicates that a dynamic state machine list is enabled
	Indicates a template
	Indicates a transition
	Indicates an internal transition
	Widget template: Indicates that a property is added to the widget template interface
	Opens a dialog to select a master image
	In the EB GUIDE Script editor it indicates a function
	In the EB GUIDE Script editor it indicates a keyword
	In the EB GUIDE Script editor it indicates a variable

6.19. Skins

Skins allow you to define different user interfaces by defining different datapool values for the same EB GUIDE model. This way you can define various looks for the same HMI as for example skins for night and day mode.

You can switch between the skins during run-time to see the effect of the different datapool values.

Skin support is only available for plain datapool values and cannot be used for scripted values or linked datapool items.

NOTE

No language support available

When you have defined a skin support for a datapool item, it is not possible to add a language support to the same item.

For instructions see section 8.5, “Working with skin support”.

6.20. State machines and states

NOTE

Changing background color of state machines and states

In EB GUIDE Studio, you are able to change the background color of the following:

- **Main** state machine
- View state
- compound state

To change the background color, select a color from the `Background color` drop-down list box in the **Properties** component.

6.20.1. State machines

A state machine is a deterministic finite automaton and describes the dynamic behavior of the system. In EB GUIDE, a state machine consists of an arbitrary number of hierarchically ordered states and of transitions between the states.

In EB GUIDE you can create the following types of state machines.

6.20.1.1. Haptic state machine

Haptic state machine allows the specification of GUI.

6.20.1.2. Logic state machine

Logic state machine allows the specification of some logic without GUI.

6.20.1.3. Dynamic state machine

Dynamic state machine runs parallel to other state machines.

Dynamic state machine does not start automatically at system start. The start and stop of dynamic state machines is initiated by another state machine.

There are two kinds of dynamic state machines:

- Haptic dynamic state machine
- Logic dynamic state machine

For instructions, see section 12.1, “Tutorial: Adding a dynamic state machine”.

6.20.2. States

EB GUIDE uses a concept of states. States determine the status and behavior of a state machine. States are linked by transitions. Transitions are the connection between states and define a state change from a source state to a destination state.

A state has the following properties:

- Entry action
- Exit action
- Internal transitions

6.20.2.1. Compound state

A compound state can have other states within it as child states. The compound state structure is hierarchical and the number of possible child states is arbitrary. Any type of state can be nested in a compound state.

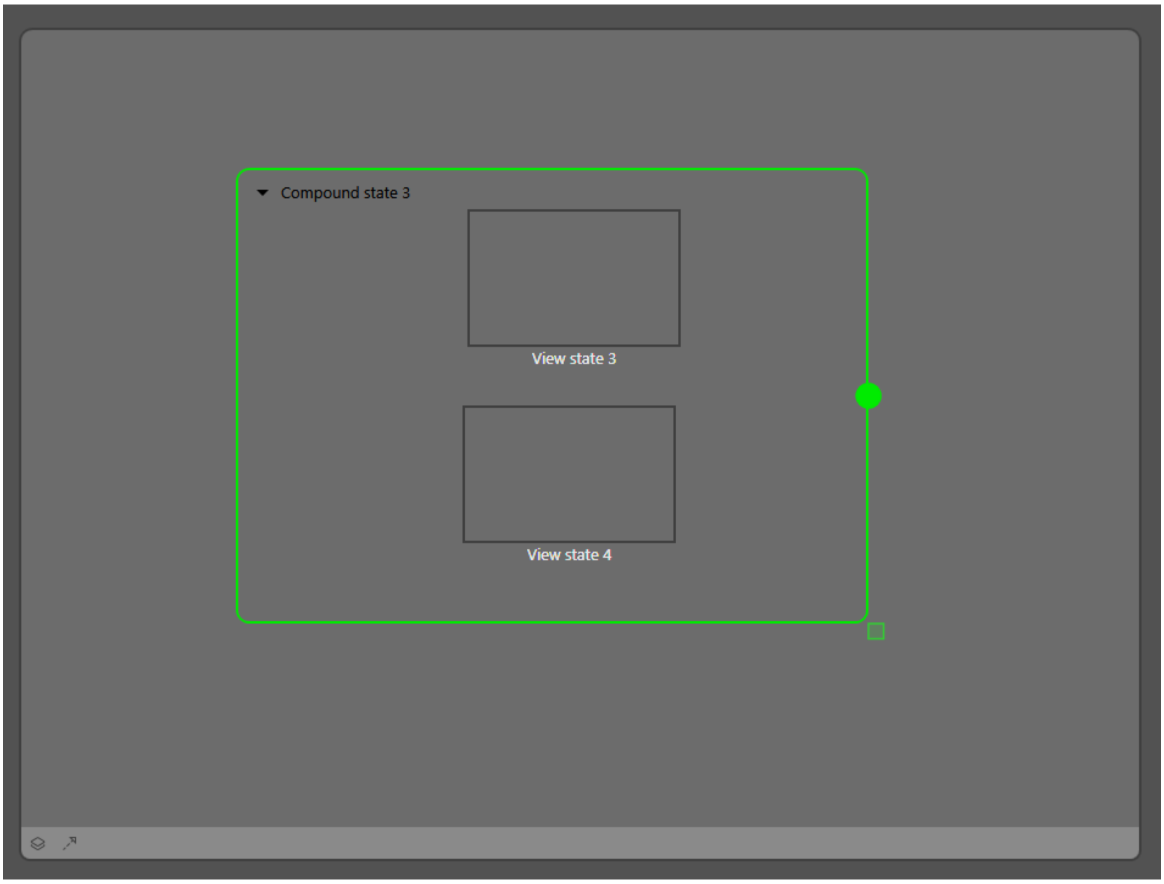


Figure 6.19. Compound states

In the **Navigation** component, the state hierarchy is shown as a tree structure.

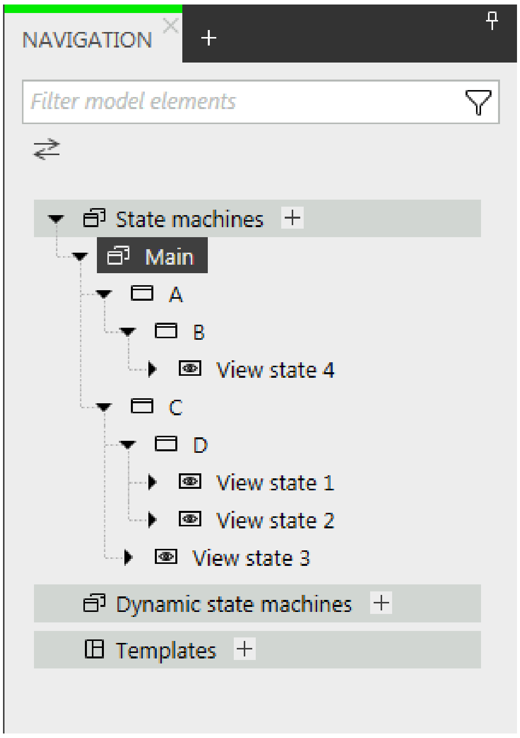


Figure 6.20. State hierarchy as a tree

A compound state can have an arbitrary number of incoming and outgoing transitions, and of internal transitions. Child states inherit the transitions of parent states.

6.20.2.2. View state

A view state contains a view. A view represents a project specific HMI screen. The view is displayed while the corresponding view state is active. The view consists of widgets which are the interface between user and system.

6.20.2.3. Initial state

An initial state defines the starting point of the state machine. An initial state has an outgoing default transition that points to the first state. An initial state has no incoming transition.

Initial state can be used as starting point of a compound state or to enter a compound state in the following ways:

- With a transition to compound state, initial state is mandatory
- With a transition to a child state of a compound state

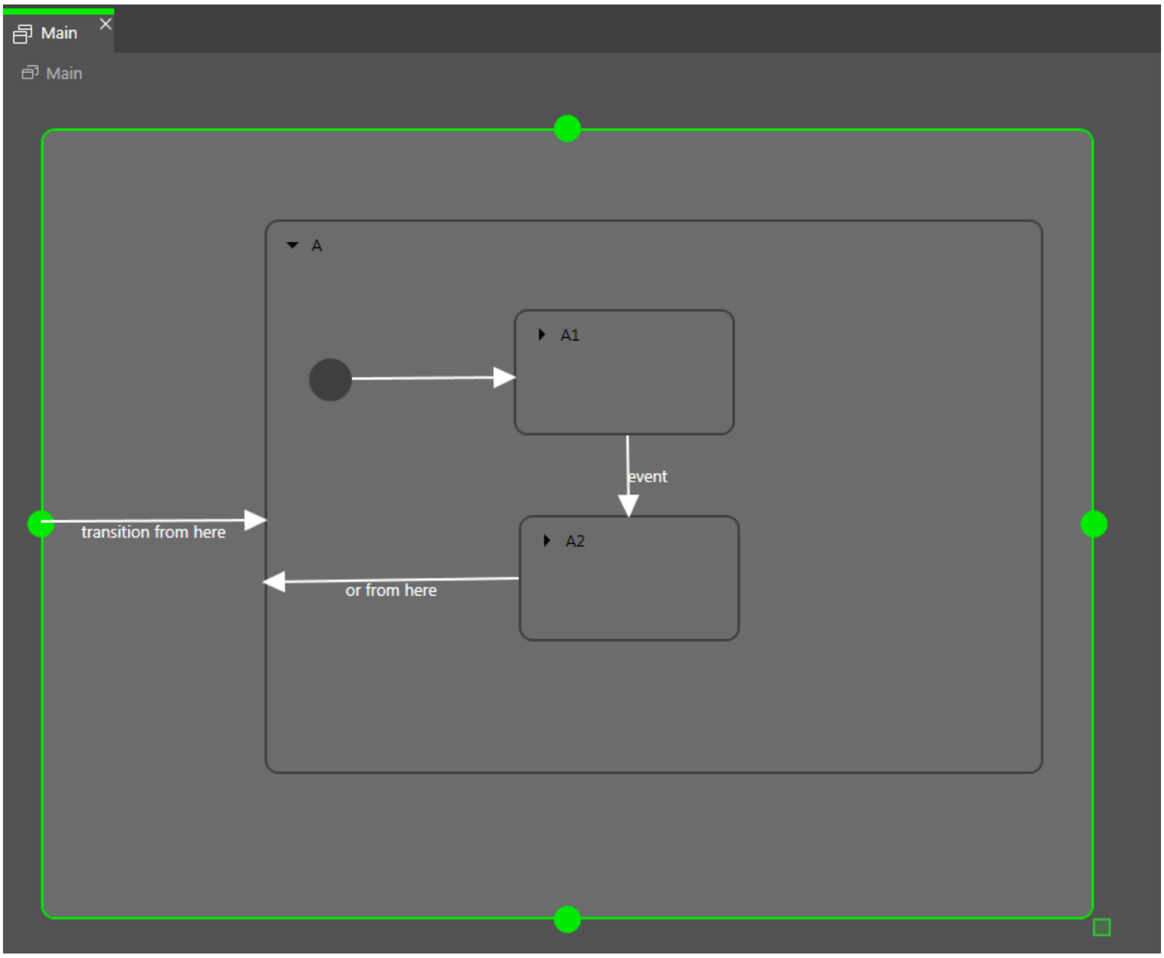


Figure 6.21. An example of an initial state

6.20.2.4. Final state

A final state is used to exit a compound state. If the final state of the state machine is entered, the state machine terminates. Any history states within the compound state are reset. A final state does not have any outgoing transitions.

A compound state can have only one final state. The final state is triggered by the following actions:

- A transition from a child state to the outside of the compound state (the transition with event z)
- An outgoing transition from the compound state (the transition with event y)
- A transition to the final state in a compound state (the transition with event x)

If a compound state contains a final state, the compound state must have an outgoing transition.

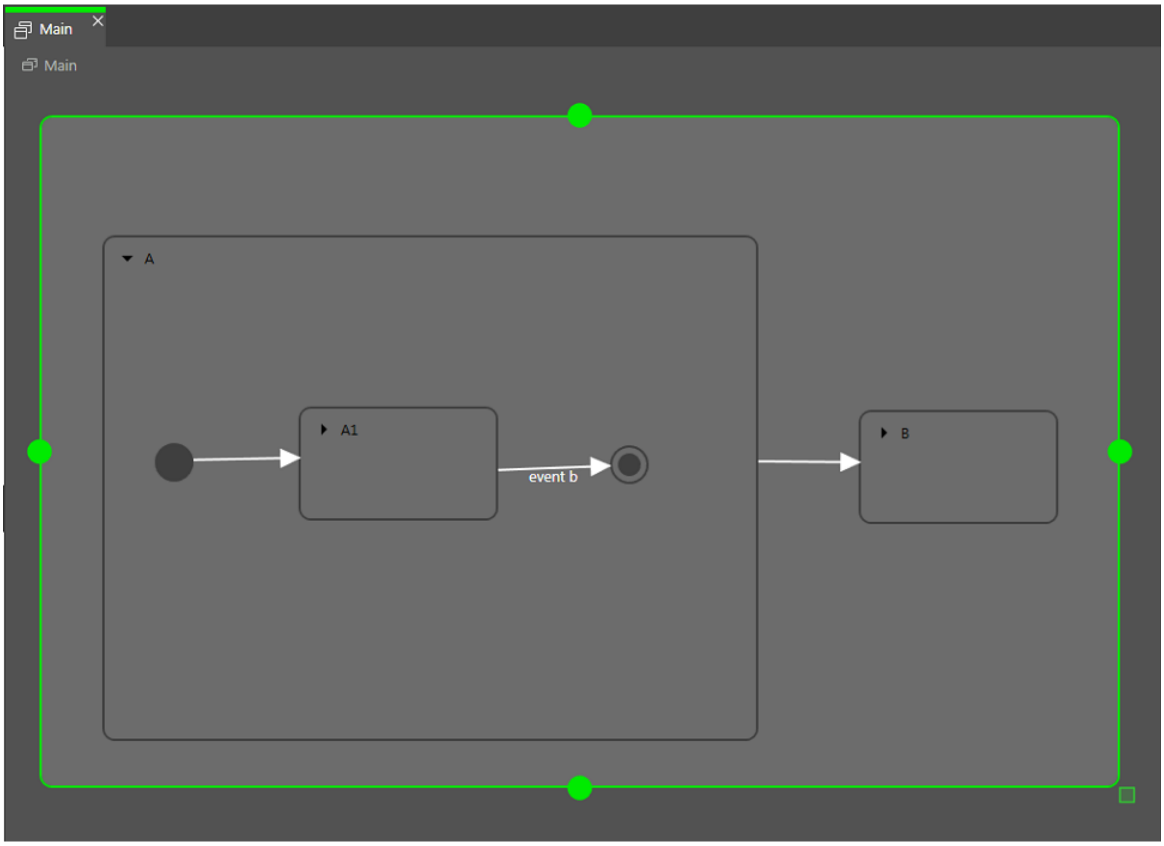


Figure 6.22. Final state usage in a compound state

6.20.2.5. Choice state

A choice state realizes a dynamic conditional branch. It is used when firing an event depends on conditions. A choice state is the connection between a source state and a destination state. A choice state can have several incoming and outgoing transitions. Every outgoing transition is assigned a condition and is only executed if the condition evaluates to `true`. One outgoing transition is the `else` transition. It is executed if all other conditions evaluate to `false`. The `else` transition is mandatory.

It is possible that several of the outgoing transitions are true, thus it is necessary to define the order in which the outgoing transitions are evaluated.

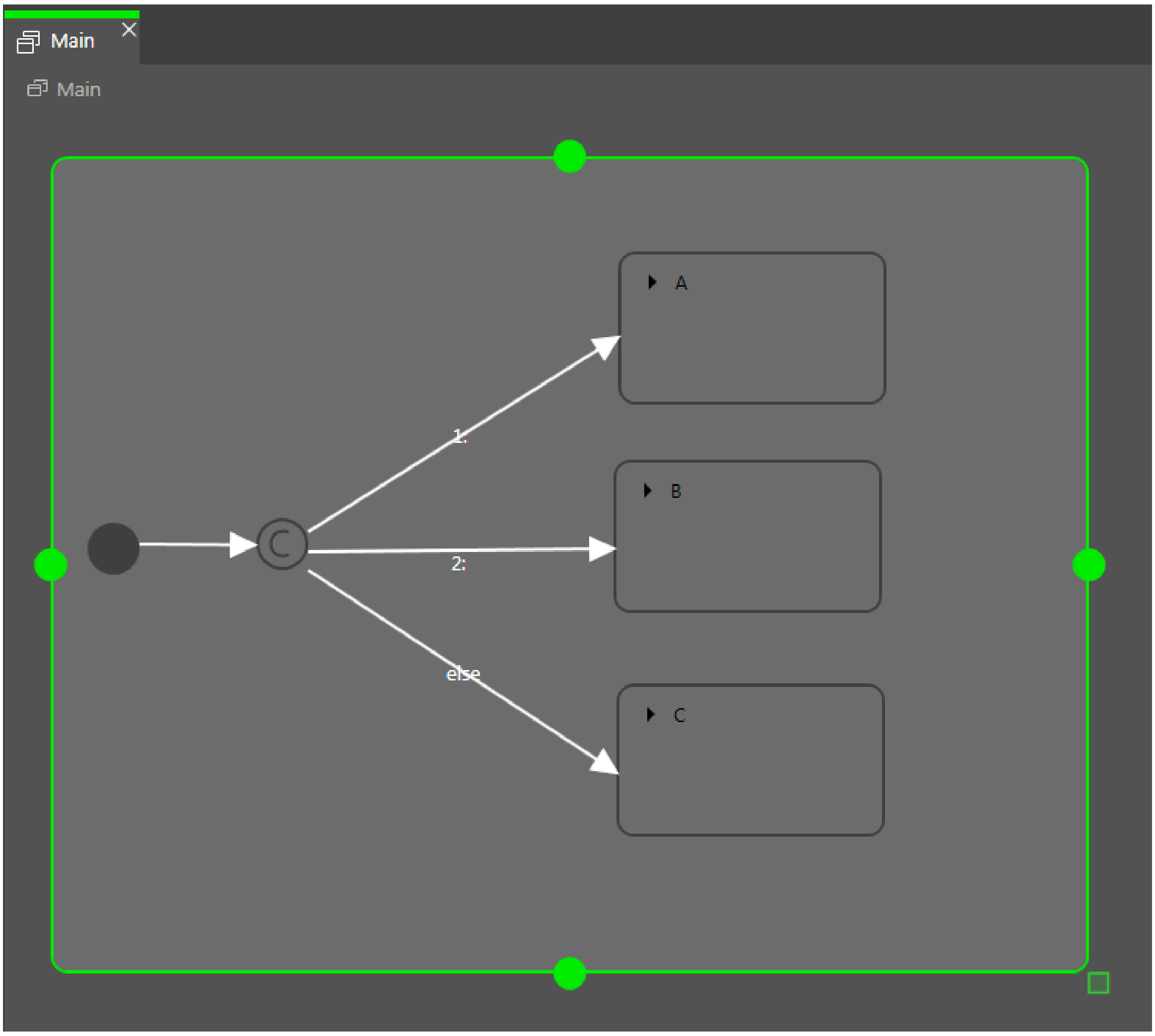


Figure 6.23. Choice state with incoming and outgoing transitions

6.20.2.6. History states

EB GUIDE supports two types of history states:

- Shallow history state stores the most recent active sub-state: the sub-state that was active just before exiting the compound state.
- Deep history state stores a compound state and its complete sub-hierarchy just before the compound state is exited.

When the parent state of a history state is entered for the first time, the last active child state is restored.

A shallow history state only remembers the last state that was active before compound state was exited. It cannot remember hierarchies.

A shallow history state restores the last active state recorded within a compound state. It has an outgoing default transition without conditions but can have multiple incoming transitions.

When a compound state is entered for the first time the shallow history state is empty. When an empty shallow history state is entered the shallow history state default transition determines the next state.

Example 6.27.
Shallow history state

A shallow history state can be used as follows.

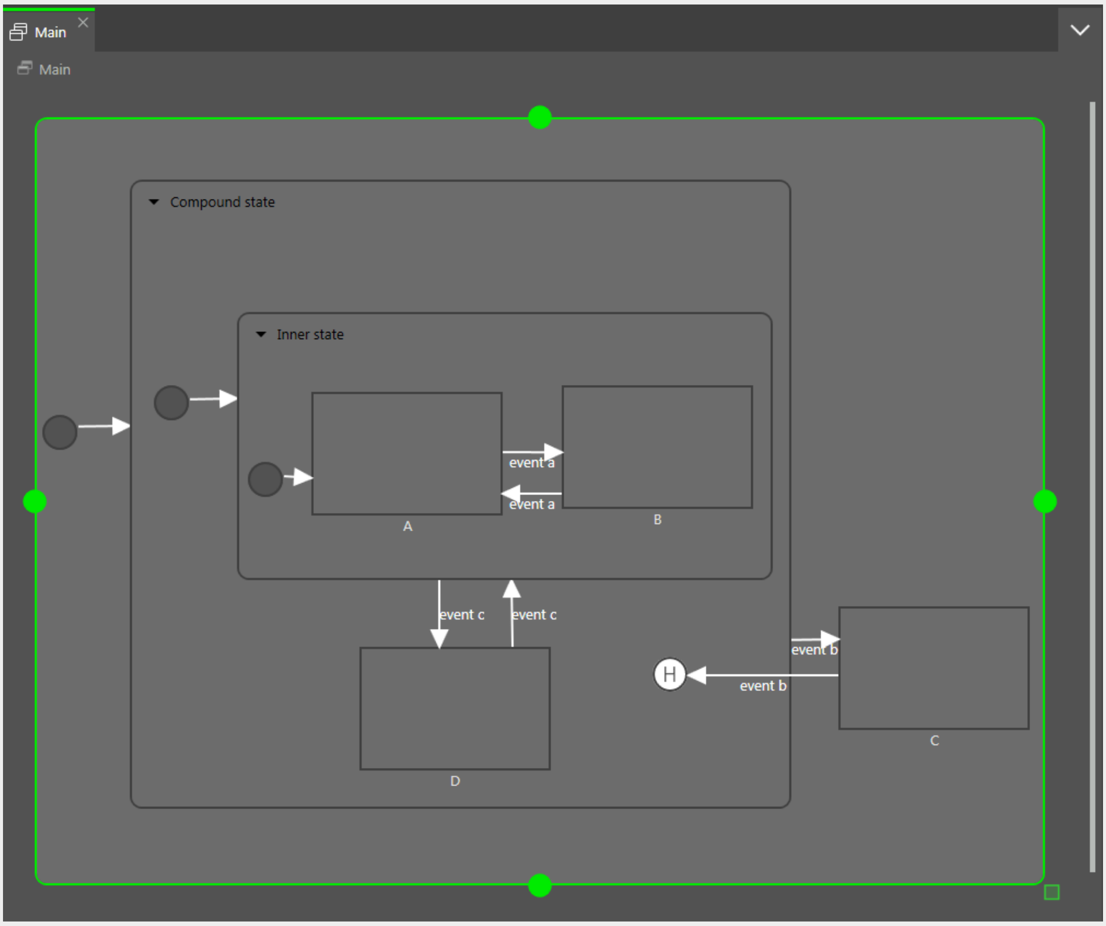


Figure 6.24. Shallow history state

- Case 1: The active state is `D`.
  1. `event b` is fired and state `C` is entered.
  2. `event b` is fired again and the shallow history state is entered.
  3. From the shallow history state, the state machine enters state `D` because state `D` was the last active state in `Compound State`.
- Case 2: The active state is `B`.

1. `event b` is fired and state `C` is entered.
2. `event b` is fired again the shallow history state is entered.
3. From the shallow history state, the state machine enters `Inner state` because shallow history states remember the state last active but cannot remember hierarchies.
4. Entering `Inner state` leads to state `A`.

A deep history state is able to save hierarchical histories.

Example 6.28.
Deep history state

A deep history state can be used as follows.

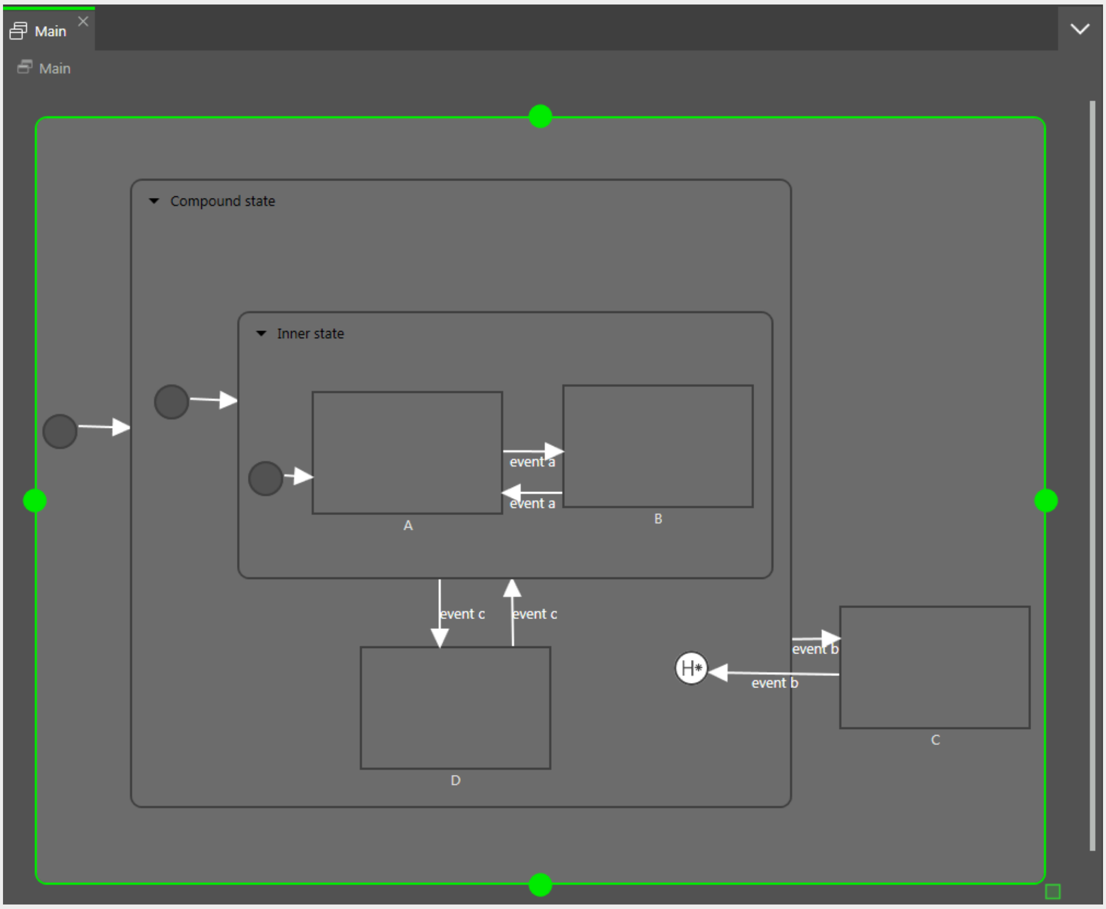


Figure 6.25. Deep history state

- Case 1: The active state is `D`.

1. `event b` is fired and state `C` is entered.
2. `event b` is fired again and the deep history state is entered.

3. From the deep history state, the state machine enters state `D` because state `D` was the last active state in `Compound State`.

- Case 2: The active state is `B`.

1. `event b` is fired and state `C` is entered.
2. `event b` is fired again and the deep history state is entered.
3. From the deep history state, the state machine enters state `B` because state `B` was the last active state and deep history state remembers state hierarchies.

One state can have either a shallow history state or deep history state. You can have a history state in a parent state and another history state in a child state.

6.20.3. Transitions

A transition is a directed relationship between a source state and a target state. It takes the state machine from one state to another. A transition has the following properties:

- A trigger to execute the transition

  A trigger can either be an event or the change of a datapool item.
- A condition that must be evaluated as `true` to execute the transition
- An action that is executed along with the transition

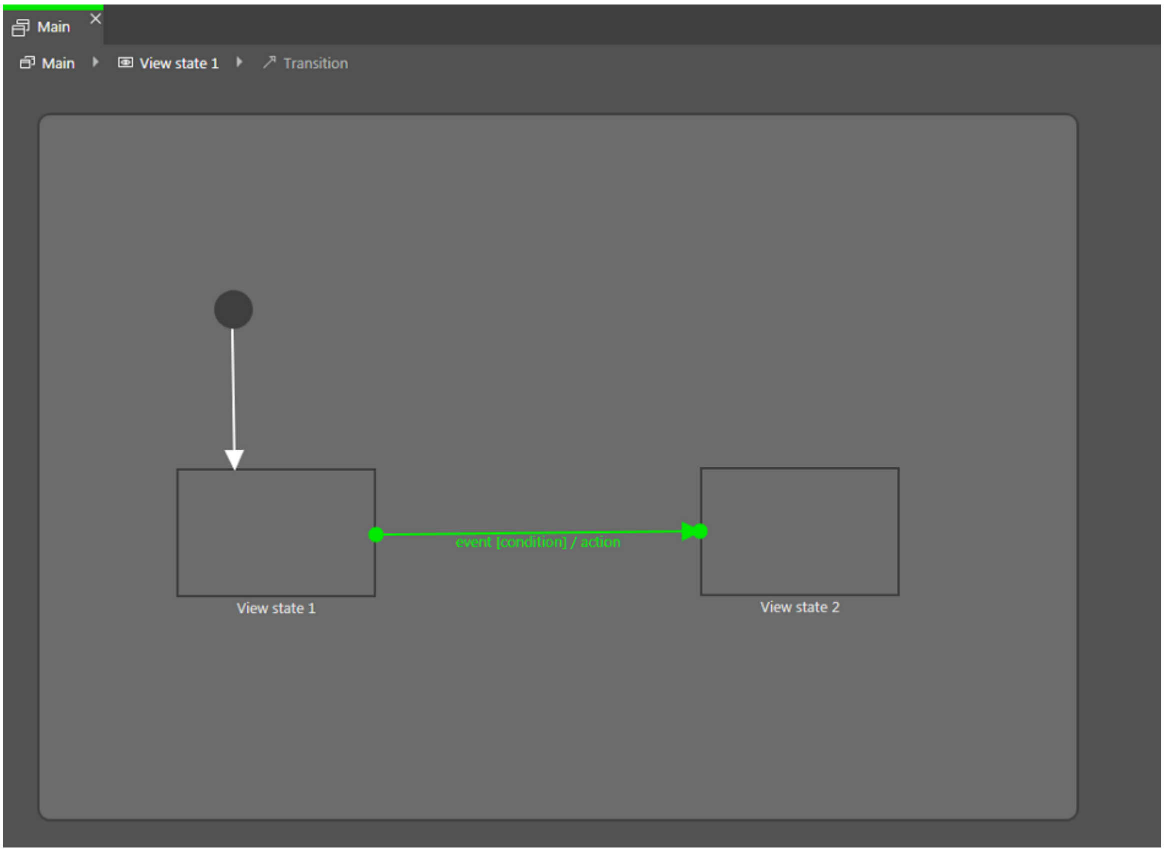


Figure 6.26. A transition

NOTE

Transitions are deterministic

It is not possible to have more than one transition from a particular source state for the same event even with different conditions. If the state machine is supposed to jump to different destination states depending on different conditions, use a choice state.

A state inherits all transitions from its parent states. If a number of states share the same transitions to another state, an enclosing compound state can be used to bundle the transitions and thus reduce the number of conditions.

Example 6.29.

Transition inheritance

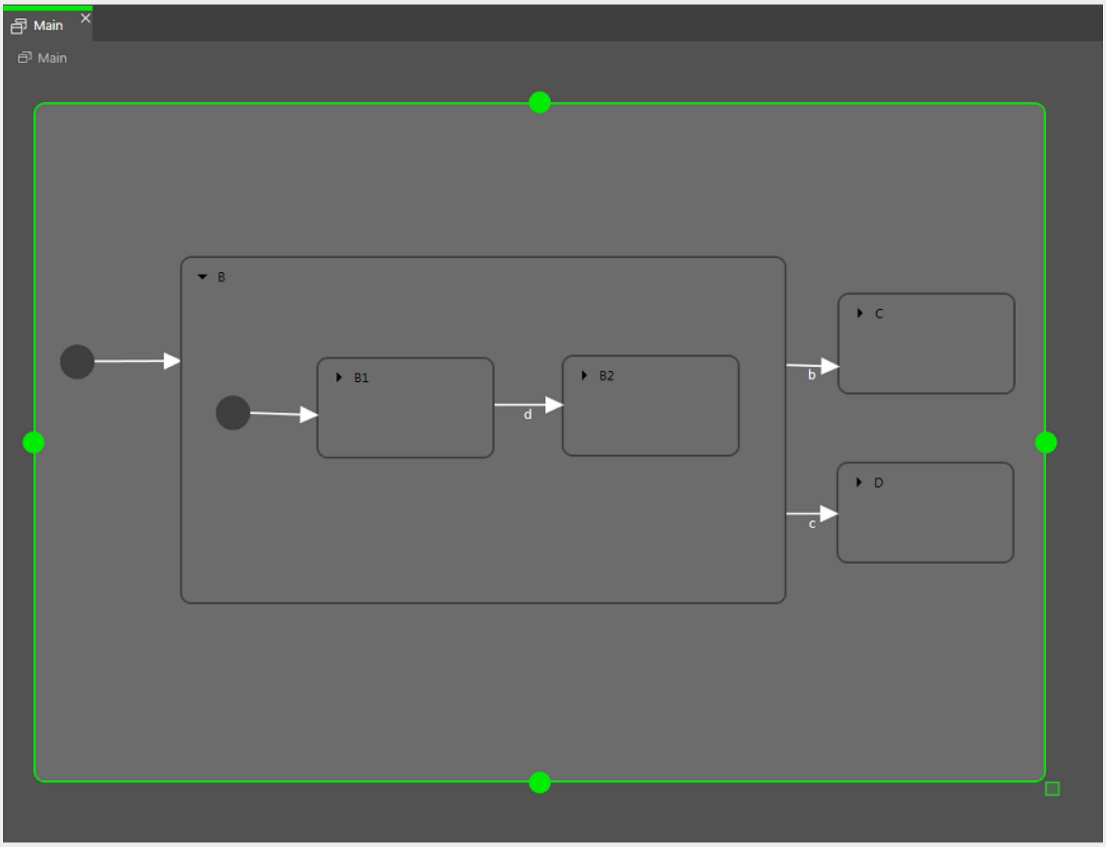


Figure 6.27. Transition inheritance

If the event `b` is fired while the state machine is in `State B1`, the transition to `State C` is executed because the child states `State B1` and `State B2` inherit the transitions of state `State B`.

If an internal transition from the child state uses the same event as the external transition from the parent state, transition inheritance is overridden.

Example 6.30.

Transition override

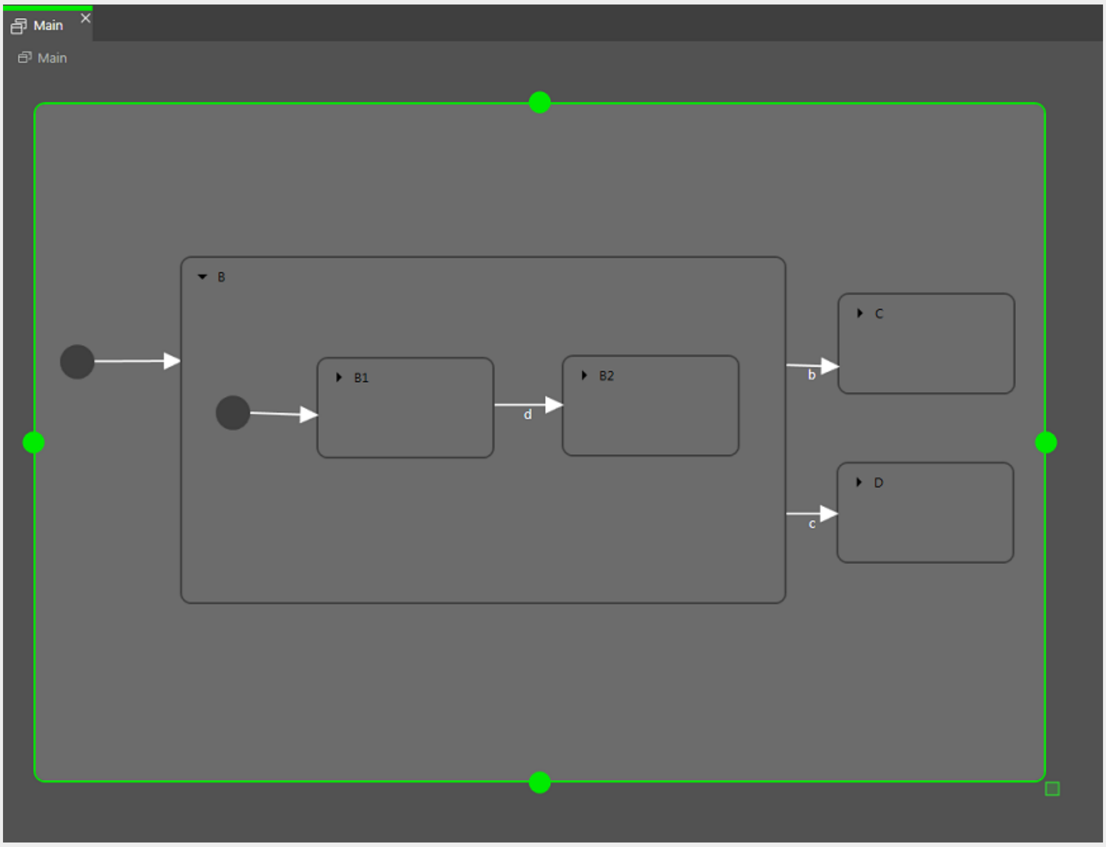


Figure 6.28. Transition override

If event `d` is fired while the state machine is in state `State B`, the transition to `State C` is executed.

If event `d` is fired while the state machine is in state `State B1`, the transition to `State B2` is executed instead of the transition to `State C`. Because the two transitions have the same name, the inner transition overrides the outer one.

NOTE

Execution hierarchy

In a state machine the hierarchy for the execution of transitions that use the same event is always from the inside out. This means internal transitions are preferred compared to external transitions.

There are different types of transitions:

- Default transition

  A default transition is triggered automatically and not by any event or datapool item update. It has no condition, but can have an action. It is used with initial state, final state, choice state, and history states.

- Choice transition

  A choice transition is an outgoing transition with a condition assigned to it. Its source state is a choice state. Choice transitions are triggered by the evaluation of their condition. They result in an action. The first choice transition that has condition `true` is executed.

- Else transition

  An else transition is the mandatory counterpart of a choice transition. Every choice state needs to have one else transition which is executed if the conditions of all its choice transitions evaluate to `false`.

- Internal transition

  An internal transition is a transition that has no destination state and thus does not change the active state. The purpose of an internal transition is to react to an event without leaving the present state. It can have a condition and it results in an action.

  It is possible to have several internal transitions for the same event in a state. The order of execution is defined.

- Self transition

  A self transition is a transition with the same state as source state and destination state. Unlike an internal transition, a self transition leaves and re-enters the state and thus executes its entry and exit actions.

6.20.4. Execution of a state machine

When a state machine is executed, at any moment in time it has exactly one active state. A state machine is event-driven.

The state machine cycle is as follows:

1. The state machine is started by entering its initial state.
2. The state machine waits for incoming events.
   a. Internal transitions are found.
      i. Start at the current state and search for the first internal transition that is triggered by the current event and has condition `true`. If such a transition is found, it is executed.
      ii. If no transition is found, go to the parent state and search for the first internal transition that is triggered by the current event and has condition `true`.
      iii. If no transition is found, repeat the previous step until the top-level state is reached.
   b. Internal transitions are processed.

      Executing an internal transition only triggers the action that is connected to the internal transition. The state is not exited and re-entered.

c. Transitions are found.

   i. Start at the current state and search for a transition that is triggered by the current event and has condition `true`. If such a transition is found, it is executed.

   ii. If no transition is found, go up to the parent state and search for a transition.

   iii. Repeat the previous step until the first fitting transition is found.

d. Transitions are processed.

   Executing a transition changes the state machine from one state to another state. The source state is exited and the destination state is entered.

   A transition is only executed when its corresponding event is fired and the condition is evaluated to `true`.

   A transition can exit and enter several compound states in the state hierarchy. Between the exit cascade and the entry cascade the transition's action is executed.

   Entering a state can require a subsequent transition, for example entering a compound state requires executing the transition of an initial state as a subsequent transition. A chain of several subsequent transitions is possible.

3. The state machine stops when the final state of the state machine is reached.

If a transition crosses several states in the state hierarchy, a cascade of exit and entry actions is executed.

Example 6.31.

Executing a transition

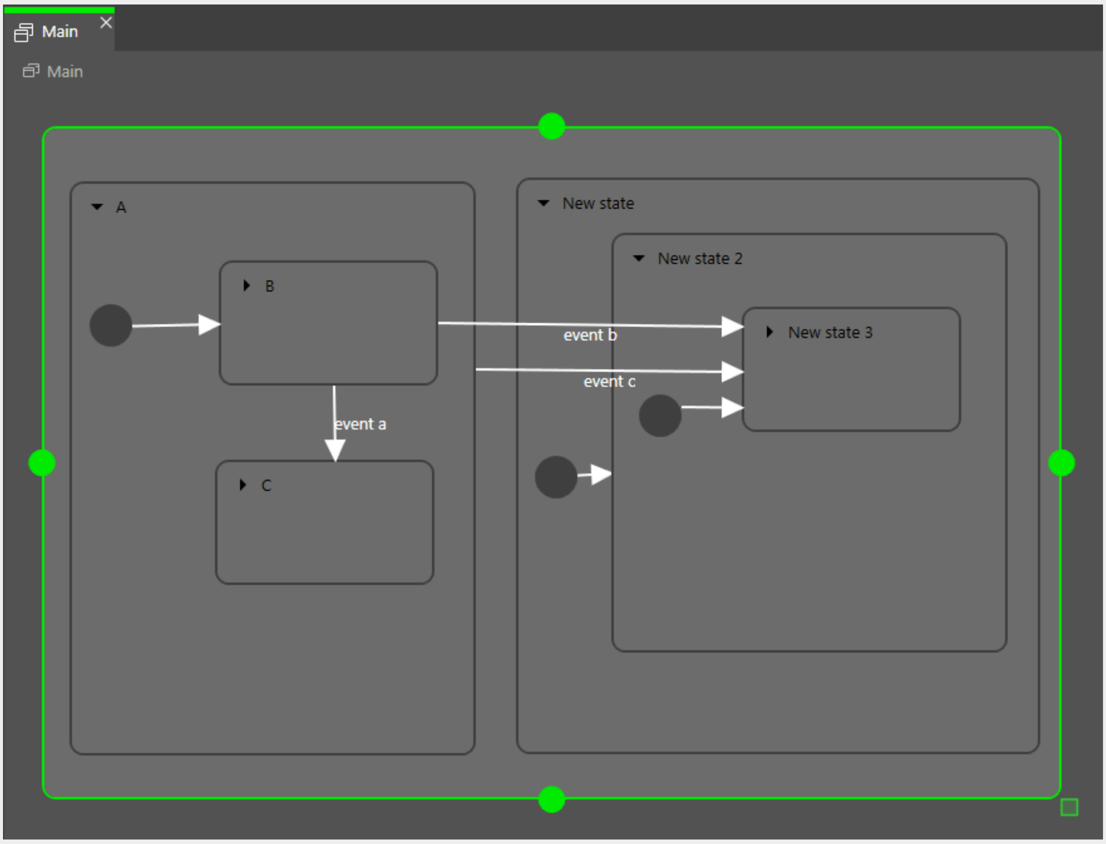


Figure 6.29. Executing a transition

When `event a` is fired, the following happens:

1. State `B` is exited.
2. State `C` is entered.

When `event b` is fired, the following happens:

1. State `B` is exited.
2. State `A` is exited.
3. State `New state` is entered.
4. State `New state 2` is entered.
5. State `New state 3` is entered.

When `event c` is fired, the following happens:

1. If state `B` or state `C` is active, state `B` or state `C` is exited.
2. State `A` is exited.

3. State `New state` is entered.
4. State `New state 2` is entered.
5. State `New state 3` is entered.

Example 6.32.
Executing a transition

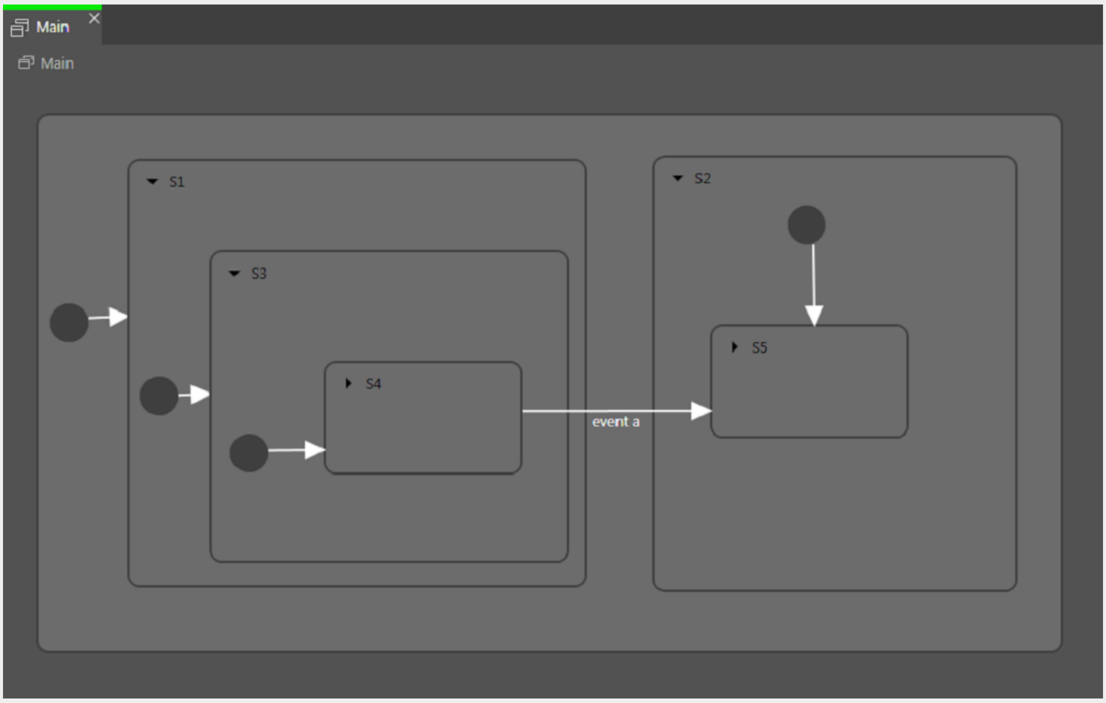


Figure 6.30. Executing a transition

When `event a` triggers the transition, the following happens:

1. State `S4` is exited.
2. State `S3` is exited.
3. State `S1` is exited.
4. State `S2` is entered.
5. State `S5` is entered.

Example 6.33.

Executing a transition

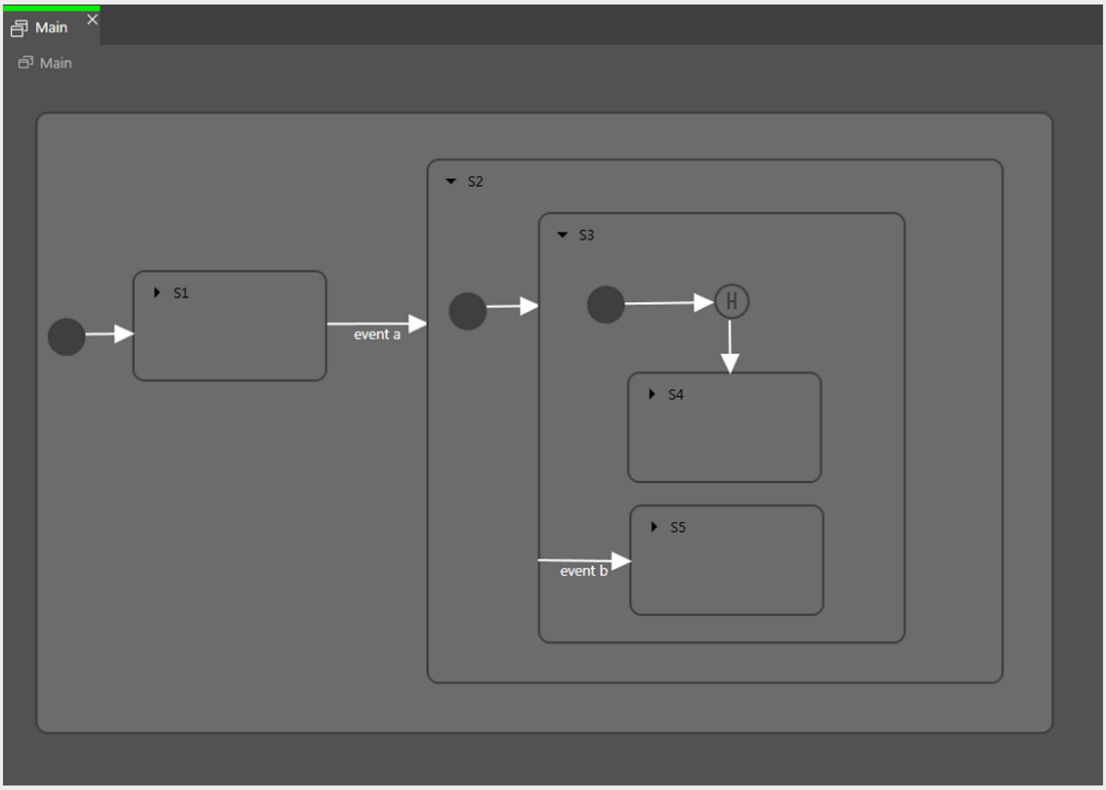


Figure 6.31. Executing a transition

The transition that is triggered by `event a` causes the following transition sequence:

1. The state machine goes to state `S2`.
2. The default transition leads to state `S3`.
3. The next default transition enters the shallow history state.
4. Shallow history state restores the last active state of state `S3`, either state `S4` or state `S5`.

