Elektrobit

EB GUIDE TF

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

Table of Contents

1. About this documentation

1.1. Target audiences of the user documentation

This chapter informs you about target audiences involved in an EB GUIDE project and the tasks they usually perform.

You can categorize your tasks and find the documentation relevant to you.

The following roles exist:

- section 1.1.1, “System integrators”
- section 1.1.2, “Application developers”
- section 1.1.3, “Extension developers”

1.1.1. System integrators

System integrators make sure that all the different system parts are integrated into one complete and working system.

System integrators perform the following tasks:

- Ensure that the different project parts are executed together
- Configure required modules and file system structures
- Integrate customer specific EB GUIDE GTF extensions and HMI applications
- Carry out settings to ensure system integrity within EB GUIDE Studio and on the target device
- Carry responsibility for the project setup in EB GUIDE Studio, for example, create a shared workspace in projects involving different people working together on one EB GUIDE model

System integrators have the profound knowledge of the following:

- The system, including the target framework used and its restrictions
- The generating mechanism that ensures compatibility of an EB GUIDE model and the target system

1.1.2. Application developers

Application developers write source code for HMI applications, such as a CD player or a radio. Such applications add distinct functionality to the system, for example control of hardware components.

Application developers perform the following tasks:

- Program additional functionality that is required by the system
- Write code to interface with EB GUIDE GTF, provide application data to the HMI, and provide communication with the HMI
- Consider the required communication data between the HMI model and its application
- Define datapool items and events
- Determine the flow of data between HMI model and application
- Communicate with modelers to know what data can be provided by hardware devices and how to use the different EB GUIDE GTF communication mechanisms

Application developers have the profound knowledge of the following:

- C++, to know how to compile for the existing EB GUIDE TF C++ interfaces
- All programming languages used, as applications can be written in any programming language
- The specifications and requirements of the domain

1.1.3. Extension developers

There may be missing features that cannot be provided through simply modeling an EB GUIDE model or adding customer-specific applications. This is when new widgets or a specific renderer may be required.

Extension developers perform the following tasks:

- Communicate with members of the EB GUIDE development team through chapter 3, “Support“ to find out if there are already solutions to problems
- Work on the framework and develop new features, EB GUIDE Studio extensions or EB GUIDE GTF extensions
- Write code for additional modules for the following items:
  - Existing EB GUIDE GTF modules such as widgets or the shaders
  - Existing EB GUIDE Studio extensions such as additional toolbar buttons

Extension developers have the profound knowledge of the following:

- EB GUIDE interfaces
- Interaction between the central modules
- Structure of the framework's data

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>}`

Font	Description	Example
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`.

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. Framework overview

5.1. Relationship between EB GUIDE Studio, EB GUIDE GTF and EB GUIDE Monitor

EB GUIDE Studio provides input to EB GUIDE GTF in form of the exported EB GUIDE model. The EB GUIDE model comprises the binary model description files that are generated during export. For more information on export, see EB GUIDE Studio user documentation.

Within the EB GUIDE product line, EB GUIDE GTF is responsible for the following:

- EB GUIDE GTF executes the content of an exported EB GUIDE model on the dedicated target platform. A target platform for EB GUIDE GTF is typically defined by concrete target architecture, operating system and graphics API.
- EB GUIDE GTF runs the EB GUIDE model for the simulation in EB GUIDE Studio.

You can connect EB GUIDE Monitor to EB GUIDE GTF, which allows you to observe, manipulate, and test an EB GUIDE model that runs on EB GUIDE GTF. EB GUIDE Monitor and EB GUIDE GTF communicate using TCP/IP.

EB GUIDE GTF interacts with third-party applications using API. Applications have access to the configuration files of the exported EB GUIDE model.