For each step the entry-exit-cascade is executed separately.

6.20.5. EB GUIDE notation in comparison to UML notation

In this section the EB GUIDE notation is compared to the Unified Modeling Language (UML) 2.5 notation.

6.20.5.1. Supported elements

The following table shows all UML 2.5 elements that are supported by EB GUIDE. The names of some elements deviate from the naming convention in UML 2.5, but the functionality behind these elements remains the same:

Name in EB GUIDE	Name in UML 2.5
Initial state	Initial (pseudostate)
Final state	Final state
Compound state	State
Choice state	Choice (pseudostate)
Deep history state	DeepHistory (pseudostate)
Shallow history state	ShallowHistory (pseudostate)
Internal transition	Internal transition
Transition	External/local transition [a]

[a]EB GUIDE does not differentiate between external and local transitions.

6.20.5.2. Not supported elements

The following UML 2.5 elements are not supported in EB GUIDE:

- Join
- Fork
- Junction
- Entry point
- Exit point
- Terminate

6.20.5.3. Deviations

Some elements of the UML 2.5 notation are not implemented in EB GUIDE. But the functionality of these elements can be modeled with EB GUIDE concepts.

Concept in UML 2.5	Workaround with EB GUIDE
Parallel states	Concept is implemented using dynamic state machines.
Number of triggers per transition	Concept is implemented using EB GUIDE Script in a datapool item or a view.

Concept in UML 2.5	Workaround with EB GUIDE
Time triggers at transitions	Concept is implemented using EB GUIDE Script (`fire_delayed`) in a state machine, a datapool item, a transition, or a view.

6.21. Touch input

EB GUIDE supports two types of touch input: Touch gestures and multi-touch input.

Each touch gesture is represented in EB GUIDE Studio as a widget feature. Enabling the widget feature adds a set of properties to a widget.

The gestures are divided into two basic types:

- Non-path gestures
- Path gestures

6.21.1. Non-path gestures

EB GUIDE implements the following non-path gestures:

- Flick
- Pinch
- Rotate
- Hold
- Long hold

Non-path gestures include multi-touch and single-touch gestures. Multi-touch gestures require an input device that supports multi-touch input. Single-touch gestures work with any supported input device.

Each gesture reacts independently of the others. If several gestures are enabled, the modeler is responsible to make sure that the EB GUIDE model behaves consistently.

6.21.2. Path gestures

Path gestures are shapes drawn by a finger on a touch screen or entered by some other input device. When a widget has the widget feature enabled, the user can enter a shape starting on the widget. The shape has to exceed a configurable minimal bounding box to be considered by the path gesture recognizer. The shape is matched against a set of known shapes and, if a match is found, a gesture is recognized.

For instructions, see section 12.3, “Tutorial: Modeling a path gesture”.

6.21.3. Input processing and gestures

Gesture recognition runs in parallel to ordinary input processing. Each gesture can request that the contact involved in the gesture is removed from ordinary input processing. The moment at which a gesture requests contact removal depends on the actual gesture and for some gestures this can be configured.

Contact removal is only relevant for fingers involved in a gesture. Once a contact is removed, it is ignored by ordinary input handling until a release event is received for the contact. On a touch screen without proximity support this implies that a contact, once removed, does not trigger any further touch reactions.

TIP

Removing a contact from ordinary input processing

Consider a window with a button and a widget feature for gestures. When a contact is involved in a gesture it should not cause the action associated with the button to be triggered, even if the contact is released while on the button.

6.21.4. Multi-touch input

EB GUIDE is able to handle multi-touch input, if a compatible multi-touch input device is used.

Multi-touch is the ability of a surface to recognize and track more than one point of contact on an input device. The typical scenario are multiple fingers touching a touch screen.

- Multi-touch event handling

  Multi-touch events are dispatched using the mechanism for touch events, in the same way events from the mouse and from single-touch touch screens are dispatched. The only difference is that each contact triggers touch reactions independently of all others. To be able to distinguish individual contacts, each touch reaction is supplied with a parameter called `fingerid`.

- Finger ID

  Each contact tracked by an input device is assigned a number that identifies it. This identifier is called `fingerid` and is unique per input device. However, the same value can be assigned to another contact at a later time when it is no longer in use.

Consider the extra touch interaction sequences the end user is allowed to make when multi-touch input is enabled. They include the following:

- The end user can interact with multiple elements of the interface at the same time, for example press a button while scrolling in a list.

- The end user can place multiple fingers on a single widget.

Two typical situations where this manifests are scrolling and dragging. They can be handled correctly by employing `fingerid`. Depending on the required behavior, possible solutions include the following:

- Allow only the first finger that pressed a widget to do scrolling and/or dragging.
- Always use the last finger to land on a widget to do scrolling and/or dragging. This is easily achieved by a slight modification of the previous approach.

6.22. Widgets

Widgets are the basic graphical elements an EB GUIDE model is composed of.

It is possible to customize widgets. Editing the properties of a widget adapts the widget to individual needs. The following elements are the example properties when being touched or moved:

- Size
- Color
- Layout
- Behavior

It is possible to combine widgets. Out of small building blocks, complex structures are created. For example, it is possible to make up a button of the following elements:

- Ellipse
- Image
- Label
- Rectangle

It is possible to nest widgets. These subordinate widgets are referred to as child widgets and the superordinate widgets are referred to as parent widgets in a widget hierarchy.

6.22.1. View

A view is the topmost widget of each scene. While modeling, the following elements are placed into views:

- Basic widgets
- 3D widgets
- Animations

- Widget templates

Every view is associated to exactly one view state. A view cannot exist without a view state.

NOTE

Changing the size of a view

In EB GUIDE Studio, to get a close-up view or to see more, you have the possibility to increase or decrease the size of a view. To zoom in and zoom out, use the slider or click the text box at the bottom of the view. The default zoom level is 100%. Alternatively, use the **Ctrl++** to zoom in, **Ctrl+-** to zoom out and **Ctrl+0** to reset the zoom level to 100%.

NOTE

Aligning elements on a master image

In EB GUIDE Studio, to align elements evenly such as basic widgets and 3D widgets, you have the possibility to add a master image to a view. To add a master image to a view, click at the bottom of the view. To hide the master image, check or clear the check box. If you close the view, you need to add the master image again.

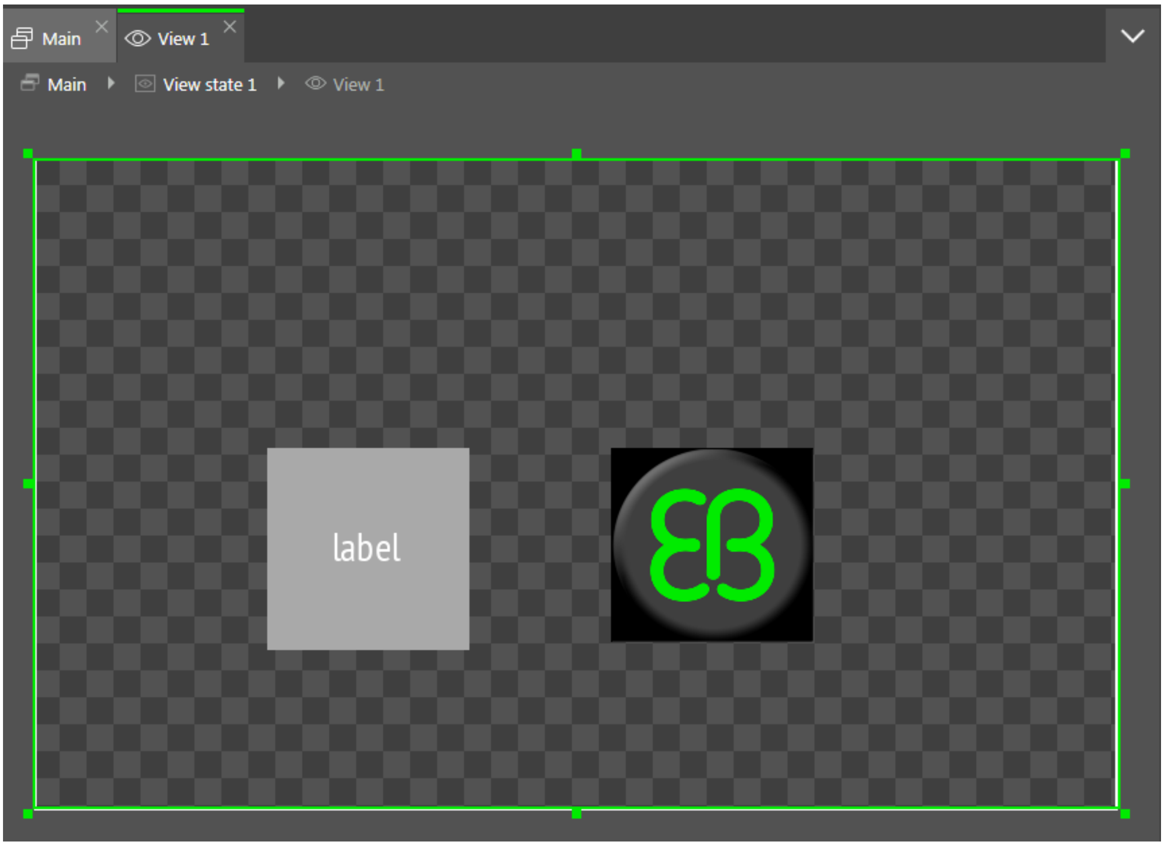


Figure 6.32. A view that contains a rectangle, a label, and an image

6.22.2. Widget categories

In the **Toolbox**, widgets are grouped by categories. The following categories are available.

- Basic widgets

  The following elements are the basic widgets:

  - Alpha mask
  - Animation
  - Container
  - Ellipse
  - Image
  - Instantiator
  - Label
  - Rectangle
- 3D widgets

  To display a 3D graphic, use widgets that are contained in the **3D widgets** category. The **3D widgets** are the following elements:

  - Ambient light
  - Camera
  - Directional light
  - Image-based light
  - Material
  - Mesh
  - PBR GGX material
  - PBR Phong material
  - Point light
  - Scene graph
  - Scene graph node
  - Spot light

  NOTE **Supported renderers**

  To display 3D graphics, OpenGL ES 2.0 or higher is required. Make sure that your graphics driver is compatible to the version of the renderer.
- Widget templates

The **Templates** category contains widget templates. It is only visible if widget templates are defined.

- Custom widgets

  The **Custom widgets** category contains customized widgets and is therefore only visible when customized widgets are added to the project. For more information, see our website https://www.elektrobit.com/ebguide/learn/resources/.

For instructions, see section 8.1, “Working with widgets”.

6.22.3. Widget properties

A widget is defined by a set of properties which specify the appearance and behavior of the widget. The **Properties** component displays the properties of the currently focused widget and allows editing the properties.

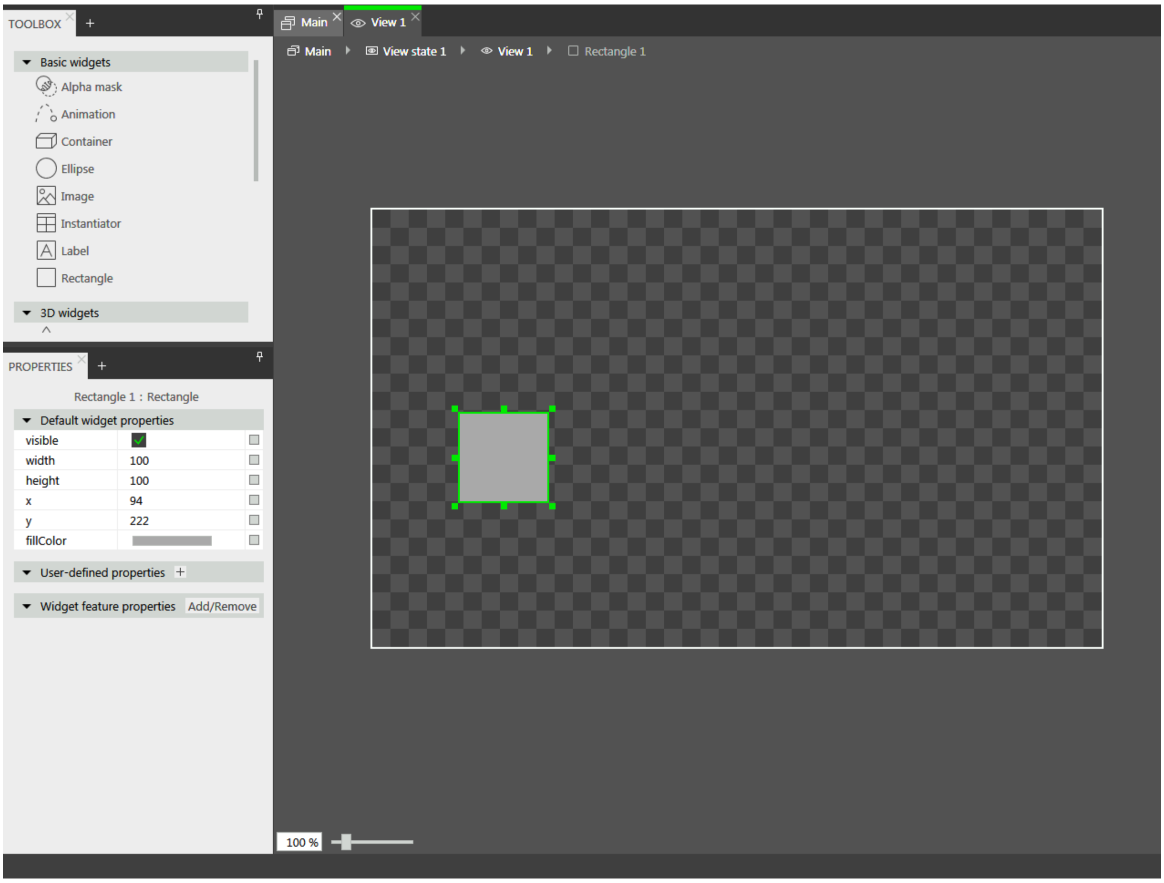


Figure 6.33. A rectangle and its properties

There are three types of widget properties:

- Default widget properties are created along with each widget instance. For a list of default properties for all widgets, see section 13.12, “Widgets”.

- User-defined widget properties are created by the modeler in addition to the default ones.
- Widget feature properties are created by EB GUIDE Studio when the modeler adds a widget feature to a widget. Widget feature properties are grouped by categories. Widget features add more functionality for the appearance and behavior of widgets.

  Example 6.34.
  Touched widget feature

  The **Touched** widget feature defines if and how a widget reacts to being touched. It adds four properties. The boolean property `touchable` determines if the widget reacts on touch input. The boolean property `touched` is set during run-time by EB GUIDE if the widget is currently touched. The two integer properties `touchPolicy` and `touchBehavior` determine how the widget reacts on touch input.

6.22.4. Widget templates

A widget template allows the definition of a customized widget that can be used multiple times in an EB GUIDE model. You have the possibility to define templates on the basis of existing widgets or derive a new template from an existing one. After creating, you modify the template according to your needs, for example, by adding properties or widget features. Widget templates thus allow you to build a library of complex widgets.

A widget template has a template interface. The template interface contains the properties of the template which are visible and accessible in widget instances. A widget instance thus inherits the properties of its template's interface. Inherited properties are called template properties. Template properties are marked with the button.

When you change the value of a template property, the property is turned into a local property. Local properties are marked with the button.

Example 6.35.
Relation of the properties of a widget template and its instances

You add a widget template `Square` to the EB GUIDE model. Let `Square` have a property `color`. `color` is added to the template interface. Let the value of `color` be `red`.

You add an instance of the widget template `Square` to a view. The instance is named `BlueSquare`.

- `BlueSquare` inherits `color` with the value `red`.
- Change the value of `color` in the `Square` template to `green`.

  => The value of `color` in `BlueSquare` changes to `green`, too.
- Change the value of `color` in `BlueSquare` to `blue`.

  Change the value of `color` in the `Square` template to `yellow`.

=> The value of `color` in `BlueSquare` remains `blue`.

For instructions, see section 8.7, “Re-using a widget”.

6.22.5. Widget features

It is possible to extend widgets and widget templates in their functionality using widget features. Widget features have predefined widget properties. Widget features are grouped into categories.

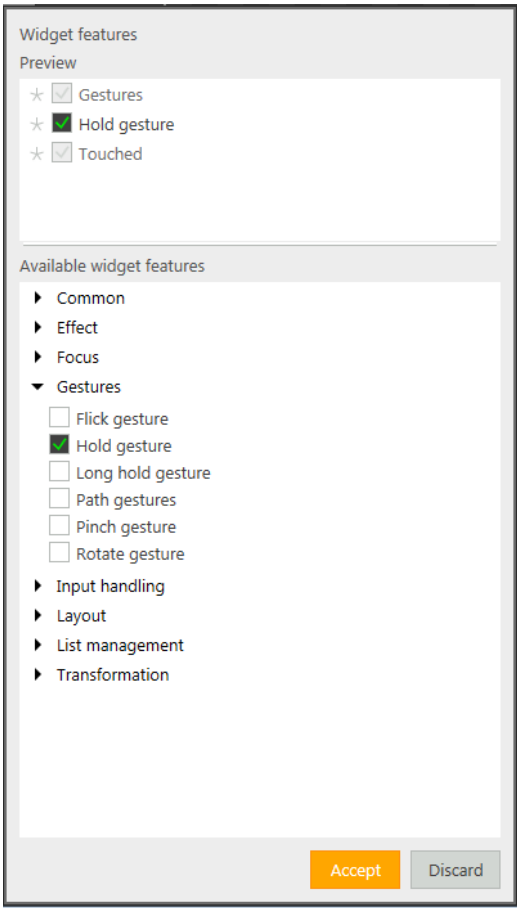


Figure 6.34. Widget features

If you add a widget feature to a widget template, any created widget template instance inherits the added widget feature. Note that you cannot add widget features to a widget template instance or to a template that was created from a template.

Restrictions for usage of widget features are as follows:

- Widget features do not have an inheritance hierarchy.
- It is not possible to add a widget feature more than once per widget.
- Some widget features are interdependent. That means, to add one widget feature, you have to add another, or widget features may exclude each other.
- It is possible to restrict widget features to a particular type of widgets.
- It is not possible to activate or deactivate widget features during run-time.

By default all widget features are disabled. If you need a specific widget feature, you must add it to a widget.

For instructions, see section 8.3, “Extending a widget by widget features”. For a list of all widget features, see section 13.13, “Widget features”.

6.22.5.1. Focus widget feature category

In EB GUIDE Studio you model the focus management of the widgets using the **Focus** widget features: **Auto focus** and **User-defined focus**.

The following two focus directions are available:

1. Forward direction: The next focusable widget is focused.
2. Backward direction: The previous focusable widget is focused.

The **Auto focus** and **User-defined focus** widget features provide a configuration for how the focus is handled for the forward direction. For the backward directions, the same focus order is used but only in reverse direction.

The **Focus** widget features have the following characteristics:

Auto focus

In this policy the focus is distributed between the focusable widgets from left to right starting with the top row. The order is defined through the structure of the widget tree.

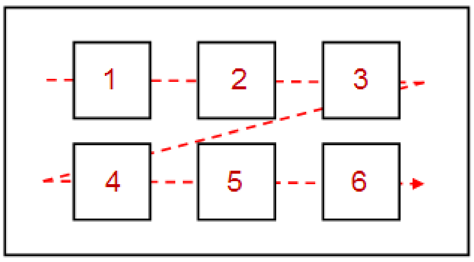


Figure 6.35. The policy of the **Auto focus** widget feature

Focusable child widgets cannot be skipped. Invisible widgets, widgets with disabled `focused` property, and widgets without the **Focused** widget feature are not recognized as valid focusable widgets. Thus they are skipped over when the currently focused widget is determined.

User-defined focus

Due to view complexity the focus sequencing through the auto focus policy may be quite difficult. In this case it is useful to determine a user-defined focus order.

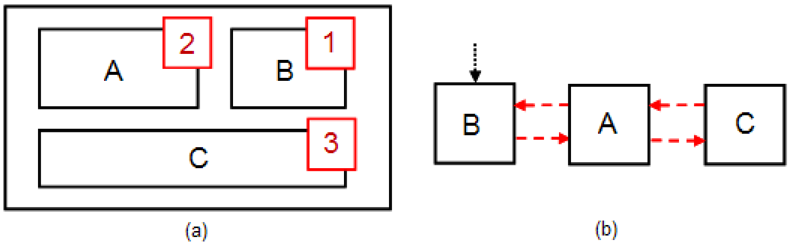


Figure 6.36. The policy of the **User-defined focus** widget feature

In figure 6.36, “The policy of the **User-defined focus** widget feature”, (a) shows the view, while (b) shows the focus order. The order, in which the focus changes are processed, may differ from the widget tree structure.

When widgets within a widget hierarchy are marked as focusable, they are part of a focus hierarchy. This focus hierarchy consists of focusable widgets and a focus policy, the **Auto focus** widget feature or the **User-defined focus** widget feature, that defines how the focus is handled within the hierarchy. It is possible to nest focus hierarchies.

6.22.5.2. List management widget feature category

The **Line index** and **Template index** widget features allow you to connect data, for example images, song titles, to the corresponding dynamically created line templates of an instantiator.

Line index

The **Line index** widget feature is used to customize the line templates of the instantiator widget. The **Line index** widget feature defines the unique position for each line of your list or table.

Example 6.36.
Line index widget feature

If you want to model a list, you would expect that each entry of the list has a specific value that reflects the entry in a list property. To access a certain entry in a list, the instance of the line template needs to know which of the instantiator's child it is. The **Line index** widget feature adds the `lineIndex` property. While the instantiator creates the instances of line templates, it fills `lineIndex` with values: The index starts with zero for the first instance. If you have two elements in the instantiator, the second element receives the `lineIndex` value `1`.

For instructions, see section 12.4, “Tutorial: Creating a list with dynamic content”.

Template index

The **Template index** widget feature allows complex data abstraction. For very complex lists or tables, to visualize an entry or a set of entries, you require more than one data list. For example, a table with mixed image and text content requires a list of images and a list of strings. To cover such complex cases, the **Template index** widget feature provides the property `lineTemplateIndex`.

Example 6.37.
Template index widget feature

If you model a list using an instantiator with the property `lineMapping` set to `0|1` and the property `numItems` set to `5`, the `lineTemplateIndex` results in `0|0|1|1|2`.

7. Modeling HMI behavior

NOTE

Default window layout
All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

7.1. Modeling a state machine

7.1.1. Adding a state machine

Adding a state machine

Step 1
In the **Navigation** component, go to **State machines**, and click +.

A menu expands.

Step 2
Select a type for the state machine.

A new state machine of the selected type is added.

Step 3
Rename the state machine.

7.1.2. Adding a dynamic state machine

Dynamic state machines run in parallel to other state machines and can be started (pushed) and stopped (popped) during run-time.

Adding a dynamic state machine

You use a dynamic state machine for example to show an error message that overlays the regular screen.

Prerequisite:

- A state machine, view state, or compound state is added to the EB GUIDE model.

Step 1
In the **Navigation** component, go to **Dynamic state machines**, and click +.

A menu expands.

Step 2
Select a type for the dynamic state machine.

A new dynamic state machine of the selected type is added.

Step 3
In the **Navigation** component, click the state machine, view state, or compound state to which you want to run in parallel the dynamic state machine.

Step 4
In the **Properties** component, select the `Dynamic state machine list` check box.

With these steps done, you use EB GUIDE Script functions that are related to dynamic state machines.

For details, see section 12.1, "Tutorial: Adding a dynamic state machine".

7.1.3. Defining an entry action for a state machine

Defining an entry action for a state machine

Step 1
Select a state machine.

Step 2
In the **Properties** component, go to the **Entry action** property, and click + .

A script editor opens.

Step 3
Enter an action using EB GUIDE Script.

For background information, see section 6.16, "Scripting language EB GUIDE Script".

Step 4
Click **Accept**.

You defined an entry action for a state machine.

7.1.4. Defining an exit action for a state machine

Defining an exit action for a state machine

Step 1

Select a state machine.

Step 2

In the **Properties** component, go to the **Exit action** property, and click + .

A script editor opens.

Step 3

Enter an action using EB GUIDE Script.

For background information, see section 6.16, “Scripting language EB GUIDE Script”.

Step 4

Click **Accept**.

You defined an exit action for a state machine.

7.1.5. Deleting a state machine

Deleting a state machine

Step 1

In the **Navigation** component, right-click the state machine.

Step 2

In the context menu, click **Delete**.

The state machine is deleted.

7.2. Modeling states

7.2.1. Adding a state

Adding a state

Prerequisite:

- The content area displays a state machine.

Step 1
Drag a state from the **Toolbox** into the state machine.

A state is added to the state machine.

NOTE

Initial state, final state, and history states are unique
You can insert initial state, final state, and history states only once per compound state.

TIP

Copying and finding states
Alternatively, you can copy and paste an existing state using the context menu or **Ctrl+C** and **Ctrl+V**.

To find a specific state within your EB GUIDE model, enter the name of the state in the search box or use **Ctrl+F**. To jump to a state, double-click it in the hit list.

7.2.2. Adding a state to a compound state

Adding a state to a compound state

To create a state hierarchy, you create a state as a child to another state. You do so by adding a state to a compound state.

Prerequisite:

- The content area displays a state machine.
- The state machine contains a compound state.

Step 1
In the **Navigation** component, double-click the compound state.

The compound state expands in the content area.

Step 2

Drag a state from the **Toolbox** into the compound state.

The state is added as a child state to the compound state.

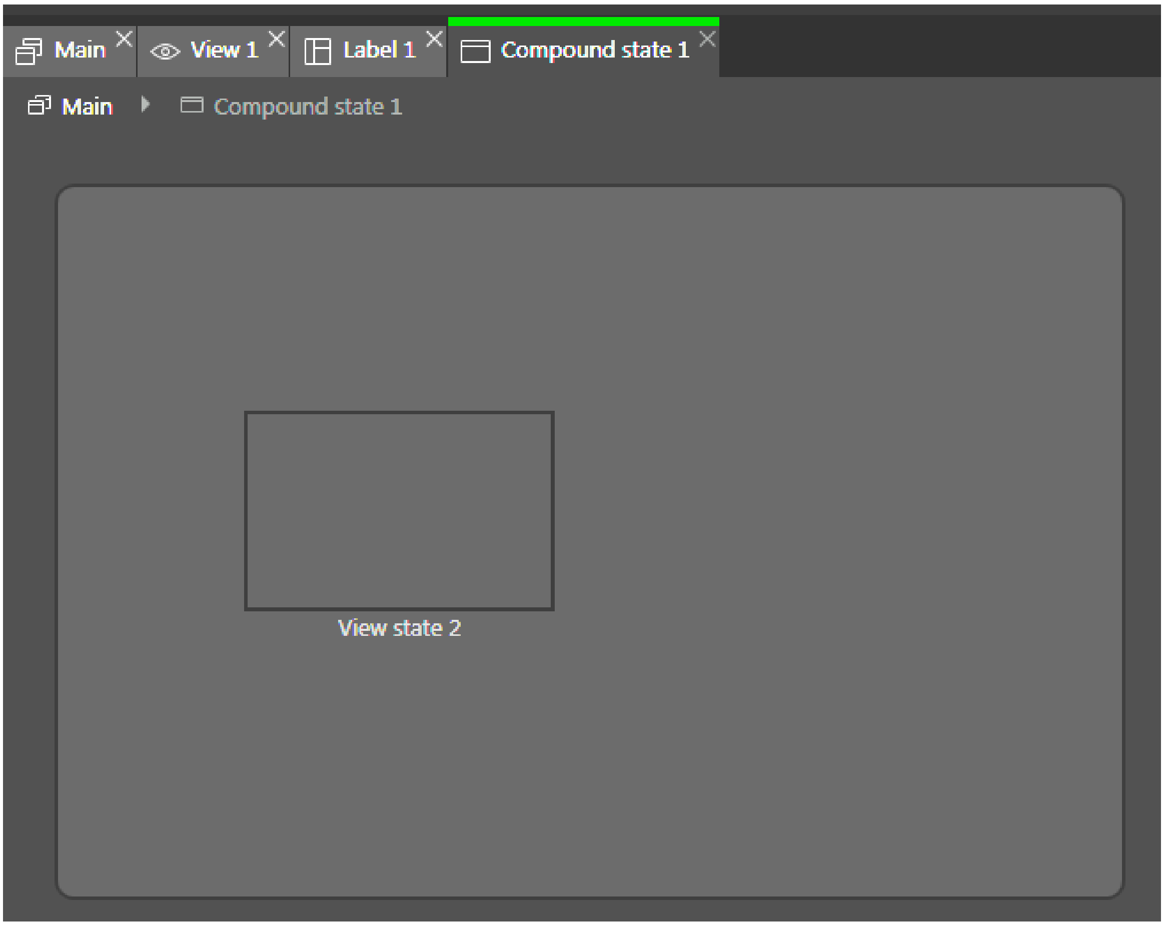


Figure 7.1. A compound state with a nested view state

7.2.3. Adding a choice state

Adding a choice state

Prerequisite:

- The content area displays a state machine.
- The state machine contains at least two states.

Step 1
Drag a choice state from the **Toolbox** into the state machine.

Step 2
Add a condition to the outgoing transition. For details see section 7.3.4, “Adding a condition to a transition”

The condition is assigned priority one. When the state machine enters the choice state, the condition with priority one is evaluated first.

Step 3
To add more choice transitions, repeat the two previous steps.

A new choice transition is assigned a lower priority than the transition that was created before.

Step 4
Add an outgoing transition from the choice state.

Step 5
In the **Navigation** component, right-click the transition. In the context menu, click **Convert to else**.

You added an else transition. The else transition is executed when all conditions which are assigned to outgoing choice transitions evaluate to `false`.

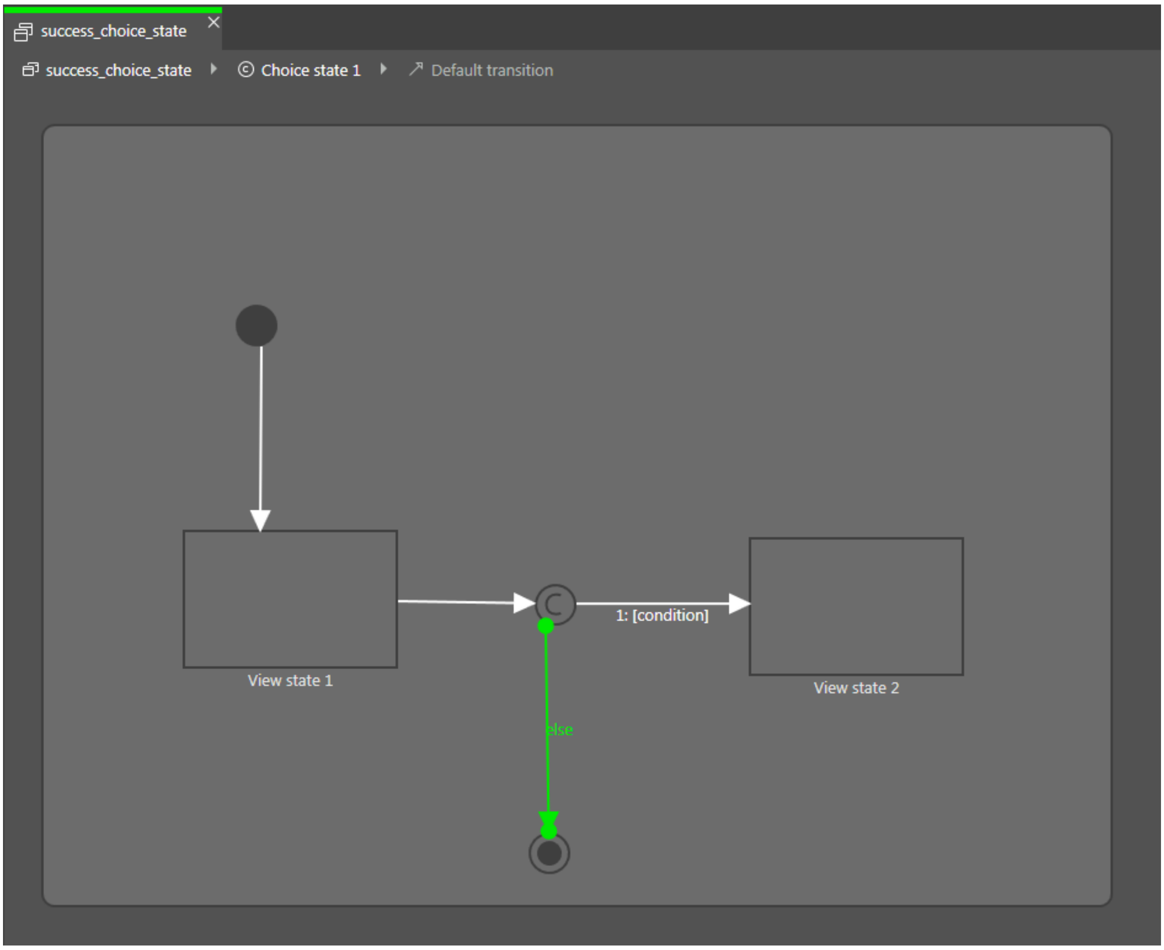


Figure 7.2. A choice state with its choice transitions

7.2.4. Defining an entry action for a state

Defining an entry action for a state

For view states and compound states you can define an entry action. The entry action is executed every time the state is entered.

Prerequisite:

- A state machine contains a view state or a compound state.

Step 1
Select a state.

Step 2
In the **Properties** component, go to the `Entry action` property, and click + .

A script editor opens.

Step 3
Enter an action using EB GUIDE Script.

For background information, see section 6.16, “Scripting language EB GUIDE Script”.

Step 4
Click **Accept**.

7.2.5. Defining an exit action for a state

Defining an exit action for a state

For view states and compound states you can define an exit action. The exit action is executed every time the state is exited.

Prerequisite:

- A state machine contains a view state or a compound state.

Step 1
Select a state.

Step 2
In the **Properties** component, go to the `Exit action` property, and click + .

A script editor opens.

Step 3
Enter an action using EB GUIDE Script.

For background information, see section 6.16, “Scripting language EB GUIDE Script”.

Step 4
Click **Accept**.

7.2.6. Deleting a model element from a state machine

Deleting a model element from a state machine

Prerequisite:

- A state machine contains at least one model element.

Step 1
In the **Navigation** component, right-click a model element.

Step 2
In the context menu, click **Delete**.

The model element is deleted.

7.3. Connecting states through transitions

7.3.1. Adding a transition between two states

Adding a transition between two states

With a transition, you connect a source state to a target state.

Prerequisite:

- The content area displays a state machine.
- The state machine contains at least two states.

Step 1
Select a state as a source state for the transition.

Step 2
Click the green drag point, and keep the mouse button pressed.

Step 3
Drag the mouse into the target state.

Step 4
When the target state is highlighted green, release the mouse button.

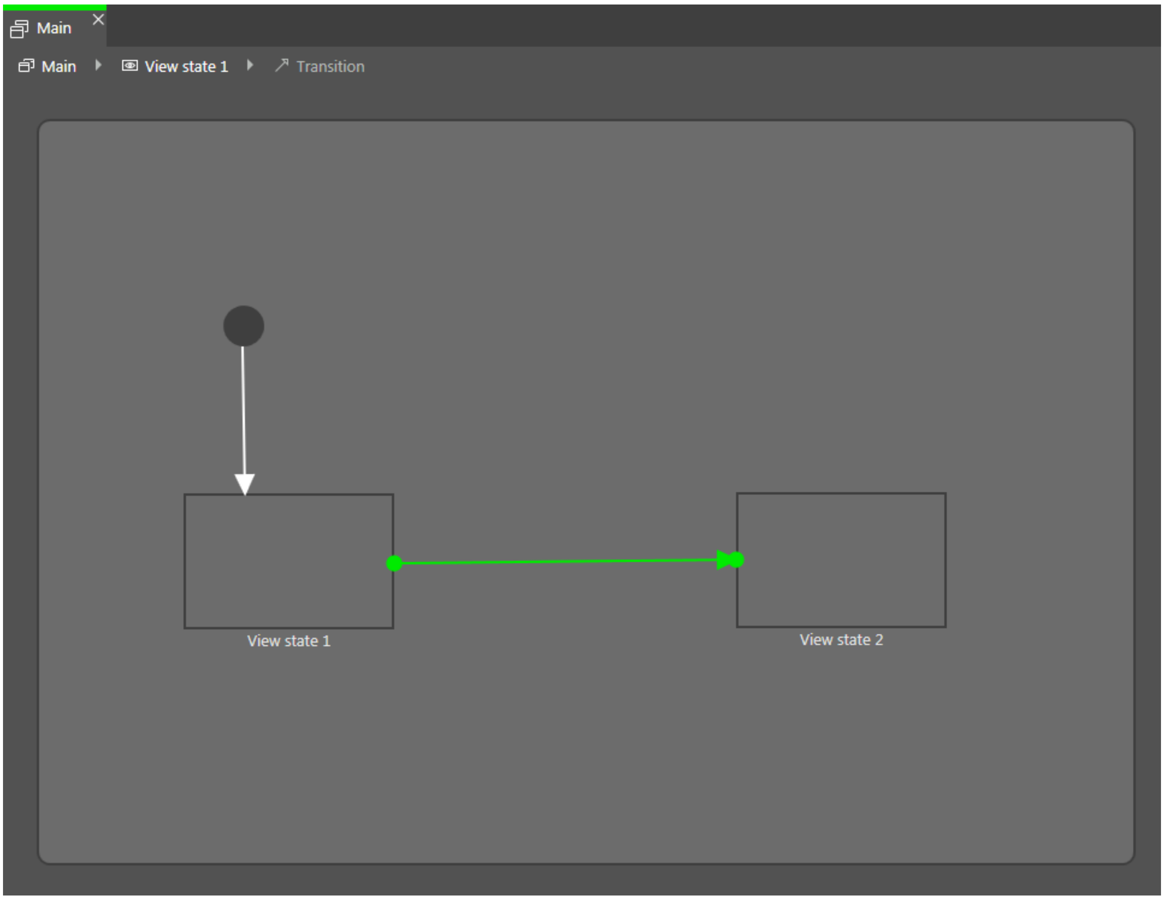


Figure 7.3. A transition

A transition is added and displayed as a green arrow.

TIP

Connect transitions to the state machine

The state machine is the top-most compound state. Therefore, you can create transitions to and from the border of the state machine. All states in the state machine inherit such a transition.

7.3.2. Moving a transition

Moving a transition

You move a transition by moving one of its end points.

Prerequisite:

- The content area displays a state machine.
- The state machine contains at least two states.
- The states are connected by a transition.

Step 1
In the content area, click a transition.

Two green drag points are displayed.

Step 2
Click the drag point you would like to move, and keep the mouse button pressed.

Step 3
Drag the mouse into a different state.

Step 4
When the state is highlighted green, release the mouse button.

The transition is moved.

7.3.3. Defining a trigger for a transition

Defining a trigger for a transition

For a transition, you can define an event that triggers it.

Prerequisite:

- A state machine contains at least two states.
- The states are connected by a transition.

Step 1
Select a transition.

Step 2
In the **Properties** component, click in the box next to **Trigger**.

Step 3
Select an event.

To create a new event, enter a name and click **Add event**.

The event is added as a transition trigger.

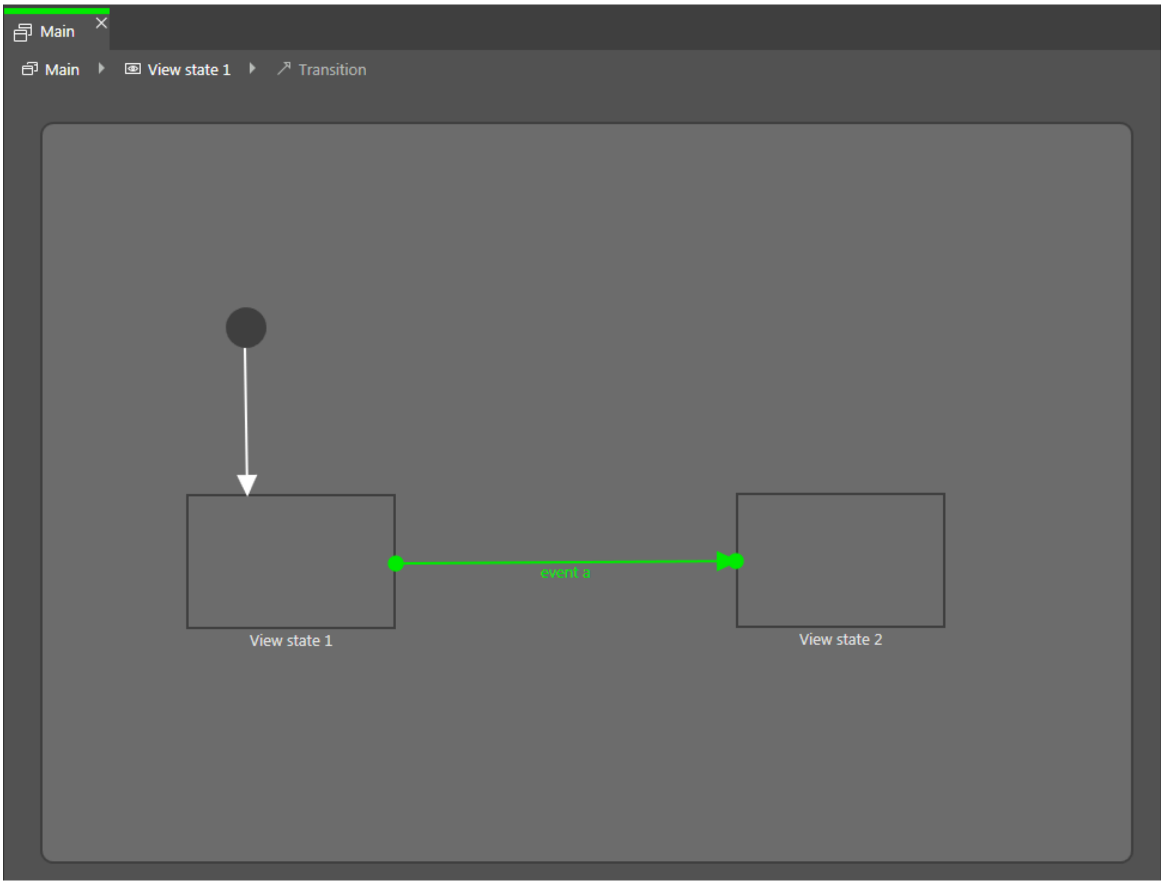


Figure 7.4. A transition with a trigger

7.3.4. Adding a condition to a transition

Adding a condition to a transition

For every transition, you can define a condition that needs to be fulfilled to execute the transition.

Prerequisite:

- A state machine contains at least two states.
- The states are connected by a transition.

Step 1
Select a transition.

Step 2
To add a condition to the transition, go to the **Properties** component. Next to the `Condition` property, click + .

A script editor opens.

Step 3
Enter a condition using EB GUIDE Script.

For background information, see section 6.16, “Scripting language EB GUIDE Script”.

Step 4
Click **Accept**.

The condition is added to the transition.

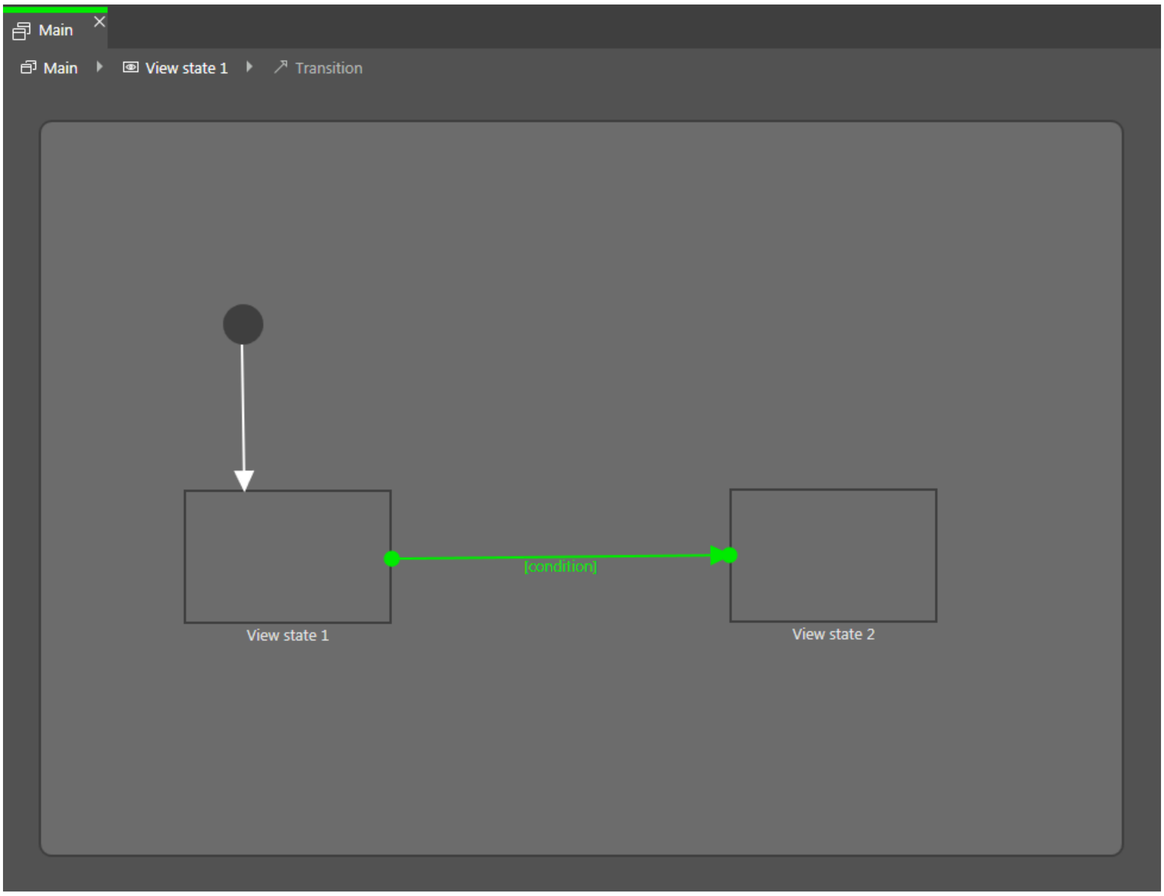


Figure 7.5. A transition with a condition

7.3.5. Adding an action to a transition

Adding an action to a transition

For every transition, you can define an action that is executed along with the transition.

Prerequisite:

- A state machine contains at least two states.
- The states are connected by a transition.

Step 1
Select a transition.

Step 2
To add an action to the transition, go to the **Properties** component. Next to the `Action` property, click + .

A script editor opens.

Step 3
Enter an action using EB GUIDE Script.

For background information, see section 6.16, “Scripting language EB GUIDE Script”.

Step 4
Click **Accept**.

The action is added to the transition.

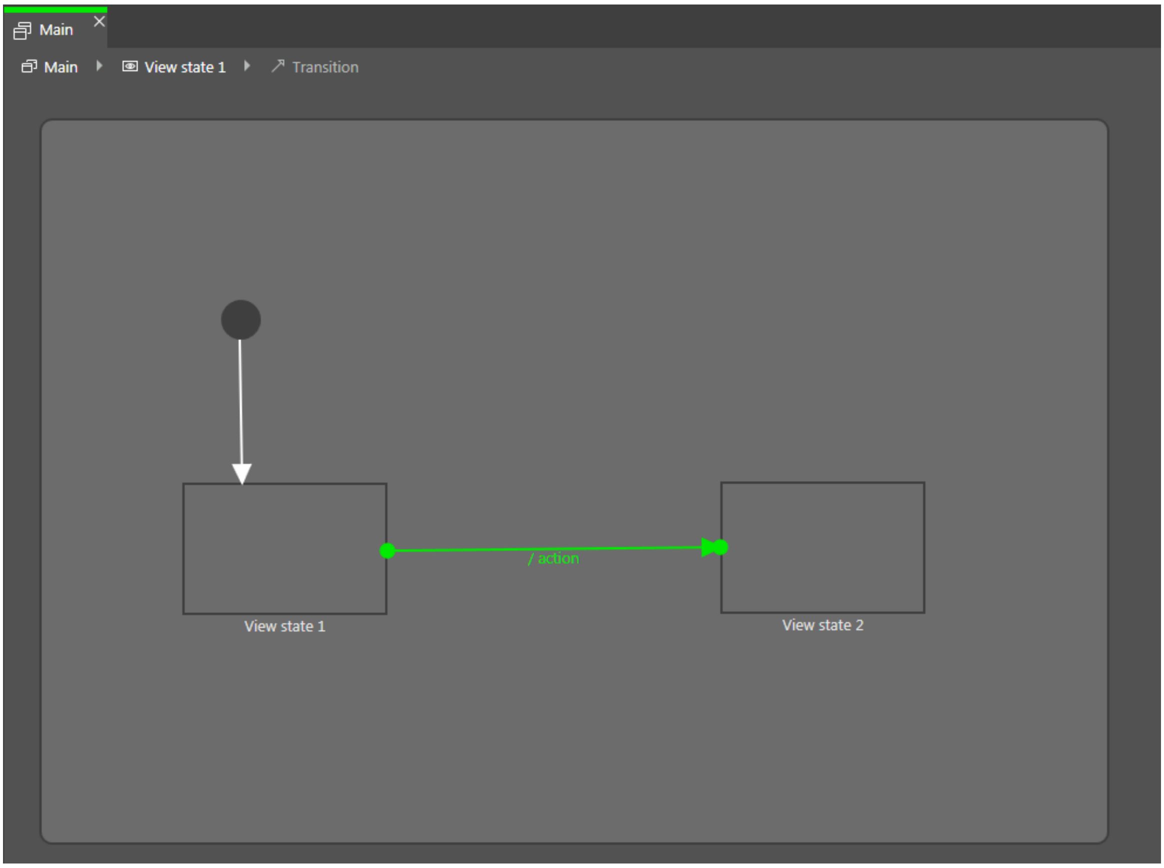


Figure 7.6. A transition with an action

7.3.6. Adding an internal transition to a state

Adding an internal transition to a state

Prerequisite:

- A state machine contains a state.

Step 1
Select a state.

Step 2
In the **Properties** component, go to **Internal transitions**, and click + .

A script editor opens.

An internal transition is added to the state. The internal transition is visible in the **Navigation** component.

8. Modeling HMI appearance

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

8.1. Working with widgets

TIP

Copying and finding views and widgets

You can copy and paste an existing view or widget using the context menu or **Ctrl**+**C** and **Ctrl**+**V**.

To find a specific view or widget within your EB GUIDE model, enter the name of the view or widget in the search box or use **Ctrl**+**F**. To jump to a view or widget, double-click it in the hit list.

8.1.1. Adding a view

Adding a view

Prerequisite:

- The content area displays a state machine.

Step 1
Drag a view state from the **Toolbox** into the state machine.

Along with the view state, a view is added to the EB GUIDE model.

Step 2
In the **Navigation** component, click the view.

Step 3
Press the **F2** key, and rename the view.

Step 4
Double-click the view state in the content area.

The content area displays the new view.

8.1.2. Adding a basic widget to a view

For details on basic widgets, see section 13.12.2, “Basic widgets”.

8.1.2.1. Adding a rectangle

Adding a rectangle

Prerequisite:

- The content area displays a view.

Step 1
Drag a rectangle from the **Toolbox** into the view.

The rectangle is added to the view.

8.1.2.2. Adding an ellipse

Adding an ellipse

Prerequisite:

- The content area displays a view.

Step 1
Drag an ellipse from the **Toolbox** into the view.

The widget is added to the view.

8.1.2.2.1. Editing an ellipse

You can draw just a sector of an ellipse and you can change the arc of an ellipse.

Creating a circular sector

Prerequisite:

- The view contains an ellipse.

Step 1
Click the ellipse and go to the **Properties** component.

Step 2
Enter the angle of the sector in the `centralAngle` text box.

Step 3
Enter the orientation of the sector in the `sectorRotation` text box.

You created a circular sector.

Creating a circular arc

Prerequisite:

- The view contains an ellipse.

Step 1
Click the ellipse and go to the **Properties** component.

Step 2
Enter a width between 0 and 50 in the `arcWidth` text box.

You created a circular arc.

8.1.2.3. Adding an image

Adding an image using **Toolbox**

Prerequisite:

- An image file is located in the `$GUIDE_PROJECT_PATH/<project name>/resources` directory. For supported file types, see section 6.15.3, “Images”.
- The content area displays a view.

Step 1
Drag an image from the **Toolbox** into the view.

Step 2
In the **Properties** component, select an image from the `image` combo box. Alternatively, drag another image from the **Assets** component into the `image` drop-down list box.

The view displays the image.

Adding an image using **Assets** component

Prerequisite:

- An image file is located in the `$GUIDE_PROJECT_PATH/<project name>/resources` directory. For supported file types, see section 6.15.3, “Images”.
- The content area displays a view.

Step 1
Drag an image file from the **Assets** component into the view.

The view displays the image.

Step 2
To change the image file, go to the **Properties** component and select an image from the `image` combo box. Alternatively, drag another image from the **Assets** component into the `image` combo box.

The view displays the image.

Adding 9-patch images

Prerequisite:

- A 9-patch image file is located in the `$GUIDE_PROJECT_PATH/<project name>/resources` directory. For background information on 9-patch images, see section 6.15.3.1, “9-patch images”.
- The content area displays a view.
- An image is added to the EB GUIDE model.

Step 1
Select the image, and go to the **Properties** component.

Step 2
From the `image` combo box, select a 9-patch image.

Step 3
Go to the **Widget features properties** and click **Add/Remove**.

The **Widget feature** dialog is displayed.

Step 4
Under **Available widget features**, expand the **Layout** category, and select **Scale mode**.

Step 5
Click **Accept**.

The related widget properties are added to the image and displayed in the **Properties** component.

Step 6
In the **Properties** component, for the `scaleMode` property select `fit to Size (1)`.

NOTE

Adding 9-patch images

If you do not add the **Scale mode** widget feature or if for the `scaleMode` property you select `original Size (0)` or `keep aspect ratio (2)`, the 9-patch image is scaled like a normal `.png` image.

8.1.2.4. Adding a label

Adding a label using **Toolbox**

Prerequisite:

- The content area displays a view.

Step 1
Drag a label from the **Toolbox** into the view.

The label is added to the view. The label has the default font `PT_Sans_Narrow.ttf`.

Adding a label using **Assets** component

Prerequisite:

- A font file is located in the `$GUIDE_PROJECT_PATH/<project name>/resources` directory. For supported file types, see section 6.15.1, “Fonts”.
- The content area displays a view.

Step 1
Drag a font file from the **Assets** component into the view.

The view displays the label with the selected font.

8.1.2.4.1. Changing the font of a label

Changing the font of a label

Prerequisite:

- A font file is located in the `$GUIDE_PROJECT_PATH/<project name>/resources` directory. For supported file types, see section 6.15.1, “Fonts”.
- The EB GUIDE model contains a view state.
- The view contains a label.

Step 1
Select the label in the view.

Step 2
In the **Properties** component, select a font from the `font` combo box.

Alternatively, drag a font file from the **Assets** component into the `font` combo box.

The view displays the label with the new font. Note that if you select an `.fnt` bitmap font, the size of the font is fixed and you cannot change it in the `font` property of the label.

NOTE

Calculation of text height and line gap

The following figure shows, how text height, line height, and line gap are calculated in EB GUIDE Studio. Take this into account when changing font style, size or line gap of a label.

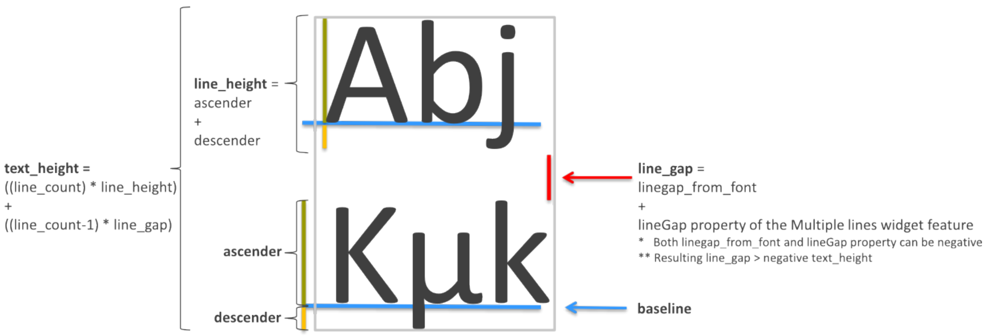


Figure 8.1. Calculation of text height, line height, and line gap

For instructions on how to use multifont support, see section 8.2.8, “Managing multifont support”.

8.1.2.5. Adding a container

Adding a container

A container allows grouping widgets.

Prerequisite:

- The content area displays a view.

Step 1
Drag a container from the **Toolbox** into the view.

Step 2
In the content area, enlarge the container by dragging one of its corners.

Step 3
Drag two or more widgets from the **Toolbox** into the container.

The widgets are modeled as child widgets of the container. Moving the container moves its child widgets along with it.

8.1.2.6. Adding an instantiator

Adding an instantiator

Prerequisite:

- The content area displays a view.

Step 1
Drag an instantiator from the **Toolbox** into the view.

Step 2
Drag a widget from the **Toolbox** into the instantiator.

The widget serves as a line template.

Step 3
Select the instantiator, and go to the **Properties** component.

Step 3.1
For the `numItems` property enter a value that is greater than one.

Step 3.2
Add one of the following widget features to the instantiator:

- **Box layout**

- **Flow layout**
- **Grid layout**
- **List layout**

For details, see section 8.3.1, “Adding a widget feature”.

In the view, the child widget is displayed as many times as specified by the `numItems` property and in the layout specified by widget features for the instantiator.

Step 4
Drag a widget from the **Toolbox** into the instantiator.

You added the second child widget that serves as the second line template.

Step 5
Select the instantiator, and go to the **Properties** component.

Step 5.1
Select the `lineMapping` and click ✎.

Step 5.2
Click the **Add** button.

A new entry is added to the table.

Step 5.3
In the `Value` text box enter `0`.

Step 5.4
Click the **Add** button.

The new entry is added to the table.

Step 5.5
In the `Value` text box enter `1`.

You defined the order in which the line templates are instantiated.

Example 8.1.
Instantiation order

The `lineMapping` property defines the order of instantiation. For example, if you enter the values `1|0`, the instantiator instantiates the line template `1` as the first child widget and the line template `0` as the second child widget.

The `lineMapping` property is applied iteratively. This means that if for the `numItems` property you enter `10`, the result is the order `1|0|1|0|1|0|1|0|1|0`.

For a detailed example of how to use instantiators, see section 12.4, “Tutorial: Creating a list with dynamic content”.

NOTE

Linking of properties of the line templates

The following are the rules for linking:

- You cannot link properties between line templates.
- You cannot link from the outside of the instantiator to its line templates.
- You can link from a line template to the corresponding instantiator.

8.1.2.7. Adding an animation

Adding an animation

For details on curves and for a description of curve properties, see section 13.12.2.2, “Animation”.

Prerequisite:

- The content area displays a view.

Step 1
Drag one of the basic widgets from the **Toolbox** into the view.

Step 2
Drag an animation from the **Toolbox** into the widget you added.

Step 3
Go to the **Animation** editor and next to **Animated properties** click +.

A menu expands.

Step 4
Under **Animation properties** select the property that you want to animate and under **Animation curves** select a respective curve.

Step 5
Click **Accept**.

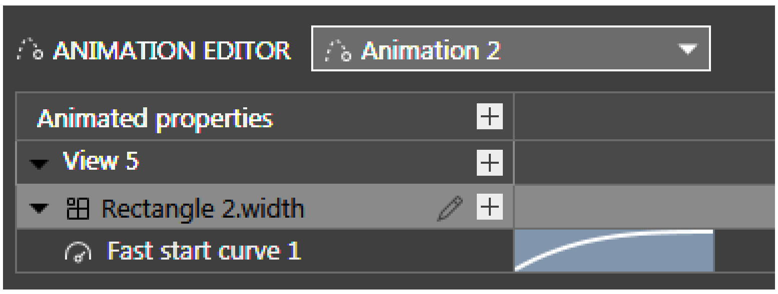


Figure 8.2. **Animation** editor with an example curve

Step 6
Select the basic widget, and add a user-defined property of type `Conditional script`. For details, see section 8.2.5, "Adding a user-defined property to a widget".

Step 7
Next to the conditional script, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 8
Enter the following EB GUIDE Script in the **On trigger** section:

```
function(v:arg0::bool)
{
f:animation_play(v:this->"Animation 1")
}
```

Animation 1 is the default name of the animation that is added first. If the animation you added in step two has a different name, replace the name in the **On trigger** script.

Step 9
Start the simulation.

The linked property of your widget gradually changes as specified by the curve you added.

As a follow-up step, you can change the properties of the animation or the curve.

Step 10
To change the curve behavior, edit the animation or curve properties in the **Properties** component.

The **Animation** editor shows a preview of the curve behavior.

For a concrete animation example, see section 12.5, "Tutorial: Making an ellipse move across the screen".

8.1.2.8. Adding an animation with a script curve

Getting the output of the script curve

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state.

Step 1
Rename the view to `FirstView` and open it.

Step 2
Drag an animation from the **Toolbox** into `FirstView`.

Step 3
In the **Animation editor**, next to **Animated properties** click + and then click `FirstView`.

A menu expands.

Step 4
Under `FirstView` select the `y` property and then **Script curve**.

Step 5
Click **Accept**.

An animation is added to the **Animation editor**.

Step 6
Rename the new animation to `ScriptCurveMonitoring`.

Step 7
In the **Properties** component, go to **Default widget properties** and next to the `curve` property click {} . EB GUIDE Script editor is displayed.

Enter the following EB GUIDE Script:

```
function(v:diff::int, v:t_anim::int)
{
 f:trace_string("Diff : "+ f:int2string(v:diff) + " t_anim: " + f:int2string(v:t_anim))
       0::int
}
```

Click **Accept**.

Step 8
In the **Properties** component, go to the **User-defined properties** category, and click +.

A menu expands.

Step 9
In the menu, select `Conditional script`.

A user-defined property of type `Conditional script` is added to the animation. Rename it to `PlayAnimation`.

Step 10
Next to the `PlayAnimation` property, click {} .

The EB GUIDE Script editor is displayed.

Step 11
Enter the following EB GUIDE Script in the **On trigger** section:

```
function(v:arg0::bool)
{
 f:animation_play(v:this)
 false
}
```

Saving and testing the EB GUIDE model

Prerequisite:

- You completed the previous instruction.

Step 1
To save the project, click in the command area.

Step 2
To start the simulation, click in the command area.

In EB GUIDE Monitor, observe the **Logger** component. `v:diff` shows the animation is executed every 16 milliseconds. `v:t_anim` shows how long the animation ran at this point since the start. See the figure below.

traceString 'Diff : 0 t_anim: 0'	12:21:07.779
traceString 'Diff : 16 t_anim: 16'	12:21:07.779
traceString 'Diff : 16 t_anim: 32'	12:21:07.779
traceString 'Diff : 17 t_anim: 49'	12:21:07.779
traceString 'Diff : 19 t_anim: 68'	12:21:07.779
traceString 'Diff : 16 t_anim: 84'	12:21:07.779
traceString 'Diff : 16 t_anim: 100'	12:21:07.779

Figure 8.3. EB GUIDE Monitor messages

8.1.2.9. Adding an alpha mask

Adding an alpha mask

For details on alpha mask, see section 13.12.2.1, “Alpha mask”.

Prerequisite:

- The `$GUIDE_PROJECT_PATH/<project name>/resources` directory contains an image.
- The content area displays a view.

Step 1
Drag the alpha mask from the **Toolbox** into the view.

Step 2
Go to the **Properties** component and select an image from the `image` drop-down list box.

NOTE

Supported image file types for alpha mask
The available image formats depend on the implementation of the renderer. The renderers for OpenGL ES 2.0 or higher support `.png` files and `.jpg` files. RGB images are converted to grayscale images before being used as alpha masks. Grayscale images are used as is. The alpha channel in the image is ignored.

Alpha mask functionality is not applied to 9-patch images. 9-patch images are handled the same way the PNG and JPEG file formats are.

Step 3
Add one of the basic widgets from the **Toolbox** as a child widget to the alpha mask.

The alpha channel, i.e. the opacity of the child widget is controlled with the alpha mask.

8.1.3. Adding a 3D widget to a view

8.1.3.1. Adding a scene graph to a view

Adding a scene graph to a view

For restrictions and recommendations, see section 6.1.2, “Settings for 3D graphic files”.

Prerequisite:

- A 3D graphic file is available. The file contains a camera, a light source, and one object containing a mesh and at least one material. For supported 3D graphic file formats, see section 6.1.1, “Supported 3D graphic formats”.
- The content area displays a view.

Step 1
Drag a scene graph from the **Toolbox** into the view.

The view displays the empty bounding box.

Step 2
In the **Properties** component, click **Import file**.

A dialog opens.

Step 3
Navigate to the directory where the 3D graphic file is stored.

Step 4
Select the 3D graphic file.

Step 5
Click **Open**.

The import starts. A dialog opens.

Step 6
Click **OK**.

The view displays the 3D graphic. The **Navigation** component displays the imported widget tree with the scene graph as a parent node. If the imported 3D scene has animations, the linear key value interpolation integer or linear key value interpolation float curve are added. Note that you cannot modify the underlying key-value pairs of these curves in EB GUIDE Studio.

TIP

Multiple import
Import of multiple 3D graphics within one scene graph is possible.

After importing, multiple 3D graphics are rendered on top of each other. To display 3D objects separately, use the `visible` property of `RootNode`.

8.1.4. Importing a `.psd` file to a view

Adding a `.psd` file into a view

For background information, see section 6.14, “Photoshop file format support”.

Prerequisite:

- A `.psd` file is available in `$GUIDE_PROJECT_PATH/<project name>/resources` or a subfolder.
- The content area displays a view.

Step 1
In the **Assets** component, select the corresponding folder.

Step 2
From the preview area, drag the `.psd` file into the content area.

An import status message appears.

Step 3
Click **OK**.

If the import was successful, the **Navigation** component displays a new widget tree. The top element is a container named after the `.psd` file.

In the `$GUIDE_PROJECT_PATH/<project name>/resources` folder, a subfolder is created, that contains all of the extracted images.

NOTE

Multiple lines

If a text layer contains text for more than one line, you need to add widget feature **Multiple lines** and edit the properties accordingly. For more information, see section 8.3.1, “Adding a widget feature” and section 13.13.1.4, “Multiple lines”.

8.1.5. Extracting images from a .psd file

Extracting images from a `.psd` file

When you extract images from a `.psd` file instead of importing them, no widget tree is created. For background information, see section 6.14, “Photoshop file format support”.

Prerequisite:

- A `.psd` file is available in `$GUIDE_PROJECT_PATH/<project name>/resources` or in subfolders.

Step 1
In the **Assets** component, select the corresponding folder.

Step 2
Right-click the `.psd` file and select **Extract images from .psd file**.

An import status message appears.

Step 3
Click **OK**.

In the `$GUIDE_PROJECT_PATH/<project name>/resources` folder, a subfolder is created, that contains all the extracted images. The subfolder is named after the `.psd` file.

8.1.6. Importing IBL files

For background information, see section 6.11, “Image-based lighting”.

Importing IBL files

To import IBL files they must be transformed into the `.ebibl` format first. This is done using IBLGenerator

Prerequisite:

- An EB GUIDE project is created.
- Your IBL files are in either `.pfm` or `.hdr` format.
- You have admin rights.
- A scene graph widget is added to the EB GUIDE model.
- A 3D file is imported.

Step 1
Open the command line prompt as administrator.

Step 2
Navigate to the IBLGenerator installation path. It is in your EB GUIDE Studio installation directory in the `tools` directory, `$GUIDE_INSTALL_PATH\tools\IBLGenerator`.

Step 3
Type the command to transform your file into to `.ebibl` format. It could look like this:

```
IBLGenerator.exe -i yourfile.hdr -o yourfile.ebibl -p latlong -q 1
```

- `-i`: The input file name
- `-o`: The output file name
- `-p`: The parameterization type. Other types are cube and sphere.
- `-q`: The quality level. The quality level with 1 as the low quality and 10 as the highest quality. Higher levels of quality need significantly more processing time.

The `.ebibl` file is placed in the directory that you provided.

TIP

IBLGenerator help

To see a list of options for IBLGenerator, run it with parameters `-h`: `IBLGenerator.exe -h`

Step 4

Copy the `.ebibl` file into the resources directory of your EB GUIDE model. Now you can use the `.ebibl` file in a scene graph node.

Step 5

In EB GUIDE Studio, in the **Toolbox** component, from the **3D widgets** drag the image-based light into a scene graph node.

Step 6

In the **Properties** component, next to the `ibl` property, select your `.ebibl` file.

The import of the IBL file is finished.

TIP

Best results for IBL

For best results, adapt the properties of the image-based light and use PBR GGX material or PBR Phong material.

To only illuminate the scene with image-based lighting, disable all other light sources.

8.1.7. Deleting a widget from a view

Deleting a widget from a view

Prerequisite:

- The EB GUIDE model contains a widget.

Step 1

In the **Navigation** component, right-click a widget.

Step 2

In the context menu, click **Delete**.

The widget is deleted.

TIP

Deleting widgets from the content area
It is also possible to delete a widget by selecting it in the content area and pressing the **Delete** key.

8.2. Working with widget properties

8.2.1. Positioning a widget

Positioning a widget

Positioning a widget means adjusting the widget's `x` and `y` properties. The point of origin where both `x` and `y` have the value `0` is the top left corner of the parent widget.

Prerequisite:

- The content area displays a view.
- The view contains a widget.

Step 1
Select a widget.

The **Properties** component displays the properties of the selected widget.

Step 2
To define the x-coordinate of the widget enter a value in the `x` text box.

Step 3
To define the y-coordinate of the widget enter a value in the `y` text box.

Step 4
Click outside the text box.

The content area displays the widget at the entered position.

TIP

Alternative approach

To position a widget by visual judgment, select the widget in the content area and move it with the mouse.

8.2.2. Resizing a widget

Resizing a widget

Prerequisite:

- The content area displays a view.
- The view contains a widget.

Step 1
Select a widget.

The **Properties** component displays the properties of the selected widget.

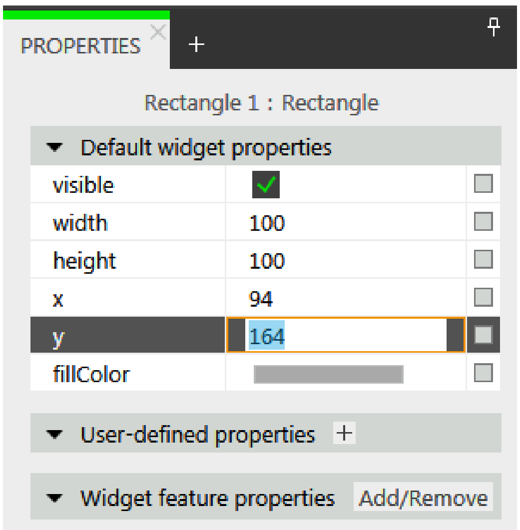


Figure 8.4. Properties of an image

Step 2
To define the height of the widget enter a value in the `height` text box.

Step 3
To define the width of the widget enter a value in the `width` text box.

Step 4
Click outside the text box.

The content area displays the widget with the entered size.

NOTE

Negative values

Do not use negative values for `height` and `width` properties. EB GUIDE Studio treats negative values as `0`, this means the respective widget will not be depicted.

TIP

Alternative approach

To resize a widget by visual judgment, select the widget in the content area and drag one of its corners with the mouse.

8.2.3. Linking between widget properties

Linking between widget properties

In order to make sure that two widget properties have the same value at all times, you can link two widget properties. As an example, the following instructions show you how to link the `width` property of a rectangle to the `width` property of a view.

You can only link the properties of widgets within the same view

You cannot link to properties of child widgets of an instantiator.

Prerequisite:

- The EB GUIDE model contains a view state.
- The view contains a rectangle.
- The `width` property of the rectangle is not a scripted value.

Step 1
Click the rectangle.

The **Properties** component displays the properties of the rectangle.

Step 2
In the **Properties** component, go to the `width` property, and click the button next to the property.

A menu expands.

Step 3
In the menu, click **Add link to widget property**.

A dialog opens.

Step 4
In the dialog, go to the view, and select its `width` property.

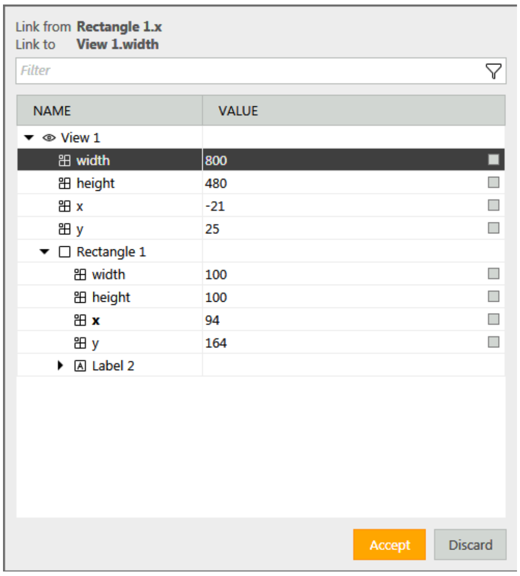


Figure 8.5. Linking between widget properties

Step 5
Click **Accept**.

The dialog closes. The button is displayed next to the `width` property. It indicates that the `width` property of the rectangle is now linked to the `width` property of the view. Whenever you change the width of the view, the width of the rectangle changes and vice versa.

NOTE

Link source and link target

The button is only displayed next the link source. It is not displayed for the link target.

TIP

Removing the link

To remove the link, click the button again. In the menu that opens click **Remove link**.

8.2.4. Linking a widget property to a datapool item

Linking a widget property to a datapool item

In order to make sure that a widget property and a datapool item have the same value at all times, you can link a widget property to a datapool item. As an example, the following instructions show you how to link the `image` property of an image to a new datapool item.

Prerequisite:

- The EB GUIDE model contains a view state.
- The view contains an image.
- The `image` property of the image is not a scripted value.

Step 1
Click the image.

The **Properties** component displays the properties of the image.

Step 2
In the **Properties** component, go to the `image` property, and click the button next to the property.

A menu expands.

Step 3
In the menu, click **Add link to datapool item**.

A dialog opens.

Step 4
To add a new datapool item, enter a name in the text box.

Step 5
Click **Add datapool item**.

Step 6
Click **Accept**.

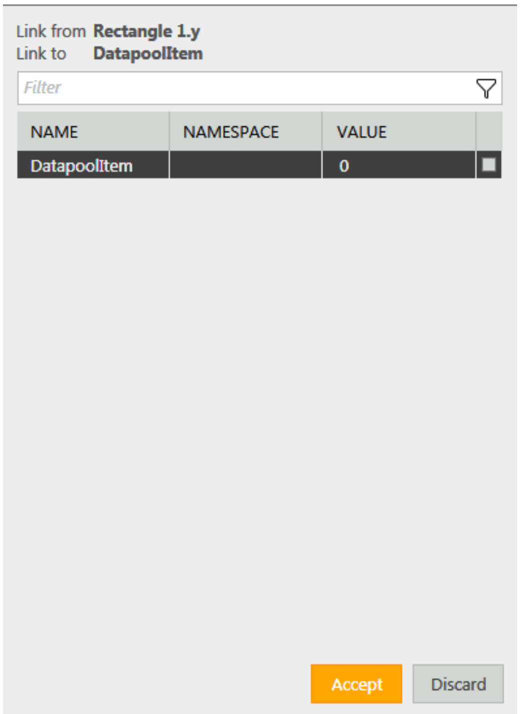


Figure 8.6. Linking to a datapool item

A new datapool item is added.

Step 7

The dialog closes. The button is displayed next to the `image` property. It indicates that the `image` property is now linked to a datapool item. Whenever you change the image, the datapool item changes and vice versa.

NOTE

Link source and link target

The button is only displayed next the link source. It is not displayed for the link target.

TIP

Removing the link

To remove the link, click the button again. In the menu that opens, click **Remove link**.

8.2.5. Adding a user-defined property to a widget

Adding a user-defined property to a widget

Prerequisite:

- The EB GUIDE model contains a view state.
- The view contains a widget.

Step 1
Select a widget.

The **Properties** component displays the properties of the selected widget.

Step 2
In the **Properties** component, go to the **User-defined properties** category, and click +.

A menu expands.

Step 3
In the menu, click a type for the user-defined property.

A new widget property of the selected type is added to the widget.

Step 4
Rename the property.

8.2.5.1. Adding a user-defined property of type `Function (): bool`

Adding a user-defined property of type `Function (): bool`

A property of type `Function (): bool` is a function that has no parameters and returns a boolean value. You call the function in EB GUIDE Script in the way you address widget properties followed by the arguments list.

Prerequisite:

- The EB GUIDE model contains a view state.
- The view contains a widget.

Step 1
Select a widget.

The **Properties** component displays the properties of the selected widget.

Step 2
In the **Properties** component, go to the **User-defined properties** category, and click +.

A menu expands.

Step 3
In the menu, click `Function (): bool`.

A new widget property of type `Function (): bool` is added to the widget.

Step 4
Rename the property.

Step 5
Next to the datapool item, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 6
Define the behavior of the new function using EB GUIDE Script.

Step 7
Click **Accept**.

Example 8.2.
Calling a property of type `Function (): bool`

In your EB GUIDE model, there is a rectangle called `Background color`. You added a property of type `Function (): bool` to it. The property is called `change`.

In any EB GUIDE Script code in the EB GUIDE model, you can call the script in the property as follows:

```
"Background color".change()
```

8.2.6. Renaming a user-defined property

Renaming a user-defined property

Prerequisite:

- The EB GUIDE model contains a widget with a user-defined property.

Step 1
In the **Navigation** component, select the widget with the user-defined property.

Step 2
In the **Properties** component, right-click the property name.

A menu expands.

Step 3
In the menu click **Rename**.

Step 4
Enter a name for the property.

Step 5
Press the **Enter** key.

8.2.7. Editing a property of type list

Editing a property of type list

For more information on properties of type list, see section 13.3.12, “List”.

For more information on resource management, see section 6.15, “Resource management” and section 6.5, “Components of the graphical user interface”.

Prerequisite:

- The EB GUIDE model contains a view state.
- The view state contains a widget.
- The widget has a property of type list.

Step 1
In the **Properties** component, select the property of type list, and next to it, click the button.

The **Edit** dialog opens.

Step 2
To add a new entry to the list, click the **Add** button.

A new row is added.

Step 3
To edit a value, in the table, click the corresponding row in the **Value** column.

NOTE

Assets component

If your property is of type font list, ibl list, image list, or mesh list, the **Edit** dialog has an additional **Assets** component. You can drag and drop the asset from the **Assets** component to the corresponding row in the **Value** column.

Step 4
When you are finished editing the list, click the **Accept** button.

The dialog closes.

8.2.8. Managing multifont support

For more information, see section 6.15.1.2, “Multifont support”.

Adding multifont support for type font

Prerequisite:

- More than one font is available in `$GUIDE_PROJECT_PATH/resources`.
- A property or a datapool item of type font exists.

Step 1
To add multifont support, do the following:

- If you want to add multifont support to a widget property, go to the **Properties** component.
- If you want to add multifont support to a datapool item, go to the **Datapool** component.

Step 2
Next to the property or datapool item, click the button.

A menu expands.

Step 3
In the menu, click **Add multifont support**.

A table is displayed below the property or the datapool item. The table has one default multifont value.

Step 4
To add a new multifont value, click the + button.

A new row is added and is filled with values based on the default font.

Step 5
In the row you can edit and define the following:

- In the column **Prio**, define the priority of the font entry.
- In the column **Font**, define the size and the font.
- In the column **Range**, define the Unicode characters that are affected by the selected font.

Step 6
Repeat the steps 2 to 5 until all required font values are added.

TIP

Removing the multifont support

To remove the multifont support, click the button again. In the menu that opens, click **Remove multifont support**.

Adding multifont support for type font list

For instructions on how to work with properties of type list, see section 8.2.7, “Editing a property of type list”.

Prerequisite:

- More than one font is available in `$GUIDE_PROJECT_PATH/resources`.
- A property or a datapool item of type font list exists and has at least one font list entry.

Step 1
To add multifont support, do the following:

- If you want to add multifont support to a widget property, go to the **Properties** component.
- If you want to add multifont support to a datapool item, go to the **Datapool** component.

Step 2
Select the property or datapool item, and next to it, click the button.

The **Edit** dialog opens.

Step 3
Next to the entry of type font, click the button.

A menu expands.

Step 4
In the menu, click **Add multifont support**.

A table is added below the property or the datapool item. The table has one default multifont value.

Step 5
To add a new multifont value, click the + button.

A new row is added and filled with values based on the default font.

Step 6
In the row you can edit and define the following:

- In the column **Prio**, define the priority of the font entry.
- In the column **Font**, define the size and the font.
- In the column **Range**, define the Unicode characters that are affected by the selected font.

Step 7
Repeat the steps 3 to 6 until all required font values are added.

TIP

Removing the multifont support

To remove the multifont support, click the button again. In the menu that opens, click **Remove multifont support**.

Step 8
When you are finished editing entries in the list, click the **Accept** button.

The dialog closes.

8.2.9. Managing order and visibility of widgets

In EB GUIDE Studio you have several possibilities, to define the order, layers, or visibility of widgets. The following possibilities are available and whether you can apply them depends on your use case.

When you just want to change which widget is on top in relation to other widgets in a view, use the **Navigation** tree. In the **Navigation** tree, the order of the widgets determines which widget is going to be displayed on top. If the widgets are on the same branch, the widget with the higher position in the tree is displayed below. If widgets are on different branches, the widget on the higher level is displayed on top.

When you want to have a pop-up that is displayed on top of everything else, use dynamic state machines. For instructions, see section 12.1, “Tutorial: Adding a dynamic state machine”.

When you have complex menus that change according to what users choose, use `Child visibility selection`. With this feature you can control the visibility of the child widgets of a widget. It overwrites the `Visibility` property of the affected widget. This features has two usage possibilities:

- Making a single child widget visible. For this use case you only need the index of the widget in the widget tree. For instructions, see “Making a single child widget visible”.
- Making multiple child widgets visible. For this use case you need to define groups of child widgets and provide one group's identifying `Value`. For instructions, see “Making multiple child widgets visible”.

Making a single child widget visible

Prerequisite:

- An EB GUIDE project is opened in EB GUIDE Studio.
- The EB GUIDE model contains widgets.

Step 1
Add the **Child visibility selection** widget feature to the parent widget. See section 8.3, “Extending a widget by widget features”.

Step 2
In the **Navigation** component, check in which position the child widget is. `0` is the first position.

Step 3
In the **Properties** component, select the row `containerIndex` and enter the position of child widget in the input field.

Step 4
Enter the position of the child widget in the `containerIndex` input field.

This child widget is now the only child widget that is visible.

Making multiple child widgets visible

Prerequisite:

- An EB GUIDE project is opened in EB GUIDE Studio.
- The EB GUIDE model contains widgets.

Step 1
Add the **Child visibility selection** widget feature. See section 8.3, “Extending a widget by widget features”.

Step 2
In the **Properties** component, select `containerMapping` and then click .

An editor opens.

Step 3
As a next step, you define a mapping between the group of widgets and child widgets.

The **Index** column contains the child widget index. In the **Value** column enter the group to which you want to map the child widgets.

Step 4
Click **Accept**.

Step 5
Select `containerIndex` and in the input field enter the group's `Value` that should be visible.

This group of widgets is now visible. Widgets that are not mapped to this group are invisible.

8.3. Extending a widget by widget features

Widget features add more functionality for the appearance and behavior of widgets. Adding a widget feature to a widget means adding one or more widget properties. The offered widget features depend on the type of the widget.

8.3.1. Adding a widget feature

Adding a widget feature

Prerequisite:

- The EB GUIDE model contains a widget.

Step 1
In the **Navigation** component, click a widget.

The **Properties** component displays the properties of the selected widget.

Step 2
In the **Properties** component, go to the **Widget feature properties** category, and click **Add/Remove**.

The **Widget features** dialog is displayed.

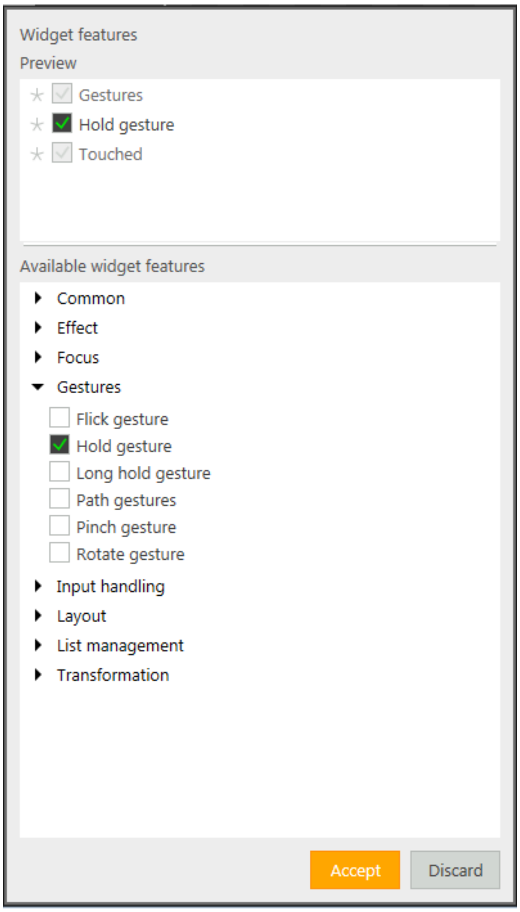


Figure 8.7. Widget features dialog

Step 3

Under **Available widget features**, expand a category, and select the widget feature you want to add.

The selected widget feature as well as dependent widget features that are activated automatically along with it, are listed under **Preview**.

Click **Accept**.

TIP

Dependencies between widget features

Some widget features require other widget features. Therefore, in some cases, if you select a widget feature, other widget features are selected automatically.

For example, you want to add the widget feature **Moveable**. In addition the widget features **Touched** and **Touch Move** are added automatically.

For a list of widget features grouped by categories, see section 13.13, "Widget features".

For tutorials, see the following:

- section 12.3, “Tutorial: Modeling a path gesture”
- section 12.4, “Tutorial: Creating a list with dynamic content”
- section 12.2, “Tutorial: Modeling button behavior with EB GUIDE Script”

8.3.2. Removing a widget feature

Removing a widget feature

Prerequisite:

- The EB GUIDE model contains a widget.
- At least one widget feature is added to the widget.

Step 1
In the **Navigation** component, click a widget.

The **Properties** component displays the properties of the selected widget.

Step 2
In the **Properties** component, go to the **Widget feature properties** category and click **Add/Remove**.

The **Widget features** dialog is displayed.

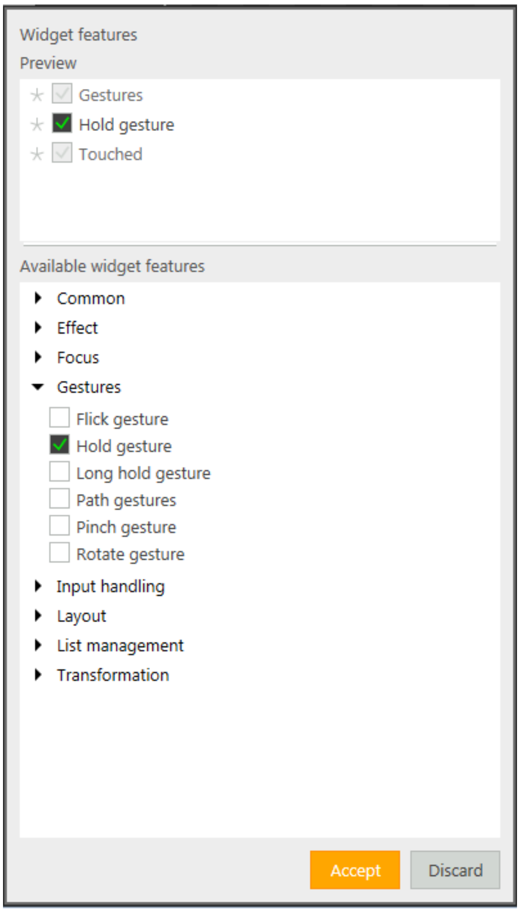


Figure 8.8. Widget features dialog

Step 3
Under **Preview** clear the widget feature you want to remove.

Click **Accept**.

The related widget feature properties are removed from the **Properties** component.

NOTE

Removing widget features with dependencies
Widget features which were added automatically due to dependencies are not deleted automatically. They cannot be removed directly. Clear the parent widget feature before you clear the child widget feature.

8.4. Working with language support

To change the language of your EB GUIDE model during run-time, you add language support and language-dependent texts.

For more information on languages in an EB GUIDE model, see section 6.12.2, “Languages in the EB GUIDE model”

8.4.1. Adding a language to the EB GUIDE model

NOTE

No skin support available

When you have defined a language support for a datapool item, it is not possible to add a skin support to the same item.

Adding a language

The following instruction shows you how to add a language to your EB GUIDE model.

Step 1

Click .

The project center opens.

Step 2

In the navigation area, click **Configure Languages**.

The available languages are displayed. The first language in the list is always the default language.

Step 3

In the content area, click **Add**.

A language is added to the table. The language uses the standard language settings as initial values.

Step 4

Enter a name for the language.

Step 5

Select a language from the **Language** drop-down list box.

Step 6

Select a country from the **Country** drop-down list box.

The language is added to the EB GUIDE model. In the project editor, in the **Language** drop-down list of the command area, you can select the new language.

You can switch between the languages during run-time, to see the effect of the different datapool values. For more information, see section 12.6, “Tutorial: Adding a language-dependent text to a datapool item”.

8.4.2. Adding language support to a datapool item

Adding language support to a datapool item

The following instruction shows you how to add a language support to a datapool item of your EB GUIDE model.

Prerequisite:

- The EB GUIDE model contains datapool items.
- At minimum two languages are added to the model.

Step 1
In the project editor go to the **Datapool** component.

Step 2
Next to the **Value** property of a datapool item, click the button.

A menu expands.

Step 3
In the menu, click **Add language support**.

The dialog closes. Next to the **Value** property, the button is displayed. It indicates that a language support is added to this datapool item and now different values for each language can be defined.

You added language support to a datapool item. Now you can define language-dependent values for this datapool item. For more information, see section 12.6, “Tutorial: Adding a language-dependent text to a datapool item”.

8.4.3. Deleting a language

NOTE

Default language

The first language in the list is always the default language and cannot be deleted.

Deleting a language

Prerequisite:

- At minimum two languages are added to the EB GUIDE model.

Step 1

Click .

The project center opens.

Step 2

In the navigation area, click **Configure** > **Languages**.

The available languages are displayed.

Step 3

In the content area, select a language.

Step 4

In the content area, click **Delete**.

The language is deleted from the table.

8.5. Working with skin support

With skin support you can define different datapool values for your model. This way you can define different looks for the same model, as for example night and day mode.

For more information on skins, see section 6.19, “Skins”.

8.5.1. Adding a skin to the EB GUIDE model

NOTE

No language support available

When you have defined a skin support for a datapool item, it is not possible to add a language support to the same item.

Adding a skin to the EB GUIDE model

Step 1

Click .

The project center opens.

Step 2

In the navigation area, click **Configure** > **Skins**.

A standard skin is added to each model by default.

Step 3

In the content area, click **Add**.

A skin is added to the table.

Step 4

Enter a name for the skin.

The new skin is added to the EB GUIDE model. In the project editor, in the **Skin** drop-down list box of the command area you can select the new skin.

8.5.2. Adding skin support to a datapool item

Adding skin support to a datapool item

To define different datapool values and thus define various looks for the your EB GUIDE model, you first need to add a skin support to the datapool item.

Prerequisite:

- The EB GUIDE model contains datapool items.
- A skin is added to the model.

Step 1
In the project editor, go to the **Datapool** component.

Step 2
Next to the **Value** property of a datapool item, click the button.

A menu expands.

Step 3
In the menu, click **Add skin support**.

The dialog closes. Next to the **Value** property, the button is displayed. It indicates that a skin support is added to this datapool item and now different values for each skin can be defined.

Step 4
To define different values for the datapool item, select the datapool in the **Datapool** component.

The **Properties** component displays a table with all skins available in the EB GUIDE model.

Step 5
Define a value for each skin in the table.

8.5.3. Switching between skins

Switching between skins

Prerequisite:

- The EB GUIDE model contains datapool items.
- A skin is added to the model.

Step 1
In the project editor go to the command area.

Step 2
Select a skin in the drop-down list box.

The content area displays the model with the datapool values valid for this skin. Also the simulation mode will display the model with the specific skin values.

8.5.4. Deleting a skin

Deleting a skin

Prerequisite:

- A skin is added to the model.

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click **Configure** > **Skins**.

All skins of the current project are listed.

Step 3
Select the skin to be deleted and click **Delete**.

The skin is deleted from the table.

8.6. Animating a view transition

8.6.1. Adding an entry animation

Adding an entry animation

The following instruction guides you through the process of adding an entry animation to a view state. The instruction also applies to exit animations, pop up on animations and pop up off animations. For more information, see section 6.2.2, “Animations for view transitions” and section 13.12.1, “View”.

Prerequisite:

- A view state and a view are added to the EB GUIDE model.
- The **VTA** component is opened.

Step 1
In the **VTA** component, click +.

Step 2
In the context menu, select **Entry animation**.

The **Animation** editor opens.

The **Properties** component displays the properties of the added entry animation.

Step 3
Animate all available widget properties in the dependent view.

Define the entry animation in the **Animation** editor.

8.6.2. Adding a change animation

Adding a change animation

The following instruction guides you through the process of adding a change animation to a view state or view template.

Prerequisite:

- **VTA** component is opened.
- The **Main** state machine contains two view states.

Step 1
Select `View state 1` in the **Main** state machine.

Step 2
In the **VTA** component, click +.

Step 3
In the context menu, select **Change animation**.

A dialog opens.

Step 4
Select `View 2`.

Step 5
Click **Accept**.

The name of the destination view is shown next to the change animation name.

The **Animation** editor opens.

The **Properties** component displays the properties of the added change animation.

The start view is shown in the project editor.

Step 6
To edit the destination view, in the **VTA** component, click ✎.

Step 7
To add animation properties, click + in the **Animation** editor to add the respective categories.

Animate all available widget properties in the dependent view.

8.6.3. Rearranging animations

Rearranging animations

The following instruction guides you through the process of rearranging animations of a view state or view template. This instruction also applies to exit animations, change animations, pop up on animations and pop up off animations.

Prerequisite:

- **VTA** component is opened.
- You added multiple entry animations.

Step 1
In the **VTA** component, select the prioritization text box next to the entry animation that you want to start first.

Step 2
Change the value of the entry animation to `0`.

The order of the entry animations is changed. The edited entry animation starts first and all following entries are incremented by one.

The value `0` means that the animation is played first. Subsequent values mean that the animations are played in a subsequent order.

8.7. Re-using a widget

For more information on templates, see section 6.22.4, “Widget templates”.

8.7.1. Adding a template

Adding a template

Step 1
In the **Navigation** component, go to **Templates**, and click +.

A menu expands.

Step 2
In the menu, click a type for the template.

A new template of the selected type is added. The content area displays the template.

Step 3
Rename the template.

Step 4
In the **Properties** component, edit the template's properties, and define the template interface.

TIP

Templates of templates
A type for the template can be an existing template. EB GUIDE thus allows creating templates from templates.

TIP

Copying and finding templates
Alternatively, you can copy and paste an existing template using the context menu or **Ctrl+C** and **Ctrl+V**.

To find a specific template within your EB GUIDE model, enter the name of the template in the search box or use **Ctrl+F**. To jump to a template, double-click it in the hit list.

8.7.2. Defining the template interface

Defining the template interface

Prerequisite:

- The EB GUIDE model contains a template.

Step 1
Select a template.

Step 2
To add a property to the template interface, in the **Properties** component, click the button next to the property. In the menu, click **Add to template interface**.

The icon is displayed next to the property.

Step 3
To remove a property from the template interface, click the button next to the property. In the menu, click **Remove from template interface**.

The icon is no longer displayed next to the property.

NOTE

Instantiator templates
For templates of instantiators, it is not possible to add properties of the instantiator's child widgets to the template interface.

8.7.3. Using a template

Using a template

Prerequisite:

- The content area displays a view.
- In the **Toolbox**, a widget template is available.
- There is at least one property in the template interface of the widget template.

Step 1
Drag a widget template from the **Toolbox** into the view.

An instance of the template is added to the view. The **Properties** component displays the properties which belong to the template interface.

TIP

Define the template interface
If the **Properties** component does not display any properties for a template instance, no properties have been added to the template interface. Define the template interface to change that.

Step 2
In the **Properties** component, edit the properties of the template instance.

After editing a property, the button changes to the button.

Step 3
To reset a property value to the value of the template, click the button next to the property. In the menu, click **Reset to template value**.

8.7.4. Deleting a template

Deleting a template

Step 1
In the **Navigation** component, right-click a template.

Step 2
In the context menu, click **Delete**.

The template is deleted.

9. Handling data

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

9.1. Adding an event

Adding an event

Step 1

In the **Events** component, from the namespace drop-down box, select the namespace, to which you want to add an event.

Step 2

Click +.

An event is added to the table.

Step 3

Rename the event.

TIP

Copying and finding events

Alternatively, you can copy and paste an existing event using the context menu or **Ctrl**+**C** and **Ctrl**+**V**. To prevent duplicates, the pasted event has a different event ID than the copied event.

To find a specific event within your EB GUIDE model, enter the name of the event in the search box or use **Ctrl**+**F**. To jump to an event, double-click it in the hit list.

9.2. Adding a parameter to an event

Adding a parameter to an event

Prerequisite:

- An event is added to the EB GUIDE model.

Step 1
In the **Events** component, click an event.

Step 2
In the events table click + next to the event.

Step 3
From the drop-down list box select a type for the parameter.

A parameter of the selected type is added to the event.

Step 4
Rename the parameter.

9.3. Addressing an event

Event IDs and event group IDs are used to address events. EB GUIDE TF uses the IDs to send and receive the events at run-time.

Adding an event group

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click **Configure** > **Event groups**.

Step 3
In the content area, click **Add**.

An event group is added to the table.

Step 4
Rename the event group.

Step 5
To change an event group ID, double-click the **ID**, and type a number.

Addressing an event for EB GUIDE TF

Prerequisite:

- An event group is added.
- An event is added to the EB GUIDE model.

Step 1
In the **Events** component, click an event.

The **Properties** component displays the properties of the selected event.

Step 2
Insert an ID in the `Event ID` text box.

Step 3
Go to the **Events** component and select an event group from the `Group` drop-down list box.

9.4. Mapping a key to an event

To fire an event, you can press a key on your keyboard, or, for example, you can press a key or use a rotary button on your target device.

In order to react on keyboard key presses, the model needs to define a mapping for these key events.

For each key, EB GUIDE GTF defines a numeric code in the C++ header file. For code numbers, see `$INSTALL_PATH$/platform/win64/include/gtf/inputmapper/KeyConstants.h`.