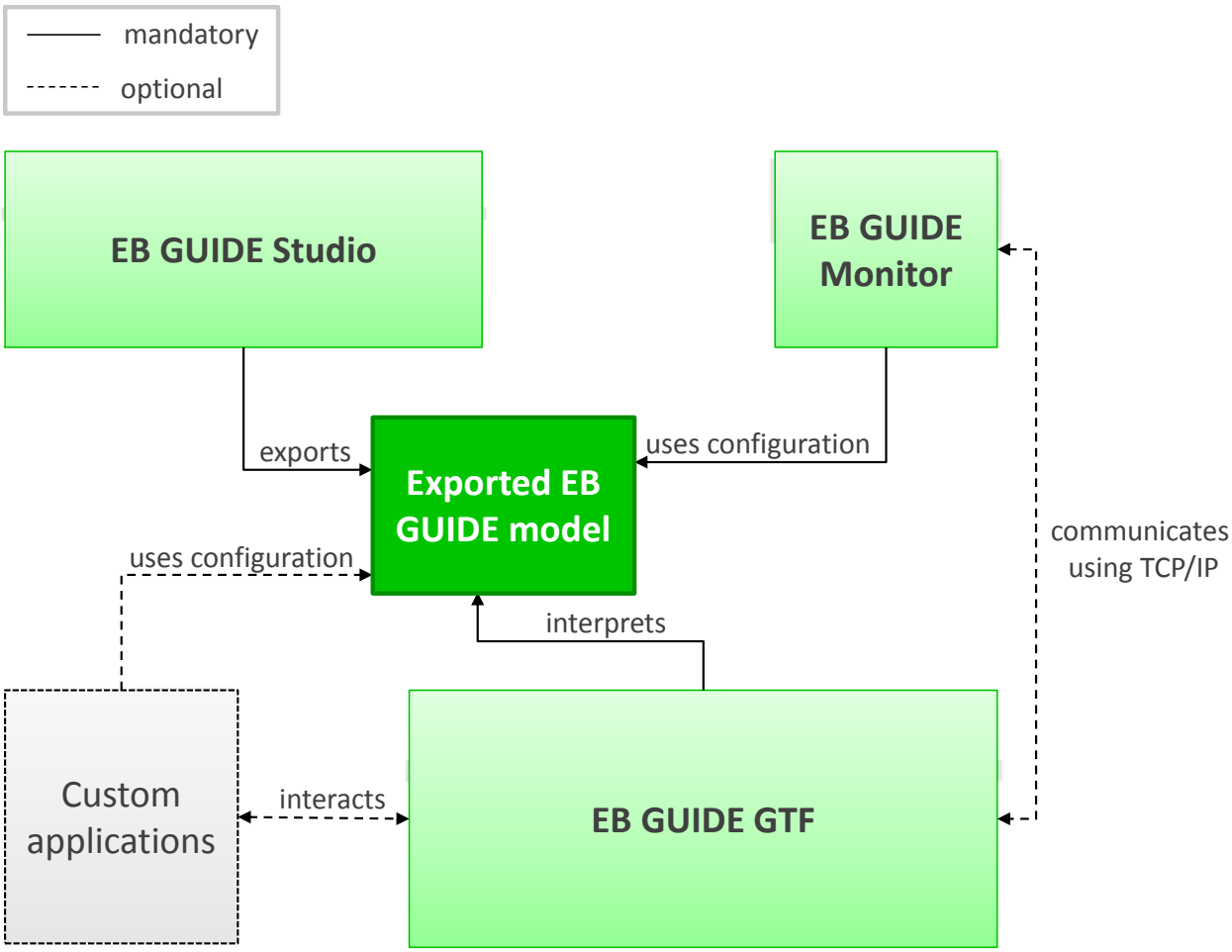


Figure 5.1. Workflow overview

5.2. Interactions between EB GUIDE GTF and extensions

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 typical EB GUIDE GTF extensions are the following:

- New widgets
- 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, into the following directory:

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

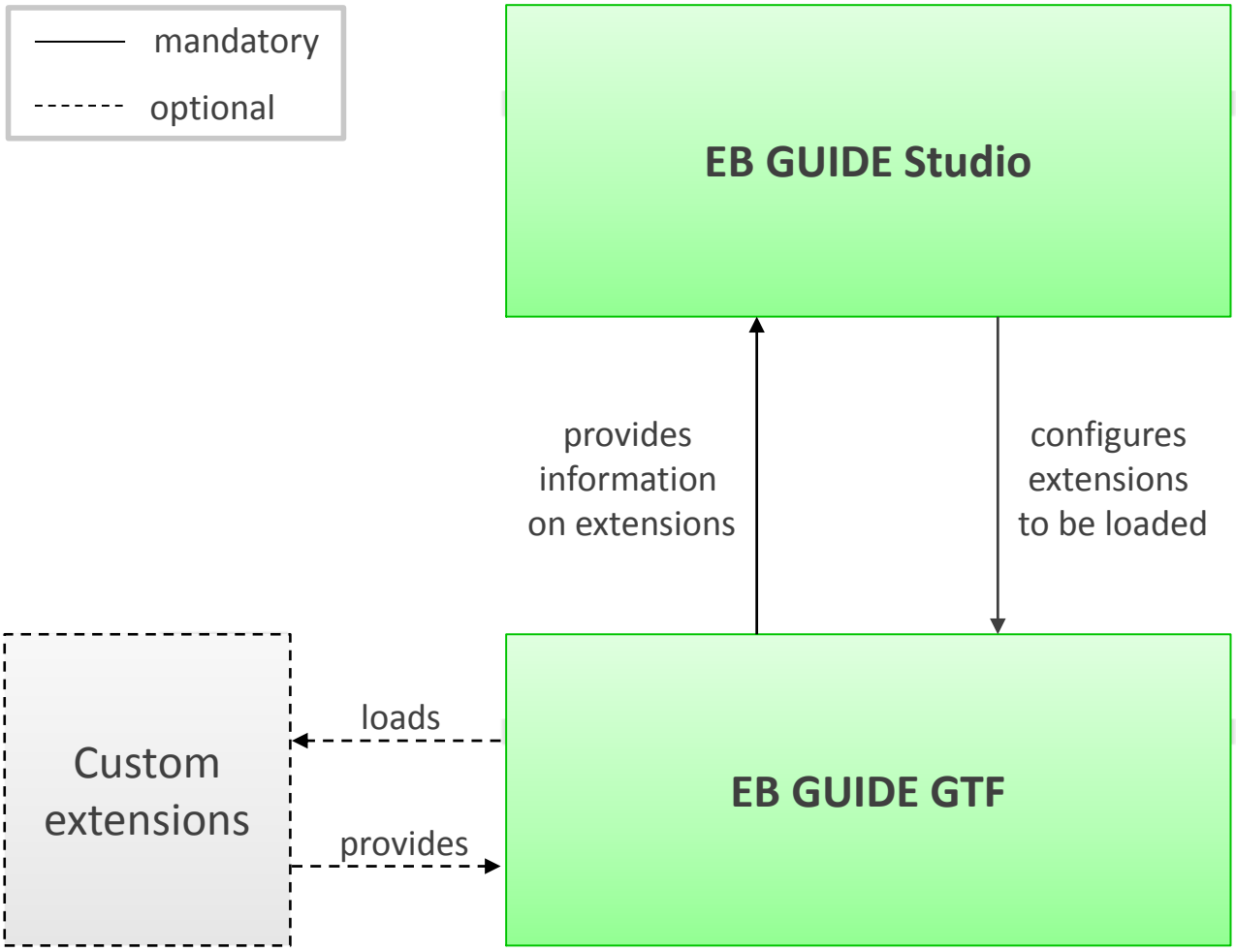


Figure 5.2. Workflow overview

EB GUIDE GTF loads an additional custom extension and provides data on the extension to EB GUIDE Studio. This is the way EB GUIDE Studio gets the necessary data and is able to configure the extension.

For instructions, see chapter 10, “Using the EB GUIDE GTF plugin mechanism“.

5.3. Interactions between EB GUIDE GTF and application

EB GUIDE product line abstracts all communication data between an application and EB GUIDE GTF in an application programming interface (API). An application is for example a media player or a navigation system. The API is defined by datapool items and events. Datapool and event system of an application exchange data with datapool and event system of EB GUIDE GTF.