Mapping a key to an event

For more information on event handling, see section 6.8, “Event handling” and section 13.5, “Events”.

Prerequisite:

- The event group **Key** (ID 10) is added in **Project Center** > **Configure** > **Event groups**.
- An event is added.

Step 1
Look up the hex code number of the key you want to map in the file `KeyConstant.h`.

Step 2
Calculate the decimal code number.

Step 3
In the **Events** component, in the **Group** column, select **Key** (ID 10).

Step 4
In the **Properties** component, into the `Event ID` text box enter the calculated decimal code number.

The selected key is now mapped to an event.

Example 9.1.
Mapping the key F1 to an event

The internal hex code number of **F1** is `12`.

The decimal code number of `12` is `18`.

Go to the **Properties** component and enter `18` into the `Event ID` text box.

The key **F1** is now mapped to the event.

NOTE

Invisible event in EB GUIDE Monitor
The event itself is not shown in EB GUIDE Monitor, but an EB GUIDE Script, which was triggered by the event, reacts.

For more information on decimal code numbers, see section 13.5.1, “Decimal codes for key events”.

9.5. Deleting an event

Deleting an event

Prerequisite:

- An event is added to the EB GUIDE model.

Step 1
In the **Events** component, select the corresponding namespace.

Step 2
Right-click the event and from the context menu select **Delete**.

The event is deleted.

9.6. Adding a datapool item

Adding a datapool item

Step 1
In the **Datapool** component, from the namespace drop-down box, select the namespace, to which you want to add a datapool item.

Step 2
Click +.

A menu expands.

Step 3
In the menu, click a type for the datapool item.

A new datapool item of the selected type is added. The datapool item is prepared for internal use.

Step 4
Rename the datapool item.

TIP

Copying and finding datapool items
Alternatively, you can copy and paste an existing datapool item using the context menu or **Ctrl**+**C** and **Ctrl**+**V**.

To find a specific datapool item within your EB GUIDE model, enter the name of the datapool item in the search box or use **Ctrl**+**F**. To jump to a datapool item, double-click it in the hit list.

9.7. Editing datapool items of a list type

Editing datapool items of a list type

Prerequisite:

- A datapool item of a list type is added.

Step 1
In the **Datapool** component, click a datapool item of a list type.

Step 2
Select the `Value` column and click .

An editor opens.

Step 3
To add an item to the list datapool item, click **Add**.

A new entry is added to the table.

Step 4
Enter a value for the new entry in the `Value` text box or select a value from the combo box.

Step 5
Repeat steps three and four to add more items to the list.

Step 6
Click **Accept**.

The content of the list is displayed in the `Value` column.

9.8. Converting a property to a scripted value

Converting a property to a scripted value

Properties of datapool items and widgets can be converted to a scripted value and back to their plain value. The following instruction shows the procedure with a datapool item value. With a widget property, the procedure is the same.

Prerequisite:

- A datapool item is added.
- The datapool item is not language-dependent.
- The datapool item is not skin-dependent.
- The datapool item is not linked.

Step 1
In the **Datapool** component, click a datapool item and click the button.

A menu expands.

Step 2
In the menu, click **Convert to script**.

The datapool item is converted to a scripted value.

Step 3
Next to the datapool item, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 4
Edit the EB GUIDE Script.

Step 5
To convert the datapool item back to its plain value, click the button.

A menu expands.

Step 6
In the menu, click **Convert to plain value**.

The datapool item is converted to its plain value.

9.9. Establishing external communication

To establish external communication for example between the EB GUIDE model and an application, you add communication contexts to the EB GUIDE model.

Adding a communication context

With communication contexts you are able to channel communication.

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click **Configure** > **Communication contexts**.

Step 3
In the content area, click **Add**.

A communication context is added to the table.

Step 4
Rename the communication context, for example to `Media`.

Step 5
To run the communication context in an own thread, select **Use own thread**.

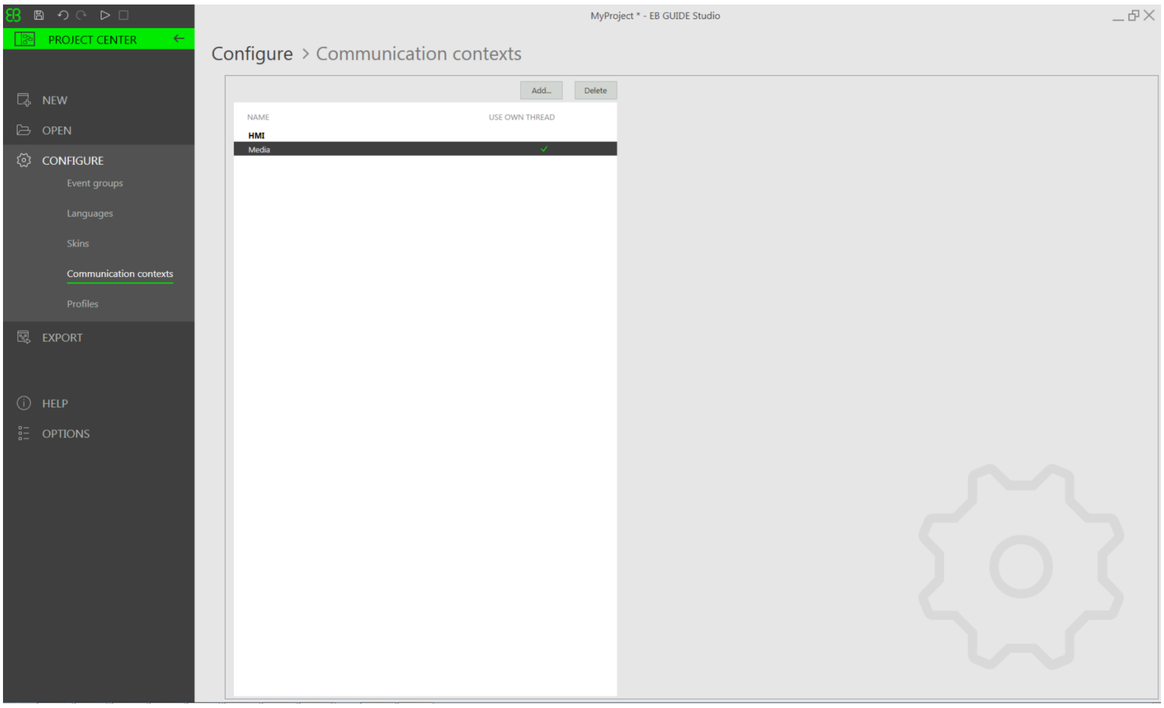


Figure 9.1. Communication context `Media`.

9.10. Linking between datapool items

Linking between datapool items

Prerequisite:

- A datapool item is added.
- The datapool item is not language-dependent.
- The datapool item is not skin-dependent.
- The datapool item is not a scripted value.

Step 1
In the **Datapool** component, click a datapool item.

Step 2
Click the button.

A menu expands.

Step 3
In the menu, click **Add link to datapool item**.

A dialog opens.

Step 4
To add a new datapool item, enter a name in the text box.

Step 5
Click **Add datapool item**.

Step 6
Click **Accept**.

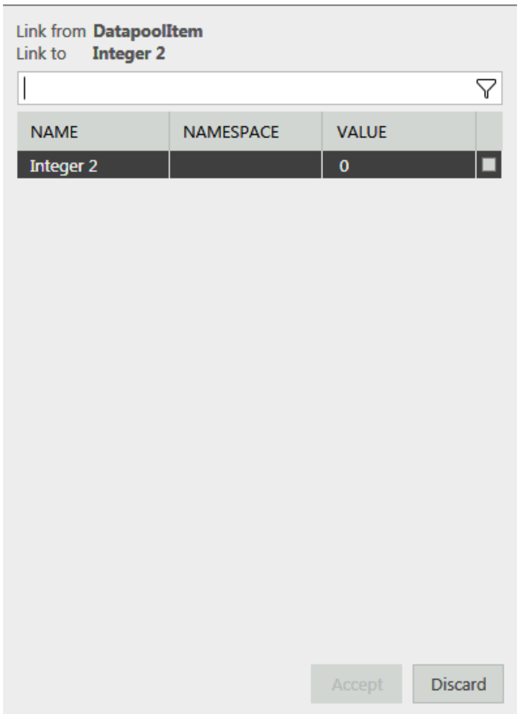


Figure 9.2. Linking between datapool items

The dialog closes. Next to the `Value` property, the button is displayed. It indicates that the `Value` property is linked to a datapool item. Whenever one of the datapool items changes its value, the value of the other datapool item changes as well.

9.11. Deleting a datapool item

Deleting a datapool item

Prerequisite:

- A datapool item is added.

Step 1
In the **Datapool** component, select the corresponding namespace.

Step 2
Right-click the datapool item and from the context menu select **Delete**.

The datapool item is deleted.

9.12. Working with namespaces

For more information on namespaces, see section 6.5, “Components of the graphical user interface” and section 6.13, “Namespaces”.

9.12.1. Adding a namespace

Adding a namespace

You can add a new namespace as a child to the root namespace, or to any existing namespace.

Step 1
As the **Namespaces** component is not available in the default window layout of EB GUIDE, do the following:

In the command area click **Layout** and select **Namespaces**.

The **Namespaces** component is displayed.

Step 2
In the **Namespaces** component, click +.

A namespace is added to the tree.

Step 3
Rename the namespace

TIP

Moving a namespace

To move a namespace, drag it to the root namespace or to another namespace. Take care of the naming conventions for namespaces to avoid naming conflicts.

9.12.2. Adding model elements to a namespace

For instructions on how to add an event to a namespace, see section 9.1, “Adding an event”.

For instructions on how to add a datapool item to a namespace, see section 9.6, “Adding a datapool item”.

9.12.3. Moving model elements between namespaces

Moving model elements between namespaces

Prerequisite:

- A model element, an event, or a datapool item is added to a namespace.
- At least two namespaces exist.

Step 1
To move an event, go to the **Events** component.

To move a datapool item, go to the **Datapool** component.

Step 2
Select the corresponding namespace and right-click the model element.

Step 3
From the context menu, select **Move to namespace...**.

A dialog opens.

Step 4
Select the target namespace and click **Accept**.

The model element is moved to the target namespace.

NOTE

Moving model elements

Alternatively, you can drag the model element to another namespace.

9.12.4. Deleting a namespace

WARNING

Deleting a namespace

When you delete a namespace, you also delete all model elements that this namespace contains.

Deleting a namespace

NOTE

Root namespace

Note that you cannot delete the root namespace.

Prerequisite:

- A namespace is added to the EB GUIDE model.

Step 1

In the **Namespace** component, right-click the namespace.

Step 2

In the context menu, click **Delete**.

The namespace is deleted.

10. Handling a project

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

10.1. Creating a project

Creating a project

Step 1

Click .

The project center opens.

Step 2

In the navigation area, click **New**.

Step 3

Enter a project name, and select a location.

Step 4

Click **Create**.

The project is created. The project editor opens and displays the new project.

10.2. Opening a project

NOTE

Invalid project

If the EB GUIDE project is not valid, EB GUIDE Studio cannot open it. An error message is shown and a log file with the error description is created in `$GUIDE_PROJECT_PATH/<project name>/<project name>_LoadingErrorLog.txt`.

For more information, see section 6.7.2, “Validation criteria for EB GUIDE project”.

10.2.1. Opening a project from the file explorer

Opening a project from the file explorer

Prerequisite:

- An EB GUIDE project is created.

Step 1

Open the file explorer, and select the EB GUIDE project file you would like to open. EB GUIDE project files have the file extension `.ebguide`.

Step 2

Double-click the EB GUIDE project file.

The project opens in EB GUIDE Studio.

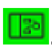
10.2.2. Opening a project within EB GUIDE Studio

Opening a project within EB GUIDE Studio

Prerequisite:

- An EB GUIDE project is created.

Step 1

Click .

The project center opens.

Step 2
In the navigation area, click the **Open** tab.

Step 3
Select a project that is listed under **Recent projects** or click **Browse**, and select the EB GUIDE project file you would like to open. EB GUIDE project files have the file extension `.ebguide`.

The project opens in EB GUIDE Studio.

10.3. Renaming model elements

Renaming model elements

The following instruction guides you through the process of renaming model elements such as states, state machines, widgets, transitions, datapool items and events.

Prerequisite:

- A model element is added to the EB GUIDE model.

Step 1
To rename a model element, perform the following:

- To rename a model element such as widget, state, or state machine, in the **Navigation** component, right-click the model element.
- To rename a datapool item, in the **Datapool** component, right-click the datapool item .
- To rename an event, in the **Events** component, right-click the datapool item.

The context menu opens.

Step 2
In the context menu, select either of the following:

- To rename only the selected model element, select **Rename**.
- To rename the selected model element, and also its occurrences in the EB GUIDE model, for example in EB GUIDE Script, select **Rename global**.

10.4. Validating and simulating an EB GUIDE model

Before exporting an EB GUIDE model to the target device, you resolve errors and simulate the model on your PC.

10.4.1. Validating an EB GUIDE model

10.4.1.1. Validating an EB GUIDE model using EB GUIDE Studio

Validating an EB GUIDE model using EB GUIDE Studio

In the **Problems** component, EB GUIDE displays the following:

- errors
- warnings

For more information, see section 6.7.2, “Validation criteria for EB GUIDE project”.

Step 1
To expand the **Problems** component, click **Problems**.

Step 2
In the **Problems** component, click .

A list of errors and warnings is displayed.

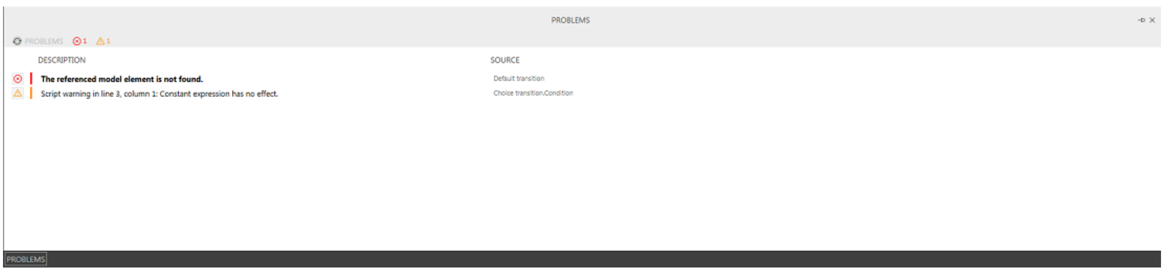


Figure 10.1. Problems component

Step 3
To navigate to the source of a problem, double-click the corresponding line.

The element that causes the problem is highlighted.

Step 4
Solve the problem.

Step 5
Click .

The problem you solved is no longer listed in the **Problems** component.

Step 6
To collapse the **Problems** component, click **Problems** once again.

If there are no errors, the EB GUIDE model is valid. The EB GUIDE model is also valid if there are some warnings.

10.4.1.2. Validating an EB GUIDE model using command line

Validating an EB GUIDE model using command line

Step 1
With command line navigate to `$GUIDE_INSTALL_PATH/Studio`.

Step 2
Enter `Studio.Console.exe -c "<logfile dir>/log.txt" -o "$GUIDE_PROJECT_PATH/project_name.ebguide"`.

The EB GUIDE model is validated and the result is saved to a logfile at the specified location `<logfile dir>`.

10.4.2. Starting and stopping the simulation

Starting and stopping the simulation

Step 1
To start the simulation, click in the command area.

The simulation and EB GUIDE Monitor start. The simulation starts with its own configuration.

To change the configuration, go to the project center, and click **Configure** > **Profiles**.

Step 2
To stop the simulation, click in the command area.

The simulation and EB GUIDE Monitor stop.

10.5. Exporting an EB GUIDE model

10.5.1. Exporting an EB GUIDE model using EB GUIDE Studio

Exporting an EB GUIDE model using EB GUIDE Studio

To copy the EB GUIDE model to the target device, you need to export it using EB GUIDE Studio.

For every export of an EB GUIDE model you select a profile.

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click the **Export** tab.

Step 3
From the `Profile` drop-down list box select a profile.

Step 4
Click **Browse**, and select a location where to export the binary files.

Step 5
Click **Select folder**.

Step 6
Click **Export**.

The binary files are exported to the selected location.

10.5.2. Exporting an EB GUIDE model using command line

Exporting an EB GUIDE model using command line

Prerequisite:

- The EB GUIDE model is free of errors and warnings.

Step 1
With command line navigate to `$GUIDE_INSTALL_PATH/Studio`.

Step 2
Enter Studio.Console.exe -e <destination dir> -p <profile> -o "$GUIDE_PROJECT_-PATH/project_name.ebguide".

The EB GUIDE model is exported to the selected location <destination dir> with the specified profile <profile>.

10.6. Changing the display language of EB GUIDE Studio

Changing the display language of EB GUIDE Studio

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click the **Options** tab.

Step 3
Select a language from the **Display language** drop-down list box.

Step 4
Restart EB GUIDE Studio.

After restarting the graphical user interface is displayed in the selected language.

10.7. Configuring profiles

EB GUIDE Studio offers the possibility to create different profiles for an EB GUIDE model.

You use profiles to do the following:

- Send messages
- Configure internal and user-defined libraries to load
- Configure a scene
- Configure a renderer

There are two default profiles: **Edit** and **Simulation**.

10.7.1. Adding a profile

Adding a profile

To add a profile in EB GUIDE Studio, clone an existing profile.

Prerequisite:

- An EB GUIDE project is opened.
- The project center is displayed.

Step 1
In the navigation area, click **Configure** > **Profiles**.

Step 2
In the content area, select the **Simulation** profile.

Step 3
Click **Clone**.

A profile is added to the table. The profile is a clone of the default profile **Simulation**.

Step 4
Double-click in the table and rename the profile to `MySimulation`.

Step 5
Select **Use for simulation**.

The `MySimulation` profile is used for simulation on the PC.

10.7.2. Adding a library

The default delivery of EB GUIDE TF runs on operating systems that support shared libraries, for example Windows 10, Linux or QNX. EB GUIDE TF is divided into executable file and a set of libraries to fit most customer projects out of the box.

The following tasks show you how to add a user-defined library that interacts with the EB GUIDE model and provides additional functionality.

Adding a library: Platform

This task shows you how to add a library or several libraries that can be used by all EB GUIDE models on the current platform.

Prerequisite:

- An EB GUIDE project is opened.
- The project center is displayed.
- In the navigation area, the tab **Configure** > **Profiles** is selected.
- A profile `MySimulation` is added.
- Libraries `MyLibraryA` and `MyLibraryB` are available in `$GTF_INSTALL_PATH/platform/<platform name>/bin`.

Step 1
In the content area, select the `MySimulation` profile.

Step 2
Click the **Platform** tab.

Step 3
Enter the following code:

```
{
  "gtf":
  {
    "core":
    {
      "pluginstoload": ["MyLibraryA", "MyLibraryB"]
    }
  }
}
```

You added libraries `MyLibraryA` and `MyLibraryB` to the start-up code.

NOTE

JSON object notation

If you configure `platform.json` within EB GUIDE Studio, use the JSON object notation.

For an example, see section 13.8.1, “Example `platform.json` in EB GUIDE Studio”.

For more information about JSON format, see http://www.json.org.

Adding a library: Model

This task shows you how to add a library or several libraries that can be used only by the current EB GUIDE model.

Prerequisite:

- An EB GUIDE project is opened.

- The project center is displayed.
- In the navigation area, the tab **Configure** > **Profiles** is selected.
- A profile `MySimulation` is added.
- Libraries `MyLibraryA` and `MyLibraryB` are available in `$GUIDE_PROJECT_PATH/<project name>/resources`.

Step 1
In the content area, select the `MySimulation` profile.

Step 2
Click the **Model** tab.

Step 3
Enter the following code:

```
{
  "gtf":
  {
    "model":
    {
      "pluginstoload": ["resources/MyLibraryA", "resources/MyLibraryB"]
    }
  }
}
```

You added libraries `MyLibraryA` and `MyLibraryB` to the start-up code.

NOTE

JSON object notation

If you configure `model.json` in EB GUIDE Studio, use the JSON object notation.

For an example, see section 13.6.1, “Example `model.json` in EB GUIDE Studio”.

For more information about JSON format, see http://www.json.org.

10.7.3. Configuring a scene

In EB GUIDE Studio it is possible to configure a scene for every state machine.

Projects can have more than one state machine for one of the following reasons:

- To separate the logic of the model into different state machines
- To use more than one display or layer

Configuring a scene

Prerequisite:

- An EB GUIDE project is opened.
- The project center is displayed.
- In the navigation area, the tab **Configure** > **Profiles** is selected.

Step 1
In the content area, click the **Scenes** tab.

Step 2
From the **State machine** drop-down list box select the state machine of your main display, for example **Main**.

Step 3
To set the initial position of the window on the PC desktop, enter a value for `x` and `y`.

Step 4
Select a renderer from the **Renderer** drop-down list box.

Step 5
Adjust further properties. For information on each property, see section 13.9, “Scenes”.

10.8. Exporting and importing language-dependent texts

EB GUIDE allows you to display text in the user's preferred langauge. To make this possible, you add language support to your EB GUIDE model. You can then export the language-dependent texts to an `.xliff` file, have them translated and import them back into your model.

NOTE

Project and language specific IDs

A unique alphanumeric `sourcelanguageid` and `targetlanguageid` is created for every project and language pair. These IDs prevent unintentional import of an `.xliff` file from another project or target language. Also each datapool item receives a unique alphanumerical ID for each language.

When the languages or datapool items are changed in the EB GUIDE Studio model while the texts are exported for translation, the translated texts can still be assigned to the right datapool item and language due to the specific ID. Refer to the import logfile, to see, if all datapool items are assigned correctly.

10.8.1. Exporting language-dependent texts

TIP

Validating the EB GUIDE model

To avoid errors during export and import of texts, validate your EB GUIDE model before you start.

For more information, see section 10.4.1.1, “Validating an EB GUIDE model using EB GUIDE Studio”.

Exporting language-dependent texts

To provide text in the user's preferred language, you export all language-dependent texts of datapool items and pass them on to your localization service provider.

Prerequisite:

- The languages to be translated are added to the EB GUIDE model. For more information, see section 8.4.1, “Adding a language to the EB GUIDE model”.
- A datapool item of type `String` or `String list` is added.
- The datapool item has language support. For information, see section 8.4.2, “Adding language support to a datapool item”.
- The EB GUIDE model is free of errors and warnings.

Step 1

Click .

The project center opens.

Step 2
In the navigation area, click **Configure** > **Languages**.

Step 3
In the content area, select the target language to be translated.

Multi-selection is possible.

Step 4
Click **Export**.

A dialog opens.

Step 5
Select a directory to export the files.

Step 6
Click **Select folder**.

Result: The export starts. The files are saved in the selected directory. The file has a language-dependent acronym and the format `.xliff`. The file contains values for the source language and values for the target language.

NOTE **Structure and content of the exported file**

- For each language you select in the project center, a separate `.xliff` file is exported.
- The source language is the default language. Therefore, if a text was not translated yet, the `target-language` element always contains the source text.

For more information on language-dependent texts in an EB GUIDE model and `.xliff` files, see section 6.12, “Languages”.

10.8.2. Importing language-dependent texts

10.8.2.1. Importing language-dependent texts using EB GUIDE Studio

Importing language-dependent texts using EB GUIDE Studio

Prerequisite:

- At minimum one translated `.xliff` file for the selected EB GUIDE model is available.

- The datapool item to be translated and the target language still exists.
- The EB GUIDE model is free of errors and warnings.

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click **Configure** > **Languages**.

Step 3
Click **Import**.

A dialog opens.

Step 4
Select the directory where the translated `.xliff` file is stored.

Step 5
Select the translated `.xliff` file.

Multi-selection is possible.

Step 6
Click **Open**.

The import starts. A dialog opens.

Step 7
Click **Close**.

Now all datapool items with language support show the corresponding language dependent text. See the logfile for more information on the import.

10.8.2.2. Importing language-dependent texts using command line

Importing language-dependent texts using command line

Prerequisite:

- At minimum one translated `.xliff` file for the selected EB GUIDE model is available.
- The datapool item that was sent to translation still exists.
- The EB GUIDE model is free of errors and warnings.

Step 1
With command line navigate to `$GUIDE_INSTALL_PATH/Studio`.

Step 2
Enter `Studio.Console.exe -l <language file> -o "$GUIDE_PROJECT_PATH/project_-name.ebguide"`.

If the import was successful, the EB GUIDE model is changed. If the import was not successful, the EB GUIDE model is not changed. In both cases a logfile is generated. A date and a time stamp are added to the name of the logfile.

10.9. Working with EB GUIDE Monitor

For more information on EB GUIDE Monitor, see section 6.5.4, “EB GUIDE Monitor”.

NOTE **Default window layout**
All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Monitor window to default layout by selecting **Layout** > **Reset to default layout**.

10.9.1. Firing an event in EB GUIDE Monitor

Firing an event in EB GUIDE Monitor

Prerequisite:

- The EB GUIDE model contains an event.
- The simulation of the EB GUIDE model is started.
- The EB GUIDE Monitor is started.

Step 1
In EB GUIDE Monitor, in the **Events** component, click + .

A dialog opens.

Step 2
Select the event to be fired and click **Accept**.

The event is added to the list.

Step 3
To fire an event, click in the **Events** component next to the event.

The event is fired. In the **Logger** component a log message appears.

Step 4
If the event has parameters, do the following:

Step 4.1
Click ▶ to expand parameters.

Step 4.2
Change parameters in the **Value** column.

Step 4.3
To fire an event, click next to the event.

The event is fired with changed parameters. In the **Logger** component a log message appears.

10.9.2. Changing value of the datapool item with EB GUIDE Monitor

Changing value of the datapool item in EB GUIDE Monitor

Prerequisite:

- The EB GUIDE model contains a datapool item.
- The simulation of the EB GUIDE model is started.
- The EB GUIDE Monitor is started.

Step 1
In EB GUIDE Monitor, in the **Datapool** component, click + .

A dialog opens.

Step 2
Select the datapool item and click **Accept**.

The datapool item is added to the list.

Step 3
Change the value of the datapool item in the **Value** column.

NOTE

Supported types

You can change datapool items of the following data types:

- Boolean
- Color
- Integer
- Float
- String

The value of the datapool item is changed. In the **Logger** component a log message appears.

10.9.3. Starting scripts in EB GUIDE Monitor

Starting scripts in EB GUIDE Monitor

Prerequisite:

- The simulation of the EB GUIDE model is started.
- The EB GUIDE Monitor is started.
- A `.cs` or a `.dll` file with a script is available on your computer. For script examples, see section 10.9.3.1, “Writing script files for EB GUIDE Monitor”.

Step 1
To open the **Scripting** component, select **Layout** > **Scripting**.

The **Scripting** component opens as a docked component.

Step 2
In the **Scripting** component click the **Open** button.

The file explorer opens.

Step 3
Select a `.cs` or a `.dll` file and click **Open**.

All applicable methods and the corresponding classes, which were included in the file, are listed in the **Script** table.

Step 4
Select a method and click the start button.

The script is started. In the **Script output** area a log message appears.

10.9.3.1. Writing script files for EB GUIDE Monitor

For more information on script methods, see the EB GUIDE Monitor API in `$GUIDE_INSTALL_PATH/doc/monitor/monitor_api.chm`.

The following is an example for basic EB GUIDE Monitor script functions.

NOTE

Using methods for states and state machines

If your EB GUIDE model has several states or state machines with identical names, use `uint` IDs. Find `uint` IDs that are relevant for your project in `$EXPORT_PATH/monitor.cfg`.

Example 10.1.
Example script file for EB GUIDE Monitor

The following is an example script `MonitorScriptSample.cs`.

```
namespace MyProject
{
    using System.Threading.Tasks;

    using System.Windows.Media; // necessary for Color type!

    using Elektrobit.Guide.Monitor.Scripting.MonitorContext;

    public class Basic
    {
        public async Task PrintMessage(IMonitorContext monitor) //❶
        {
            await monitor.Write("Hello World");
        }

        public async Task FireEvent(IMonitorContext monitor) //❷
        {
            await monitor.FireEvent("nextView");
        }
    }

    public class Events
    {
        public async Task FireEventWithParameter(IMonitorContext monitor)
        {
            await monitor.FireEvent("setBool", true);
        }

        public async Task WaitForEvent(IMonitorContext monitor) //❸
```

```
        {
            var ev = await monitor.WaitForEvent("nextView");
            await monitor.Write("Even occured: " + ev.EventModel.Name);
        }

        public async Task WaitForEventWithParameters(IMonitorContext monitor)
        {
            var ev = await monitor.WaitForEvent("setBool");

            bool mv1 = ev["value"]; // read parameter via name
            bool mv2 = ev[0]; // read the parameter via index

            await monitor.Write("Parameter 'value' is: " + mv1);
            await monitor.Write("Parameter [0] is: " + mv2);
        }
    }

    public class Datapool
    {
        public async Task WriteDpValue(IMonitorContext monitor) //❹
        {
            await monitor.WriteDatapool("Boolean 1", true);
        }

        public async Task ReadDatapoolValue(IMonitorContext monitor) //❺
        {
            bool boolValue = await monitor.ReadDatapool("Boolean 1");
            string stringValue = await monitor.ReadDatapool("String 1");
            int integerValue = await monitor.ReadDatapool("Integer 1");
            float floatValue = await monitor.ReadDatapool("Float 1");

            await monitor.Write("Boolean: " + boolValue);
            await monitor.Write("String: " + stringValue);
            await monitor.Write("Integer: " + integerValue);
            await monitor.Write("Float: " + floatValue);
        }

        public async Task ReadColor(IMonitorContext monitor)
        {
            Color colorValue = await monitor.ReadDatapool("Color 1");
            await monitor.Write("Boolean: " + colorValue);
        }
    }

    public class StateMachines
    {
        public async Task WaitForStateChanges(IMonitorContext monitor)
```

```
        {
            var leftState = await monitor.WaitForStateExit
                        ("Main", "State 1"); //❻
            await monitor.Write(string.Format("State {0} left",
                        leftState.Name));

            var enteredState = await monitor.WaitForStateEnter
                        ("Main", "State 2"); //❼
            await monitor.Write(string.Format("State {0} entered",
                        enteredState.Name));
        }

        public async Task WaitForStateMachineChanges(IMonitorContext monitor)
        {
            var startedStateMachine = await monitor.WaitForStateMachineStart
                       ("Dynamic state machine 1"); //❽
            await monitor.Write(string.Format("State Machine {0} started",
                       startedStateMachine.Name));

            var stoppedStateMachine = await monitor.WaitForStateMachineStop
                       ("Dynamic state machine 1"); //❾
            await monitor.Write(string.Format("State Machine {0} stopped",
                       stoppedStateMachine.Name));
        }
    }

    public class Advanced
    {
        public async Task CaptureScreenshot(IMonitorContext monitor) //❿
        {
            // make sure remote framebuffer is enabled in profile
            uint sceneId = 0;
            await monitor.CaptureScreenshot(sceneId, @"d:/image.png");
        }

        public async Task CountTo10(IMonitorContext monitor)
        {
            for (var i = 0; i < 10; i++)
            {
                await monitor.Write("Hello World: " + i);
                await Task.Delay(1000, monitor.CancellationToken);

                monitor.CancellationToken.ThrowIfCancellationRequested();
            }
        }
```

```
        public async Task WaitForEventWithTimeout(IMonitorContext monitor) //11
        {
            // Disclaimer:
            // this is just one of many opportunities provided by
            // the .NET's "Task Parallel Library"

            var eventWaitTask = monitor.WaitForEvent("nextView");

            await Task.WhenAny(eventWaitTask, Task.Delay(5000));

            if (!eventWaitTask.IsCompleted || eventWaitTask.IsFaulted)
            {
                return;
            }

            await monitor.Write("event occured");
        }
    }

    namespace MonitorScripting.EventScripts //12
    {
        using Elektrobit.Guide.Monitor.Scripting.MonitorContext;
        using System.Threading;
        using System.Threading.Tasks;
        using Elektrobit.Guide.Monitor.Model.Event;
        using Elektrobit.Guide.Monitor.Model.Value;

        public class MonitorScripts
        {
            public async Task FireEventInNamespace(IMonitorContext monitor)
            {
                string[] namespacePath = { "Foo_namespace", "go_to_view2" };
                var identifier = new QualifiedIdentifier(namespacePath);

                await monitor.FireEvent(identifier);
            }

            public async Task FireEventInNestedNamespace(IMonitorContext monitor)
            {
                string[] namespacePath =
                    { "Foo_namespace", "sub_namespace_under_foo", "go_to_view4" };
                var identifier = new QualifiedIdentifier(namespacePath);

                await monitor.FireEvent(identifier);
            }

            public async Task FireEventInRootNamespace(IMonitorContext monitor)
```

```
                {
                    await monitor.FireEvent("go_to_view3");
                }
            }
        }
}
```

❶ Method to print out a message
❷ Method to fire an event
❸ Method to wait for an event
❹ Method to write a datapool value
❺ Method to read a datapool value
❻ Method to wait until the state is entered and then to report it
❼ Method to wait until the state is exited and then to report it
❽ Method to wait until the state machine is started and then to report it
❾ Method to wait until the state machine is stopped and then to report it
❿ Method to capture a screenshot
11 Method to wait for an event with timeout
12 Example how to work with namespaces

10.9.4. Starting EB GUIDE Monitor as a stand-alone application

EB GUIDE Monitor starts automatically in EB GUIDE Studio during the simulation of an EB GUIDE model. But you can also start EB GUIDE Monitor as a stand-alone application in `$GUIDE_INSTALL_PATH/tools/monitor` or using the command line.

Starting EB GUIDE Monitor using command line

Prerequisite:

- EB GUIDE is installed.
- An EB GUIDE model is exported to `$EXPORT_PATH`.

Step 1
In the file explorer, navigate to `$GUIDE_INSTALL_PATH/tools/monitor`.

Step 2
Open command line and enter `Monitor.exe`

EB GUIDE Monitor starts.

Changing the display language of EB GUIDE Monitor

Prerequisite:

- EB GUIDE Monitor is started as a stand-alone application.

Step 1
From **File** > **Display language** select a language.

Step 2
Restart EB GUIDE Monitor.

After restarting, the graphical user interface is displayed in the selected language.

NOTE

EB GUIDE Monitor inherits the language from EB GUIDE Studio
If EB GUIDE Monitor is started in EB GUIDE Studio, you cannot change the display language of the graphical user interface. EB GUIDE Monitor has the same display language as EB GUIDE Studio.

10.9.5. Loading configurations into EB GUIDE Monitor

Loading a configuration file into EB GUIDE Monitor

Prerequisite:

- EB GUIDE Monitor is started as a stand-alone application.
- An EB GUIDE model is exported to `$EXPORT_PATH`.

Step 1
Select **File** > **Load configuration.**

A dialog opens.

Step 2
Navigate to `$EXPORT_PATH` and select the `monitor.cfg` configuration file.

Step 3
Click **Open**.

The configuration of your project is loaded into EB GUIDE Monitor.

Loading a recent configuration file into EB GUIDE Monitor

Prerequisite:

- EB GUIDE Monitor is started as a stand-alone application.
- One or more configurations have been used recently.

Step 1
Select **File** > **Recent configuration files**.

A dialog opens.

Step 2
Navigate to the respective location and select a configuration file.

The configuration is loaded into EB GUIDE Monitor.

NOTE

EB GUIDE Monitor disconnects from EB GUIDE GTF
Before a new configuration is loaded, EB GUIDE Monitor is automatically disconnected from the current EB GUIDE GTF.

EB GUIDE Monitor reconnects and loads the new configuration.

10.9.6. Exporting and importing watch lists

The events and datapool items you use for a project are stored in watch lists. In order to use the items for other projects you can export the watch lists as an `.xml` file and later import them into your new projects.

Exporting all watch lists

Prerequisite:

- EB GUIDE Monitor is started.
- An EB GUIDE model is already set up with items stored in **Datapool** or **Events** tabs.

Step 1
To export all watch lists, select **File** > **Export all watch lists**.

A dialog opens.

Step 2
Select a destination directory and enter a file name.

All datapool items and events are exported.

Exporting a single watch list

Prerequisite:

- EB GUIDE Monitor is started.
- An EB GUIDE model is already set up with items stored in **Datapool** or **Events** components.

Step 1
Go to the **Datapool** or **Events** component that contains the items you want to export.

Step 2
To save the list of items of this tab, click **Export**.

A dialog opens.

Select a destination directory and enter a file name.

The datapool items or events of the component are exported.

Importing watch lists

Prerequisite:

- EB GUIDE Monitor is started.
- An exported watch list is already available.

Step 1
To import a watch list, select **File** > **Import watch lists**.

A dialog opens.

Step 2
Select a watch list file to import.

The datapool items or events are opened in new **Datapool** or **Events** components.

NOTE

Layout is not imported

Only the datapool items and events are imported, not the layout.

The default layout is used for the newly opened **Datapool** and **Events** components.

11. Best practices

The topics in this chapter are sorted alphabetically.

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

11.1. Best practice: `Conditional script`

Properties of datapool items and widgets converted to a scripted value lead to EB GUIDE Script execution each time the property is read. In some use cases, to minimize the number of EB GUIDE Script executions and improve the performance, do the following:

1. If you have a scripted value, reset the property type: Use properties which are not converted to a scripted value but have their plain value. For more information, see section 9.8, “Converting a property to a scripted value” and section 8.2.5, “Adding a user-defined property to a widget”.
2. To compute and set the current value, add a user-defined property of type `Conditional script`. Consider that this action has to be executed only if it is necessary, for example, on initialization or when an input property was changed.

12. Tutorials

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

12.1. Tutorial: Adding a dynamic state machine

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

Dynamic state machines allow pop-ups during run-time. You use dynamic state machines for example to display error messages that overlay the regular display.

The following instructions guide you through the process of creating a dynamic state machine. The instructions show you how to model a dynamic state machine for volume control. For best results, work through the following steps in the order presented.

Approximate duration: 20 minutes.

Adding events and datapool items

The following instructions guide you through the process of adding events and datapool items. These events are used to change the volume afterwards. The purpose of the datapool item is to change the position of a graphical element in a later section.

Step 1
Go to the **Events** component and click +.

An event is added to the table.

Step 2
Rename the event to `Volume up`.

Step 3
Add an event, and rename it to `Volume down`.

Step 4
Add an event, and rename it to `Close volume control`.

Step 5
Go to the **Datapool** component and click +.

A menu expands.

Step 6
In the menu, click **Integer**.

A datapool item of type `Integer` is added.

Step 7
Rename the datapool item to `Volume indicator`.

You added three events and a datapool item.

Adding a dynamic state machine and modeling the behavior

The following instructions guide you through the process of adding a dynamic state machine. The haptic dynamic state machine that you model is used to control the volume.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, go to **Dynamic state machines** and click +.

A menu expands.

Step 2
In the menu, click **Haptic dynamic state machine**.

A haptic dynamic state machine is added and displayed in the content area.

Step 3
Rename the dynamic state machine to `Volume control`.

Step 4
Drag an initial state from the **Toolbox** into the dynamic state machine.

Step 5
Drag a view state from the **Toolbox** into the dynamic state machine.

Along with the view state, a view is added to the EB GUIDE model.

Step 6
In the **Navigation** component, click the view state.

Step 7
Press the **F2** key, and rename the view state to `Volume`.

Step 8
Add a transition from the initial state to the `Volume` view state.

Modeling a slider

The following instructions guide you through the process of modeling a horizontal slider indicator. The slider indicator shows the volume during run-time.

The slider indicator consists of two rectangles. One rectangle represents the background of the slider. The second rectangle indicates the volume.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, expand the `Volume` view state. Double-click the view.

The content area displays the view.

Step 2
Drag a rectangle from the **Toolbox** into the view.

Step 3
In the **Navigation** component, click the rectangle, and press the **F2** key.

Step 4
Rename the rectangle to `Slider background`.

Step 5
To change the appearance of `Slider background`, click the rectangle, and go to the **Properties** component.

Step 5.1
Enter `500` in the `width` text box.

Step 5.2
Enter `125` in the `x` text box.

Step 5.3
Enter `300` in the `y` text box.

Step 6
Drag a rectangle from the **Toolbox** into `Slider background` in the **Navigation** component.

The rectangle is added as a child widget to `Slider background`.

Step 7
In the **Navigation** component, click the rectangle, and press the **F2** key.

Step 8
Rename the rectangle to `Indicator`.

Step 9
To change the appearance of `Indicator`, click the rectangle, and go to the **Properties** component.

Step 9.1
Enter `40` in the `width` text box.

Step 9.2
Enter `80` in the `height` text box.

Step 9.3
Next to the `x` property, click the button.

A menu expands.

Step 9.4
In the menu, click **Add link to datapool item**.

A dialog opens.

Step 9.5
From the list, select the `Volume indicator` datapool item.

Step 9.6
Click **Accept**.

The dialog closes. The button is displayed next to the `x` property. The values of `x` and `Volume indicator` are now linked.

Step 9.7
Enter `10` in the `y` text box.

Step 9.8
Select black for the `fillColor` property.

You added two rectangles to the view. You changed the appearance of the rectangles.

Step 10
In the **Datapool** component, click the `Volume indicator` datapool item.

Step 11
In the `Value` text box enter `10`.

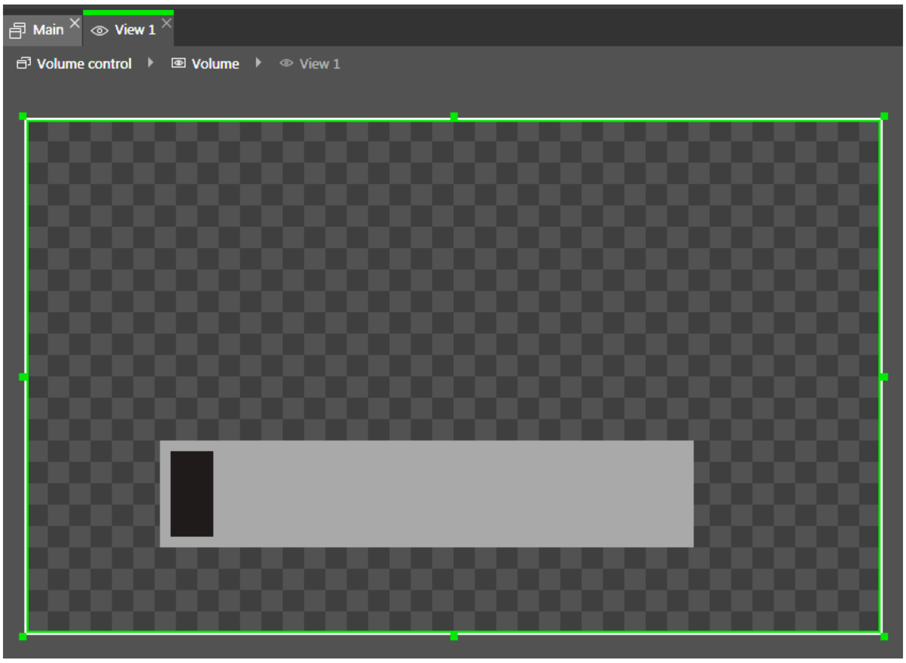


Figure 12.1. Appearance of `View 1` with two rectangles

In the content area, the `Indicator` rectangle changes the position.

The `Volume indicator` datapool item controls the `x` position of the `Indicator` rectangle.

Adding states to the **Main** state machine

In the following instructions, you add an initial state and a view state to the **Main** state machine. You use the view state to run the dynamic state machine in parallel to other state machines.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, double-click **Main**.

The **Main** state machine is displayed in the content area.

Step 2
Drag an initial state from the **Toolbox** into the state machine.

Step 3
Drag a view state from the **Toolbox** into the state machine.

Along with the view state, a view is added to the EB GUIDE model.

Step 4
Rename the view state to `Home`.

Step 5
In the content area, click the initial state.

Step 6
Add a transition from the initial state to the `Home` view state.

Step 7
In the **Navigation** component, click **Main**.

Step 8
In the **Properties** component, select the `Dynamic state machine list` check box.

With these steps done, you can use EB GUIDE Script functions that are related to dynamic state machines.

You added an initial state and a view state to the **Main** state machine. The haptic dynamic state machine runs in parallel to the **Main** state machine.

Adding internal transitions to the **Main** state machine

In the following instruction, you add internal transitions. You use the internal transitions to start (push) and stop (pop) the dynamic state machine during run-time.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, click the **Main** state machine.

Step 2
In the **Properties** component, go to **Internal transitions**, and click + .

An internal transition is added to the state machine. The internal transition is visible in the **Navigation** component.

Step 3
Add two more internal transitions.

Step 4
In the **Navigation** component, click the first internal transition.

Step 4.1
Go to the **Properties** component.

Step 4.2
In the **Trigger** combo box, search for the `Volume up` event and double-click it.

Step 4.3
Next to the **Action** property, double-click + .

Step 4.4
Enter the following EB GUIDE Script:

```
function()
  {
  dp:"Volume indicator" = dp:"Volume indicator" + 20
  f:pushDynamicStateMachine(popup_stack:Main, sm:"Volume control", 0)
  }
```

Step 4.5
Click **Accept**.

The action is added to the transition. In the **Navigation** component, the internal transition is renamed to `Volume up`.

Step 5
In the **Navigation** component, click the second internal transition.

Step 5.1
Go to the **Properties** component.

Step 5.2
In the **Trigger** combo box, search for the `Volume down` event and double-click it.

Step 5.3
Next to the **Action** property, click + .

Step 5.4
Enter the following EB GUIDE Script:

```
function()
  {
  dp:"Volume indicator" = dp:"Volume indicator" - 20
  f:pushDynamicStateMachine(popup_stack:Main, sm:"Volume control", 0)
  }
```

Step 5.5
Click **Accept**.

The action is added to the transition. In the **Navigation** component, the internal transition is renamed to `Volume down`.

Step 6
In the **Navigation** component, click the third internal transition.

Step 6.1
Go to the **Properties** component.

Step 6.2
In the **Trigger** combo box, search for the `Close volume control` event and double-click it.

Step 6.3

Next to the **Action** property, click + .

Step 6.4

Enter the following EB GUIDE Script:

```
function()
 {
 f:popDynamicStateMachine(popup_stack:Main,sm:"Volume control")
 }
```

Step 6.5

Click **Accept**.

The action is added to the transition. In the **Navigation** component, the internal transition is renamed to `Close volume control`.

You added three internal transitions which start and stop the dynamic state machine. Furthermore, the internal transitions `Volume up` and `Volume down` change the position of the `Indicator` rectangle.

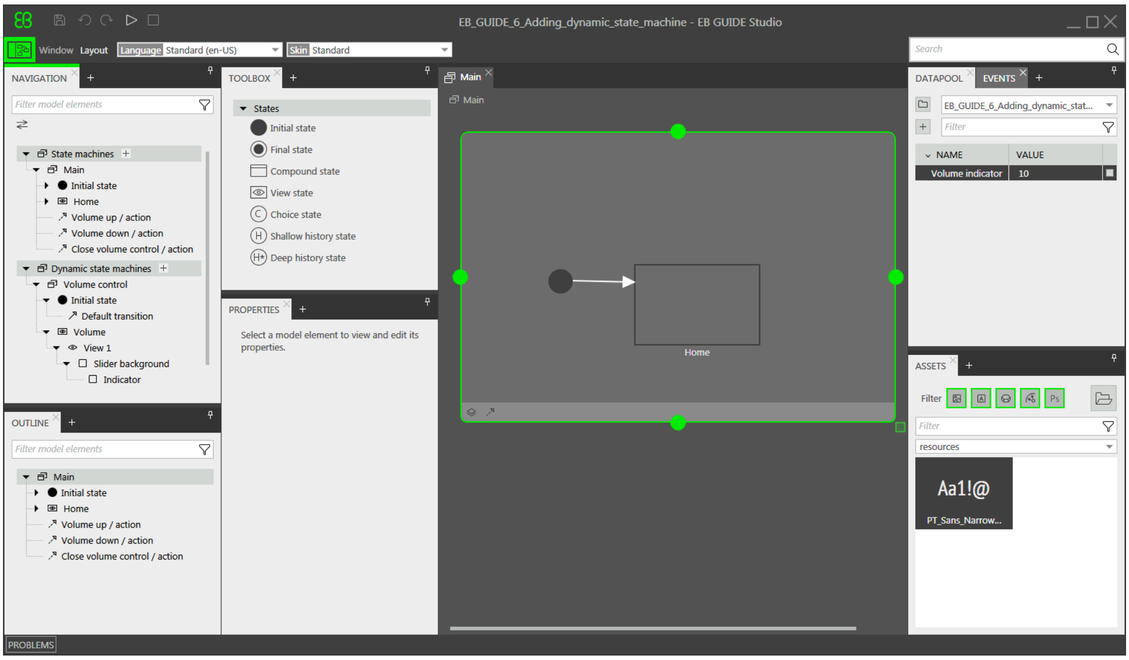


Figure 12.2. EB GUIDE model with all model elements

Starting the simulation and testing the EB GUIDE model

Prerequisite:

- You completed the previous instruction.

To start the simulation, click ▷ in the command area.

The simulation and EB GUIDE Monitor start. The EB GUIDE model displays the `Home` view state.
Step 1
In EB GUIDE Monitor in the **Events** component, click + , select the `Volume up` event and then click to fire the event.

The dynamic state machine is started and shows the slider indicator. The dynamic state machine overlays the `Home` view state.

When you fire the events `Volume up` or `Volume down` the black `Indicator` rectangle moves. If you fire the event `Close volume control`, the slider disappears from the view.

If you add additional states to the **Main** state machine, the `Volume control` dynamic state machine will overlay the other states as well.

12.2. Tutorial: Modeling button behavior with EB GUIDE Script

NOTE

Default window layout
All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

With EB GUIDE Script you can express property values, actions, or conditions and evaluate them during runtime.

The following instructions guide you through the process of using EB GUIDE Script to model the behavior of a button. The button increases in size when it is clicked and shrinks back to its original size when it reaches a defined maximum size. For best results, work through the steps in the order presented.

Approximate duration: 10 minutes.

Adding widgets

Prerequisite:

- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state.

- The content area displays the view.

Step 1
Drag a rectangle from the **Toolbox** into the view.

Step 2
In the **Navigation** component, click the rectangle, press the **F2** key, and rename the rectangle to `Background`.

Step 3
Drag a rectangle from the **Toolbox** into the **Navigation** component. Place it as a child widget to the `Background` rectangle.

Step 4
In the **Navigation** component, click the new rectangle, press the **F2** key, and rename the rectangle to `Button`.

Step 5
Drag a label from the **Toolbox** into the **Navigation** component. Place the label as a child widget to the `Button` rectangle.

Step 6
In the **Navigation** component, click the label, press the **F2** key, and rename the label to `Button text`.

Your widget hierarchy now looks as follows.

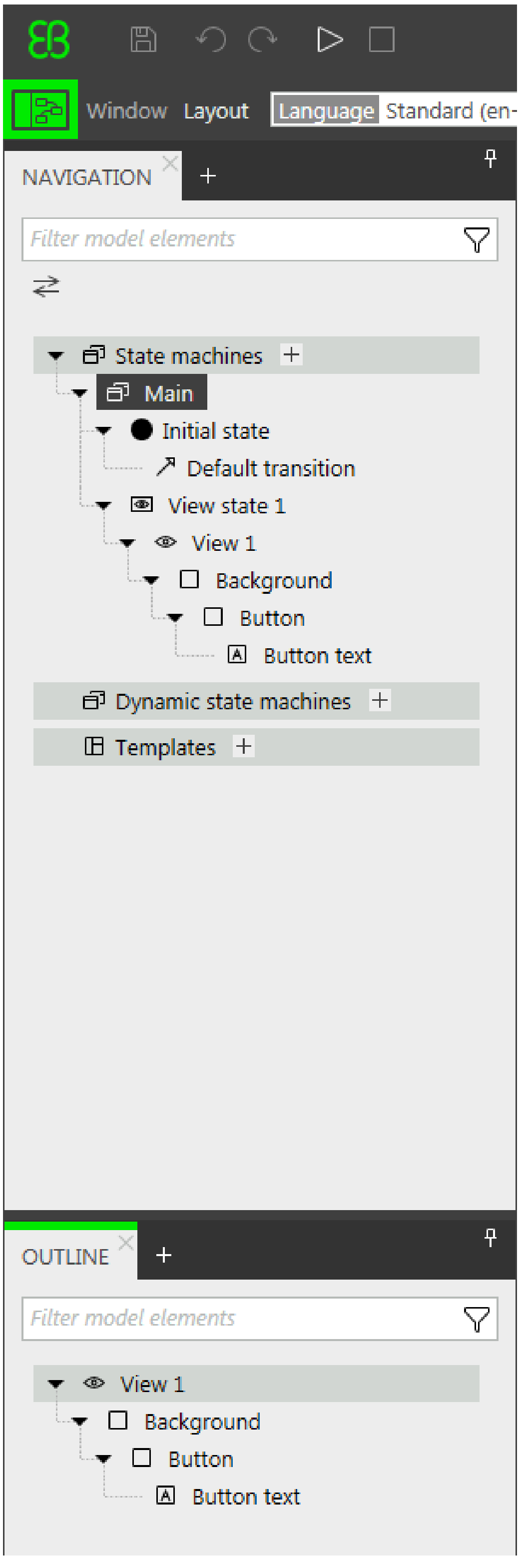


Figure 12.3. Widget hierarchy

Configuring the background

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, click the `Background` rectangle, and go to the **Properties** component.

Step 2
Next to the `width` property, click the button.

A menu expands.

Step 3
In the menu, click **Add link to widget property**.

A dialog opens.

Step 4
In the dialog, go to the view, and select its `width` property.

Step 5
Click **Accept**.

The dialog closes. The button is displayed next to the `width` property.

Step 6
Link the `height` property of the `Background` rectangle to the `height` property of the view.

Step 7
Link the `x` property of the `Background` rectangle to the `x` property of the view.

Step 8
Link the `y` property of the `Background` rectangle to the `y` property of the view.

The `Background` rectangle covers the exact size and position of the view.

Defining the maximum button width

A datapool item holds the value for the maximum width of the button. It can be changed during run-time.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Datapool** component, click +.

A menu expands.

Step 2
In the menu, click **Integer**.

A new datapool item of type `Integer` is added.

Step 3
Rename the datapool item to `Maximum width`.

Step 4
In the `Value` text box, enter `400`.

Configuring the button

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, click the `Button` rectangle, and go to the **Properties** component.

Step 1.1
Enter `50` in the `height` text box.

Step 1.2
Enter `350` in the `x` text box.

Step 1.3
Enter `215` in the `y` text box.

Step 1.4
Select blue for the `fillColor` property.

The button is now colored blue.

Step 2
In the **Widget feature properties** category, click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 3
Under **Available widget features**, expand the **Input handling** category, and select the **Touch pressed** widget feature.

Step 4
Click **Accept**.

The related widget feature properties are added to the `Button` rectangle and displayed in the **Properties** component.

Step 5
Next to the `touchPressed` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 6
Replace the existing EB GUIDE Script with the following code:

```
function(v:touchId::int, v:x::int, v:y::int, v:fingerId::int)
     {
         if (v:this.width > dp:"Maximum width") // If the button has grown
                                            // beyond its maximum size...
      {
  // ...reset its dimensions to the default values.
  v:this.height = 50
  v:this.width = 100
  v:this.x = 350
  v:this.y = 215
      }
      else // Otherwise...
      {

  // ... increase button size...
  v:this.width += 80
  v:this.height += 40

  // ...and move the button to keep it centered.
  v:this.x -= 40
  v:this.y -= 20
      }
         false
     }
```

Step 7
Click **Accept**.

You configured the `Button` rectangle and wrote an EB GUIDE Script which changes the size of the `Button` rectangle in run-time.

Configuring the button text

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, click the `Button text` label, and go to the **Properties** component.

Step 2
Enter `grow!` in the `text` text box.

Step 3
Link the `width` property of the `Button text` label to the `width` property of the `Button` rectangle.

Step 4
Link the `height` property of the `Button text` label to the `height` property of the `Button` rectangle.

Step 5
Enter `0` in the `x` text box.

Step 6
Enter `0` in the `y` text box.

Step 7
Next to the `horizontalAlign` property, select `center(1)`.

Now the `Button text` label and the `Button` rectangle are equal in size and position.

Saving and testing the EB GUIDE model

Prerequisite:

- You completed the previous instruction.

Step 1
To save the project, click in the command area.

Step 2
To start the simulation, click ▷ in the command area.

Result:

The simulation starts the EB GUIDE model you created. It behaves as follows.

1. First, it displays a grey screen with a blue button in its center. The screen looks as follows.

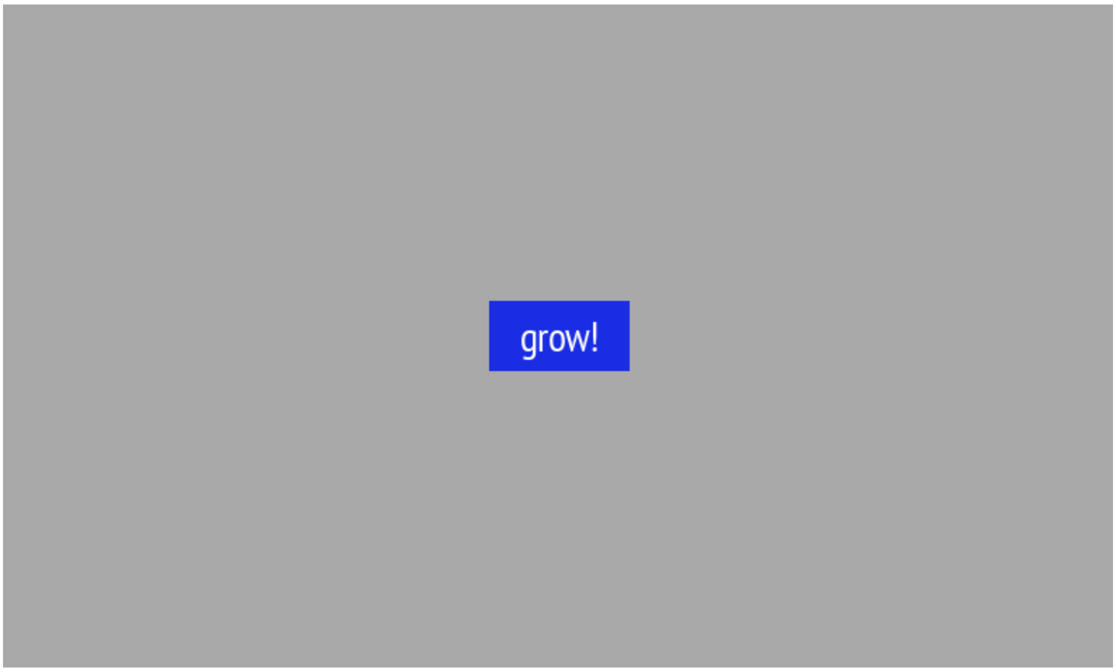


Figure 12.4. Result

2. Whenever you click the button, it increases in size but keeps its position at the center of the screen.
3. As soon as the button width reaches the value of the `Maximum width` datapool item, it shrinks back to its original size and position.

12.3. Tutorial: Modeling a path gesture

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

Path gestures are shapes drawn by a finger on a touch screen or entered by some other input device.

The following instructions guide you through the process of modeling a path gesture.

Approximate duration: 10 minutes

Adding widgets and configuring default widget properties

Prerequisite:

- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state.
- The content area displays a view.

Step 1
Drag a rectangle from the **Toolbox** into the view.

Step 2
Drag a label from the **Toolbox** into the rectangle.

The label is added as a child widget to the rectangle.

The **Properties** component displays the properties of the label.

Step 3
In the **Properties** component, enter `500` in the `width` text box.

Step 4
Select the rectangle.

The **Properties** component displays the properties of the rectangle.

Step 5
Enter `500` in the `width` text box.

Step 6
In the **Properties** component, go to **fillColor**, and select red.

You added two widgets and configured default widget properties.

Adding widget features to a rectangle

To enable the user to enter a shape starting on the widget, you add the widget feature **Path gesture** to the rectangle. The shape is matched against a set of known shapes and, if a match is found, a gesture is recognized.

Prerequisite:

- You completed the previous instruction.

Step 1
Select the rectangle.

The **Properties** component displays the properties of the rectangle.

Step 2
In the **Properties** component, go to **Widget feature properties**, and click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 3
Under **Available widget features**, expand the **Gestures** category, and select `Path gestures`.

The **Touched** widget feature is automatically selected, as it is required for the **Gestures** widget feature.

Step 4
Click **Accept**.

The related widget feature properties are added to the rectangle and displayed in the **Properties** component.

Step 5
For the **Path gestures** widget feature edit the following properties:

Step 5.1
Next to the `onPath` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 5.2
Enter the following EB GUIDE Script:

```
function(v:gestureId::int)
    {
    v:this->"Label 1".text = "recognized path gesture #"
    + f:int2string(v:gestureId);
    }
```

Step 5.3
Click **Accept**.

Step 5.4
Next to the `onPathStart` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 5.5
Enter the following EB GUIDE Script:

```
function()
    {
    v:this->"Label 1".text = "path gesture start";
    }
```

Step 5.6
Click **Accept**.

Step 5.7
Next to the `onPathNotRecognized` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 5.8
Enter the following EB GUIDE Script:

```
function()
    {
    v:this->"Label 1".text = "shape not recognized";
    }
```

Step 5.9
Click **Accept**.

Step 6
To start the simulation, click ▷ in the command area.

The simulation and EB GUIDE Monitor start. To see a reaction, draw a shape with the mouse inside the rectangle.

12.4. Tutorial: Creating a list with dynamic content

NOTE **Default window layout**

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

Instantiators allow creating lists dynamically during run-time. Based on a datapool item of a list type, an instantiator displays all list elements in a pre-defined layout. If the content of the datapool item is modified, so is the appearance of the instantiator.

The following instructions guide you through the process of creating a list with dynamic content. Each list element consists of a labeled rectangle.

Approximate duration: 15 minutes.

Adding a datapool item

The following instructions guide you through the process of adding a datapool item of type `String list`. The datapool item provides a value for every list element of the instantiator. If the content of the datapool item is modified, so is the appearance of the instantiator.

Prerequisite:

- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state.

Step 1
To display content in your list, add a datapool item of type `String list`.

In the **Datapool** component, click +.

A menu expands.

Step 2
In the menu, click **String list**.

A new datapool item of type `String list` is added.

Step 3
Rename the datapool item to `MyStringList`.

Step 4
Select the **Value** column and click button.

An editor opens.

Step 4.1
Click **Add**.

A new entry is added to the table.

Step 4.2
Enter `One` in the `Value` text box.

Step 4.3
Add the values `Two`, `Three`, `Four`, and `Five` to the `MyStringList` datapool item.

Step 4.4
Click **Accept**.

You added a datapool item of type `String list`. The datapool item contains five entries.

The content of the list is displayed next to the `Value` property.

Adding widgets

Prerequisite:

- You completed the previous instruction.

Step 1
To add widgets to your view, double-click the view state in the content area.

The view is displayed in the content area.

Step 2
In the **Navigation** component, expand the view state and the view.

Step 3
Drag an instantiator from the **Toolbox** into the view. Rename the instantiator to `MyInstantiator`.

Step 4
Drag a rectangle from the **Toolbox** into the instantiator. Rename the rectangle to `MyRectangle`.

Step 5
Drag a label from the **Toolbox** into the rectangle. Rename the label to `MyLabel`.

The widget hierarchy now looks as follows.

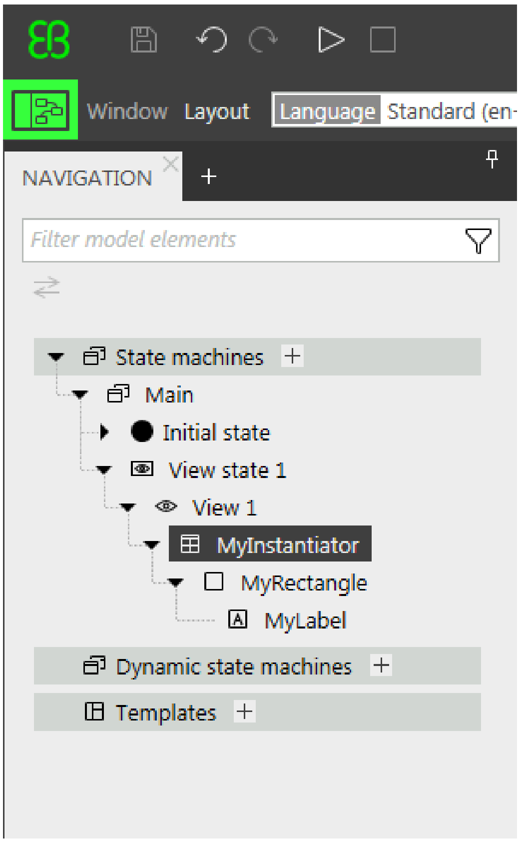


Figure 12.5. Widget hierarchy with an instantiator

Configuring the instantiator

Prerequisite:

- You completed the previous instruction.

Step 1
To change the properties of `MyInstantiator`, select the instantiator and go to the **Properties** component.

Step 2
Enter `300` in the `width` text box, and in the `height` text box.

Step 3
Enter `250` in the `x` text box.

Step 4
Enter `150` in the `y` text box.

Step 5
To calculate the length of the list dynamically, add a conditional script.

In the **User-defined properties** category, click +.

A menu expands.

Step 5.1
In the menu, click **Conditional script**.

Step 5.2
Rename the property to `calculateNumItems`.

Step 5.3
Next to the property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 5.4
Under **Triggers**, enter `dp:MyStringList`.

Step 5.5
Enter the following **On trigger** script:

```
function(v:arg0::bool)
{
 v:this.numItems = length dp:MyStringList;
 false
}
```

You added a script which automatically changes the number of list entries depending on the content of `MyStringList`.

Step 6
To arrange all labels within the instantiator, add a layout to it.

In the **Widget feature properties** category, click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 6.1
Under **Available widget features**, expand the **Layout** category, and select the **Box layout** widget feature to arrange the labels side by side.

The related widget feature properties are added to the instantiator and displayed in the **Properties** component.

Step 6.2
Click **Accept**.

Step 6.3
Enter `5` in the `gap` text box to set a spacing of 5 px between each list element.

Step 6.4
From the `layoutDirection` drop-down list box select `vertical (=1)`, to arrange the labels among each other.

You configured the instantiator which defines the visual appearance of the list and adapts the number of list items dynamically.

Configuring list element texts

Prerequisite:

- You completed the previous instruction.

Step 1
To change the appearance of the label, select the `MyLabel` and go to the **Properties** component.

Step 2
Enter `0` in the `x` and `y` text boxes.

Step 3
Add a link from the label's `width` property to the rectangle's `width` property.

Step 3.1
Next to the `width` property, click the button.

A menu expands.

Step 3.2
In the menu, click **Add link to widget property**.

A dialog opens.

Step 3.3
In the dialog, go to the rectangle, and select its `width` property.

Step 3.4
Click **Accept**.

The dialog closes. The button is displayed next to the `width` property.

Step 4
Add a link from the label's `height` property to the rectangle's `height` property.

Step 5
Next to the `horizontalAlign` property, select `center(1)`.

You changed the appearance of the label. The label is now centered in the rectangle.

Configuring list elements

Prerequisite:

- You completed the previous instruction.

Step 1
To change the appearance of the rectangle, select the rectangle and go to the **Properties** component.

Step 2
To make sure that the list elements use the available width, add a link from the rectangle's `width` property to the instantiator's `width` property.

Step 3
Enter `50` in the `height` text box.

Step 4
To define a unique position for each line of your list, add the **Line index** widget feature.

Step 4.1
In the **Widget feature properties** category, click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 4.2
Under **Available widget features**, expand the **List management** category, and select the **Line index** widget feature.

The `lineIndex` property is added to the rectangle's properties.

Step 5
To fill the labels of the list with the content of `MyStringList`, add a conditional script.

Step 5.1
Next to the **User-defined properties** category, click +.

A menu expands.

Step 5.2
In the menu, click **Conditional script**.

Step 5.3
Rename the property to `setText`.

Step 5.4
Next to the `setText` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 5.5
Under **Triggers**, enter `v:this.lineIndex` and `dp:MyStringList`.

Step 5.6
Enter the following **On trigger** script:

```
function(v:arg0::bool)
    {
    v:this->MyLabel.text=dp:MyStringList[v:this.lineIndex];
    false
    }
```

You changed the appearance of the rectangle. With the `setText` property, the labels of `MyStringList` are filled automatically with the content of `MyStringList`.

Testing the EB GUIDE model

Prerequisite:

- You completed the previous instruction.

Step 1
To start the simulation, click ▷ in the command area.

Result:

Since `MyStringList` contains five datapool items, five rectangles that are labeled from one to five are displayed in vertical arrangement.

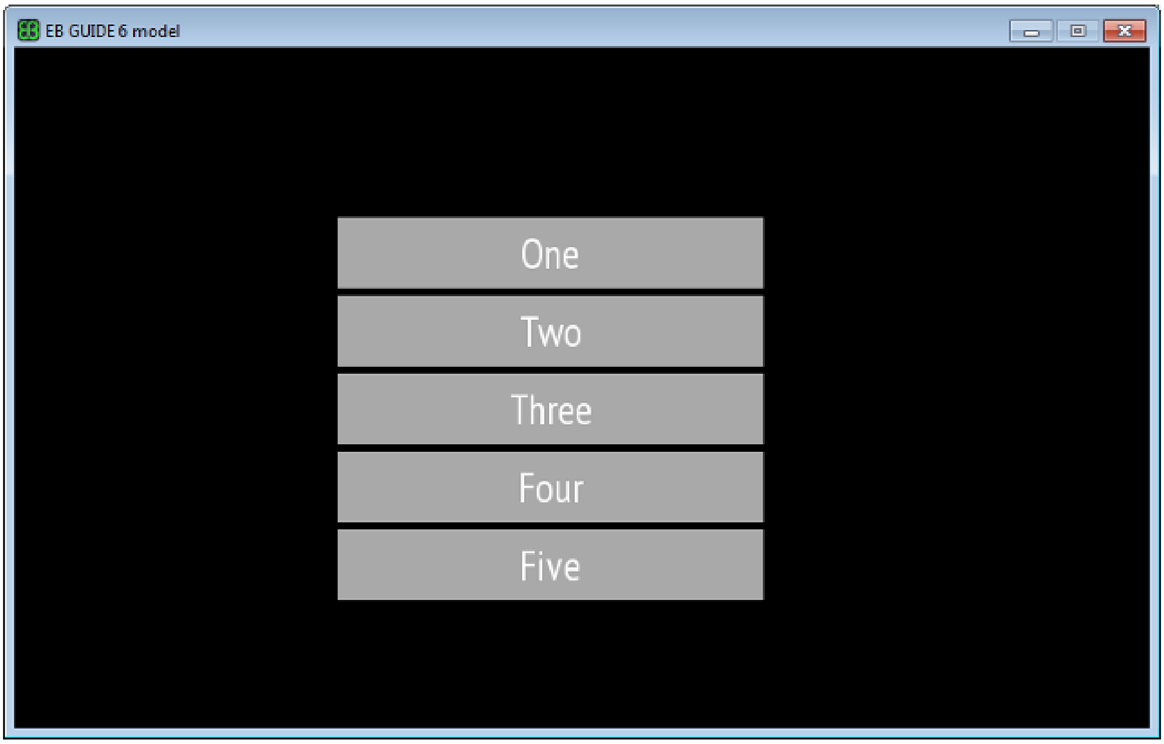


Figure 12.6. List created with an instantiator

12.5. Tutorial: Making an ellipse move across the screen

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

The following instructions guide you through the process of animating an ellipse so that it continually moves across the screen when the simulation starts.

Approximate duration: Five minutes.

Adding widgets

In the following steps, you add three widgets to the view and organize the hierarchy of the widgets.

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state

Step 1
In the content area, double-click the view state.

The view is displayed in the content area.

Step 2
Drag an ellipse from the **Toolbox** into the view.

Step 3
Drag an animation from the **Toolbox** into the ellipse.

Step 4
In the **Navigation** component, click the animation, and press the **F2** key. Rename the animation to `MyAnimation`.

Now, if you start the simulation, an ellipse is displayed in a view. The ellipse does not move yet.

Adding a user-defined property of type `Conditional script`

As a next step, you add a user-defined property to the ellipse. With the conditional script property, rendering the ellipse during simulation starts the animation.

Prerequisite:

- You completed the previous instruction.

Step 1
Select the ellipse.

Step 2
In the **Properties** component, go to the **User-defined properties** category, and click +.

A menu expands.

Step 3
In the menu, click `Conditional script`.

A user-defined property of type `Conditional script` is added to the ellipse.

Step 4
Rename the property to `startAnimation`.

Step 5
Next to the `startAnimation` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 6
Enter the following EB GUIDE Script:

```
function(v:arg0::bool)
    {
 f:animation_play(v:this->^->MyAnimation)
    }
```

Making the animation visible

The following instructions guide you through the process of making the animation visible.

Prerequisite:

- You completed the previous instruction.
- The content area displays the `View 1` view.

Step 1
Go to the **Animation editor**. Next to **Animation properties**, click + and select `View 1`.

A menu expands.

Step 2
Under `Ellipse 1` select the `x` property and then the **Linear interpolation curve**.

Step 3
Click **Accept**.

The button is displayed next to the `target` property.

Step 4
Link the `end` property to the view's `width` property.

With these settings, when the animation starts, the `x` property of the ellipse changes from zero to the width of the view. Thus the ellipse moves from the left boundary to the right boundary of the view.

Step 5
To make the animation run in infinite repetitions, enter `0` in the `repeat` property.

Step 6
Save the project.

Step 7
To start the simulation, click ▷ in the command area.

Result:

The ellipse continually moves from the left side of the view to the right side of the view.

12.6. Tutorial: Adding a language-dependent text to a datapool item

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

EB GUIDE offers the possibility to display texts in the user's preferred language. The following instructions show you how to model a label that changes with an English, French, and German user interface.

Approximate duration: 15 minutes

NOTE

Prerequisites to language dependency

To add language support to a datapool item, do the following:

- If its `Value` property is linked to another datapool item or widget property, remove the link.
- If its `Value` property is a scripted value, convert the property to a plain value.

Linking a widget property to a datapool item

The following instructions guide you through the process of linking the label's `text` property to a datapool item. In run-time the displayed text is provided by the datapool item.

Prerequisite:

- Three languages are added to the EB GUIDE model: English, German, and French.The name of **Language 1** is set to `German` and the name of **Language 2** is set to `French`.
- The Main state machine contains an initial state and a view state.
- The initial state has a transition to the view state.
- The content area displays the view.
- The view state contains a label.

Step 1

Click the label.

Step 2
In the **Properties** component, go to the `text` property, and click the button next to the property.

Step 3
In the menu, click **Add link to datapool item**.

A dialog opens.

Step 4
To add a new datapool item, enter `Welcome_text` in the text box.

Step 5
Click **Add datapool item**.

Step 6
Click **Accept**.

The datapool item `Welcome_text` is added.

In the content area, the label no longer displays any text.

Enter language-dependent text to the datapool item

The following instructions guide you through the process of adding language-dependent text to the datapool item. For every language the `Value` property has a different text.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Datapool** component, click the `Welcome_text` datapool item.

Step 2
Click the button.

Step 3
In the menu, click **Add language support**.

In the **Properties** component, the language properties are displayed.

Step 4
In the **Datapool** component, in the `Value` text box, enter `Welcome`.

In the content area, the label displays `Welcome`.

Step 5
Go to the **Properties** component.

Step 6
In the `German` text box, enter `Willkommen`.

In the `Language` box in the upper left corner, change the language to `German`.

In the content area, the label displays `Willkommen`.

Step 7
In the `French` text box, enter `Bienvenue`.

In the `Language` box in the upper left corner, change the language to `French`.

In the content area, the label displays `Bienvenue`.

You have added language support for English, German and French and defined a language-dependent text label.

Changing the language during run-time

The following instructions guide you through the process of creating a script for changing the language during run-time. Each time, the user clicks the label, the display language changes.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Datapool** component, click +.

A menu expands.

Step 2
In the menu, click `Integer`.

A datapool item of type `Integer` is added.

Step 3
Rename the datapool item to `SelectedLanguage`.

Step 4
In the **Navigation** component, click the `Label 1` label.

Step 5
In the **Properties** component, go to the **Widget feature properties** and click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 6
Under **Available widget features**, expand the **Input handling** category, and select the **Touch pressed** widget feature.

Step 7
Click **Accept**.

The related widget feature properties are added to the label and displayed in the **Properties** component.

Step 8

Next to the `touchPressed` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 9

Replace the existing EB GUIDE Script with the following code:

```
function(v:touchId::int, v:x::int, v:y::int, v:fingerId::int)
{
    if (dp:SelectedLanguage == 0)  // Standard selected
    {
     f:language(l:German)
     dp:SelectedLanguage = 1
    }
    else if (dp:SelectedLanguage == 1)  // German selected
    {
     f:language(l:French)
     dp:SelectedLanguage = 2
    }
    else if (dp:SelectedLanguage == 2)  // French selected
    {
     f:language(l:Standard)
     dp:SelectedLanguage = 0
    }
    false
}
```

Step 10

Click **Accept**.

You configured the label and wrote an EB GUIDE Script which changes the language of the label during run-time.

Result:

You added a datapool item of type `String` to the EB GUIDE model. The datapool item has different values for languages. In English the value is `Welcome`. In German the value is `Willkommen`. In French the value is `Bienvenue`. The datapool item is linked to the `text` property of the label. Every time you change the language of the EB GUIDE model the text of the label changes too.

12.7. Tutorial: Working with a 3D graphic

NOTE

Default window layout

All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

EB GUIDE Studio offers the possibility to use 3D graphics in your EB GUIDE model.

The following instructions guide you through the process of adding a 3D graphic to your EB GUIDE model. The instructions show you how to import a 3D graphic and how to modify the appearance of the imported 3D graphic using widget features. For best results, work through the following steps in order presented.

NOTE

3D graphic

To create a 3D graphic file, use third-party 3D modeling software.

Only the renderers for OpenGL ES 2.0 or higher can display 3D graphics. Make sure that your graphics driver is compatible to the version of the renderer. The supported 3D graphic formats are COLLADA (`.dae`) and Filmbox (`.fbx`). For best results, use the Filmbox format.

To be able to apply textures to a mesh, a 3D object needs to have texture coordinates. To add texture coordinates, use third-party 3D modeling software.

Approximate duration: 15 minutes.

Importing a 3D graphic

The following instructions guide you through the process of importing a 3D graphic file to an EB GUIDE project.

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state.
- A 3D graphic file is available. The file contains a camera, a light source, and one object containing a mesh and at least one material.

Step 1

In the content area, double-click the view state.

The view is displayed in the content area.

Step 2
Drag a scene graph from the **Toolbox** into the view.

The view displays the empty bounding box.

Step 3
Rename the scene graph to `My3DGraphic`.

Step 4
In the **Properties** component, click **Import file**.

A dialog opens.

Step 5
Navigate to the directory where the 3D graphic file is stored.

Step 6
Select the 3D graphic file.

Step 7
Click **Open**.

The import starts. The **Import successful** dialog is displayed. Here you have the possibility to check the import log file.

Step 8
Click **OK**.

The view displays the 3D graphic. The **Navigation** component displays the imported widget tree with the scene graph as a parent node. `My3DGraphic` contains a `RootNode` that has at least one mesh with material, camera and several other child widgets depending on the content of your 3D graphic file.

Adding widgets

The following instructions guide you through the process of adding an additional light source to your 3D graphic.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, expand `RootNode`.

Step 2
Drag a directional light from the **Toolbox** to `RootNode`.

You added a directional light to `My3DGraphic`. You can manipulate and transform this directional light with the transformation properties of the `RootNode`.

Step 3
To add the light source and place it with default widget properties different from the `RootNode` scene graph, do the following:

Step 3.1
Drag a scene graph node from the **Toolbox** to `RootNode`.

Step 3.2
Rename the scene graph node to `MyLight`.

Step 3.3
Drag a directional light from the **Toolbox** to `MyLight`.

You added a directional light to `My3DGraphic`. To change the placing of the directional light, change the properties of `MyLight`.

Changing meshes

Prerequisite:

- You completed the previous instruction.
- The `$GUIDE_PROJECT_PATH/<project name>/resources/<3D graphic name>` directory contains an additional `.ebmesh` file.

Step 1
In the **Navigation** component, click `Mesh 1`, and go to the **Properties** component.

Step 2
From the `mesh` combo box select the `.ebmesh` file from the resource folder mentioned above.

The view displays the scene graph with the new mesh.

Step 3
Alternatively, drag an `.ebmesh` file from the **Assets** component into the `mesh` drop-down list box.

The view displays the scene graph with the new mesh.

Changing textures

The following instructions guide you through the process of adding and modifying textures of your 3D graphic.

Prerequisite:

- You completed the previous instruction.

- The `$GUIDE_PROJECT_PATH/<project name>/resources/<3D graphic name>` directory contains a `.png` or `.jpg` image file.

Step 1
In the **Navigation** component, click the material, and go to the **Properties** component.

Step 2
In the **Widget feature properties** category, click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 3
Under **Available widget features**, expand the **3D** category, and select a texture widget feature, for example **Diffuse texture**.

Step 4
Click **Accept**.

The related widget feature properties are added to the material and displayed in the **Properties** component.

Step 5
In the **Properties** component, select an image from the `diffuseTexture` combo box.

The view displays a scene graph with the new texture.

NOTE

Usage of 3D widget features

These instructions are valid for the following widget features from the category **3D**:

- **Ambient texture**
- **Diffuse texture**
- **Emissive texture**
- **Light map texture**
- **Normal map texture**
- **Opaque texture**
- **Reflection texture**
- **Specular texture**

Displaying 3D object several times

The following instructions guide you through the process of adding an additional camera to be able to display the 3D object of your 3D graphic several times. You will be able to have different points of view of the same object.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, click `My3DGraphic` and go to the **Properties** component.

Step 2
Enter `800` in the `width` text box and `480` in the `height` text box.

The `My3DGraphic` scene graph has the size of the view.

Step 3
In the **Navigation** component, expand `RootNode` and `Camera001`.

Step 4
Click `Camera 1` and go to the **Properties** component.

Step 5
In the **Widget feature properties** category, click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 6
Under **Available widget features**, expand the **3D** category, and select **Camera viewport**.

Step 7
Click **Accept**.

The related widget feature properties are added to `Camera 1` and displayed in the **Properties** component.

Step 8
Drag a camera from the **Toolbox** to the scene graph node `Camera001`.

You added a second camera.

Step 9
Click `Camera 2` and go to the **Properties** component.

Step 10
In the `nearPlane`, `farPlane` and `fieldOfView` text boxes enter the same values that `Camera 1` has.

Both `Camera 1` and `Camera 2` have the same viewing position.

Step 11
In the **Widget feature properties** category, click **Add/Remove**.

The **Widget features** dialog is displayed.

Step 12
Under **Available widget features**, expand the **3D** category, and select **Camera viewport**.

Step 13
Click **Accept**.

The related widget feature properties are added to `Camera 2` and displayed in the **Properties** component.

Step 14
In the **Properties** component, enter `100` in `viewportX` and `viewportY` text boxes.

In the view, the 3D object is displayed two times with different x-coordinate and y-coordinate.

12.8. Tutorial: Rendering gamma correctly

NOTE **Default window layout**
All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

In EB GUIDE Studio you can perform gamma correction for the following:

- Display
- Image
- Texture

The following instructions show you how to configure gamma correction. For best results, work through the steps in order presented.

Approximate duration: 15 minutes.

Configuring gamma encoding for displays

The following steps guide you through the process of setting up EB GUIDE Studio to output gamma-encoded values to the display.

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state.

Step 1
Click .

The project center opens.

Step 2
In the navigation area, click **Configure** > **Profiles**.

Step 3
For each profile that is sRGB-aware configure the following:

Step 3.1
In the content area, click the **Scenes** tab.

Step 3.2
From the `colorMode` drop-down list box, select the following:

- If your rendering pipeline uses the hardware sRGB support of the GPU, select `32-bit sRGB (=4)`
- If your rendering hardware does not support sRGB, select `32-bit sRGB (=5)`.

To apply changes in the edit mode, restart EB GUIDE Studio.

NOTE

Rendering hardware

The OpenGL 3 renderer always has hardware sRGB support. The OpenGL renderer, which uses the OpenGL ES 2.0 API, uses only hardware sRGB support if the hardware supports it through the appropriate OpenGL ES extensions. This is automatically detected. In case your OpenGL ES 2.0 hardware does not support sRGB, the renderer automatically falls back to `32-bit sRGB (Emulated)(=5)`, which uses fragment shaders for the conversion. Note that hardware sRGB mode does not work on all systems, even if they support OpenGL ES 3.0. In that case, switch to `32-bit sRGB (Emulated)(=5)`.

Configuring gamma encoding for images

Prerequisite:

- You completed the previous instruction.
- An image file is available in the resource directory.

Step 1
Go to the project editor and double-click the view. Drag an image from the **Toolbox** into the view.

Step 2
In the **Properties** component, from the `image` drop-down box, select an image file.

The image probably appears too bright.

This is caused by the sRGB color mode that was configured in the scene properties and now applies gamma-correction to an already gamma-corrected image.

Step 3
To configure the image to be gamma encoded, in the **Properties** component select `sRGB`.

The image is now displayed and processed correctly in blending operations.

Configuring gamma encoding for textures

Prerequisite:

- You completed the previous instructions.
- A 3D file, for example an `.fbx` file, with at least one textured 3D object is available with a diffuse texture.

Step 1
Drag a scene graph from the **Toolbox** into the view.

Step 2
In the **Properties** component, click **Import file** and select a 3D file.

Step 3
In the **Navigation** component locate the imported scene graph. Within the scene graph structure, select a material widget that uses a diffuse texture.

Step 4
In the **Properties** component, in the **Widget feature properties**, click **Add/Remove**.

Step 5
From the **3D** category, select **Diffuse texture**.

Step 6
Select the `diffuseSRGB` property.

The texture is treated as gamma-encoded image and is linearized before it is being used in lighting computations.

12.9. Tutorial: Using view transition animations

NOTE

Default window layout
All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

View transition animation (VTA) is an animation that is done while moving from one view to another. The following instructions guide you through the process of creating these animations. You are going to create a model with views and animations that are played when you change views. You are going to create the following elements:

- Two view states
- Navigation elements, such as buttons and labels
- Events to trigger changes to the view states

- Animations that are played when you transition to another state

Approximate duration: 30 minutes.

Creating the first view state

Create the first view and a button.

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state and a view state.
- The initial state has a transition to the view state

Step 1
In the **Navigation** component, rename the view state to `FirstState` and the view to `FirstView`.

Step 2
Open the `FirstView`.

Step 3
From the **Toolbox** component, drag a rectangle into the `FirstView` and rename it to `RectNextView`.

This rectangle is for the button that triggers the transition.

Step 4
In the **Properties** component, go to the **Widget feature properties** category and click **Add/Remove**. The **Widget features** dialog is displayed.

Step 5
Under **Available widget features**, expand the **Effect** category and select **Border**.

Step 6
Click **Accept**.

The **Properties** component displays the related widget feature properties.

Step 7
In the **Properties** component, enter the following:

- In the `width` text box, enter `220`.
- In the `height` text box, enter `70`.
- In the `x` text box, enter `290`.
- In the `y` text box, enter `150`.
- Set `fillColor` to black.
- In the `borderThickness` text box, enter `2`.
- Set `borderColor` to white.

Step 8
From the **Toolbox** component, drag a label into the **Navigation** component and add it as a child widget of `FirstView`.

Step 9
Rename the label to `LabelNextView`.

Step 10
In the **Properties** component, enter the following:

- In the `text` text box, enter `Go to the next view`.
- In the `font` text box, enter `25`.
- Set the `horizontalAlign` to centered.

Step 11
Link the dimensions of the label to the dimensions of the rectangle. Link the following properties:

- Link the `width` property of `LabelNextView` to `width` of `RectNextView`.
- Link the `height` property of `LabelNextView` to `height` of `RectNextView`.
- Link the `x` property of `LabelNextView` to `x` of `RectNextView`.
- Link the `y` property of `LabelNextView` to `y` of `RectNextView`.

Figure 12.7. The `FirstView` with the button

Creating the second view state

The second view contains a button. Create this view by copying and renaming the elements you already created.

Prerequisite:

- You completed the previous instruction.

Step 1
Select the **Main** tab.

Step 2
Copy and paste the `FirstState` state.

Step 3
In the **Navigation** component, find the new state you created and rename the following widgets:

- Rename the state to `SecondState`.
- Rename the view to `SecondView`.
- Rename `RectNextView` to `RectGoBack`.
- Rename `LabelNextView` to `LabelGoBack`.

Step 4
Double-click `LabelGoBack`, and in the `text` text box, enter `Go back`.

Figure 12.8. The `SecondView` with the button

Creating transitions and events

Prerequisite:

- You completed the previous instruction.

Step 1
Double-click the **Main** state machine.

Step 2
Create transitions from the edges of the **Main** state machine to both view states.

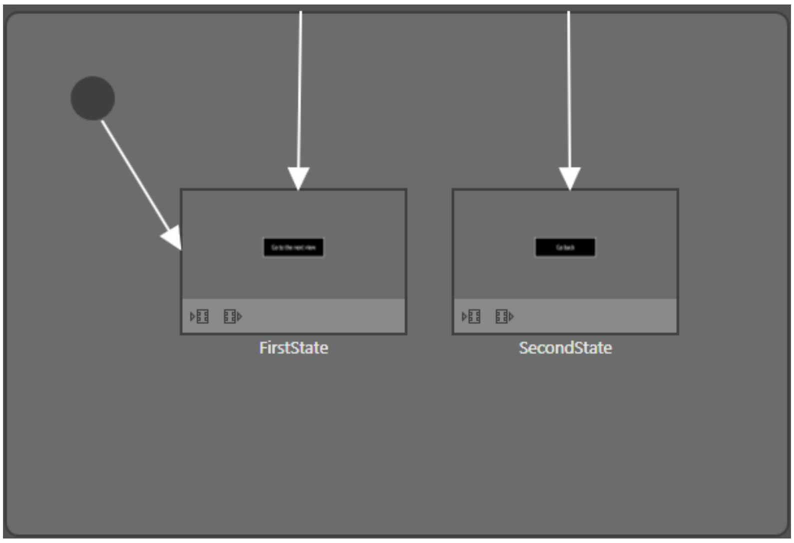


Figure 12.9. The `Main` state machine with transitions

Step 3
Select the transition from the **Main** state machine to `FirstState`.

Step 4
In the **Properties** component, in the **Trigger** combo box, enter `goToFirstState` and click **Add event**.

A new event is created.

Step 5
Select the transition to `SecondState`.

Step 6
In the **Properties** component, in the **Trigger** combo box, enter `goToSecondState` and click **Add event**.

A new event is created.

Connecting buttons and events

Now you define the following behavior: when a button is clicked, the transition to another state is triggered. For this you use EB GUIDE Script.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component, double-click `LabelNextView`.

Step 1.1
In the **Properties** component, go to the **Widget feature properties** category and click **Add/Remove**. The **Widget features** dialog is displayed.

Step 1.2
Under **Available widget features**, expand the **Input handling** category and select **Touch released**.

Step 1.3
Click **Accept**.

The related widget feature properties are added to the **Properties** component.

Step 1.4
Next to the `touchShortReleased` property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 1.5
Enter the following EB GUIDE Script that fires the `goToSecondState` event when the button is clicked:

```
function(v:touchId::int, v:x::int, v:y::int, v:fingerId::int)
{
    fire ev:goToSecondState()
    false
}
```

Step 1.6
Click **Accept**.

Step 2
In the **Navigation** component, double-click `LabelGoBack`.

Step 2.1
In the **Properties** component, go to the **Widget feature properties** category and click **Add/Remove**. The **Widget features** dialog is displayed.

Step 2.2
Under **Available widget features**, expand the **Input handling** category and select **Touch released**.

Step 2.3
Click **Accept**.

The related widget feature properties are added to the **Properties** component.

Step 2.4
Next to the **touchShortReleased** property, select the **Value** column and click {} .

An EB GUIDE Script editor opens.

Step 2.5
Enter the following EB GUIDE Script that fires the `goToFirstState` event when the button is clicked:

```
function(v:touchId::int, v:x::int, v:y::int, v:fingerId::int)
{
    fire ev:goToFirstState()
```

```
    false
}
```

Step 2.6
Click **Accept**.

Creating the `FirstView` entry animation

Enable the **VTA** component and create an animation that moves the button in from the right. To create an animation, you need to define which property is animated, how long the animation lasts, where it starts, and where it ends. In this tutorial only the `x` property is used.

Prerequisite:

- You completed the previous instruction.

Step 1
The **VTA** component is not visible in the default layout. You need to enable it.

In the command area click **Layout** > **VTA**.

The **VTA** component is displayed.

Step 2
In the **Navigation** component double-click `FirstState`.

Step 3
In the **VTA** component click + and select **Entry animation**

The **Entry animation** table is displayed.

The **Animation editor** is displayed below the content area.

Step 4
Click + and select **Exit animation**.

The **Exit animation** table is displayed.

Step 5
In the **Animation editor** in the drop-down list select `Entry animation 1`.

Step 6
Click + and select **Destination:** `FirstView`. The **Animation properties** dialog is displayed.

Step 7
Click `RectNextView`, then `x` then **Fast start curve** and **Accept**.

A new animation is added to the **Animated properties** list.

Step 8
In the **Properties** component enter the following:

- In the `start` text box, enter `900`.
- In the `end` text box, enter `290`.

When you start the simulation you can see the button move in.

Creating the `FirstView` exit animation

Create an animation that moves the button out to the right.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Animation editor** in the drop-down list select `Exit animation 1`.

Step 2
Click + and select **Source:** `FirstView`. The **Animation properties** dialog is displayed.

Step 3
Click `RectNextView` then `x` then **Fast start curve** and **Accept**.

A new animation is added to the **Animated properties** list.

Step 4
In the **Properties** component enter the following:

- In the `duration` text box, enter `500`.
- In the `start` text box, enter `290`.
- In the `end` text box, enter `800`.

Creating the `SecondView` entry animation

Create an animation that moves the button in from the right.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Navigation** component double-click `SecondView`.

`SecondView` is displayed in the content area.

Step 2
In the **VTA** tab click +

Step 3
Add an **Entry animation** and an **Exit animation**.

The **Animation editor** is displayed below the content area.

Step 4
In the **Animation editor**, in the drop-down list, select `Entry animation 2`.

Step 5
Click + and select **Destination:** `SecondView`. The **Animation properties** dialog is displayed.

Step 6
Click `RectGoBack` then `x` then **Fast start curve** and **Accept**.

A new animation is added to the **Animated properties** list.

Step 7
In the **Properties** component enter the following:

- In the `start` text box, enter `900`.
- In the `end` text box, enter `290`.

Creating the `SecondView` exit animation

Create an animation that moves the button out to the left.

Prerequisite:

- You completed the previous instruction.

Step 1
In the **Animation editor**, in the drop-down list, select `Exit animation 2`.

Step 2
Click + and select **Source:** `SecondView`. The **Animation properties** dialog is displayed.

Step 3
Click `RectGoBack` then `x` then **Fast start curve** and **Accept**.

A new animation is added to the **Animated properties** list.

Step 4
In the **Properties** component enter the following:

- In the `duration` text box, enter `500`.
- In the `start` text box, enter `290`.
- In the `end` text box, enter `800`.

Saving and testing the EB GUIDE model

Prerequisite:

- You completed the previous instruction.

Step 1
To save the project, click in the command area.

Step 2
To start the simulation, click ▷ in the command area.

Click `Go to the next view`. The view changes and an animation is played.

Click `Go back`. The view changes back to the first view and an animation is played.

12.10. Tutorial: Using script curves for animations

NOTE

Default window layout
All instructions and screenshots use the default window layout. If you want to follow the instructions, we recommend to set the EB GUIDE Studio window to default layout by selecting **Layout** > **Reset to default layout**.

Use a script curve when you want to define your own curve for an animation. Defining your own curve can be necessary when the other animation curves are not suitable or when you just want to define a custom curve. In this tutorial you are going to create a simple model with two script curves for two animations. You are going to create the following elements:

- A view state
- Two rectangle widgets
- Two animation widgets with script curves that animate the positions of the rectangle widgets

This results in a model with two rectangles. One rectangle moves down. The other moves to the side.

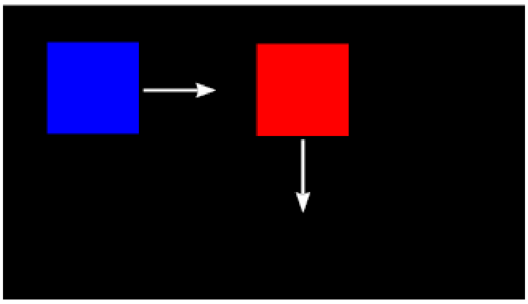


Figure 12.10. The rectangles with their movement direction

Approximate duration: 15 minutes.

Creating the first script curve

Prerequisite:

- The content area displays the **Main** state machine.
- The **Main** state machine contains an initial state and a view state called `FirstView`.
- The initial state has a transition to `FirstView`

Step 1
From the **Toolbox** component, drag a rectangle into the view and rename it to `BlueRectangle`

Step 2
In the **Properties** component, set the `fillColor` to blue.

Step 3
From the **Toolbox** component, drag an animation into the view and rename it to `MoveAnimation`.

Step 4
In the **Datapool** component, add a datapool item of type `Float` and rename it to `xFloat`.

Step 5
In the **Navigation** component, select `BlueRectangle`.

Step 6
In the **Properties** component, go to the **User-defined properties** category, and click +.

A menu opens.

Step 7
In the menu, select **Conditional script**.

`Conditional script 1` is added to the **User-defined properties**.

Step 8
Rename `Conditional script 1` to `StartBlueAnimation`.

Step 9

Next to `StartBlueAnimation`, click {} .

The EB GUIDE Script editor opens.

Step 10

Enter the following script:

```
function(v:arg0::bool)
{
    f:animation_play(v:this->^->"MoveAnimation")
}
```

Step 11

In the **Navigation** component, select `FirstView`.

Step 12

In the **Animation editor**, next to the **Animated properties** click + and select `FirstView`.

The **Animation properties** dialog opens.

Step 13

Under `BlueRectangle`, select the `x` property and then the `Script curve`

Step 14

Click **Accept**.

`Script curve 1` is added to the **Animation editor**.

Step 15

Rename `Script curve 1` to `BlueCurve`.

Step 16

In the **Properties** component, next to the `curve` property click {} .

The EB GUIDE Script editor opens.

Step 17

Enter the following script:

```
function(v:diff::int, v:t_anim::int)
{
 dp:xFloat+=0.2
 f:floor(dp:xFloat*dp:xFloat)
}
```

Creating the second script curve

Prerequisite:

- You have finished the previous instruction.

Step 1
From the **Toolbox** component, drag a rectangle into **FirstView** and rename it to `RedRectangle`.

Step 2
In the **Properties** component, set the `fillColor` to red.

Step 3
In the **Datapool** component, add a datapool item of type `Integer` and rename it to `1_diff`.

Step 4
Add another datapool item of type `Integer` and rename it to `2t_anim`.

Step 5
Select `RedRectangle`.

Step 6
In the **Properties** component, go to the **User-defined properties** category, and click +.

`Conditional script 2` is added.

Step 7
Rename `Conditional script 2` to `StartRedAnimation`.