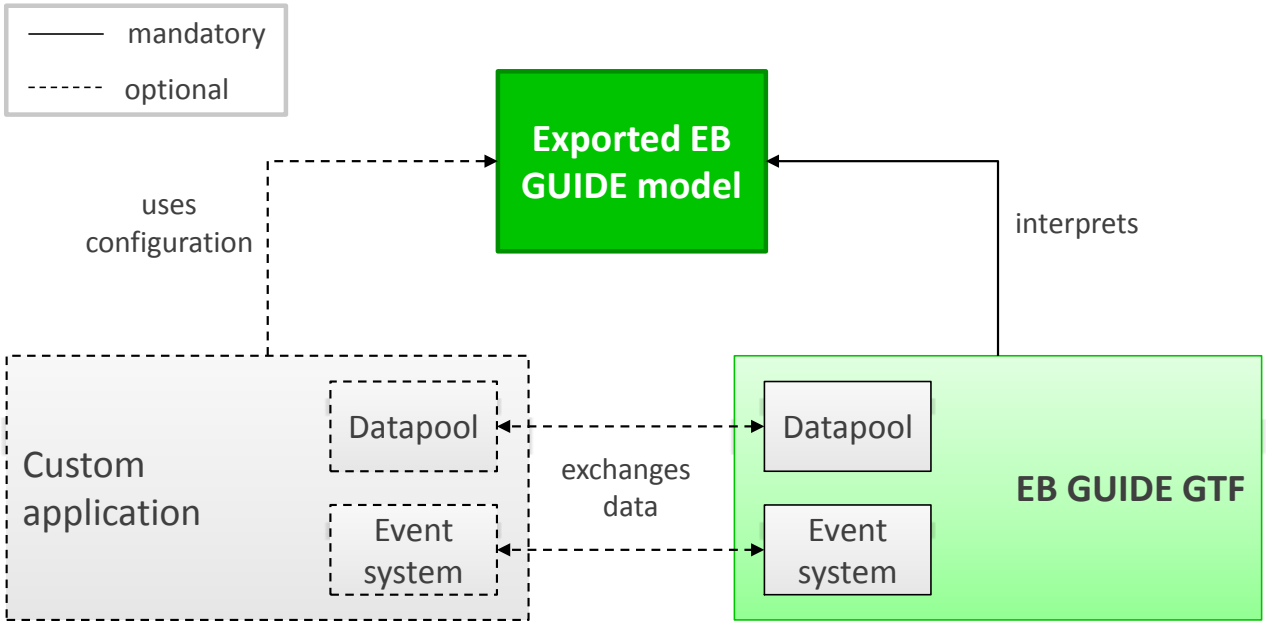


Figure 5.3. Workflow overview

5.4. Interaction between EB GUIDE GTF instances

Since every EB GUIDE GTF instance executes only one EB GUIDE model, to make your HMI model more configurable, you can use several EB GUIDE GTF instances. Multiple EB GUIDE GTF instances may exchange information, for example datapool items or events, using the inter-process communication (IPC) mechanism.

Example 5.1.
Multiple EB GUIDE GTF instances

Depending on your HMI model, you can have several combinations for using EB GUIDE GTF:

- You have only one target device that is connected to several displays, and one EB GUIDE GTF instance that interprets one EB GUIDE model.
- You have only one target device that is connected to several displays, and several EB GUIDE GTF instances. Each of these EB GUIDE GTF instances interprets a separate EB GUIDE model.
- You have several target devices and several EB GUIDE GTF instances. Each of these target devices connects to a separate display. Each of these EB GUIDE GTF instances interprets a separate EB GUIDE model.

If you have several displays and want to avoid blocking, it is advisable to have several EB GUIDE models and therefore use several EB GUIDE GTF instances.