Step 8
Next to `StartRedAnimation` click {} .

The EB GUIDE Script editor opens.

Step 9
Enter the following script:

```
function(v:arg0::bool)
{
    f:animation_play(v:this->^->"MoveAnimation")
}
```

Step 10
In the **Navigation** component, select `FirstView`.

Step 11
In the **Animation editor**, next to the **Animated properties** click + and select `FirstView`.

The **Animation properties** dialog opens.

Step 12
Under `RedRectangle`, select the `y` property and then the `Script curve`.

Step 13
Click **Accept**.

`Script curve 2` is added to the **Animation editor**.

Step 14
Rename `Script curve 2` to `RedCurve`.

Step 15

In the **Properties** component, next to the `curve` property, click {}.

The EB GUIDE Script editor opens.

Step 16

Enter the following script:

```
function(v:diff::int, v:t_anim::int)
{
 dp:"1_diff"=v:diff
 dp:"2t_anim"=v:t_anim
    v:t_anim/2::int
}
```

Saving and testing the EB GUIDE model

Prerequisite:

- You completed the previous instruction.

Step 1

To save the project, click in the command area.

Step 2

To start the simulation, click ▷ in the command area.

The animation is played at the start of the simulation.

13. References

The following chapter provides you with lists and tables for example parameters, properties, and identifiers.

13.1. Android events

Android events belong to the `SystemNotifications` event group and have event group ID 13.

Table 13.1. Android events

Event ID	Name	Description
1	`RendererEnabled`	Is sent by the application when Android life cycle management stops or starts the renderer Parameters: ▶ `enabled`: If true, the renderer is enabled. If false, the renderer is set to sleep mode.
2	`setKeyboardVisibility`	Is sent by the EB GUIDE model if a virtual keyboard is intended to be shown Parameters: ▶ `visibility`: If true, a virtual keyboard is made visible. If false, it is invisible.
3	`onKeyboardVisibilityChanged`	Is sent by the application if a virtual keyboard is shown Parameters: ▶ `visibility`: If true, a virtual keyboard is visible. If false, it is invisible.
4	`onLayoutChanged`	Is sent by the application when the visible area of the screen changes Parameters (in pixels): ▶ `x`: The x-coordinate of the top left corner of the visible screen area

Event ID	Name	Description
		▶ `y`: The y-coordinate of the top left corner of the visible screen area ▶ `width`: The width of the visible screen area ▶ `height`: The height of the visible screen area

13.2. Datapool items

Table 13.2. Properties of a datapool item

Property name	Description
`Value`	The initial value of the datapool item

13.3. Data types

The following section describes data types in EB GUIDE. You can add user-defined properties and datapool items from the types listed below.

13.3.1. Boolean

Boolean properties can have the values true and false.

Available operations are as follows:

- equal (==)
- not equal (!=)
- negation (!)
- and (&&)
- or (||)
- assign (writable properties) (=)

It is possible to store boolean properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.2. Color

Colors are stored in the RGBA8888 format.

Example: Red without transparency is (255, 0, 0, 255).

Available operations are as follows:

- equal (==)
- not equal (!=)
- assign (writable properties) (=)

It is possible to store color properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.3. Conditional script

Conditional scripts are used to react on initialization and on trigger. When you edit conditional scripts, the content area is divided into the following sections:

- In the **Trigger** section, you can add an event, datapool item, or widget property that triggers the execution of the **On trigger** script.
- In the **On trigger** section, you can add an EB GUIDE Script that is called on initialization, an event trigger, or after a value update of a datapool item or a widget property.

  The parameter of the **On trigger** EB GUIDE Script indicates the cause for the execution of the script.

  The `arg0` refers to the fact whether the EB GUIDE Script is executed during initialization or by a trigger. Consider the following:

  - If the EB GUIDE Script is executed during initialization, `arg0` is true.
  - If the EB GUIDE Script is executed by a trigger, `arg0` is false.

  The return value of the **On trigger** EB GUIDE Script controls change notifications for the property.

  The return value of the **On trigger** EB GUIDE Script regulates whether the EB GUIDE Script must produce a notification or not. Consider the following:

  - If the return value is true, a notification is generated.

- If the return value is false, a notification is not generated.

To be able to execute the **On trigger** script, the conditions are to be fulfilled during the following:

- On initialization, for example, in case of datapool items during EB GUIDE model startup, or in case of widget properties during the view creation.
- On processing an event from the trigger script. The EB GUIDE Script is executed once for each matching event.
- On processing the datapool notifications of one or more items from the trigger script. Multiple notifications may be processed at once.
- On processing the notifications of one or more widget properties from the trigger script. Multiple notifications may be processed at once.

13.3.4. Float

Float-point number data type represents a single-precision 32-bit IEEE 754 value.

Available operations are as follows:

- equal (==)
- not equal (!=)
- greater (>)
- greater or equal (>=)
- less (<)
- less or equal (<=)
- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)
- assign (writable properties) (=)

It is possible to store float properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.5. Font

To add a font to an EB GUIDE project, copy the font file to the following directory: `$GUIDE_PROJECT_PATH/<project name>/resources`

Available operations are as follows:

- assign (writable properties) (=)

It is possible to store font properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.6. `Function () : bool`

By means of `Function () : bool` you can create an own function.

The available operation for this data type is a read/run operation for all properties.

13.3.7. Ibl

Ibl is a data format that stores lighting information generated by the IBLGenerator.

To add an ibl to an EB GUIDE project, copy the `.ebibl` file to the following directory: `$GUIDE_PROJECT_PATH/<project name>/resources`

Available operations are as follows:

- assign (writable properties) (=)

It is possible to store ibl properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.8. Image

To add an image to an EB GUIDE project, copy the image file to the following directory: `$GUIDE_PROJECT_PATH/<project name>/resources`

Available operations are as follows:

- assign (writable properties) (=)

It is possible to store image properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.9. Integer

EB GUIDE supports signed 32-bit integers.

Available operations are as follows:

- equal (==)
- not equal (!=)
- greater (>)
- greater or equal (>=)
- less (<)
- less or equal (<=)
- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)
- modulo (%)
- assign (writable properties) (=)

It is possible to store integer properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.10. Mesh

Mesh defines the shape of the 3D object.

To add a mesh to an EB GUIDE project, copy the `.ebmesh` file to the following directory: `$GUIDE_PROJECT_PATH/<project name>/resources`

Available operations are as follows:

- assign (writable properties) (=)

It is possible to store mesh properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.11. String

EB GUIDE supports character strings, for example *Hello world*.

Available operations are as follows:

- equal (case sensitive) (==)
- not equal (case sensitive) (!=)
- equal (case insensitive, only in the ASCII range) (=Aa=)
- greater (>)
- greater or equal (>=)
- less (<)
- less or equal (<=)
- concatenation (+)
- assign (writable properties) (=)

It is possible to store string properties in a list. For details about lists, see section 13.3.12, “List”.

13.3.12. List

EB GUIDE supports a list of values with the same data type.

The following list types are available:

- Boolean list
- Color list
- Float list
- Font list
- Ibl list
- Image list
- Integer list
- Mesh list
- String list

The following types cannot be used in lists:

- List
- Property reference
- List element reference

Available operations are as follows:

- length: (length)
- element accessor: ([])

13.4. EB GUIDE Script

13.4.1. EB GUIDE Script keywords

The following is a list of reserved keywords in EB GUIDE Script. If you want to use these words as identifiers in a script, you must quote them.

Keyword	Description
`cancel_fire`	Cancels an event that is fired with `fire_delayed`.
`color:`	A color parameter follows, for example {0,255,255}.
`dp:`	A datapool item follows.
`l:`	A language follows. Is used on `f:language`.
`else`	An `if` condition is completed. The following block is executed as an alternative.
`ev:`	An event follows.
`f:`	A user-defined function follows.
`false`	A boolean literal value
`fire`	Fires an event
`fire_delayed`	Fires an event after a specified time. The time is specified in milliseconds.
`if`	A statement which tests a boolean expression follows. If the expression is true, the statement is executed.
`in`	Is a separator between a local variable declaration and the variable's scope of usage Is used with `match_event` and `let`.
`function`	Declares a function
`length`	The length of a property
`let`	Declares a local variable that is accessible in the scope
`list`	Declares a list type, for example an integer list
`match_event`	Checks if the current event corresponds to an expected event and declares variables like `let`

Keyword	Description
`popup_stack`	The dynamic state machine list which defines the priority of dynamic state machines
`s:`	A skin follows. Is used on `f:skin`.
`sm:`	A state machine follows
`true`	A boolean literal value
`unit`	A value of type void
`v:`	A local variable follows
`while`	Repeats a statement as long as the condition is true

13.4.2. EB GUIDE Script operator precedence

The following is a list of the operators in EB GUIDE Script together with their precedence and associativity. Operators are listed top to bottom, in descending precedence.

Table 13.3. EB GUIDE Script operator precedence

Operator	Associativity
(()), ({})	none
([])	none
(->)	left
(.)	none
(::)	left
length	none
(&)	right
(!), (-) unary minus	right
(*), (/), (%)	left
(+), (-)	left
(<), (>), (<=), (>=)	left
(!=), (==), (=Aa=)	left
(&&)	left
(||)	left
(=), (+=), (-=), (=>)	right
(,)	right

Operator	Associativity
(;)	left

13.4.3. EB GUIDE Script standard library

The following chapter provides a description of all EB GUIDE Script functions.

13.4.3.1. EB GUIDE Script functions A - B

13.4.3.1.1. `abs`

The function returns the absolute value of the integer number `x`.

Table 13.4. Parameters of `abs`

Parameter	Type	Description
`x`	integer	The number to return the absolute value from
`<return>`	integer	The return value

13.4.3.1.2. `absf`

The function returns the absolute value of the float number `x`.

Table 13.5. Parameters of `absf`

Parameter	Type	Description
`x`	float	The number to return the absolute value from
`<return>`	float	The return value

13.4.3.1.3. `acosf`

The function returns the principal value of the arc cosine of `x`.

Table 13.6. Parameters of `acosf`

Parameter	Type	Description
`x`	float	The number to return the arc cosine from
`<return>`	float	The return value

13.4.3.1.4. `animation_before`

The function checks if a running animation has passed a given point in time.

Table 13.7. Parameters of `animation_before`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`time`	integer	The point in time
`<return>`	boolean	If true, the animation has not yet passed the point in time.

13.4.3.1.5. `animation_beyond`

The function checks if a running animation has passed a given point in time.

Table 13.8. Parameters of `animation_beyond`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`time`	integer	The point in time
`<return>`	boolean	If true, the animation has passed the point in time.

13.4.3.1.6. `animation_cancel`

The function cancels an animation and leaves edited properties in the current state.

Table 13.9. Parameters of `animation_cancel`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

13.4.3.1.7. `animation_cancel_end`

The function cancels an animation and sets edited properties to the end state where possible.

Table 13.10. Parameters of `animation_cancel_end`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

13.4.3.1.8. `animation_cancel_reset`

The function cancels an animation and resets edited properties to the initial state where possible.

Table 13.11. Parameters of `animation_cancel_reset`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

13.4.3.1.9. `animation_pause`

The function pauses an animation.

Table 13.12. Parameters of `animation_pause`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

13.4.3.1.10. `animation_play`

The function starts or continues an animation.

Table 13.13. Parameters of `animation_play`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate

Parameter	Type	Description
`<return>`	boolean	If true, the animation is not running yet.

13.4.3.1.11. `animation_reverse`

The function plays an animation backwards.

Table 13.14. Parameters of `animation_reverse`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the animation is not running yet.

13.4.3.1.12. `animation_running`

The function checks if an animation is currently running.

Table 13.15. Parameters of `animation_running`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the animation is running.

13.4.3.1.13. `animation_set_time`

The function sets the current time of an animation, can be used to skip or replay an animation.

Table 13.16. Parameters of `animation_set_time`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`time`	integer	time
`<return>`	boolean	If true, the function succeeded.

13.4.3.1.14. `asinf`

The functions calculates the principal value of the arc sine of `x`.

Table 13.17. Parameters of `asinf`

Parameter	Type	Description
`x`	float	The number to return the arc sine from
`<return>`	float	The return value

13.4.3.1.15. `atan2f`

The function calculates the principal value of the arc tangent of `y/x`, using the signs of the two arguments to determine the quadrant of the result.

Table 13.18. Parameters of `atan2f`

Parameter	Type	Description
`y`	float	Argument `y`
`x`	float	Argument `x`
`<return>`	float	The return value

13.4.3.1.16. `atan2i`

The function calculates the principal value of the arc tangent of `y/x`, using the signs of the two arguments to determine the quadrant of the result.

Table 13.19. Parameters of `atan2i`

Parameter	Type	Description
`y`	integer	Argument `y`
`x`	integer	Argument `x`
`<return>`	float	The return value

13.4.3.1.17. `atanf`

The function calculates the principal value of the arc tangent of `x`.

Table 13.20. Parameters of `atanf`

Parameter	Type	Description
`x`	float	The number to return the arc tangent from

Parameter	Type	Description
`<return>`	float	The return value

13.4.3.1.18. `bool2string`

The function converts a boolean variable to either the string `true` or `false`.

Table 13.21. Parameters of `bool2string`

Parameter	Type	Description
`x`	bool	The value to convert to a string
`<return>`	string	`true` in case `x` was true, and `false` otherwise

13.4.3.2. EB GUIDE Script functions C - H

13.4.3.2.1. `ceil`

The function returns the smallest integral value that is not less than the argument.

Table 13.22. Parameters of `ceil`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

13.4.3.2.2. `changeDynamicStateMachinePriority`

The function changes the priority of a dynamic state machine.

Table 13.23. Parameters of `changeDynamicStateMachinePriority`

Parameter	Type	Description
`state`		The state with the dynamic state machine list
`sm`	integer	The dynamic state machine
`priority`	integer	The priority of the dynamic state machine in the list

13.4.3.2.3. `character2unicode`

The function returns the Unicode value of the first character in a string.

Table 13.24. Parameters of `character2unicode`

Parameter	Type	Description
`str`	string	The input string
`<return>`	integer	The character as Unicode value 0 in case of errors

13.4.3.2.4. `clampf`

The function clamps a floating-point value to a defined range `[xmin, xmax]`, this means the function computes `max (xmin, min (xmax, x))`.

Table 13.25. Parameters of `clampf`

Parameter	Type	Description
`x`	float	The value to clamp
`xmin`	float	The minimum range
`xmax`	float	The maximum range
`<return>`	float	The `x` value clamped to the `[xmin, xmax]` range

13.4.3.2.5. `clampi`

The function clamps an integer value to a defined range `[xmin, xmax]`, this means the function computes `max (xmin, min (xmax, x))`.

Table 13.26. Parameters of `clampi`

Parameter	Type	Description
`x`	int	The value to clamp
`xmin`	int	The minimum range
`xmax`	int	The maximum range
`<return>`	int	The `x` value clamped to the `[xmin, xmax]` range

13.4.3.2.6. `clearAllDynamicStateMachines`

The function removes all dynamic state machines from the dynamic state machine list.

Table 13.27. Parameters of `clearAllDynamicStateMachines`

Parameter	Type	Description
`state`		The state with the dynamic state machine list

13.4.3.2.7. `color2string`

The function converts a color to eight hexadecimal values.

Table 13.28. Parameters of `color2string`

Parameter	Type	Description
`value`	color	The color to convert to string
`<return>`	string	The color formatted as a string of hexadecimal digits with `#` as prefix

NOTE

Formatting examples

The format of the returned string is `#RRGGBBAA` with two digits for each of the color channels red, green, blue and alpha.

For example, opaque pure red is converted to `#ff0000ff`, semi-transparent pure green is converted to `#00ff007f`.

13.4.3.2.8. `cosf`

The function returns the cosine of `x`, where `x` is given in radians.

Table 13.29. Parameters of `cosf`

Parameter	Type	Description
`x`	float	The number to return the cosine from
`<return>`	float	The return value

13.4.3.2.9. `deg2rad`

The function converts an angle from degrees to radians.

Table 13.30. Parameters of `deg2rad`

Parameter	Type	Description
`x`	float	The angle to convert from degrees to radians

Parameter	Type	Description
`<return>`	float	The return value

13.4.3.2.10. `expf`

The function returns the value of `e`, the base of natural logarithms, raised to the power of `x`.

Table 13.31. Parameters of `expf`

Parameter	Type	Description
`x`	float	The exponent
`<return>`	float	The return value

13.4.3.2.11. `float2string`

The function converts simple float to string.

Table 13.32. Parameters of `float2string`

Parameter	Type	Description
`value`	float	The value to convert to string
`<return>`	string	The float value, formatted as string

13.4.3.2.12. `floor`

The function returns the largest integral value not greater than the parameter value.

Table 13.33. Parameters of `floor`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

13.4.3.2.13. `fmod`

The function computes the remainder of the floating-point division `x/y`.

Table 13.34. Parameters of `fmod`

Parameter	Type	Description
`x`	float	The floating point numerator
`y`	float	The floating point denominator
`<return>`	float	The remainder of the division `x/y`

13.4.3.2.14. `focusMoveTo`

The function forces the focus manager to forward the focus to a dedicated focusable element.

Table 13.35. Parameters of `focusMoveTo`

Parameter	Type	Description
`widget`	widget	The widget on which the focus is moved.
`<return>`	void	

13.4.3.2.15. `focusNext`

The function forces the focus manager to forward the focus to the next focusable element.

Table 13.36. Parameters of `focusNext`

Parameter	Type	Description
`<return>`	void	

13.4.3.2.16. `focusPrevious`

The function forces the focus manager to return the focus to the previous focusable element.

Table 13.37. Parameters of `focusPrevious`

Parameter	Type	Description
`<return>`	void	

13.4.3.2.17. `format_float`

The function formats a float value.

Table 13.38. Parameters of `format_float`

Parameter	Type	Description
`format`	string	A string of the following structure: %[flags] [width] [.precision] type ▶ flags: Optional character or characters that control output justification and output of signs, blanks, leading zeros, decimal points, and octal and hexadecimal prefixes. ▶ width: Optional decimal number that specifies the minimum number of characters that are output. ▶ precision: Optional decimal number that specifies the number of significant digits or the number of digits after the decimal-point character . ▶ type: Required conversion specifier character that determines whether the associated argument is interpreted as a character, a string, an integer, or a float number.
`useDotAsDelimiter`	boolean	Defines the delimiter sign. Possible values: ▶ true: Use a dot as delimiter. ▶ false: Use a comma as delimiter.
`value`	float	The number to format

WARNING

Adhere to `printf` specification for C++

The `format` parameter is defined according to the `printf` specification for C++.

Using values that do not comply with this specification can lead to unexpected behavior.

For example, allowed types for `format_float` are `f`, `a`, `g` and `e`, and not more than one type character is allowed.

13.4.3.2.18. `format_int`

The function formats an integer value.

Table 13.39. Parameters of `format_int`

Parameter	Type	Description
`format`	string	A string of the following structure:

Parameter	Type	Description
		%[flags] [width] [.precision] type ▶ flags: Optional character or characters that control output justification and output of signs, blanks, leading zeros, decimal points, and octal and hexadecimal prefixes. ▶ width: Optional decimal number that specifies the minimum number of characters that are output. ▶ precision: Optional decimal number that specifies the minimum number of digits that are printed. ▶ type: Required conversion specifier character that determines whether the associated argument is interpreted as a character, a string, an integer, or a float number.
`value`	int	The number to format

WARNING

Adhere to `printf` specification for C++

The `format` parameter is defined according to the `printf` specification for C++.

Using values that do not comply with this specification can lead to unexpected behavior.

For example, allowed types for `format_int` are `d`, `i`, `o`, `x` and `u`, and not more than one type character is allowed.

13.4.3.2.19. `frac`

The function computes the fractional part of a floating-point value. The return value lies in the interval [0, 1]. For example, the function returns 0.5 for the parameter value x=1.5 or x=-1.5.

Table 13.40. Parameters of `frac`

Parameter	Type	Description
`x`	float	The floating point value
`<return>`	float	The fractional part of the floating-point value.

13.4.3.2.20. `getConfigItem`

The function fills a datapool item with a configuration item value.

Table 13.41. Parameters of `getConfigItem`

Parameter	Type	Description
`itemId`	dp_id	The datapool ID where the configuration item is to be stored

Parameter	Type	Description
`name`	string	The configuration item name
`<return>`	boolean	True if datapool item is successfully filled with a configuration item value

13.4.3.2.21. `getFontAscender`

The function returns the ascender of the font passed as parameter.

Table 13.42. Parameters of `getFontAscender`

Parameter	Type	Description
`x`	font	The font to be evaluated Note that if you have the multifont support added, only the default font is evaluated.
`<return>`	integer	The ascender of the font

13.4.3.2.22. `getFontDescender`

The function returns the descender of the font passed as parameter.

Table 13.43. Parameters of `getFontDescender`

Parameter	Type	Description
`x`	font	The font to be evaluated Note that if you have the multifont support added, only the default font is evaluated.
`<return>`	integer	The descender of the font

13.4.3.2.23. `getFontLineGap`

The function returns the line gap of the font passed as parameter.

Table 13.44. Parameters of `getFontLineGap`

Parameter	Type	Description
`x`	font	The font to be evaluated

Parameter	Type	Description
		Note that if you have the multifont support added, only the default font is evaluated.
`<return>`	integer	The line gap of the font

13.4.3.2.24. `getImageHeight`

The function returns the height in pixels of an image passed as parameter.

Table 13.45. Parameters of `getImageHeight`

Parameter	Type	Description
`x`	image widget	The widget to evaluate
`<return>`	integer	The height in pixels of an image

13.4.3.2.25. `getImageWidth`

The function returns the width in pixels of an image passed as parameter.

Table 13.46. Parameters of `getImageWidth`

Parameter	Type	Description
`x`	image widget	The widget to be evaluated
`<return>`	integer	The width in pixels of an image

13.4.3.2.26. `getLabelTextHeight`

The function returns the total height in pixels of a label's text. The total height is calculated using the formula:

```
total_height = line_height * line_count + line_gap * (line_count - 1)
```

The `line_gap` is calculated as the sum of the font line gap and the `lineGap` property of the **Multiple lines** widget feature. Both font line gap and the `lineGap` property can be negative.

Table 13.47. Parameters of `getLabelTextHeight`

Parameter	Type	Description
`widget`	label widget	The widget to be evaluated

Parameter	Type	Description
`<return>`	integer	The height in pixels of the text

13.4.3.2.27. `getLabelTextWidth`

The function returns the width of the longest line of a label's text.

Table 13.48. Parameters of `getLabelTextWidth`

Parameter	Type	Description
`widget`	label widget	The widget to evaluate
`<return>`	integer	The width in pixels of the longest line of the text

13.4.3.2.28. `getLineCount`

The function returns the number of lines of a label's text.

Table 13.49. Parameters of `getLineCount`

Parameter	Type	Description
`widget`	label widget	The widget to be evaluated
`<return>`	integer	The number of lines of the text

13.4.3.2.29. `getLineHeight`

The function returns the height of a line written with the font passed as parameter.

Table 13.50. Parameters of `getLineHeight`

Parameter	Type	Description
`x`	font	The font to be evaluated Note that if you have the multifont support added, only the default font is evaluated.
`<return>`	integer	The height of a line written with the specified font

13.4.3.2.30. `getProductString`

The function returns a string with the product name of EB GUIDE GTF.

Table 13.51. Parameters of `getProductString`

Parameter	Type	Description
`<return>`	string	The product name

13.4.3.2.31. `getTextHeight`

The function returns the height of a text with regard to its font resource. The height represents the sum of the font ascender and descender.

Table 13.52. Parameters of `getTextHeight`

Parameter	Type	Description
`text`	string	The text to evaluate
`font`	font	The font to evaluate
`<return>`	integer	The height of the text If the size of the font is `0` or negative, the function returns `0`.

NOTE

`getTextHeight`

The function always calculates the height value assuming that the text has a single line.

13.4.3.2.32. `getTextLength`

The function returns the number of characters in a text.

Table 13.53. Parameters of `getTextLength`

Parameter	Type	Description
`text`	string	The text to evaluate
`<return>`	integer	The number of characters in the text

NOTE

Escape sequences

EB GUIDE Script does not resolve escape sequences like `\n` and counts every character. For example, for the text `Label\n` the `getTextLength` function returns `7`.

13.4.3.2.33. `getTextWidth`

The function returns the width of a text with regard to its font resource.

Table 13.54. Parameters of `getTextWidth`

Parameter	Type	Description
`text`	string	The text to evaluate
`font`	font	The font to evaluate
`<return>`	integer	The width of the text If the size of the font is `0` or negative, the function returns `0`.

NOTE

The function always calculates the width value assuming that the text has a single line.

13.4.3.2.34. `getVersionString`

The function returns a string with the version number of EB GUIDE GTF.

Table 13.55. Parameters of `getVersionString`

Parameter	Type	Description
`<return>`	string	The version string

13.4.3.2.35. `has_list_window`

The function checks if the index is valid for a datapool item of type list. For windowed lists it also checks if the index is located inside at least one window.

Table 13.56. Parameters of `has_list_window`

Parameter	Type	Description
`itemId`	dp_id	The ID of the datapool item of type list
`index`	integer	The index within the datapool item
`<return>`	boolean	If true, the index within a datapool item is valid and located inside at least one window.

13.4.3.2.36. `hsba2color`

The function converts an HSB/HSV color to an EB GUIDE GTF color.

Table 13.57. Parameters of `hsba2color`

Parameter	Type	Description
`hue`	integer	The color value in degrees from 0 to 360
`saturation`	integer	The saturation in percent
`brightness`	integer	The brightness in percent
`alpha`	integer	The alpha value between 0 (totally transparent) and 255 (opaque)
`<return>`	color	The resulting EB GUIDE GTF color with the alpha value applied

13.4.3.3. EB GUIDE Script functions I - R

13.4.3.3.1. `int2float`

The function returns the integer value converted to a float point value.

Table 13.58. Parameters of `int2float`

Parameter	Type	Description
`value`	integer	The value to convert to float
`<return>`	float	The integer value, converted to float

13.4.3.3.2. `int2string`

The function converts a simple integer to string.

Table 13.59. Parameters of `int2string`

Parameter	Type	Description
`value`	integer	The value to convert to string
`<return>`	string	The integer value, in decimal notation, converted to string

13.4.3.3.3. `isDynamicStateMachineActive`

The function checks if the state with the dynamic state machine list is active.

Table 13.60. Parameters of `isDynamicStateMachineActive`

Parameter	Type	Description
`state`		The state with the dynamic state machine list
`sm`	integer	The dynamic state machine

13.4.3.3.4. `isWidgetOnActiveStatemachine`

The function checks if the widget belongs to an active state machine.

Table 13.61. Parameters of `isWidgetOnActiveStatemachine`

Parameter	Type	Description
`widget`	widget	The widget to be evaluated
`<return>`	boolean	True if the widget belongs to an active state machine

13.4.3.3.5. `language`

The function switches the language of all datapool items. This operation is performed asynchronously.

Table 13.62. Parameters of `language`

Parameter	Type	Description
`language`	languageType	The language to switch to, for example `f:language(l:German)`
`<return>`	void	

13.4.3.3.6. `lerp`

The function calculates the linear interpolation of two values x and y using the formula `(1-s) * x + s * y`

Table 13.63. Parameters of `lerp`

Parameter	Type	Description
`x`	float	The first value
`y`	float	The second value
`s`	float	A value that linearly interpolates between the `x` and `y` values
`<return>`	float	Returns the linear interpolation `(1-s) * x + s * y`

13.4.3.3.7. `localtime_day`

The function extracts the day [1:31] in local time from a system time value.

Table 13.64. Parameters of `localtime_day`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted day

13.4.3.3.8. `localtime_hour`

The function extracts the hours from the local time of a system time value.

Table 13.65. Parameters of `localtime_hour`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted hour

13.4.3.3.9. `localtime_minute`

The function extracts the minutes from the local time of a system time value.

Table 13.66. Parameters of `localtime_minute`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted minute

13.4.3.3.10. `localtime_month`

The function extracts the month [0:11] from the local time of a system time value.

Table 13.67. Parameters of `localtime_month`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time

Parameter	Type	Description
`<return>`	integer	The extracted month

13.4.3.3.11. `localtime_second`

The function extracts the seconds from the local time of a system time value.

Table 13.68. Parameters of `localtime_second`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted second

13.4.3.3.12. `localtime_weekday`

The function extracts the week day [0:6] from the local time of a system time value. 0 is Sunday.

Table 13.69. Parameters of `localtime_weekday`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted weekday

13.4.3.3.13. `localtime_year`

The function extracts the year from the local time of a system time value.

Table 13.70. Parameters of `localtime_year`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted year

13.4.3.3.14. `log10f`

The function returns the base 10 logarithm of `x`.

Table 13.71. Parameters of `log10f`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

13.4.3.3.15. `logf`

The function returns the natural logarithm of `x`.

Table 13.72. Parameters of `logf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

13.4.3.3.16. `maxf`

The function computes the maximum of two floating-point values.

Table 13.73. Parameters of `maxf`

Parameter	Type	Description
`x`	float	The first value
`y`	float	The second value
`<return>`	float	The maximum of x and y

13.4.3.3.17. `maxi`

The function computes the maximum of two integer values.

Table 13.74. Parameters of `maxi`

Parameter	Type	Description
`x`	int	The first value
`y`	int	The second value

Parameter	Type	Description
`<return>`	int	The maximum of `x` and `y`

13.4.3.3.18. `minf`

The function computes the minimum of two floating-point values.

Table 13.75. Parameters of `minf`

Parameter	Type	Description
`x`	float	The first value
`y`	float	The second value
`<return>`	float	The minimum of `x` and `y`

13.4.3.3.19. `mini`

The function computes the minimum of two integer values.

Table 13.76. Parameters of `mini`

Parameter	Type	Description
`x`	int	The first value
`y`	int	The second value
`<return>`	int	The minimum of `x` and `y`

13.4.3.3.20. `nearbyint`

The function rounds to nearest integer.

Table 13.77. Parameters of `nearbyint`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

13.4.3.3.21. `popDynamicStateMachine`

The function removes the specified dynamic state machine from the priority queue.

Table 13.78. Parameters of `popDynamicStateMachine`

Parameter	Type	Description
`state`		The state with the dynamic state machine list
`sm`	string	The dynamic state machine

13.4.3.3.22. `powf`

The function returns the value of `x` raised to the power of `y`.

Table 13.79. Parameters of `powf`

Parameter	Type	Description
`x`	float	The argument `x`
`y`	float	The argument `y`
`<return>`	float	The return value

13.4.3.3.23. `pushDynamicStateMachine`

The function inserts the dynamic state machine in a priority queue.

Table 13.80. Parameters of `pushDynamicStateMachine`

Parameter	Type	Description
`state`		The state with the dynamic state machine list
`sm`	string	The dynamic state machine
`priority`	integer	The priority of the dynamic state machine in the list. Note that `0` is handled with a higher priority than `1`.

13.4.3.3.24. `rad2deg`

The function converts an angle form radians to degree.

Table 13.81. Parameters of `rad2deg`

Parameter	Type	Description
`x`	float	The argument

Parameter	Type	Description
`<return>`	float	The return value

13.4.3.3.25. `rand`

The function gets a random value between 0 and 2^{31}-1.

Table 13.82. Parameters of `rand`

Parameter	Type	Description
`<return>`	integer	A random number between 0 and 2^{31}-1

13.4.3.3.26. `rgba2color`

The function converts from RGB color space to EB GUIDE GTF color.

Table 13.83. Parameters of `rgba2color`

Parameter	Type	Description
`red`	integer	The red color coordinate, ranging from 0 to 255
`green`	integer	The green color coordinate, ranging from 0 to 255
`blue`	integer	The blue color coordinate, ranging from 0 to 255
`alpha`	integer	The alpha value, ranging from 0 (totally transparent) to 255 (opaque)
`<return>`	color	The color converted from RGB color space to EB GUIDE GTF color, with the alpha value applied

13.4.3.3.27. `round`

The function rounds to nearest integer, but rounds halfway cases away from zero.

Table 13.84. Parameters of `round`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

13.4.3.4. EB GUIDE Script functions S - W

13.4.3.4.1. `saturate`

The function clamps a floating-point value to [0, 1] range, i.e. the function computes `max (0, min (1, x))` and acts as a shorthand notation for `clampf(0, 1, x)`

Table 13.85. Parameters of `saturate`

Parameter	Type	Description
`x`	float	The value to clamp
`<return>`	float	The `x` value clamped to the [0, 1] range

13.4.3.4.2. `seed_rand`

The function sets the seed of the random number generator.

Table 13.86. Parameters of `seed_rand`

Parameter	Type	Description
`seed`	integer	The value to seed the random number generator
`<return>`	void	

13.4.3.4.3. `shutdown`

The function requests the framework to shutdown the program.

13.4.3.4.4. `sinf`

The function returns the sine of `x`, where `x` is given in radians.

Table 13.87. Parameters of `sinf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

13.4.3.4.5. `skin`

The function switches the skin of all datapool items. This operation is performed asynchronously.

Table 13.88. Parameters of `skin`

Parameter	Type	Description
`skin`	skinType	The skin to switch to, for example `f:skin(s:Standard)`
`<return>`	void	

13.4.3.4.6. `smoothstep`

The function computes the smooth hermite interpolation `3z² - 2z³ with z = (x - xmin) / (xmax - xmin)` in case it is in range `[xmin, xmax]` and 0 otherwise. The function returns a value in the interval `[0,1]`.

Table 13.89. Parameters of `smoothstep`

Parameter	Type	Description
`xmin`	float	The xmin value
`xmax`	float	The xmax value
`x`	float	The value to be interpolated
`<return>`	float	Returns the hermite interpolation `3z² - 2z³ with z = (x-xmin) / (xmax-xmin)`

13.4.3.4.7. `sqrtf`

The function returns the non-negative square root of `x`.

Table 13.90. Parameters of `sqrtf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

13.4.3.4.8. `string2float`

The function converts the initial part of a string to float.

The expected form of the initial part of the string is as follows:

1. Optional leading white space
2. Optional plus ('+') or minus ('-') sign
3. One of the following:
   - Decimal number
   - Hexadecimal number
   - Infinity
   - NAN (not-a-number)

Table 13.91. Parameters of `string2float`

Parameter	Type	Description
`str`	string	The string value
`<return>`	float	The return value

13.4.3.4.9. `string2int`

The function converts the initial part of a string to integer. The result is clipped to the range from 2147483647 to -2147483648, if the input exceeds the range. If the string does not start with a number, the function returns 0.

Table 13.92. Parameters of `string2int`

Parameter	Type	Description
`str`	string	The string value
`<return>`	integer	The return value

13.4.3.4.10. `string2string`

The function is used to truncate a string to a given number of characters.

Table 13.93. Parameters of `string2string`

Parameter	Type	Description
`str`	string	The string to truncate
`len`	integer	The maximum length of the string
`<return>`	string	The truncated string

13.4.3.4.11. `substring`

The function creates a substring copy of the string. Negative end indexes are supported.

Examples:

- `substring("abc", 0, -1)` returns `abc`.
- `substring("abc", 0, -2)` returns `ab`.
- `substring ("abcd", 1, 3)` returns `bc`.

Table 13.94. Parameters of `substring`

Parameter	Type	Description
`str`	string	The input string
`startIndex`	integer	The first character index of the result string
`endIndex`	integer	The first character index that is not part of the result
`<return>`	string	The language string

13.4.3.4.12. `system_time`

The function gets the current system time in seconds. The result is intended to be passed to the `localtime_*` functions.

Table 13.95. Parameters of `system_time`

Parameter	Type	Description
`<return>`	integer	The system time in seconds

13.4.3.4.13. `system_time_ms`

The function gets the current system time in milliseconds.

Table 13.96. Parameters of `system_time_ms`

Parameter	Type	Description
`<return>`	integer	The system time in milliseconds

13.4.3.4.14. `tanf`

The function returns the tangent of `x`, where `x` is given in radians.

Table 13.97. Parameters of `tanf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

13.4.3.4.15. `trace_dp`

The function writes debugging information about a datapool item to the trace log and the connection log.

Table 13.98. Parameters of `trace_dp`

Parameter	Type	Description
`itemId`	dp_id	The datapool ID of the item to trace debug information about
`<return>`	void	

13.4.3.4.16. `trace_string`

The function writes a string to the trace log and the connection log.

Table 13.99. Parameters of `trace_string`

Parameter	Type	Description
`str`	string	The text to trace
`<return>`	void	

13.4.3.4.17. `transformToScreenX`

The function takes a widget and a local coordinate and returns x-position in the screen-relative world coordinate system.

Table 13.100. Parameters of `transformToScreenX`

Parameter	Type	Description
`widget`	widget	The widget to which the coordinates are relative
`localX`	integer	The x-position of the local coordinate
`localY`	integer	The y-position of the local coordinate

Parameter	Type	Description
`<return>`	integer	The x-position of the screen coordinate

13.4.3.4.18. `transformToScreenY`

The function takes a widget and a local coordinate and returns y-position of a position in the screen-relative world coordinate system.

Table 13.101. Parameters of `transformToScreenY`

Parameter	Type	Description
`widget`	widget	The widget to which the coordinates are relative
`localX`	integer	The x-position of the local coordinate
`localY`	integer	The y-position of the local coordinate
`<return>`	integer	The y-position of the screen coordinate

13.4.3.4.19. `transformToWidgetX`

The function takes a widget and a screen coordinate as provided to the touch reactions and returns x-position in the widget-relative local coordinate system.

Table 13.102. Parameters of `transformToWidgetX`

Parameter	Type	Description
`widget`	widget	The widget to which the coordinates are relative
`screenX`	integer	The x-position of the screen coordinate
`screenY`	integer	The y-position of the screen coordinate
`<return>`	integer	The x-position of the local coordinate

13.4.3.4.20. `transformToWidgetY`

The function takes a widget and a screen coordinate as provided to the touch reactions and returns y-position in the widget-relative local coordinate system.

Table 13.103. Parameters of `transformToWidgetY`

Parameter	Type	Description
`widget`	widget	The widget to which the coordinates are relative

Parameter	Type	Description
`screenX`	integer	The x-position of the screen coordinate
`screenY`	integer	The y-position of the screen coordinate
`<return>`	integer	The y-position of the local coordinate

13.4.3.4.21. `trunc`

The function rounds to the nearest integer value, always towards zero.

Table 13.104. Parameters of `trunc`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

13.4.3.4.22. `widgetGetChildCount`

The function obtains the number of child widgets of the given widget.

Table 13.105. Parameters of `widgetGetChildCount`

Parameter	Type	Description
`widget`	widget	The widget of which to obtain the number of child widgets
`<return>`	integer	The number of child widgets

13.5. Events

Table 13.106. Properties of an event

Property name	Description
`Name`	The name of the event
`Event ID`	A numeric value that EB GUIDE TF uses to send and receive the event
`Event group`	The name of the event group An event group has an ID that EB GUIDE TF uses to send and receive the event.

13.5.1. Decimal codes for key events

Table 13.107. Decimal codes of numpad keys

Numpad key	Decimal code
0	5
1	6
2	7
3	8
4	9
5	10
6	11
7	12
8	13
9	14

Table 13.108. Decimal codes of function keys

Function key	Decimal code
F1	18
F2	19
F3	20
F4	21
F5	22
F6	23
F7	24
F8	25
F9	26
F10	27
F11	28
F12	29

Table 13.109. Decimal codes of ASCII keys

ASCII key	Decimal code
Space	32
a	97
b	98

ASCII key	Decimal code
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

13.6. `model.json` configuration file

The `model.json` is an EB GUIDE TF configuration file that contains configuration items which are relevant for a single EB GUIDE model.

The `model.json` file is a part of the exported EB GUIDE model.

The following table is used as documentation for all default configuration parameters.

NOTE

JSON object notation

If you configure `model.json` in EB GUIDE Studio, use the JSON object notation.

For an example, see section 13.6.1, “Example `model.json` in EB GUIDE Studio”.

For more information about the `JSON` format, see http://www.json.org.

Table 13.110. Common

Configuration item	Type	Description	Default value
`gtf.eventsystem.maxQueue`	integer	Maximum size of the event queues	`0`
`gtf.model.traces`	boolean	Enables the tracing of the `f:trace_string` script function	`true`
`gtf.model.identifier`	string	Unique identifier of the EB GUIDE model (equal to the EB GUIDE Studio project `UUID`)	empty
`gtf.model.identifier.short`	integer	Short identifier of the EB GUIDE model	`0xdeadbeaf`
`gtf.model.initialLanguage.id`	string	Unique identifier of the language that is activated after the EB GUIDE model start-up (equal to `UUID` of the language in the EB GUIDE Studio project). Find the `UUID` identifiers of the languages as part of the EB GUIDE Studio project export in `include\export\aspects\languages.h`.	undefined
`gtf.model.initialLanguage.force`	boolean	If `true`, the value of `gtf.model.initialLanguage.id` is preferred over the active language of the	`false`

Configuration item	Type	Description	Default value
		already running EB GUIDE model. Otherwise, the active language of the already running EB GUIDE model has the higher priority.	
`gtf.model.initialSkin.id`	string	Unique identifier of the skin that is activated after the EB GUIDE model start-up (equal to `UUID` of the skin in the EB GUIDE Studio project). Find the `UUID` identifiers of the skins as part of the EB GUIDE Studio project export in `include\export\aspects\skins.h`.	undefined
`gtf.model.initialSkin.force`	boolean	If `true`, the value of `gtf.model.initialSkin.id` is preferred over the active skin of the already running EB GUIDE model. Otherwise, the active skin of the already running EB GUIDE model has the higher priority.	`false`

Table 13.111. Files and paths

Configuration item	Type	Description	Default value
`gtf.model.path`	string	Path to the EB GUIDE model	`None`
`gtf.model.config`	string	Full path to the EB GUIDE model configuration	`<gtf.model.path>/model.json`
`gtf.datapool.descriptionFile`	string	Name of the datapool description file	`datapool.gtf`

Configuration item	Type	Description	Default value
`gtf.model.files.sm`	string	Name of the state machine description file	`model.bin`
`gtf.model.files.rm`	string	Name of the resources description file	`resources.bin`
`gtf.model.files.views`	string	Name of the view description file	`views.bin`
`gtf.model.files.types`	string	Name of the type description file	`types.bin`
`gtf.model.pluginstoload`	string list	Names of EB GUIDE model plugins to load	empty string list
`gtf.eventsystem.mapFile`	string	Name of the event system mapping file	`eventMap.gtf`

The option `gtf.model.coreNames` is a string list that contains the names of all configured cores. The following table contains configuration items for every core.

Table 13.112. Cores

Configuration item	Type	Description	Default value
`gtf.model.cores.<corename>.ownThread`	boolean	Specifies if the core uses an own thread to run	`false`
`gtf.model.cores.<corename>.id`	integer	Specifies the core context identifier	`0`

The option `gtf.model.sceneNames` is a string list that contains the names of all configured scenes. For every scene, the configuration items in the following table are found.

Table 13.113. Scenes

Configuration item	Type	Description	Default value
`gtf.model.scenes.<scenename>.visible`	boolean	Determines the visibility of the scene	`true`
`gtf.model.scenes.<scenename>.width`	integer	Width of the scene	`800`
`gtf.model.scenes.<scenename>.height`	integer	Height of the scene	`480`
`gtf.model.scenes.<scenename>.x`	integer	Coordinates of the scene's starting point	`0`
`gtf.model.scenes.<scenename>.y`	integer	Coordinates of the scene's starting point	`0`

Configuration item	Type	Description	Default value
`gtf.model.scenes.<scenename>.projectName`	string	Name of the working project	
`gtf.model.scenes.<scenename>.windowCaption`	string	Displayed window name text	
`gtf.model.scenes.<scenename>.sceneId`	integer	Identifier for the scene	`0`
`gtf.model.scenes.<scenename>.maxFPS`	integer	The redraw rate (FPS = Frames per second). Set to 0 for an unlimited redraw rate.	`60`
`gtf.model.scenes.<scenename>.hwLayerId`	integer	Specifies the core context identifier	`0`
`gtf.model.scenes.<scenename>.colorMode`	integer	Specifies the color mode: ▶ 1: 32-bit (RGBA8888) ▶ 2: 16-bit (RGB565) ▶ 3: 24-bit (RGB888) ▶ 4: 32-bit sRGB ▶ 5: 32-bit sRGB (Emulated)	`1`
`gtf.model.scenes.<scenename>.multisampling`	integer	Specifies the multisampling of the scene ▶ 0: no multisampling ▶ 1: 2x multisampling ▶ 2: 4x multisampling	`0`
`gtf.model.scenes.<scenename>.enableRemoteFramebuffer`	boolean	If `true`, the transfer of the off-screen buffer to the simulation window is enabled	`false`

Configuration item	Type	Description	Default value
gtf.model.scenes.<scenename>.showWindowFrame	boolean	Determines if the renderer window frame should be displayed	true
gtf.model.scenes.<scenename>.showWindow	boolean	If true, an additional window for simulation is opened on Windows based systems	true
gtf.model.scenes.<scenename>.disableVsync	boolean	If true, the vertical synchronization for the renderer is disabled.	false
gtf.model.scenes.<scenename>.showFPS	integer	Possible values: ▶ 0: Do not show FPS ▶ 1: Show FPS on the screen ▶ 2: Show FPS on the console ▶ 3: Show FPS on the screen and on the console ▶ 4: Show FPS on the console in an enlarged text ▶ 5: Show FPS on the screen and on the console in an enlarged text	0
gtf.model.scenes.<scenename>.renderer	string	Name of the renderer to use: OpenGLRenderer or OpenGL3Renderer	

Table 13.114. Rendering common

Configuration item	Type	Description	Default value
gtf.model.fontCache.width	integer	Width of the font cache atlas texture	512

Configuration item	Type	Description	Default value
`gtf.model.fontCache.height`	integer	Height of the font cache atlas texture	`512`
`gtf.model.fontCache.age`	integer	Maximum allowed age before the refresh operation of the font cache has to be done	`100`
`gtf.model.traversalStackSize`	integer	The renderers traver-sal stack size in bytes	`32768`

The configuration items in the following table belong together. This means that the renderer expects that the same number of items is in all three lists. The entry with an index in one list belongs to the entries with the same index in other lists.

Table 13.115. Renderer display extensions

Configuration item	Type	Description	Default value
`gtf.model.displayId`	integer list	Identifiers of the scenes	
`gtf.model.maxCacheSize`	integer list	Maximum texture caches for the scenes	
`gtf.model.driverName`	string list	`OS` specific driver names for the scenes, e.g. `/dev/fb0`	

The configuration items in the following table are used to configure the `TextEngine` component. `TextEngine` is based on the FreeType third-party library. The following parameters are passed to the FreeType implementation. For more information about FreeType, see https://www.freetype.org/freetype2/docs/reference/ft2-cache_subsystem.html.

Due to the way EB GUIDE TF handles font sizes, `ft_size` objects are not cached separately from `ft_face` objects. Consider that the values for `max_sizes` can be limited by the hardware of your target platform.

Table 13.116. `TextEngine` configuration items

Configuration item	Type	Description	Default value
`gtf.model.textengine.replacementG-lyph`	integer	Unicode character that should be used in case the dedicated font character is not found in the current font	`0xfffd`
`gtf.model.textengine.maxFaces`	integer	Maximum number of cached font faces	`0`

Configuration item	Type	Description	Default value
`gtf.model.textengine.maxSizes`	integer	Maximum number of cached font sizes	`0`
`gtf.model.textengine.maxBytes`	integer	Maximum number of memory in bytes that can be used for caches	`0`
`gtf.model.textengine.fontCacheThreshold`	integer	Preferred number of cached fonts	`2`
`gtf.model.textengine.enablePlainFileStream`	boolean	Determines the font access configuration. If `true`, the plain file I/O access is used. If `false`, the ROM-mapped file access is used.	`false`

NOTE

Configuration items for bitmap fonts

For `.fnt` bitmap fonts you can use only the `replacementGlyph` configuration item. You cannot use other configuration items in table 13.116, “`TextEngine` configuration items” for bitmap fonts.

NOTE

ROM-mapped file approach vs. plain file I/O approach

The ROM-mapped file approach in general provides higher performance. But on some systems, for example QNX, it consumes more memory than the plain file I/O approach. Plain file I/O approach in general consumes less memory than the ROM-mapped file approach. But it can lead to lower performance.

The option `gtf.model.touchDevicesNames` is a string list containing the names of all configured touch devices. For every touch device the configuration items listed in the following table are available.

Table 13.117. Touch devices

Configuration item	Type	Description	Default value
`gtf.model.touchDevices.<deviceName>.touchscreenType`	integer	Defines the touch device type: ▶ `0`: `Galaxy` ▶ `1`: `imx WVGA` ▶ `2`: `Mouse` ▶ `3`: `General`	`3`

Configuration item	Type	Description	Default value
		▶ `4: Lil-liput_889GL` ▶ `5: GeneralMul-titouch` ▶ `6: Lilliput with automat-ic calibra-tion` ▶ `7: Generic-TouchConfigu-ration`	
`gtf.mod-el.touchDevices.<deviceName>.dis-playManagerId`	integer	Specifies the scene ID for which the device is valid	0
`gtf.mod-el.touchDevices.<deviceName>.-touchId`	integer	Specifies the ID of the device	0
`gtf.mod-el.touchDevices.<deviceName>.min-imalDistanceToMove`	integer	Threshold for react-ing on touch position changes	0
`gtf.mod-el.touchDevices.<deviceName>.-touchMoveRepeatTimeout`	integer	Delay between touch position change notifi-cations	0
`gtf.mod-el.touchDevices.<deviceName>.width`	integer	Width of the touchable device area	0
`gtf.mod-el.touchDevices.<deviceName>.-height`	integer	Height of the touch-able device area	0
`gtf.mod-el.touchDevices.<deviceName>.x_-high`	integer	Maximum horizontal resolution extend of the touchable device area	0
`gtf.mod-el.touchDevices.<deviceName>.y_-high`	integer	Maximum vertical res-olution extend of the touchable device area	0

Configuration item	Type	Description	Default value
`gtf.model.touchDevices.<deviceName>.x_low`	integer	Minimal horizontal resolution extend of the touchable device area	`0`
`gtf.model.touchDevices.<deviceName>.y_low`	integer	Minimal vertical resolution extend of the touchable device area	`0`
`gtf.model.touchDevices.<deviceName>.devicePath`	string	Name of the driver used for touch, e.g. `/dev/input0`	

The configuration items in the following table are used to configure the binary shader cache in the renderer..