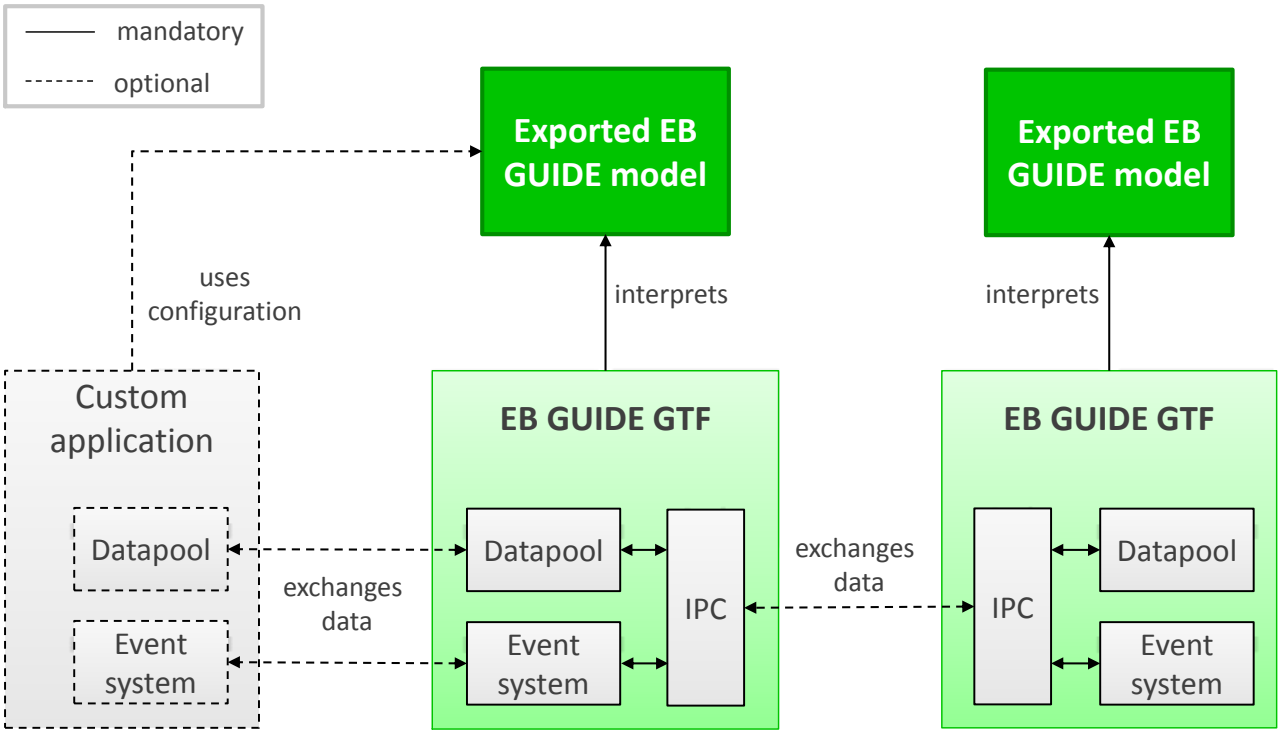


Figure 5.4. Workflow overview for several EB GUIDE GTF instances with optional application

The figure above shows the typical interactions between several EB GUIDE GTF instances that are valid for the IPC and non-IPC use cases.

5.5. Dependencies to target platform

To be able to run an EB GUIDE GTF instance, make sure that your target platform meets the following minimal requirements.

Table 5.1. Minimal requirements

Compiler	Support of C++11 is required.
Operating system	Operating system with Posix-compliant interface: Windows, Linux, QNX, Greenhills integrity
GPU	DirectX 11 for Windows OpenGL ES 2 and 3 for target platforms if applicable.

	Note that for Windows an emulation for both GPUs is used.
File system	EB GUIDE GTF runs on systems with a file system.
CPU	32-bit, little endian CPU, 64-bit

NOTE

Contact support

For more information on customization and target platforms, contact support.

5.6. Structure of EB GUIDE GTF

EB GUIDE GTF is based on a microkernel architecture. This means EB GUIDE GTF consists of a core with base functionality and plugins, i.e. extensions and applications, that enable the execution of EB GUIDE models.