Table 13.118. Shaders

Configuration item	Type	Description	Default value
`gtf.model.binShadersLocation`	string	Name of the folder under the GTF binaries directory in which the binary shaders should be located. Can also be an absolute path.	
`gtf.model.readBinShaders`	boolean	Determines if binary shaders written at a previous run should be read from disk.	`false`
`gtf.model.writeBinShaders`	boolean	Determines if binary shaders should be written to disk to be reused at a later run.	`false`

13.6.1. Example `model.json` in EB GUIDE Studio

Example 13.1.
`model.json` in EB GUIDE Studio

```
{
  "gtf": {
    "model": {
      "coreNames": [
        "HMI"
```

```
        ],
        "cores": {
          "HMI": {
            "id": 0,
            "ownThread": false
          }
        },
        "sceneNames": [
          "Main"
        ],
        "scenes": {
          "Main": {
            "name": "Main",
            "visible": true,
            "width": 800,
            "height": 480,
            "x": 0,
            "y": 0,
            "projectName": "project",
            "windowCaption": "EB GUIDE 6 model",
            "sceneId": 0,
            "maxFPS": 60,
            "hwLayerId": 0,
            "colorMode": 1,
            "multisampling": 0,
            "enableRemoteFramebuffer": false,
            "showWindowFrame": true,
            "showWindow": true,
            "disableVSync": false,
            "showFPS": 0,
            "renderer": "OpenGL3Renderer",
            "context": 0
          }
        },
       "identifier": "29691ce7-cb4c-4337-8852-93c90c62e624",
        "pluginstoload": [
          "GtfGui",
          "GtfGuiOpenGLES3"
        ]
      }
    }
  }
```

13.7. OpenGL ES extensions

OpenGL and OpenGL3 renderers use several OpenGL ES extensions that are listed below.

In case that the extensions are not available, there are limitations in rendering.

NOTE

Multiple extensions in table below

Multiple extensions listed with `OR`: The limitations apply if one of the listed extensions is not available.

Multiple extensions listed with `AND`: The limitations apply if all of the listed extensions are not available.

Table 13.119. OpenGL extensions

OpenGL extension	Limitation if not available on the OpenGL renderer	Limitation if not available on the OpenGL3 renderer
`GL_OES_get_program_binary`	Binary shaders cache not supported	None

`GL_EXT_texture_compression_s3tc`
 	
`AND`
 	
`GL_EXT_texture_compression_dxt1`
	DXT1 texture compression not supported	DXT1 texture compression not supported

`GL_EXT_texture_compression_s3tc`	DXT3 and DXT5 texture compression not supported	DXT3 and DXT5 texture compression not supported
`GL_IMG_texture_compression_pvrtc`	PVRTC texture compression not supported	PVRTC texture compression not supported
`GL_IMG_texture_compression_pvrtc2`	PVRTC2 texture compression not supported	PVRTC2 texture compression not supported
`GL_KHR_texture_compression_astc_ldr`	ASTC texture compression not supported	ASTC texture compression not supported

`GL_OES_depth32`
 	
`AND`
 	
`GL_OES_depth24`
	▶ Scene graphs rendered with less depth precision
	▶ Number of widgets limited to 65534 instead of 16 millions
	▶ At the EB GUIDE SDK interface framebuffers cannot be
		None

OpenGL extension	Limitation if not available on the OpenGL renderer	Limitation if not available on the OpenGL3 renderer
	created with more than 16-bit depth precision	

`GL_EXT_shader_texture_lod`
 	
OR
 	
`GL_EXT_texture_rg`
 	
OR
 	
`GL_OES_texture_float`
 	
OR
 	
`GL_OES_texture_half_float`
	Image-based lighting not supported	None

`GL_EXT_multisampled_render_to_texture`
 	
AND
 	
`GL_IMG_multisampled_render_to_texture`
 	
AND
 	
(`GL_NV_framebuffer_blit` OR `GL_NV_framebuffer_multisample`)
 	
AND
 	
(`GL_ANGLE_framebuffer_blit` OR `GL_ANGLE_framebuffer_multisample`)
	Multisampling not supported for scene graphs and EB GUIDE SDK offscreen framebuffers	None

`GL_OES_texture_float`	Tone mapping for scene graphs not supported	None

`GL_EXT_texture_border_clamp`
 	
AND
 	
`GL_OES_texture_border_clamp`
	Visual glitches in area outside of alpha mask widget possible	Visual glitches at area outside of alpha mask widget possible

OpenGL extension	Limitation if not available on the OpenGL renderer	Limitation if not available on the OpenGL3 renderer
AND GL_NV_texture_border_clamp		
GL_OES_element_index_uint	No functional limitation, but decreased performance possible	None
GL_OES_packed_depth_stencil	No functional limitation, but decreased performance possible	None
GL_OES_vertex_half_float	No functional limitation, but decreased performance and increased graphics memory consumption possible	None
GL_OES_vertex_type_10_10_10_2	No functional limitation, but decreased performance and increased graphics memory consumption possible	None
GL_EXT_sRGB	No functional limitation, but decreased performance possible for scene graphs in sRGB color mode or for sRGB textures	None

13.8. `platform.json` configuration file

The `platform.json` is an EB GUIDE TF configuration file which contains common and platform dependent items.

The `platform.json` file is a part of the exported EB GUIDE model.

The following table is used as documentation for all default configuration parameters.

NOTE

JSON object notation

If you configure `platform.json` within EB GUIDE Studio, use the JSON object notation.

For an example, see section 13.8.1, “Example `platform.json` in EB GUIDE Studio”.

For more information about the `JSON` format, see http://www.json.org.

Table 13.120. Platform configuration

Configuration item	Type	Description	Default value
`gtf.servicemapper.port`	integer	Connection port for the services (e.g. EB GUIDE Monitor)	`60000`
`gtf.core.pluginstoload`	string list	List of core plugins that should be loaded (relative to binary folder or absolute path)	`None`
`gtf.launcher.editmode`	boolean	Defines if EB GUIDE TF is running in EB GUIDE Studio. This is a read-only item.	`false`
`gtf.platform.config`	string	Full path to the `platform.json` file. This is a read-only item.	<model_folder>/platform.json
`gtf.framework.path`	string	Path to the `GtfStartup` executable. This is a read-only item.	<binary_folder>
`gtf.diagnostic.memory.interval`	integer	Specifies the time interval for the memory diagnostic. If value is `0` the diagnostic is deactivated.	`0`
`gtf.ipc.role`	string	The role of the IPC node. Possible values are `server` or `client`	`server`
`gtf.ipc.discovery.network`	string	The IPv4 network address which will be used for the server-client discovery	`255.255.255.255`

Configuration item	Type	Description	Default value
		mechanism. In case of direct connection, this represents the servers' network address.	
`gtf.ipc.discovery.port`	integer	The network port which will be used for the server-client discovery mechanism. In case of direct connection, this has to be equal to the item `gtf.servicemapper.port` from the server configuration.	`4711`
`gtf.ipc.datapool.config`	string	The configuration file containing the datapool items that should be part of IPC communication	`ipc_datapool.gtf`
`gtf.ipc.discovery.mode`	string	The discovery mode used for connecting the server and the clients. Possible options are: "`broadcast`", "`multicast`" and "`direct`".	`broadcast`
`gtf.ipc.client.timeout`	integer	Retry period of the client connection to the server, expressed in milliseconds.	`5000`
`gtf.osal.threading.pool.threadCount`	integer	The number of threads created and used by threadpool. The valid range is between 1 and 32. All other values will be clamped.	`1`
`gtf.resourcesytem.residentformats`	string list	The resource system formats that once attached to a resource	For the default value, see "`residentformats`" in sec-

Configuration item	Type	Description	Default value
		cannot be removed from it. You can only replace the previous resident format by attaching a new resident format to this resource.	tion 13.8.1, “Example `platform.json` in EB GUIDE Studio”.
`gtf.resourcesytem.defaultdecoding`	object list	Pairs of input and output formats used by the resource system to decode in advance, before the actual decode request.	For the default value, see "`default-decoding`" in section 13.8.1, “Example `platform.json` in EB GUIDE Studio” .
`gtf.resourcesystem.cache.softlimit`	integer	Specifies the limit in MB of the resource system cache. If the limit is exceeded, the cleanup action will be enqueued.	`-1`
`gtf.resourcesystem.cache.hardlimit`	integer	Specifies the limit in MB of the resource system cache. If the limit is exceeded, the cleanup action will be performed. Unlike the `softlimit` case, the cleanup action will be performed immediately.	`-1`

13.8.1. Example `platform.json` in EB GUIDE Studio

Example 13.2.
`platform.json` in EB GUIDE Studio

```
{
  "gtf": {
    "core": {
      "pluginstoload": [
        "GtfService",
```

```
          "GtfRuntime"
        ]
      },
      "servicemapper": {
        "port": 60000
      },
      "resourcesystem": {
        "defaultdecoding": [
          {
            "inputformat": "gtf::decoder::ImagePath",
            "outputformat": "gtf::decoder::EncodedMemImage"
          },
          {
            "inputformat": "gtf::decoder::MeshPath",
            "outputformat": "gtf::decoder::EncodedMeshData"
          },
          {
            "inputformat": "gtf::decoder::IBLPath",
            "outputformat": "gtf::decoder::EncodedIBLData"
          },
          {
            "inputformat": "gtf::decoder::EncodedMemImage",
            "outputformat": "gtf::decoder::MemImage"
          },
          {
            "inputformat": "gtf::decoder::DescriptorResource::font",
            "outputformat": "gtf::decoder::FontData"
          },
          {
            "inputformat": "gtf::decoder::DescriptorResource::image",
            "outputformat": "gtf::decoder::ImagePath"
          },
          {
            "inputformat": "gtf::decoder::DescriptorResource::ibl",
            "outputformat": "gtf::decoder::IBLPath"
          },
          {
            "inputformat": "gtf::decoder::DescriptorResource::mesh",
            "outputformat": "gtf::decoder::MeshPath"
          },
          {
            "inputformat": "gtf::decoder::MemImage",
            "outputformat": "gtf::scdr::decoder::NinePatchImage"
          },
          {
            "inputformat": "gtf::scdr::decoder::NinePatchImage",
            "outputformat": "gtf::scdr::decoder::DefaultMemImage"
```

```
        }
      ],
      "residentformats": [
        "gtf::decoder::FontData",
        "gtf::decoder::IBLPath",
        "gtf::decoder::ImagePath",
        "gtf::decoder::MeshPath"
      ]
    }
  }
}
```

13.9. Scenes

Table 13.121. Properties of a scene

Property name	Description
`height`	The height of the area in which the views of a haptic state machine are rendered on a target device
`width`	The width of the area in which the views of a haptic state machine are rendered on a target device
`x`	The x-offset of the area in which the views of a haptic state machine are rendered on a target device
`y`	The y-offset of the area in which the views of a haptic state machine are rendered on a target device
`visible`	If true, the state machine and its child widgets are visible.
`projectName`	The name of the EB GUIDE project
`windowCaption`	The text that is shown on the window frame
`sceneID`	The unique scene identifier which can be used, for example, for input handling
`maxFPS`	The redraw rate (FPS = Frames per second) Set to `0` for an unlimited redraw rate.
`hwLayerID`	The ID of the hardware layer on the target device's display that is mapped to the current state machine
`colorMode`	Possible values: ▶ `32-bit (=1)`: RGBA8888 ▶ `16-bit (=2)`: RGB565

Property name	Description
	▶ `24-bit (=3)`: RGB888 ▶ `32-bit sRGB (=4)`: This value uses GPU hardware support. Use this value, if you want to have sRGB support for an image widget or for the **Diffuse texture** widget feature. ▶ `32-bit sRGB (Emulated) (=5)`: Use this value only if `32-bit sRGB` does not yield correct results.
`multisampling`	Possible values: ▶ `Off (= 0)`: no multisampling ▶ `2x (=1)`: 2x multisampling ▶ `4x (=2)`: 4x multisampling Also see "Settings for multisampling".
`enableRemoteFramebuffer`	If true, transfer of the off-screen buffer to the simulation window is enabled
`showWindowFrame`	If true, a frame is displayed on the simulation window. The frame allows the window to be grabbed and moved.
`showWindow`	If true, an additional window for simulation is opened on Windows based systems.
`disableVSync`	If true, vertical synchronization for the renderer is disabled.
`showFPS`	Possible values: ▶ `Off (=0)`: Do not show FPS ▶ `On screen (=1)`: Show FPS on the screen ▶ `Console (=2)`: Show FPS on the console ▶ `Console & on screen (=3)`: Show FPS on the screen and on the console ▶ `On screen (large text) (=4)` ▶ Console & on screen (large text) (=5)
`Renderer`	Defines a renderer for the scene. Possible values: ▶ `OpenGLRenderer`

Property name	Description
	▶ `OpenGL3Renderer`

TIP

Settings for multisampling

The higher the resolution for multisampling is the better the quality of the rendering result. However, be aware that multisampling decreases the rendering performance, especially on a target device. At small displays with high resolution the multisampling has almost no effect.

Start with no multisampling and, if the performance is good, try the settings 2x or 4x multisampling. If there is no big difference with higher multisampling, use a lower setting.

TIP

Settings for multisampling are hardware-dependent

If the required multisampling settings are not possible from hardware side, information about it is available in the logfile.

NOTE

Using `sceneID` in the scene configuration

When using the same `sceneID` in the scene configuration, multiple state machines react to input handling at the same time.

To avoid that and to achieve that only one state machine reacts to input handling, assign different `sceneID` values to each state machine in the scene configuration.

13.10. Touch screen types supported by EB GUIDE GTF

The supported types depend on the target device.

Table 13.122. Touch screen types supported by EB GUIDE GTF

Value	Description	Platform
0	Galaxy	Linux
1	IMX WVGA	Linux
2	Touch screen connected to mouse interface	All
3	General platform-dependent touch-screen interface	All
4	Lilliput 889GL	QNX

Value	Description	Platform
5	General platform-dependent multitouch touch-screen interface	Linux

13.11. `tracing.json` configuration file

The `tracing.json` is an EB GUIDE TF configuration file which contains `Tracing` dependent items.

The `tracing.json` file is not part of the exported EB GUIDE model or of the standard delivery. Therefore, you need to create the `tracing.json` file on your PC.

NOTE

JSON object notation

For an example, see section 13.11.2, “Example `tracing.json`”.

For more information about the `JSON` format, see http://www.json.org.

In the following table you find all default configuration parameters.

Table 13.123. Tracing configuration items

Item	Type	Description	Default value
`gtf.tracing.Plugins`	string list	Tracing output plugins	Empty list, integrated output plugins are used such as `StdErr/StdErr/StdOut` for all platforms besides Android and `LogCat` for Android.
`gtf.tracing.ChannelSeverities`	string list	Custom severities set for specified channels	
`gtf.tracing.DefaultSeverity`	string	Default tracing severity For more information on the severity levels, see section 13.11.1, “Severity levels”.	`Notice`
`gtf.tracing.output.path`	boolean	The location to which the tracing output file `log.txt` is exported	The same directory where `tracing.json` is stored
`gtf.tracing.output.enabled`	boolean	Enables exporting tracing output files	`true`

Item	Type	Description	Default value
`gtf.tracing.config.path`	string	Path to the `tracing.json` file	
`gtf.tracing.gtffileoutput.severity`	string	Severity for the `GtfFileOutput` plugin	
`gtf.tracing.gtfkerneleventoutput.severity`	string	Severity for the `GtfKernelEventOutput` plugin	
`gtf.tracing.gtfvsdebugoutput.severity`	string	Severity for the `GtfVsDebugOutput` plugin	
`gtf.tracing.stdoutput.severity`	string	Severity for `StdErrOutput`	
`gtf.tracing.logcatoutput.severity`	string	Severity for `LogCatOutput`	
`gtf.tracing.output.printChannelname`	boolean	Enables printing the channel name inside the trace message. `output` from the configuration item name should be replaced with the tracing output name, e.g. `gtffileoutput`, `gtfkerneleventoutput`, `stdoutput`, `gtfvsdebugoutput`.	`true`
`gtf.tracing.output.printTimestamp`	boolean	Enables printing the time stamp inside the trace message. `output` from the configuration item name should be replaced with the tracing output name, e.g. `gtffileoutput`, `gtfkerneleventoutput`, `stdoutput`, `gtfvsdebugoutput`.	`true`
`gtf.tracing.output.printSeverity`	boolean	Enables printing the severity inside the trace	`true`

Item	Type	Description	Default value
		message. `output` from the configuration item name should be replaced with the tracing output name, e.g. `gtffileoutput`, `gtfkerneleventoutput`, `stdoutput`, `gtfvsdebugoutput`.	
`gtf.tracing.PeriodicOutputFlush`	integer	Value in milliseconds for periodic output flush. Value `0` means that flush is enforced after each trace. A value lower than `0` means that no flush is enforced at all.	`-1`

13.11.1. Severity levels

The following table lists the trace severity levels that you can use.

Table 13.124. Severity levels

Severity level	Description
`None`	The level indicates that the traces are disabled and are not created at all.
`Fatal`	The level indicates severe errors that may lead to the application abort.
`Error`	The level indicates the error events that might still allow the application to continue running.
`Warning`	The level indicates potentially harmful situations, or that some interfaces are not used as expected.
`Notice`	The level indicates information messages that describe the progress of an application at a high level.
`Info`	The level indicates information messages that roughly describe the progress of an application.
`Debug`	The level indicates information messages that describe the progress of an application in detail. This level is useful for debugging an application.

NOTE

Included levels

The levels that are listed in table 13.124, “Severity levels” automatically include information from the levels above, except for the `None` level. This means that if you use the `Error` severity level, the `Fatal` severities are also contained. And if you select the `Debug` severity level, all above mentioned severities, except for `None`, will be included.

13.11.2. Example `tracing.json`

Example 13.3.
`tracing.json`

```
{
    "gtf":
    {
        "tracing":
        {
            "Plugins": ["GtfFileoutput"],
            "ChannelSeverities":
            ["GTF_Launcher", "Warning",
            "GTF_PluginLoader", "Notice"],
            "DefaultSeverity": "Info"
        }
    }
}
```

For another example on using traces, see the EB GUIDE SDK examples. Download the `EB_GUIDE_Examples.zip` archive with all EB GUIDE SDK examples from https://www.elektrobit.com/ebguide/learn/resources/. For instructions on how to work with the EB GUIDE SDK examples, see the `EB GUIDE Studio Howto Using examples in EB GUIDE Studio.pdf` file enclosed in the `.zip` archive.

13.12. Widgets

13.12.1. View

Table 13.125. Properties of a view

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels

Property name	Description
`visible`	If true, the widget and its child widgets are visible
`x`	The x-coordinate of the widget
`y`	The y-coordinate of the widget

View states and view templates have additional properties for view transition animations. View transition animations apply for entry animations, exit animations, change animations, pop up on animations and pop up off animations.

Table 13.126. Properties of a view transition animation

Property name	Description
`enabled`	Defines whether the animation is executed.
`repeat`	The number of repetitions, `0` for infinite number.
`alternating`	If true, the animation is executed repeatedly back and forth, i.e. bidirectional. If false, the animation is executed repeatedly only in one direction, i.e. unidirectional. The number of repetitions is defined in the `repeat` property.
`scale`	The factor by which the animation time is multiplied.
`onPlay`	The reaction that is executed when the animation is started or continued. Parameters: Start time and play direction (true for forwards, false for backwards).
`onPause`	The reaction that is executed when the animation is paused. Parameter: Current animation time.
`onTerminate`	The reaction that is executed when the animation completes. First parameter: Animation time. Second parameter: Reason for the termination, encoded as follows: ▶ 0: Animation is completed ▶ 1: Animation is cancelled, triggered by `f:animation_cancel` ▶ 2: Widget is destroyed due to view transition ▶ 3: Animation jumps to its last step, triggered by `f:animation_cancel_end` ▶ 4: Animation jumps to its first step and is then canceled, triggered by `f:animation_cancel_reset`

13.12.2. Basic widgets

There are eight basic widgets.

- Alpha mask
- Animation
- Container
- Ellipse
- Image
- Instantiator
- Label
- Rectangle

The following sections list the properties of basic widgets.

NOTE

Unique names

Use unique names for two widgets with the same parent widget.

NOTE

Negative values

Do not use negative values for `height` and `width` properties. EB GUIDE Studio treats negative values as `0`, this means the respective widget will not be depicted.

13.12.2.1. Alpha mask

An alpha mask is a container widget that controls the alpha channel, i.e. the opacity, of its child widgets with an image.

Table 13.127. Properties of the alpha mask

Property name	Description
`visible`	If true, the widget and its child widgets are visible
`width`	The width of the widget in pixels
`height`	The height of the widget in pixels
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`enabled`	If true, the alpha mask is applied to the child widgets
`image`	The image that controls the alpha channel, i.e. the opacity of the child widgets

Property name	Description
`horizontalAlign`	The horizontal alignment of the image file within the boundaries of the widget
`verticalAlign`	The vertical alignment of the image file within the boundaries of the widget
`scaleMode`	The scale mode of the image. Possible values: ▶ `original size (0)` ▶ `fit to size (1)` ▶ `keep aspect ratio (2)`

NOTE

Supported image file types for alpha mask

The available image formats depend on the implementation of the renderer. The renderers for OpenGL ES 2.0 or higher support `.png` files and `.jpg` files. RGB images are converted to grayscale images before being used as alpha masks. Grayscale images are used as is. The alpha channel in the image is ignored.

Alpha mask functionality is not applied to 9-patch images. 9-patch images are handled the same way the PNG and JPEG file formats are.

13.12.2.2. Animation

An animation defines the movement of a widget along a view. To define the appearance of an animation, add curves in the **Animation** editor.

Table 13.128. Properties of the animation

Property name	Description
`enabled`	Defines if the animation is executed
`repeat`	The number of repetitions, `0` for infinite number
`alternating`	If true, the animation is executed repeatedly back and forth / bidirectional. If false, the animation is executed repeatedly only in one direction / unidirectional. The number of repetitions is defined in the `repeat` property.
`scale`	The factor by which the animation time is multiplied
`onPlay`	The reaction that is executed when the animation is started or continued. Parameters: Start time and play direction (true for forwards, false for backwards)
`onPause`	The reaction that is executed when the animation is paused. Parameter: Current animation time.

Property name	Description
`onTerminate`	The reaction that is executed when the animation completes. First parameter: Animation time. Second parameter: Reason for the termination, encoded as follows: ▶ 0: Animation is completed ▶ 1: Animation is cancelled, triggered by `f:animation_cancel` ▶ 2: Widget is destroyed due to view transition ▶ 3: Animation jumps to its last step, triggered by `f:animation_cancel_end` ▶ 4: Animation jumps to its first step and is then canceled, triggered by `f:animation_cancel_reset`

13.12.2.2.1. Constant curve

A constant curve sets a target value after a defined delay. Constant curves are available for integer, boolean, float, and color types.

Table 13.129. Properties of the constant curve

Property name	Description
`enabled`	Defines if the animation is executed
`delay`	The delay in ms relative to the animation start
`duration`	The duration of the curve segment in ms
`repeat`	The number of repetitions, with `0` for endless repetitions
`alternating`	Defines if the animation is executed repeatedly
`relative`	Defines if update values are applied on the initial value
`value`	The resulting constant value
`target`	The target property the resulting value is assigned to

13.12.2.2.2. Fast start curve

A fast start curve periodically sets a value that increases fast in the beginning but loses speed constantly until the end. Fast start curves are available for integer, float, and color types.

Table 13.130. Properties of the fast start curve

Property name	Description
`enabled`	Defines if the animation is executed

Property name	Description
delay	The delay in ms relative to the animation start
duration	The duration of the curve segment in ms
repeat	The number of repetitions, with 0 for endless repetitions
alternating	Defines if the animation is executed repeatedly
relative	Defines if update values are applied on the initial value
start	The initial value
end	The final value
target	The target property the resulting value is assigned to

13.12.2.2.3. Slow start curve

A slow start curve periodically sets a value that increases slowly in the beginning but rises constantly until the end. Slow start curves are available for integer, float, and color types.

Table 13.131. Properties of the slow start curve

Property name	Description
enabled	Defines if the animation is executed
delay	The delay in ms relative to the animation start
duration	The duration of the curve segment in ms
repeat	The number of repetitions, with 0 for endless repetitions
alternating	Defines if the animation is executed repeatedly
relative	Defines if update values are applied on the initial value
start	The initial value
end	The final value
target	The target property the resulting value is assigned to

13.12.2.2.4. Quadratic curve

A quadratic curve periodically sets a value using a quadratic function curve. Quadratic curves are available for integer, float, and color types.

Table 13.132. Properties of the quadratic curve

Property name	Description
enabled	Defines if the animation is executed
delay	The delay in ms relative to the animation start

Property name	Description
`duration`	The duration of the curve segment in ms
`repeat`	The number of repetitions, with `0` for endless repetitions
`alternating`	Defines if the animation is executed repeatedly
`relative`	Defines if update values are applied on the initial value
`acceleration`	The acceleration of the curve
`velocity`	The velocity to calculate the result
`constant`	The constant value to calculate the result
`target`	The target property the resulting value is assigned to

13.12.2.2.5. Sinus curve

A sinus curve periodically sets a value using a sinus function curve. Sinus curves are available for integer, float, and color types.

Table 13.133. Properties of the sinus curve

Property name	Description
`enabled`	Defines if the animation is executed
`delay`	The delay in ms relative to the animation start
`duration`	The duration of the curve segment in ms
`repeat`	The number of repetitions, with `0` for endless repetitions
`alternating`	Defines if the animation is executed repeatedly
`relative`	Defines if update values are applied on the initial value
`amplitude`	The amplitude of the sinus curve
`constant`	The constant value to calculate the result
`frequency`	The frequency of the curve in hertz
`phase`	The angular phase translation in radians
`target`	The target property the resulting value is assigned to

13.12.2.2.6. Script curve

The script curve is a curve that you can define yourself through EB GUIDE Script. Use the script curve in cases where you want to have an animation that is not possible with the other curves or that is your own, custom animation. This curve is especially useful if you want to have a customized trajectory for the movement of a widget. Script curves are available for integer, boolean, float, and color types.

Table 13.134. Properties of the script curve

Property name	Description
`enabled`	Defines if the animation is executed
`delay`	The delay in ms relative to the animation start
`duration`	The duration of the curve segment in ms
`repeat`	The number of repetitions, with `0` for endless repetitions
`alternating`	Defines if the animation is executed repeatedly
`relative`	Defines whether update values are applied on the initial value
`curve`	Defines your curve function in EB GUIDE Script. Provides two parameters: ▶ `diff`: The time in ms since the last execution. At the start of the animation `diff` is 0. ▶ `t_anim`: The time in ms since the start of the animation.
`target`	The target property the resulting value is assigned to

13.12.2.2.7. Linear curve

A linear curve periodically sets a value using a linear progression curve. Linear curves are available for integer, float, and color types.

Table 13.135. Properties of the linear curve

Property name	Description
`enabled`	Defines if the animation is executed
`delay`	The delay in ms relative to the animation start
`duration`	The duration of the curve segment in ms
`repeat`	The number of repetitions, with `0` for endless repetitions
`alternating`	Defines if the animation is executed repeatedly
`relative`	Defines if update values are applied on the initial value
`velocity`	The velocity to calculate the result
`constant`	The constant value to calculate the result
`target`	The target property the resulting value is assigned to

13.12.2.2.8. Linear interpolation curve

A linear interpolation curve widget periodically sets a value using a linear interpolation curve. Linear interpolation curves are available for integer, float, and color types.

NOTE

Linear key value interpolation curve

During import of a 3D graphic file, if the imported 3D scene has animations, linear key value interpolation integer curve and linear key value interpolation float curve are created. The underlying key-value pairs of these curves cannot be modified in EB GUIDE Studio.

Table 13.136. Properties of the linear interpolation curve

Property name	Description
`enabled`	Defines if the animation is executed
`delay`	The delay in ms relative to the animation start
`duration`	The duration of the curve segment in ms
`repeat`	The number of repetitions, with `0` for endless repetitions
`alternating`	Defines if the animation is executed repeatedly
`relative`	Defines if update values are applied on the initial value
`target`	The target property the resulting value is assigned to

13.12.2.3. Container

A container holds several widgets as child widgets and thus groups the widgets.

Table 13.137. Properties of the container

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels
`visible`	If true, the widget and its child widgets are visible
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget

13.12.2.4. Ellipse

An ellipse draws a colored ellipse with the dimensions and coordinates of the widget into a view. The widget can also be used to draw a sector or an arc.

Table 13.138. Properties of the ellipse

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels

Property name	Description
`visible`	If true, the widget and its child widgets are visible
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`fillColor`	The color that fills the ellipse
`arcWidth`	The width of the arc of the ellipse
`centralAngle`	The angle in degrees which defines a sector of the ellipse
`sectorRotation`	The angle in degrees which describes the rotation of the ellipse's sector

13.12.2.5. Image

An image places a picture into a view.

Table 13.139. Properties of the image

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels
`visible`	If true, the widget and its child widgets are visible
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`image`	The image the widget displays
`sRGB`	If this property is enabled, the image that is selected in `image`, is rendered using sRGB color space. Note that to use sRGB functionality, in the project center under **Configure** > **Profiles** for the `colorMode` property select `32-bit sRGB (=4)` or `32-bit sRGB (Emulated) (=5)`.
`horizontalAlign`	The horizontal alignment of the image file within the boundaries of the widget
`verticalAlign`	The vertical alignment of the image file within the boundaries of the widget

NOTE

Supported image file types

The available image formats depend on the implementation of the renderer. The renderers for OpenGL ES 2.0 or higher support `.png` files and `.jpg` files.

13.12.2.6. Instantiator

An instantiator creates widget instances during run-time. You can use the instantiator to model lists or tables with dynamic or static content. The child widgets of an instantiator serve as line templates for the list or table which is created during run-time. By default the instantiator only instantiates the first line template.

Table 13.140. Properties of the instantiator

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels
`visible`	If true the widget and its child widgets are visible
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`numItems`	The number of instantiated child widgets. If `numItems` is `0`, no child widgets are created.
`lineMapping`	Defines which child widget is the line template for which line, i.e. defines the order of instantiation

13.12.2.7. Label

A label places text into a view.

Table 13.141. Properties of the label

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels
`visible`	If true, the widget and its child widgets are visible
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`text`	The text the label displays. If the text does not fit into the widget area it is truncated at the end by default.
`textColor`	The color in which the text is displayed
`font`	The font in which the text is displayed
`horizontalAlign`	The horizontal alignment of the text within the boundaries of the label
`verticalAlign`	The vertical alignment of the text within the boundaries of the label

13.12.2.8. Rectangle

A rectangle draws a colored rectangle with the dimensions and coordinates of the widget into a view.

Table 13.142. Properties of the rectangle

Property name	Description
`height`	The height of the widget in pixels
`width`	The width of the widget in pixels
`visible`	If true, the widget and its child widgets are visible
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`fillColor`	The color that fills the rectangle

13.12.3. 3D widgets

13.12.3.1. Ambient light

An ambient light is a light that uniformly illuminates the scene. An ambient light affects the `ambient` color property of material, PBR GGX material, and PBR Phong material.

Table 13.143. Properties of the ambient light

Property name	Description
`enabled`	If true, the widget is enabled
`color`	The color of the light
`intensity`	The intensity of the light, with the lower limit value `0.0` as no ambient light

13.12.3.2. Camera

A camera defines the view of the scene from a particular point of view. Use several cameras to show the scene from different points of view.

Table 13.144. Properties of the camera

Property name	Description
`enabled`	If true, the widget is enabled
`nearPlane`	The nearest distance from the camera in view direction at which the scene becomes visible. The measurement unit is defined when you create a 3D model in third-party 3D modeling software.

Property name	Description
`farPlane`	The farthest distance from the camera in view direction up to which the scene is visible. The measurement unit is defined when you create a 3D model in third-party 3D modeling software.
`fieldOfView`	The camera's vertical viewing angle in degrees, with the maximum value of 180
`projectionType`	Defines the projection type of the camera. The objects are rendered either with `perspective (=0)` or `orthographic (=1)` projection. If the projection type is orthographic, the viewing volume is calculated by using the `fieldOfView` angle.

13.12.3.3. Directional light

A directional light illuminates the scene from one direction.

Table 13.145. Properties of the directional light

Property name	Description
`enabled`	If true, the widget is enabled
`color`	The light's color
`intensity`	The intensity of the light, with the lower limit value `0.0` as no directional light

13.12.3.4. Image-based light

An image-based light is a light that illuminates the scene by lighting information of the real world that was stored in a `.pfm` or `.hdr` file. The `.pfm` or `.hdr` files serve as input data for the IBLGenerator to create an `.ebibl` file.

Table 13.146. Properties of the image-based light

Property name	Description
`enabled`	If true, the widget is enabled
`ibl`	The IBL file `.ebibl` created manually.
`intensity`	The intensity of the light, with `0.0` as no image-based light

13.12.3.5. Material

A material defines the visual appearance of the mesh surface using the Phong reflection model.

Table 13.147. Properties of the material

Property name	Description
`ambient`	The color that the object reflects when it is illuminated by ambient light. If no ambient light is added to the parent scene graph, this property has no effect.
`diffuse`	The color that the object reflects evenly in all directions when it is illuminated by pure white light. If the **Diffuse texture** widget feature is added, this property has no effect.
`emissive`	The self-illumination color of the object. If the **Emissive texture** widget feature is added, this property has no effect.
`shininess`	The shininess factor Note that only values between `0.0` and `1.0`, as for example `0.3`, are valid. When the **Shininess texture** widget feature is used, the `shininess` property is ignored.
`specular`	The color that an object with a shiny surface reflects. If the **Specular texture** widget feature is added or the `shininess` property is set to `0.0`, the `specular` property has no effect.
`opacity`	The opacity value Note that only values between `0.0` and `1.0`, as for example `0.3`, are valid.

13.12.3.6. Mesh

A mesh defines the shape of the 3D object.

Table 13.148. Properties of the mesh

Property name	Description
`visible`	If true, the widget and its child widgets are visible
`mesh`	The automatically created mesh file `*.ebmesh`
`culling`	Defines whether no triangles (0), only front-facing triangles (1), or only back-facing triangles (2) are culled from the mesh

13.12.3.7. PBR GGX material

A PBR GGX material defines the visual appearance of the mesh surface using the physically correct Cook-Torrance model.

Table 13.149. Properties of the PBR GGX material

Property name	Description
`ambient`	The color that the object reflects when it is illuminated by ambient light. If the **Ambient texture** widget feature is added, this property has no effect.
`diffuse`	The color that the object reflects evenly in all directions when it is illuminated by pure white light. If the **Diffuse texture** widget feature is added, this property has no effect.
`emissive`	The self-illumination color of the object. If the **Emissive texture** widget feature is added, this property has no effect.
`specular`	The color that an object with a shiny surface reflects. If the **Specular texture** widget feature is added or the `shininess` property is set to `0.0`, the `specular` property has no effect.
`metallic`	The value for the surface quality of being metallic This value interpolates between the diffuse and the specular contribution. Note that only values between `0` and `1` are valid, as for example `0.3`.
`roughness`	The value for the surface quality of being rough This value controls the surface’s microstructure. Note that only values between `0` and `1` are valid, as for example `0.3`.
`opacity`	The opacity value Note that only values between `0` and `1` are valid, as for example `0.3`.

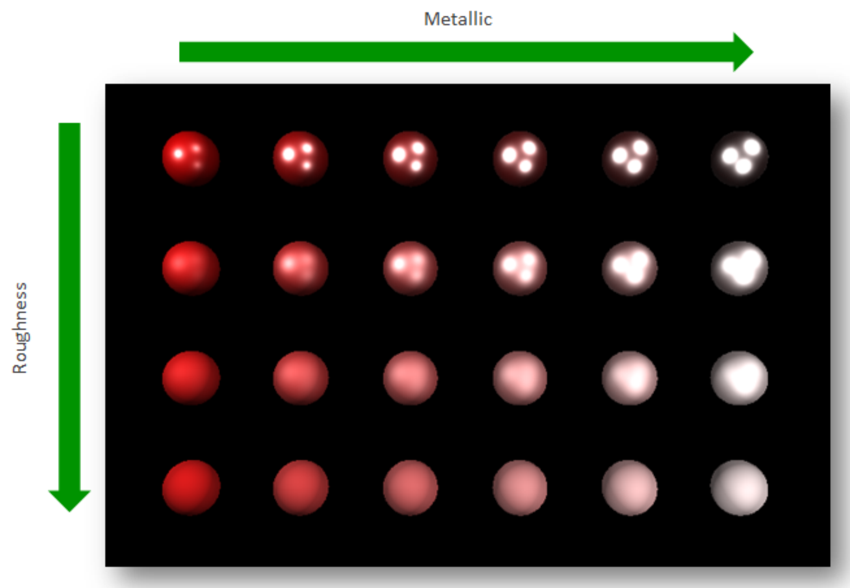


Figure 13.1. Example for a physically-based material

13.12.3.8. PBR Phong material

A PBR Phong material defines the visual appearance of the surface of the mesh using the physically correct Phong reflection model.

Table 13.150. Properties of the PBR Phong material

Property name	Description
`ambient`	The color that the object reflects when it is illuminated by ambient light. If the **Ambient texture** widget feature is added, this property has no effect.
`diffuse`	The color that the object reflects evenly in all directions when it is illuminated by pure white light. If the **Diffuse texture** widget feature is added, this property has no effect.
`emissive`	The self-illumination color of the object. If the **Emissive texture** widget feature is added, this property has no effect.
`shininess`	The shininess factor
`specular`	The color that an object with a shiny surface reflects. If the **Specular texture** widget feature is added or the `shininess` property is set to `0.0`, the `specular` property has no effect.
`metallic`	The value for the surface quality of being metallic This value interpolates between the diffuse and the specular contribution.

Property name	Description
	Note that only values between `0` and `1` are valid, as for example `0.3`.
`opacity`	The opacity value Note that only values between `0` and `1` are valid , as for example `0.3`.

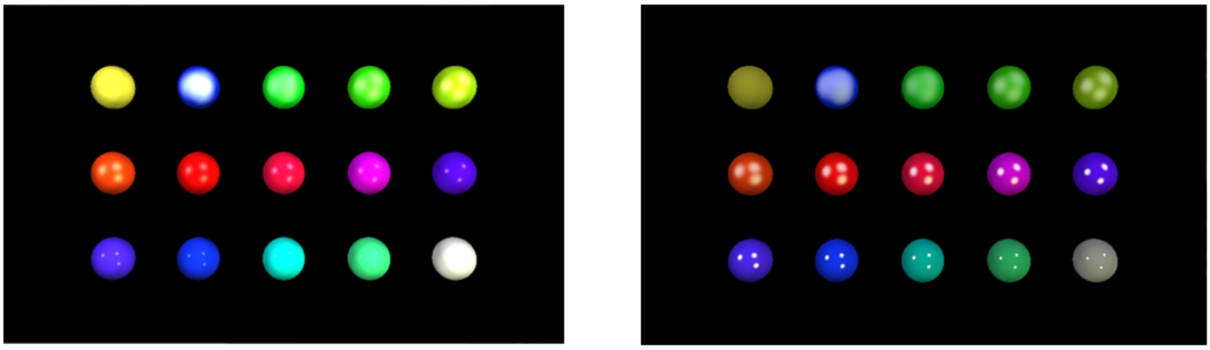


Figure 13.2. Example for a non-normalized material (left) and a normalized material (right)

13.12.3.9. Point light

A point light adds a light to the scene that emits light in all directions like a light bulb.

Table 13.151. Properties of the point light

Property name	Description
`enabled`	If true, the widget is enabled
`color`	The light's color
`intensity`	The intensity of the light, with the lower limit value `0.0` as no point light and the upper limit value depending on attenuation factors
`attenuationConstant`	The constant factor by which the light intensity weakens with increasing distance. The `0.0` value means that the factor is not used.
`attenuationLinear`	The linear factor by which the light intensity weakens with increasing distance. The `0.0` value means that the factor is not used.
`attenuationQuadratic`	The quadratic factor by which the light intensity weakens with increasing distance. The `0.0` value means that the factor is not used.

13.12.3.10. Scene graph

A scene graph places a 3D object into a view.

Table 13.152. Properties of the scene graph

Property name	Description
`visible`	If true, the widget and its child widgets are visible
`width`	The width of the widget in pixels
`height`	The height of the widget in pixels
`x`	The x-coordinate of the widget relative to its parent widget
`y`	The y-coordinate of the widget relative to its parent widget
`gamma`	Corrects the luminance output of the scene graph. The default value is set to 2.-2.

13.12.3.11. Scene graph node

A scene graph node is a child node and is added to the scene graph or to another scene graph node. You use scene graph nodes to place 3D widgets in the 3D scene with transformation properties. You can add the following 3D widgets to the scene graph node:

- Camera
- Directional light
- Image-based light
- Mesh
- Point light
- Spot light

Table 13.153. Properties of the scene graph node

Property name	Description
`visible`	If true, the widget and its child widgets are visible
`rotationX`	The rotation around the x-axis
`rotationY`	The rotation around the y-axis
`rotationZ`	The rotation around the z-axis
`scalingX`	The scaling along the x-axis
`scalingY`	The scaling along the y-axis
`scalingZ`	The scaling along the z-axis
`translationX`	The translation along the x-axis
`translationY`	The translation along the y-axis

Property name	Description
`translationZ`	The translation along the z-axis

13.12.3.12. Spot light

A spot light adds a light which restricts illumination to a cone of influence.

Table 13.154. Properties of the spot light

Property name	Description
`enabled`	If true, the widget is enabled
`color`	The light's color
`intensity`	The intensity of the light, with the lower limit value `0.0` as no spot light and the upper limit value depending on attenuation factors
`attenuationConstant`	The constant factor by which the light intensity weakens with increasing distance
`attenuationLinear`	The linear factor by which the light intensity weakens with increasing distance
`attenuationQuadratic`	The quadratic factor by which the light intensity weakens with increasing distance
`coneAngleInner`	The light's inner cone angle in degrees, with the maximum value of 180
`coneAngleOuter`	The light's outer cone angle in degrees, with the maximum value of 180

13.13. Widget features

The following list contains a description of all widget features that are implemented, with a brief description on how to use them in an EB GUIDE model.

13.13.1. Common

13.13.1.1. Child visibility selection

The **Child visibility selection** widget feature handles the visibility of child widgets. You can define a single widget to be visible or you can define groups of child widgets to be visible at the same time. To define groups, map the index of child widgets to the same group value.

Table 13.155. Properties of the **Child visibility selection** widget feature

Property name	Description
`containerIndex`	Controls the visibility of child widgets. If `containerMapping` is not filled, `containerIndex` makes a single child widget visible. The child widget that is visible is identified by its order in the widget tree. The topmost child has `containerIndex` 0, next `containerIndex` 1 etc. If `containerMapping` is filled, `containerIndex` refers to a group of child widgets. Define the group in `containerMapping`.
`containerMapping`	Use this property to create groups of child widgets. The **Index** column identifies the child widget. The **Value** column defines the group. The number of rows must match the number of child widgets. Otherwise the mapping is not used.

13.13.1.2. Enabled

The **Enabled** widget feature adds an `enabled` property to a widget.

Table 13.156. Properties of the **Enabled** widget feature

Property name	Description
`enabled`	If true, the widget reacts on touch and press input

13.13.1.3. Focused

The **Focused** widget feature enables a widget to have input focus.

Table 13.157. Properties of the **Focused** widget feature

Property name	Description
`focusable`	Defines whether the widget receives the focus or not. Possible values: ▶ `not focusable (=0)` ▶ `only by touch (=1)` ▶ `only by key (=2)` ▶ `focusable (=3)`

Property name	Description
`focused`	If true, the widget has focus

13.13.1.4. Multiple lines

The **Multiple lines** widget feature enables line breaks.

Restrictions:

- The **Multiple lines** widget feature is only available for the label widget.

Table 13.158. Properties of the **Multiple lines** widget feature

Property name	Description
`lineGap`	The size of the gap between the lines. A negative value decreases the gap, a positive value increases the gap. When the `line gap` is too small (high negative value), it has no effect anymore and the text is rendered in one line. This occurs for example, when the font style is set to `PT_Sans_Narrow`, size is set to `30` and the `line gap` is defined as `-50`.
`maxLineCount`	The maximum number of visible lines. 0 = no limitation

TIP

Number of lines used

With the script function `getLineCount`, you can obtain the number of lines of the text.

For more information on this, see section 13.4.3.2.28, “`getLineCount`”.

NOTE

Character replacement

Sequences of '\\' '\\' are replaced by '\\' . Sequences of '\\' 'n' are replaced by '\n'.

If the size of the label is increased so that one line is sufficient to display the text, '\n' is replaced by ' '.

13.13.1.5. Pressed

The **Pressed** widget feature defines that a widget can be pressed.

Restrictions:

- Adding the **Pressed** widget feature automatically adds the **Focused** widget feature.

Table 13.159. Properties of the **Pressed** widget feature

Property name	Description
`pressed`	If true, a key is pressed while the widget is focused

Combining the **Touched** widget feature with the **Touch pressed** widget feature allows modeling a push button.

13.13.1.6. Selected

The **Selected** widget feature adds a `selected` property to a widget. It is typically set by the application or the HMI modeler. It is not changed by any other component of the framework.

Table 13.160. Properties of the **Selected** widget feature

Property name	Description
`selected`	If true, the widget is selected

13.13.1.7. Selection group

The **Selection group** widget feature is used to model a list of radio buttons. In the list, every radio button has the **Selection group** widget feature and a unique button ID.

Use a datapool item for the `buttonValue` property. Assign the datapool item to all widgets in the radio button array.

Selecting and deselecting a widget within the button group can be done by an application that sets the `buttonValue` property. Alternatively, changes can be triggered by touch or key input as well as by adding a condition that sets the button value.

Restrictions:

- Adding the **Selection group** widget feature automatically adds the **Selected** widget feature.

Table 13.161. Properties of the **Selection group** widget feature

Property name	Description
`buttonId`	The ID that identifies a button within a button group
`buttonValue`	The current value of a button. If this value matches the `buttonId`, the button is selected.
`selected`	Evaluates if `buttonID` and `buttonValue` are identical. If true, the button is selected.

13.13.1.8. Spinning

The **Spinning** widget feature turns a widget into a rotary button. A widget with the **Spinning** widget feature reacts to increment and decrement events by changing an internal value. The **Spinning** widget feature can be used to create a scale, a progress bar, or a widget with a preview value.

Table 13.162. Properties of the **Spinning** widget feature

Property name	Description
`currentValue`	The current rotary value
`maxValue`	The maximum value for the `currentValue` property
`minValue`	The minimum value for the `currentValue` property
`incValueTrigger`	If true, the `currentValue` property is incremented by 1
`incValueReaction`	The reaction to an incrementation of the `currentValue` property
`decValueTrigger`	If true, the current value is decremented by 1
`decValueReaction`	Reaction to a decrementation of the `currentValue` property
`steps`	The number of steps to calculate the increment or decrement for the `currentValue` property
`valueWrapAround`	Possible values: ▶ true: The `currentValue` property continues at the inverse border, if `minValue` or `maxValue` is exceeded. ▶ false: The `currentValue` property does not decrease/increase, if `minValue` or `maxValue` is exceeded.

13.13.1.9. Text truncation

The **Text truncation** widget feature truncates the content of the `text` property if it does not fit into the widget area. The widget feature enables a different truncation than the default setting `trailing`.

Restrictions:

- The **Text truncation** widget feature is only available for the label widget.

Table 13.163. Properties of the **Text truncation** widget feature

Property name	Description
`truncationPolicy`	For single-line texts, the `truncationPolicy` property defines the position of the truncation. Possible values: ▶ `leading (=0)`: Text is replaced at the beginning of the text ▶ `trailing (=1)`: Text is replaced at the end of the text

Property name	Description
	For multi-line texts, the `truncationPolicy` property defines where text is replaced. Possible values: ▶ `leading (=0)`: Lines at the beginning are replaced and text of the first visible line is truncated at the beginning of the text. ▶ `trailing (=1)` Lines at the end are replaced and text of the last visible line is truncated at the end of the text.
`truncationSymbol`	The string that is shown instead of the replaced text part

13.13.1.10. Touched

The **Touched** widget feature enables a widget to react to touch input.

Table 13.164. Properties of the **Touched** widget feature

Property name	Description
`touchable`	If true, the widget reacts on touch input
`touched`	If true, the widget is currently touched
`touchPolicy`	Defines how to handle touch and movement that crosses widget boundaries. Possible values: ▶ `Press then react (=0)`: Press first, then the widget reacts. Notifications of moving and releasing are only active within the widget area. ▶ `Press and grab (=1)`: Press to grab the contact. The contact remains grabbed even if it moves away from the widget area. ▶ `Press then react on contact (=2)`: Even if the contact enters the pressed state outside the widget boundaries, the subsequent move and release events are delivered to the widget.
`touchBehavior`	Defines touch evaluation. Possible values: ▶ `Whole area (=0)`: To identify the touched widget, the renderer evaluates the widget's clipping rectangle. ▶ `Visible pixels (=1)`: To identify the touched widget, the renderer evaluates the widget the touched pixel belongs to. Transparent pixels in an image with alpha transparency or pixels inside letters such as in O or A are not touchable. Note that the `Visible pixels` value has no effect on labels.

Combining the **Touched** widget feature with the **Pressed** widget feature allows modeling a push button.

TIP

Performance recommendation

If performance is an important issue in your project, set the `touchBehavior` property to `Whole area (=0)`. EB GUIDE GTF evaluates `Whole area (=0)` faster than `Visible pixels (=1)`.

13.13.2. Effect

13.13.2.1. Border

The **Border** widget feature adds a configurable border to the widget. The border starts at the widget boundaries and is placed within the widget.

Restrictions:

- The widget feature is available for rectangles.

Table 13.165. Properties of the **Border** widget feature

Property name	Description
`borderThickness`	The thickness of the border in pixels
`borderColor`	The color that is used to render the border
`borderStyle`	The style that is used to render the border

13.13.2.2. Coloration

The **Coloration** widget feature colors the widget and its widget subtree. It also affects transparency if the alpha value is not opaque.

Example 13.4.
Usage of the Coloration widget feature

For all colors with RGBA components between 0.0 and 1.0, the algorithm in the **Coloration** widget feature multiplies the current color values of a widget by the `colorationColor` property value. Multiplication is done per pixel and component-wise.

A semi-transparent gray colored by an opaque blue results in semi-transparent darker blue as follows:

```
(0.5, 0.5, 0.5, 0.5) * (0.0, 0.0, 1.0, 1.0) = (0.0, 0.0, 0.5, 0.5)
```

Table 13.166. Properties of the **Coloration** widget feature

Property name	Description
`colorationEnabled`	If true, coloration is used
`colorationColor`	The color used for the coloration

13.13.2.3. Stroke

The **Stroke** widget feature activates a configurable text outline, i.e. a label border.

Restrictions:

- The widget feature is available for labels.

Table 13.167. Properties of the **Stroke** widget feature

Property name	Description
`strokeEnabled`	If true, stroke is used
`strokeThickness`	The thickness of the outline in pixels
`strokeColor`	The color that is used to render the outline

13.13.3. Focus

The **Focus** widget feature category provides the widget features relating to focus management.

13.13.3.1. Auto focus

With the **Auto focus** widget feature, the order in which child widgets are focused is pre-defined. The **Auto focus** widget feature checks the widget subtree for child widgets with the `focusable` property.

The order of the widgets in the layout is used to calculate focus order. Depending on layout orientation, the algorithm begins in the upper left or upper right corner.

Restrictions:

- The widget feature **Auto focus** automatically adds the **Focused** widget feature.

Table 13.168. Properties of the **Auto focus** widget feature

Property name	Description
`focusNext`	The condition on which the focus index is incremented
`focusPrev`	The condition on which the focus index is decremented
`focusFlow`	The behavior for focus changes within the hierarchy. Possible values: ▶ `stop at hierarchy (=0)` ▶ `wrap within hierarchy level (=1)` ▶ `step up in hierarchy (=2)`
`focusedIndex`	The index of the currently focused child widget as the n-th child widget which is focusable
`initFocus`	The index defines the focused child widget at initialization. If the widget is not focusable, the next focusable child is used.

13.13.3.2. User-defined focus

The **User-defined focus** widget feature enables additional focus functionality for the widget. A widget that uses the feature manages a local focus hierarchy for its widget subtree.

Restrictions:

- The widget feature **User-defined focus** automatically adds the **Focused** widget feature.

Table 13.169. Properties of the **User-defined focus** widget feature

Property name	Description
`focusNext`	The trigger that assigns the focus to the next child widget
`focusOrder`	The `focusOrder` property makes it possible to skip child widgets when assigning focus. The ID of a child widget corresponds to its position in the subtree. Child widgets that are not focusable are skipped by default. Order in which the child widgets are focused: ▶ defined: User-defined widget order is used ▶ not defined: Default widget order is used instead Each child widget requires the **Focused** widget feature, otherwise widgets are ignored for focus management. Example: focusOrder=1|0|2 means the second widget receives focus first, then the first widget receives focus, and finally the third widget.
`focusPrev`	The trigger that assigns the focus to the previous child

Property name	Description
`focusFlow`	The behavior for focus changes within the hierarchy. Possible values: ▶ `stop at hierarchy level (=0)` ▶ `wrap within hierarchy level (=1)` ▶ `step up in hierarchy (=2)`
`focusedIndex`	The index defines the position of the child widget in the `focusOrder` list. If the widget is not focusable, the child next in the list is used.
`initFocus`	The index of the focused child widget at initialization

13.13.4. Gestures

13.13.4.1. Flick gesture

A quick brush of a contact over a surface

Restrictions:

- Adding the **Flick gesture** widget feature automatically adds the **Gestures** and **Touched** widget features.

Table 13.170. Properties of the **Flick gesture** widget feature

Property name	Description
`onGestureFlick`	The reaction that is triggered once the gesture is recognized Reaction arguments: ▶ `speed`: relative speed of the flick gesture Speed in pixels/ms divided by `flickMinLength`/`flickMaxTime` ▶ `directionX`: x-part of the direction vector of the gesture ▶ `directionY`: y-part of the direction vector of the gesture
`flickMaxTime`	The maximal time in milliseconds the contact may stay in place for the gesture to be recognized as a flick gesture
`flickMinLength`	The minimal distance in pixels a contact has to move on the surface to be recognized as a flick gesture

13.13.4.2. Hold gesture

A hold gesture without movement

Restrictions:

- Adding the **Hold gesture** widget feature automatically adds the **Gestures** and **Touched** widget features.
- The **Hold gesture** widget feature does not trigger the **Touch lost** widget feature.

Table 13.171. Properties of the **Hold gesture** widget feature

Property name	Description
`onGestureHold`	The reaction that is triggered once the gesture is recognized. The reaction is triggered only once per contact: when `holdDuration` is expired and the contact still is in a small boundary box around the initial touch position. Reaction arguments: - `x`: x-coordinate of the contact position - `y`: y-coordinate of the contact position
`holdDuration`	The minimal time in milliseconds the contact must stay in place for the gesture to be recognized as a hold gesture

13.13.4.3. Long hold gesture

A long hold gesture without movement

Restrictions:

- Adding the **Long hold gesture** widget feature automatically adds the **Gestures** and **Touched** widget features.
- The **Long hold gesture** widget feature does not trigger the **Touch lost** widget feature.

Table 13.172. Properties of the **Long hold gesture** widget feature

Property name	Description
`onGestureLongHold`	The reaction that is triggered once the gesture is recognized. The reaction is triggered only once per contact: when `longHoldDuration` has expired and the contact still is in a small boundary box around the initial touch position. Reaction arguments: - `x`: x-coordinate of the contact position - `y`: y-coordinate of the contact position
`longHoldDuration`	The minimal time in milliseconds the contact must stay in place for the gesture to be recognized as a long hold gesture

13.13.4.4. Path gestures

A shape drawn by one contact is matched against a set of known shapes.

Restrictions:

- Adding the **Path gesture** widget feature automatically adds the **Gestures** and **Touched** widget features.

Table 13.173. Properties of the **Path gesture** widget feature

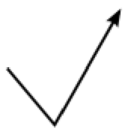
Property name	Description
`onPath`	The reaction that is triggered when the entered shape matches. The reaction is only triggered if `onPathStart` has been triggered already. Reaction argument: ▶ `gestureId`: ID of the path that was matched
`onPathStart`	The reaction that is triggered once a contact moves beyond the minimal box (`pathMinXBox`, `pathMinYBox`.)
`onPathNotRecognized`	The reaction that triggered when the entered shape does not match. The reaction is only triggered if `onPathStart` has been triggered already.
`pathMinXBox`	The x-coordinate of the minimal distance in pixels a contact must move so that the path gesture recognizer starts considering the input
`pathMinYBox`	The y-coordinate of the minimal distance in pixels a contact must move so that the path gesture recognizer starts considering the input

13.13.4.4.1. Gesture IDs

Gesture identifiers depend on the configuration of the path gesture recognizer. The following table shows an example configuration which is included in EB GUIDE.

Table 13.174. Path gesture samples configuration included in EB GUIDE

ID	Shape	Description
0		Roof shape left to right
1		Roof shape right to left

ID	Shape	Description
2		Horizontal line left to right
3		Horizontal line right to left
4		Check mark
5		Wave shape left to right
6		Wave shape right to left

13.13.4.5. Pinch gesture

Two contacts that move closer together or further apart

Restrictions:

- Adding the **Pinch gesture** widget feature automatically adds the **Gestures** and **Touched** widget features.

Table 13.175. Properties of the **Pinch gesture** widget feature

Property name	Description
`onGesturePinchStart`	The reaction that is triggered once the start of the gesture is recognized. Reaction arguments: - `ratio`: Current contact distance to initial contact distance ratio

Property name	Description
	`centerX`: x-coordinate of the current center point between the two contacts `centerY`: y-coordinate of the current center point between the two contacts
`onGesturePinchUpdate`	The reaction that is triggered when the pinch ratio or center point change. Reaction arguments: `ratio`: Current contact distance to initial contact distance ratio `centerX`: x-coordinate of the current center point between the two contacts `centerY`: y-coordinate of the current center point between the two contacts
`onGesturePinchEnd`	The reaction that is triggered once the gesture is finished. Reaction arguments: `ratio`: Current contact distance to initial contact distance ratio `centerX`: x-coordinate of the current center point between the two contacts `centerY`: y-coordinate of the current center point between the two contacts
`pinchThreshold`	The minimal distance in pixels each contact has to move from its initial position for the gesture to be recognized

13.13.4.6. Rotate gesture

Two contacts that move along a circle

Restrictions:

- Adding the **Rotate gesture** widget feature automatically adds the **Gestures** and **Touched** widget features.

Table 13.176. Properties of the **Rotate gesture** widget feature

Property name	Description
`onGestureRotateStart`	The reaction that is triggered once the start of the gesture is recognized
`onGestureRotateUpdate`	The reaction that is triggered when the recognized angle or center point changes
`onGestureRotateEnd`	The reaction that is triggered once the gesture is finished

Property name	Description
`rotateThreshold`	The minimal distance in pixels each contact has to move from its initial position for the start of the gesture to be recognized

Reaction arguments for `onGestureRotateEnd`, `onGestureRotateStart`, and `onGestureRotateUpdate`:

- `angle`: Angle between the line specified by the initial position of the two involved contacts and the line specified by the current position of the two contacts. The angle is measured counter-clockwise.
- `centerX`: x-coordinate of the current center point between the two contacts
- `centerY`: y-coordinate of the current center point between the two contacts

13.13.5. Input handling

13.13.5.1. Gestures

The **Gestures** widget feature enables the widget to react on touch gestures.

Restrictions:

- Adding the **Gestures** widget feature automatically adds the **Touched** widget feature.
- The **Gestures** widget feature has no additional properties.

13.13.5.2. Key pressed

The **Key pressed** widget feature enables a widget to react on a key being pressed.

Restrictions:

- Adding the **Key pressed** widget feature automatically adds the **Pressed** and **Focused** widget features.

Table 13.177. Properties of the **Key pressed** widget feature

Property name	Description
`keyPressed`	The widget's reaction on a key being pressed Reaction argument: - `keyId`: The ID of the key that is processed

13.13.5.3. Key released

The **Key released** widget feature enables a widget to react on a key being released.

Restrictions:

- Adding the **Key released** widget feature automatically adds the **Pressed** and **Focused** widget features.

Table 13.178. Properties of the **Key released** widget feature

Property name	Description
`keyShortReleased`	The widget's reaction on a key being released Reaction argument: ▶ `keyId`: The ID of the key that is processed

13.13.5.4. Key status changed

The **Key status changed** widget feature enables a widget to react on a key being pressed or released. It defines the reaction to key input such as **short press**, **long**, **ultra long** and **continuous**.

Restrictions:

- Adding the **Key status changed** widget feature automatically adds the **Pressed** and **Focused** widget features.

Table 13.179. Properties of the **Key status changed** widget feature

Property name	Description
`keyStatusChanged`	The widget's reaction on a key being pressed or released Reaction arguments: ▶ `keyId`: The ID of the key that is processed ▶ `status`: The numeric ID of the status change

13.13.5.5. Key unicode

The **Key unicode** widget feature enables a widget to react on Unicode key input.

Restrictions:

- Adding the **Key unicode** widget feature automatically adds the **Pressed** and **Focused** widget features.

Table 13.180. Properties of the **Key unicode** widget feature

Property name	Description
`keyUnicode`	The widget's reaction on a Unicode key input Reaction argument: ▶ `keyId`: The ID of the key that is processed

13.13.5.6. Move in

The **Move in** widget feature enables a widget to react on movement into its boundaries.

Restrictions:

- Adding the **Move in** widget feature automatically adds the **Touched** widget feature.

Table 13.181. Properties of the **Move in** widget feature

Property name	Description
`moveIn`	The widget's reaction on a movement into its boundaries Reaction arguments: ▶ `touchId`: The ID of the touch screen the user has clicked or released ▶ `x`: The x-coordinate ▶ `y`: The y-coordinate ▶ `fingerId`: The ID of the contact that moves across the widget

13.13.5.7. Move out

The **Move out** widget feature enables a widget to react on movement out of its boundaries.

Restrictions:

- Adding the **Move out** widget feature automatically adds the **Touched** widget feature.

Table 13.182. Properties of the **Move out** widget feature

Property name	Description
`moveOut`	The widget's reaction on a movement out of its boundaries Reaction arguments: ▶ `touchId`: The ID of the touch screen the user has clicked or released

Property name	Description
	- `x`: The x-coordinate - `y`: The y-coordinate - `fingerId`: The ID of the contact that moves across the widget

13.13.5.8. Move over

The **Move over** widget feature enables a widget to react on movement within its boundaries.

Restrictions:

- Adding the **Move over** widget feature automatically adds the **Touched** widget feature.

Table 13.183. Properties of the **Move over** widget feature

Property name	Description
`moveOver`	The widget's reaction on a movement within its boundaries Reaction arguments: - `touchId`: The ID of the touch screen the user has clicked or released - `x`: The x-coordinate - `y`: The y-coordinate - `fingerId`: The ID of the contact that moves across the widget

13.13.5.9. Moveable

The **Moveable** widget feature enables a widget to be moved by touch.

Restrictions:

- Adding the **Moveable** widget feature automatically adds the **Touched** and **Touch moved** widget features.

Table 13.184. Properties of the **Moveable** widget feature

Property name	Description
`moveDirection`	The direction into which the widget moves. Possible values: - `horizontal (=0)` - `vertical (=1)` - `free (=2)`

13.13.5.10. Rotary

The **Rotary** widget feature enables a widget to react on being rotated.

Restrictions:

- Adding the **Rotary** widget feature automatically adds the **Focused** widget feature.

Table 13.185. Properties of the **Rotary** widget feature

Property name	Description
`rotaryReaction`	The widget's reaction on being rotated. If true, the widget reacts on an incoming rotary event. Reaction arguments: - `rotaryId`: integer ID - `increment`: number of units the rotary input shifts when the incoming event is sent

13.13.5.11. Touch lost

The **Touch lost** widget feature enables a widget to react on a lost touch contact.

A contact can disappear when it is part of a gesture or leaves the touch screen without releasing. In these cases the `touchShortReleased` reaction is not executed.

Restrictions:

- Adding the **Touch lost** widget feature automatically adds the **Touched** widget feature.
- If you add **Touch lost**, in the `touchPolicy` drop-down box of the **Touched** widget feature, select `Press and grab`.

  Touch lost does not work with the other touch policies.

Table 13.186. Properties of the **Touch lost** widget feature

Property name	Description
`onTouchGrabLost`	The widget's reaction on a lost touch contact Reaction arguments: - `touchId`: The ID of the touch screen the user has clicked or released - `x`: The x-coordinate - `y`: The y-coordinate

Property name	Description
	▶ `fingerId`: The ID of the contact that moves across the widget

13.13.5.12. Touch move

The **Touch move** widget feature enables a widget to react on being touched and moved.

Restrictions:

- Adding the **Touch move** widget feature automatically adds the **Touched** widget feature.

Table 13.187. Properties of the **Touch move** widget feature

Property name	Description
`touchMoved`	The widget's reaction on being touched and moved Reaction arguments: ▶ `touchId`: The ID of the touch screen the user has clicked or released ▶ `x`: The x-coordinate ▶ `y`: The y-coordinate ▶ `fingerId`: The ID of the contact that moves across the widget

13.13.5.13. Touch pressed

The **Touch pressed** widget feature enables a widget to react on being pressed.

Restrictions:

- Adding the **Touch pressed** widget feature automatically adds the **Touched** widget feature.

Table 13.188. Properties of the **Touch pressed** widget feature

Property name	Description
`touchPressed`	The widget's reaction on being pressed Reaction arguments: ▶ `touchId`: The ID of the touch screen the user has clicked or released ▶ `x`: The x-coordinate ▶ `y`: The y-coordinate ▶ `fingerId`: The ID of the contact that moves across the widget

13.13.5.14. Touch released

The **Touch released** widget feature enables a widget to react on being released.

Restrictions:

- Adding the **Touch released** widget feature automatically adds the **Touched** widget feature.

Table 13.189. Properties of the **Touch released** widget feature

Property name	Description
`touchShortReleased`	The widget's reaction on being released Reaction arguments: ▶ `touchId`: The ID of the touch screen the user has clicked or released ▶ `x`: The x-coordinate ▶ `y`: The y-coordinate ▶ `fingerId`: The ID of the contact that moves across the widget

13.13.5.15. Touch status changed

The **Touch status changed** widget feature enables a widget to react on changes of its touch status.

Restrictions:

- Adding the **Touch status changed** widget feature automatically adds the **Touched** widget feature.

Table 13.190. Properties of the **Touch status changed** widget feature

Property name	Description
`touchStatusChanged`	The widget's reaction on changes of its touch status Reaction arguments: ▶ `touchId`: The ID of the touch screen the user has clicked or released ▶ `x`: The x-coordinate ▶ `y`: The y-coordinate ▶ `touchStatus`: The ID of the type of touch Possible values: ▶ `0`: new contact ▶ `1`: touch press ▶ `2`: touch move

Property name	Description
	- `3`: touch released - `4`: movement without touch - `5`: touch gone - `fingerId`: The ID of the contact that moves across the widget

13.13.6. Layout

13.13.6.1. Absolute layout

The **Absolute layout** widget feature of a parent widget defines the position and size of the child widgets. Invisible child widgets are ignored. The added widget feature properties consist of integer lists. Each list element is mapped to one child widget.

Restrictions:

- The **Absolute layout** widget feature excludes the following widget features:
  - **Box layout**
  - **Flow layout**
  - **Grid layout**
  - **List layout**

Table 13.191. Properties of the **Absolute layout** widget feature

Property name	Description
`itemLeftOffset`	An integer list that stores the offset from the left border for the child widgets. Each list element is mapped to a child widget.
`itemTopOffset`	An integer list that stores the offset from the top border for the child widgets. Each list element is mapped to a child widget.
`itemRightOffset`	An integer list that stores the offset from the right border for the child widgets. Each list element is mapped to a child widget.
`itemBottomOffset`	An integer list that stores the offset from the bottom border for the child widgets. Each list element is mapped to a child widget.

13.13.6.2. Box layout

The **Box layout** widget feature defines position and size of each child widget.

Position and size properties of child widgets are set by the parent widget. Invisible child widgets are ignored in the calculation.

Restrictions:

- The **Box layout** widget feature excludes the following widget features:
  - **Absolute layout**
  - **Flow layout**
  - **Grid layout**
  - **List layout**

Table 13.192. Properties of the **Box layout** widget feature

Property name	Description
`gap`	The space between two child widgets, depending on the layout direction
`layoutDirection`	The direction in which the list elements i.e. the child widgets are positioned. Possible values: - `horizontal (=0)` - `vertical (=1)`

13.13.6.3. Flow layout

The **Flow layout** widget feature defines position and size of each child widget.

Position and size properties of child widgets are set by the parent widget. Invisible child widgets are ignored in the calculation.

Restrictions:

- The **Flow layout** widget feature excludes the following widget features:
  - **Absolute layout**
  - **Box layout**
  - **Grid layout**
  - **List layout**

Table 13.193. Properties of the **Flow layout** widget feature

Property name	Description
`horizontalGap`	The horizontal space between two child widgets
`verticalGap`	The vertical space between two child widgets

Property name	Description
`layoutDirection`	The direction in which the list elements i.e. the child widgets are positioned. Possible values: ▶ `horizontal (=0)` ▶ `vertical (=1)`
`horizontalChildAlign`	The horizontal alignment of child widgets. Possible values: ▶ `leading (=0)`: The child widget is placed on the left side. ▶ `center (=1)`: The child widget is placed in the center. ▶ `trailing (=2)`: The child widget is placed on the right side.
`verticalChildAlign`	The vertical alignment of child widgets. Possible values: ▶ `center (=0)`: The child widget is placed in the center. ▶ `top (=1)`: The child widget is placed at the top ▶ `bottom (=2)`: The child widget is placed at the bottom.

13.13.6.4. Grid layout

The **Grid layout** widget feature defines position and size of each child widget.

Position and size properties of child widgets are set by the parent widget. Invisible child widgets are ignored in the calculation.

Restrictions:

- The **Grid layout** widget feature excludes the following widget features:
  - **Absolute layout**
  - **Box layout**
  - **Flow layout**
  - **List layout**

Table 13.194. Properties of the **Grid layout** widget feature

Property name	Description
`horizontalGap`	The horizontal space between two child widgets
`verticalGap`	The vertical space between two child widgets
`numRows`	Defines the number of rows
`numColumns`	Defines the number of columns

13.13.6.5. Layout margins

The **Layout margins** widget feature adds configurable margins to a widget that uses the **Flow layout**, **Absolute layout**, **Box layout**, or **Grid layout** widget feature.

Table 13.195. Properties of the **Layout margins** widget feature

Property name	Description
`leftMargin`	The margin of the left border
`topMargin`	The margin of the top border
`rightMargin`	The margin of the right border
`bottomMargin`	The margin of the bottom border

13.13.6.6. List layout

The **List layout** widget feature defines position and size of each child widget in pixels.

Position properties of child widgets and the `listIndex` property of the **List index** widget feature are set by the parent widget.

Best used in conjunction with instantiators to create the child widgets.

For details about the **List index** widget feature, see section 13.13.7.2, “List index”.

Restrictions:

- The **List layout** widget feature is intended to be used with instantiator.
- The **List layout** widget feature excludes the following widget features:
  - **Absolute layout**
  - **Box layout**
  - **Flow layout**
  - **Grid layout**

Table 13.196. Properties of the **List layout** widget feature

Property name	Description
`layoutDirection`	The direction in which the list elements i.e. the child widgets are positioned. Possible values: - `horizontal (=0)` - `vertical (=1)`
`scrollOffset`	The number of pixels to scroll the list

Property name	Description
scrollOffsetRebase	If the scrollOffsetRebase property changes, the current scrollOffset is translated to scrollIndex. The remaining offset is written to the scrollOffset property.
firstListIndex	The list index of the first visible list element, defined by the widget feature
scrollIndex	The base list index the scrollOffset property applies to. Scrolling starts at the list elements given in the scrollIndex property.
scrollValue	The current scroll value in pixels
scrollValueMax	The maximum scroll value in pixels, which is mapped to the end of the list
scrollValueMin	The minimum scroll value in pixels, which is mapped to the beginning of the list
bounceValue	The bounceValue property is zero as long as the scrollOffset property results in a position inside the valid scroll range. It has a positive value if the scroll position exceeds the beginning of the list and a negative value if the scroll position exceeds the end of the list. If bounceValue is added to scrollOffset, the scroll position is back in range.
bounceValueMax	The maximum value which scrollOffset can move outside the valid scroll range. scrollOffset is truncated if the user tries to scroll further.
segments	For horizontal layout direction: the number of rows For vertical layout direction: the number of columns
listLength	The number of list elements
wrapAround	Possible values: ▶ true: The scrollValue property continues at the inverse border, if scrollValueMin or scrollValueMax is exceeded. ▶ false: The scrollValue property does not decrease/increase, if scrollValueMin or scrollValueMax is exceeded.

13.13.6.7. Scale mode

The **Scale mode** widget feature defines how an image is displayed if its size differs from the size of the widget.

Restrictions:

- The **Scale mode** widget feature is only available for the widget image.

Table 13.197. Properties of the **Scale mode** widget feature

Property name	Description
scaleMode	The scale mode of the image. Possible values:

Property name	Description
	- `0 = original size` - `1 = fit to size` - `2 = keep aspect ratio`

13.13.7. List management

13.13.7.1. Line index

The **Line index** widget feature defines the unique position for each line of your list or table.

Restrictions:

- The **Line index** widget feature is intended to be used in combination with instantiators.

Table 13.198. Properties of the **Line index** widget feature

Property name	Description
`lineIndex`	The index of the current line in a table

13.13.7.2. List index

The **List index** widget feature defines the unique position of a widget in a list.

Restrictions:

- The **List index** widget feature is intended to be used in combination with the **List layout** widget feature.

Table 13.199. Properties of the **List index** widget feature

Property name	Description
`listIndex`	The index of the current widget in a list

13.13.7.3. Template index

The **Template index** widget feature defines the unique position of the used line template.

Restrictions:

- The **Template index** widget feature is intended to be used in combination with instantiators.

Table 13.200. Properties of the **Template index** widget feature

Property name	Description
`lineTemplateIndex`	The index of the used line template

13.13.7.4. Viewport

The **Viewport** widget feature clips oversized elements at the widget borders.

Restrictions:

- The **Viewport** widget feature is intended to be used in combination with containers or lists.
- The **Viewport** widget feature takes effect on the following model elements:
    - Child widgets of the widget you added **Viewport** to are clipped inside the dimensions of the widget.
    - The widget you added **Viewport** is clipped inside the dimensions of its parent view.

Table 13.201. Properties of the **Viewport** widget feature

Property name	Description
`xOffset`	The horizontal offset of the visible clipping within the drawn area of child widgets
`yOffset`	The vertical offset of the visible clipping within the drawn area of child widgets

13.13.8. 3D

Widget features in the **3D** category are only available for 3D widgets.

13.13.8.1. Camera viewport

The **Camera viewport** widget feature defines the camera's drawing region within the scene graph.

Restrictions:

- The **Camera viewport** widget feature is available for camera.

Table 13.202. Properties of the **Camera viewport** widget feature

Property name	Description
`viewportX`	The x-origin of the viewport within the scene graph
`viewportY`	The y-origin of the viewport within the scene graph
`viewportWidth`	The viewport's width in pixels
`viewportHeight`	The viewport's height in pixels

13.13.8.2. Ambient texture

The **Ambient texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Ambient texture** widget feature is available for material, PBR Phong material, and PBR GGX material.
- When the **Ambient texture** is added, the `ambient` property is ignored.

Table 13.203. Properties of the **Ambient texture** widget feature

Property name	Description
`ambientTexture`	The file name of the texture
`ambientTextureAddressModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`ambientTextureAddressModeV`	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`ambientFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.
`diffuseSRGB`	If this property is enabled, the texture that is selected in `ambientTexture`, is rendered using sRGB color space. Note that to use sRGB functionality, open the project center, go to **Configure** > **Profiles** and for the `colorMode` property select `32-bit sRGB (=4)` or `32-bit sRGB (Emulated) (=5)`.

13.13.8.3. Diffuse texture

The **Diffuse texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Diffuse texture** widget feature is available for material, PBR Phong material, and PBR GGX material.
- When the **Diffuse texture** is added, the `diffuse` property is ignored.

Table 13.204. Properties of the **Diffuse texture** widget feature

Property name	Description
`diffuseTexture`	The file name of the texture
`diffuseTextureAddressModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`diffuseTextureAddressModeV`	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`diffuseFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.
`diffuseSRGB`	If this property is enabled, the texture that is selected in `diffuseTexture`, is rendered using sRGB color space. Note that to use sRGB functionality, open the project center, go to **Configure** > **Profiles** and for the `colorMode` property select `32-bit sRGB (=4)` or `32-bit sRGB (Emulated) (=5)`.

13.13.8.4. Emissive texture

The **Emissive texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Emissive texture** widget feature is available for material, PBR Phong material, and PBR GGX material.
- When the **Emissive texture** is added, the `emissive` property is ignored.

Table 13.205. Properties of the **Emissive texture** widget feature

Property name	Description
`emissiveTexture`	The file name of the texture
`emissiveTextureAddressModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`emissiveTextureAddressModeV`	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`emissiveFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.
`diffuseSRGB`	If this property is enabled, the texture that is selected in `emissiveTexture`, is rendered using sRGB color space. Note that to use sRGB functionality, open the project center, go to **Configure** > **Profiles** and for the `colorMode` property select `32-bit sRGB (=4)` or `32-bit sRGB (Emulated) (=5)`.

13.13.8.5. Light map texture

The **Light map texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Light map texture** widget feature is available for material, PBR Phong material, and PBR GGX material.

Table 13.206. Properties of the **Light map texture** widget feature

Property name	Description
`lightMapTexture`	The file name of the texture
`lightMapTextureAddress-ModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`lightMapTextureAddressMod-eV`	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`lightMapFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.

13.13.8.6. Metallic texture

The **Metallic** widget feature adds extended configuration values to a material. The texture controls the metallic parameter of the PBR GGX material and PBR Phong material widgets.

Restrictions:

- The **Metallic texture** widget feature is available for the PBR GGX material and PBR Phong material.
- The **Metallic texture** is a grayscale image. For RGB color images, only the red channel is used.
- When the **Metallic texture** is added, the `metallic` property is ignored.

Table 13.207. Properties of the **Metallic texture** widget feature

Property name	Description
metallicTexture	The file name of the texture
metallicMinFactor	The minimal metallic parameter as a float to interpolate the texture values
metallicMaxFactor	The maximal metallic parameter as a float to interpolate the texture values
metallicTextureAddressModeU	The address mode of the texture along the u-direction. Possible values: ▶ repeat (=0): When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ clamp (=1): When accessed outside the texture bounds, the pixels at the edge of the texture are used.
metallicTextureAddressModeV	The address mode of the texture along the v-direction. Possible values: ▶ repeat (=0): When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ clamp (=1): When accessed outside the texture bounds, the pixels at the edge of the texture are used.
metallicFilterMode	The filtering mode of the texture. Possible values: ▶ point (=0): Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ linear (=1): Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ trilinear (=2): Most expensive, but yields better results than linear filtering.

13.13.8.7. Normal map texture

The **Normal map** widget feature adds extended configuration values to a material.

Restrictions:

- The **Normal map texture** widget feature is available for material, PBR Phong material, and PBR GGX material.

Table 13.208. Properties of the **Normal map** widget feature

Property name	Description
normalMapTexture	The file name of the texture

Property name	Description
normalMapTextureAddress-ModeU	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
normalMapTextureAddress-ModeV	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
normalMapFilterMode	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.

13.13.8.8. Opaque texture

The **Opaque texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Opaque texture** widget feature is available for material, PBR Phong material, and PBR GGX material.

Table 13.209. Properties of the **Opaque texture** widget feature

Property name	Description
opaqueTexture	The file name of the texture
opaqueTextureAddressModeU	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.

Property name	Description
opaqueTextureAddressModeV	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
opaqueFilterMode	The filter mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.

13.13.8.9. Reflection texture

The **Reflection texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Reflection texture** widget feature is available for material, PBR Phong material, and PBR GGX material.

Table 13.210. Properties of the **Reflection texture** widget feature

Property name	Description
reflectionTopTexture	The file name of the texture
reflectionBottomTexture	The file name of the texture
reflectionLeftTexture	The file name of the texture
reflectionRightTexture	The file name of the texture
reflectionFrontTexture	The file name of the texture
reflectionBackTexture	The file name of the texture
reflectionFilterMode	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts.

Property name	Description
	▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.

NOTE

Reflection texture widget feature

EB GUIDE Studio displays the **Reflection texture** widget feature, only when an image file is selected for all of the following properties:

- `reflectionTopTexture`
- `reflectionBottomTexture`
- `reflectionLeftTexture`
- `reflectionRightTexture`
- `reflectionFrontTexture`
- `reflectionBackTexture`

The image files must have the same size and rectangular shape.

13.13.8.10. Roughness texture

The **Roughness texture** widget feature adds extended configuration values to a material. The texture controls the roughness parameter of the PBR GGX material widget.

Restrictions:

- The **Roughness texture** widget feature is available for the PBR GGX material.
- The **Roughness texture** is a grayscale image. For RGB color images, only the red channel is used.
- When the **Roughness texture** is active, the `roughness` property is ignored.

Table 13.211. Properties of the **Roughness texture** widget feature

Property name	Description
`roughnessTexture`	The file name of the texture
`roughnessMinFactor`	The minimal roughness parameter as a float to interpolate the texture values
`roughnessMaxFactor`	The maximal roughness parameter as a float to interpolate the texture values
`roughnessTextureAddress-ModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile

Property name	Description
	▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`roughnessTextureAddress-ModeV`	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`roughnessFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.

13.13.8.11. Shininess texture

The **Shininess texture** widget feature adds extended configuration values to a material. The texture modulates the shininess strength by multiplying the texture value with the scalar `shininess` property.

Restrictions:

- The **Shininess texture** widget feature is available for the material and PBR Phong material.
- The **Shininess texture** is a grayscale image. For RGB color images, only the red channel is used.
- When the **Shininess texture** widget feature is used, the `shininess` property is ignored.

Table 13.212. Properties of the **Shininess texture** widget feature

Property name	Description
`shininessTexture`	The file name of the texture
`shininessMinFactor`	The minimal shininess parameter as a float to interpolate the texture values
`shininessMaxFactor`	The maximal shininess parameter as a float to interpolate the texture values
`shininessTextureAddress-ModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile

Property name	Description
	▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`shininessTextureAddress-ModeV`	The address mode of the texture along the v-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`shininessFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.

13.13.8.12. Specular texture

The **Specular texture** widget feature adds extended configuration values to a material.

Restrictions:

- The **Specular texture** widget feature is available for material, PBR Phong material, and PBR GGX material.
- When the **Specular texture** is added, the `specular` property is ignored.

Table 13.213. Properties of the **Specular texture** widget feature

Property name	Description
`specularTexture`	The file name of the texture
`specularTextureAddress-ModeU`	The address mode of the texture along the u-direction. Possible values: ▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`specularTextureAddressMod-eV`	The address mode of the texture along the v-direction. Possible values:

Property name	Description
	▶ `repeat (=0)`: When accessed outside the texture bounds, the texture is repeated. Also known as wrap or tile ▶ `clamp (=1)`: When accessed outside the texture bounds, the pixels at the edge of the texture are used.
`specularFilterMode`	The filtering mode of the texture. Possible values: ▶ `point (=0)`: Texture is not smoothed at all. Least expensive but prone to aliasing artifacts when texture is minimized. ▶ `linear (=1)`: Also known as bilinear filtering. Smoothens the texture when minimized to reduce aliasing artifacts. ▶ `trilinear (=2)`: Most expensive, but yields better results than linear filtering.
`diffuseSRGB`	If this property is enabled, the texture that is selected in `specularTexture`, is rendered using sRGB color space. Note that to use sRGB functionality, open the project center, go to **Configure** > **Profiles** and for the `colorMode` property select `32-bit sRGB (=4)` or `32-bit sRGB (Emulated) (=5)`.

13.13.8.13. Texture coordinate transformation

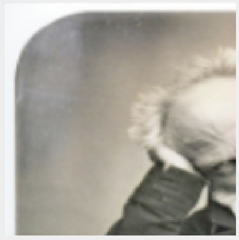
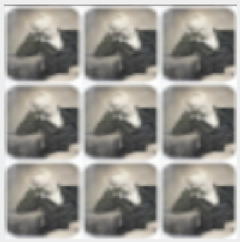
The **Texture coordinate transformation** widget feature makes it possible to modify the coordinates of the material texture. This feature is available for material, PBR Phong material, and PBR GGX material.

Table 13.214. Properties of the **Texture coordinate transformation** widget feature

Property name	Description
uOffset	Defines the offset of the texture coordinates in u-direction
vOffset	Defines the offset of the texture coordinates in v-direction
uScale	Defines the scaling of the texture coordinates in u-direction
vScale	Defines the scaling of the texture coordinates in v-direction

Example 13.5.
Example for Texture coordinate transformation

uOffset: 1.0 vOffset: 1.0 uScale: 1.0	0.5 1.0 1.0	1.0 0.5 1.0	1.0 1.0 0.5	1.0 1.0 3.0

vScale: 1.0	1.0	1.0	0.5	3.0
				

13.13.8.14. Tone mapping

The **Tone mapping** widget feature enables tone mapping, i.e. the technique to map a luminance value to a limited range, for the scene graph.

Restrictions:

- The **Tone mapping** widget feature is available for the scene graph.

Note that the **Tone mapping** widget feature implements the global tone mapping operator described by Erik Reinhard et al. [1]

Table 13.215. Properties of the **Tone mapping** widget feature

Property name	Description
`pureWhiteLuminance`	The smallest luminance value that is mapped to pure white. Note that only values bigger or equal to `0` are valid.

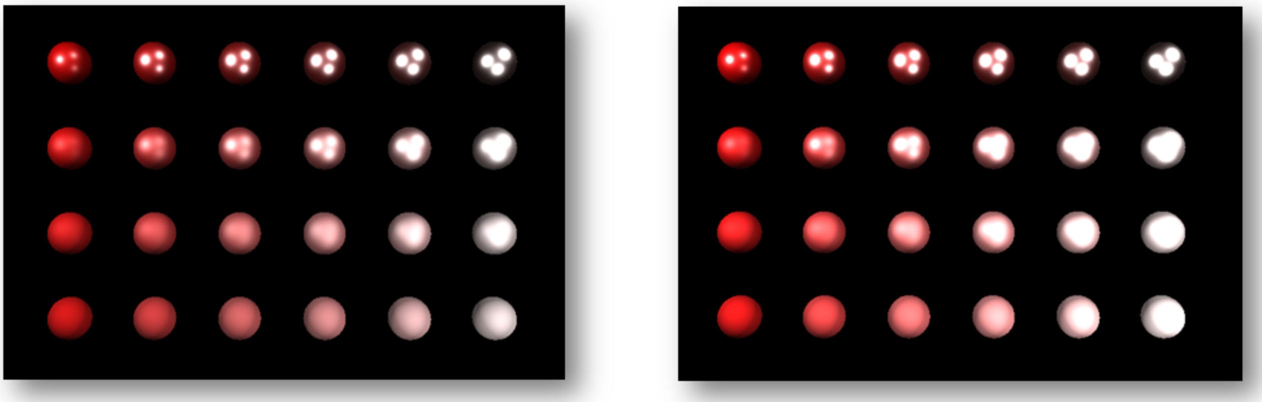


Figure 13.3. Example for image without tone mapping (left) and with tone mapping (right)

[1] *Photographic tone reproduction for digital images* Reinhard, Erik et al. in "Proceedings of the 29th annual conference on Computer graphics and interactive techniques" 2002, Pages 267-276

13.13.9. Transformation

The widget features of the category **Transformation** modify location, form, and size of widgets.

The order in which transformations are executed is equal to the order in the widget tree. If multiple transformations are applied to one widget at the same widget tree hierarchy level, the order is as follows:

1. Translation
2. Shearing
3. Scaling
4. Rotation around z-axis
5. Rotation around y-axis
6. Rotation around x-axis

13.13.9.1. Pivot

The **Pivot** widget feature defines the pivot point of transformations which are applied to the widget. If no pivot point is configured, the default pivot point is at (0.0, 0.0, 0.0).

Restrictions:

- Adding the **Pivot** widget feature automatically adds the **Rotation**, **Scaling** and **Shearing** widget features.

Table 13.216. Properties of the **Pivot** widget feature

Property name	Description
`pivotX`	The pivot point on the x-axis relative to parent widget
`pivotY`	The pivot point on the y-axis relative to parent widget
`pivotZ`	The pivot point on the z-axis relative to parent widget if widget is a scene graph

13.13.9.2. Rotation

The **Rotation** widget feature is used to rotate the widget and its subtree.

Table 13.217. Properties of the **Rotation** widget feature

Property name	Description
`rotationEnabled`	Defines whether rotation is used or not
`rotationAngleX`	The rotation angle on the x-axis. This property only affects scene graph.

Property name	Description
rotationAngleY	The rotation angle on the y-axis. This property only affects scene graph.
rotationAngleZ	The rotation angle on the z-axis

13.13.9.3. Scaling

The **Scaling** widget feature is used to scale the widget and its subtree.

Table 13.218. Properties of the **Scaling** widget feature

Property name	Description
scalingEnabled	Defines whether scaling is used or not
scalingX	The scaling on the x-axis in percent
scalingY	The scaling on the y-axis in percent
scalingZ	The scaling on the z-axis in percent if widget is a scene graph

13.13.9.4. Shearing

The **Shearing** widget feature is used to distort the widget and its subtree.

Table 13.219. Properties of the **Shearing** widget feature

Property name	Description
shearingEnabled	Defines whether shearing is used or not
shearingXbyY	The shearing of x-axis by y-axis
shearingXbyZ	The shearing of x-axis by z-axis if widget is a scene graph
shearingYbyX	The shearing of y-axis by x-axis
shearingYbyZ	The shearing of y-axis by z-axis if widget is a scene graph
shearingZbyX	The shearing of z-axis by x-axis if widget is a scene graph
shearingZbyY	The shearing of z-axis by y-axis if widget is a scene graph

13.13.9.5. Translation

The **Translation** widget feature is used to translate the widget and its subtree. It moves widgets in x, y and z directions.

Table 13.220. Properties of the **Translation** widget feature

Property name	Description
`translationEnabled`	Defines whether translation is used or not
`translationX`	The translation on the x-axis
`translationY`	The translation on the y-axis
`translationZ`	The translation on the z-axis if widget is a scene graph

14. Installation of EB GUIDE Studio

14.1. Background information

14.1.1. Restrictions

NOTE

Compatibility

EB GUIDE product line 6 is not compatible with any previous major version.

NOTE

EB GUIDE Speech Extension

EB GUIDE Speech Extension is licensed as an add-on product that is enabled only when purchased.

NOTE

User rights

To install EB GUIDE on Windows 7 or Windows 10 systems, you require administrator rights.

14.1.2. System requirements

Observe the following settings:

Table 14.1. Recommended settings for EB GUIDE Studio

Hardware	PC with quad core CPU with at least 2 GHz CPU speed and 8 GB RAM
Operating system	Windows 7 (64-bit), Windows 10 (64-bit)
Screen resolution	1920 x 1080 pixels or more Two separate monitors recommended
Software	Microsoft .NET Framework 4.7

Table 14.2. Recommended settings for EB GUIDE SDK

Development environment (IDE)	Microsoft Visual Studio 2013 or newer
File integration	CMake

14.2. Downloading EB GUIDE

To download the community edition of EB GUIDE, go to https://www.elektrobit.com/ebguide/try-eb-guide/ and follow the instructions.

To download the enterprise edition of EB GUIDE, go to EB Command.

NOTE

Activate your account

After ordering a product, you receive an email from sales department. Click the link in the email. Follow the steps to create an account as directed in the email and in the browser, then proceed to log in.

EB Command is the server from which you are going to download the EB GUIDE product line software. For the instructions on how to download from EB Command, see https://www.elektrobit.com/support/downloading-from-eb-command/.

14.3. Installing EB GUIDE

Installing EB GUIDE

Prerequisite:

- You downloaded the setup file `studio_setup.exe`.
- You have administrator rights on the operating system.

Step 1
Double-click the setup file `studio_setup.exe`.

A dialog opens.

Step 2
Click **Yes**.

The **Setup - EB GUIDE Studio** dialog opens.

Step 3
Accept the license agreement and click **Next**.

Step 4
Select a directory for installation.

The default installation directory is `C:/Program Files/Elektrobit/EB GUIDE <version>`.

Step 5
Click **Next**.

A summary dialog displays all selected installation settings.

Step 6
To confirm the installation with the settings displayed, click **Install**.

The installation starts.

Step 7
To exit the setup click **Finish**.

You have installed EB GUIDE.

TIP

Multiple installations
It is possible to install more than one EB GUIDE versions.

14.4. Uninstalling EB GUIDE

Uninstalling EB GUIDE

NOTE

Removing EB GUIDE permanently
If you follow the instruction, you remove EB GUIDE permanently from your PC.

Prerequisite:

- EB GUIDE is installed.
- You have administrator rights on the operating system.

Step 1
On the Windows **Start** menu, click **All Programs**.

Step 2
On **Elektrobit** menu, click the version you want to uninstall.

Step 3
On the submenu, click **Uninstall**.

Glossary

#

3D graphic
: A 3D graphic is a virtual picture of a 3D scene. A 3D scene is a collection of 3D models (meshes or shapes), materials, light sources, and cameras. Materials define the visual appearance of 3D models through colors and textures and the behavior under virtual lighting. A camera provides the view point from where a virtual picture of the 3D scene is taken.

A

API
: Application programming interface

application
: In the context of EB GUIDE, an application is computer software that interacts with one or more EB GUIDE models at EB GUIDE GTF run-time by means of, for example, the event system and datapool. An application is, for example, entertainment software like media player, communication software like phone, etc.
See Also API.

aspect
: In EB GUIDE, an aspect is an appearance-related modification of an EB GUIDE model that is applied at EB GUIDE GTF run-time. Two types of aspects exist: skins, with which you can define different looks for your EB GUIDE model, and languages.

C

communication context
: The communication context describes the environment in which communication occurs. Each communication context is identified by a unique numerical ID.

D

datapool
: The datapool is a data cache in an EB GUIDE model that provides access to datapool items during run-time. It is used for data exchange between the application and the HMI.

datapool item
: Datapool items store and exchange data. Each item in the datapool has a communication direction.

E

EB GUIDE GTF	EB GUIDE GTF is the graphics target framework of the EB GUIDE product line and is part of EB GUIDE TF. EB GUIDE GTF represents the run-time environment to execute EB GUIDE models on target devices.
EB GUIDE GTF SDK	EB GUIDE GTF SDK is the development environment contained in EB GUIDE GTF. It is a sub-set of the EB GUIDE SDK. Another sub-set is the EB GUIDE Studio SDK.
EB GUIDE model	An EB GUIDE model is the description of an HMI created with EB GUIDE Studio.
EB GUIDE product line	The EB GUIDE product line is a collection of software libraries and tools which are needed to specify an HMI model and convert the HMI model into a graphical user interface that runs on an embedded environment system.
EB GUIDE Script	EB GUIDE Script is the scripting language of the EB GUIDE product line. EB GUIDE Script enables accessing the datapool, model elements such as widgets and the state machine, and system events.
EB GUIDE SDK	EB GUIDE SDK is a product component of EB GUIDE. It is the software development kit for the EB GUIDE product line. It includes the EB GUIDE Studio SDK and the EB GUIDE GTF SDK.
EB GUIDE Studio	EB GUIDE Studio is the tool for modeling and specifying an HMI with a graphical user interfaces.
EB GUIDE Studio SDK	EB GUIDE Studio SDK is an application programming interface (API) to communicate with EB GUIDE Studio. It is a sub-set of the EB GUIDE SDK. Another sub-set is the EB GUIDE GTF SDK.
EB GUIDE TF	EB GUIDE TF is the run-time environment of the EB GUIDE. It consists of EB GUIDE GTF and EB GUIDE STF. It is required to run an EB GUIDE model.
extension	In EB GUIDE, an extension is an addition to any of the EB GUIDE products. An extension is a plug-in in form of libraries (`.dll` or `.so` files) that add a certain functionality to EB GUIDE Studio, EB GUIDE GTF, or EB GUIDE Monitor. Such functionality could be, for example, a data exporter or an additional widget feature.

G

GL	Graphical library

GUI	Graphical user interface

H

HMI	Human machine interface

I

IBL	Image-based lighting
IBLGenerator	IBLGenerator is the tool to process environment lighting information.

L

library	A library is a collection of pre-compiled software parts, sub-routines, or programs that are used in EB GUIDE. Libraries that are necessary for an EB GUIDE project are defined in the project center. Two file types are supported: `.dll` and `.so`.

M

model element	A model element is an object within an EB GUIDE model, for example a state, a widget, or a datapool item. See Also EB GUIDE model.
multifont support	Aggregation of multiple fonts for different character ranges acting as a single font.
MVC	Model-view-controller

N

namespace	In EB GUIDE Studio, with namespaces you create groups of model elements like datapool items and events. These groups have usually a defined functionality. Each namespace creates a naming scope for model elements so that model elements in different namespaces can have the same name.

O

OS	Operating system

P

PBR	Physically-based rendering
profile	In the project center, a profile is a set of specifications. In a profile you define libraries, messages and scenes for your project. During export of an EB GUIDE model the data in the profile is written to the `model.json` configuration file.
project center	All project-related functions are located in the project center, for example profiles and languages.
project editor	In the project editor you model the behavior and the appearance of the human machine interface.

R

resource	A resource is a data package that is part of the EB GUIDE project. Examples for resources are fonts, images, meshes. Resources are stored outside of the EB GUIDE model, for example in files, depending on the operating system.
RomFS	Read-only memory file system

S

shared library	A shared library, as opposed to a static library, can be loaded when preparing a program for execution. On Windows platforms shared libraries are called dynamic link libraries and have a `.dll` file extension. On Unix systems shared libraries are called shared objects and have an `.so` file extension.
state	A state defines the status of the state machine. States and state transitions are modeled in state charts.
state machine	A state machine is a set of states, transitions between those states, and actions. A state machine describes the dynamic behavior of the system.

T

transition	A transition defines the change from one state to another. A transition is usually triggered by an event.

U

UI	User interface

V

view	A view is a graphical representation of a project-specific HMI-screen and is related to a specific state machine state. A view consists of a tree of widgets.
VTA	View transition animation

W

widget	A widget is a basic graphical element. Widgets are used for interaction with a graphical user interface.

Index

Symbols

A

B

C

D

E

F

G

T

U

V

W