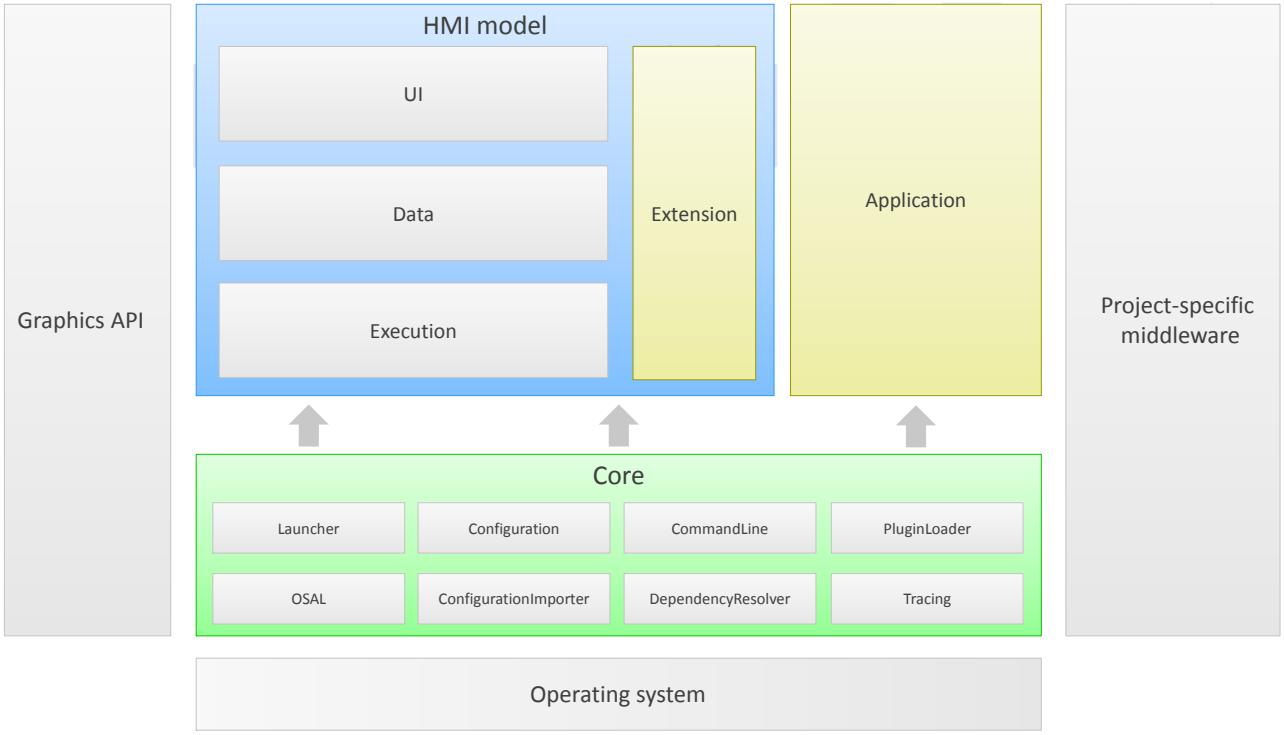


Figure 5.5. EB GUIDE GTF overview

EB GUIDE GTF is embedded in an environment that contains an operating system, a graphics API, and project-specific middleware.

The core includes the following:

- Abstraction of the dedicated operating system that is called Operating System Abstraction Layer (OSAL)
- The `Launcher` interface that launches EB GUIDE GTF
- `DependencyResolver`, a dependency resolving mechanism
- The `PluginLoader` interface that loads plugins
- The `Configuration` interface that accesses configuration items
- The `ConfigurationImporter` interface that loads configuration files
- The `CommandLine` interface that parses given command line parameters
- Interface for tracing purposes

Applications are plugins for EB GUIDE GTF. This means the core loads applications at the start-up before the start of an EB GUIDE model is initiated. Applications are bound to the global accessible API and cannot directly access EB GUIDE model content.

Extensions are plugins for an EB GUIDE model. Extensions are initiated during the EB GUIDE model start-up. Extensions can access model-specific content directly and extend the EB GUIDE model content, for example using widgets, features or EB GUIDE script functions.

The HMI model in this context is the summary of all components that are necessary to run an exported EB GUIDE model. This includes components that manage the user interface, the data and the execution of the dynamic behavior of an EB GUIDE model.

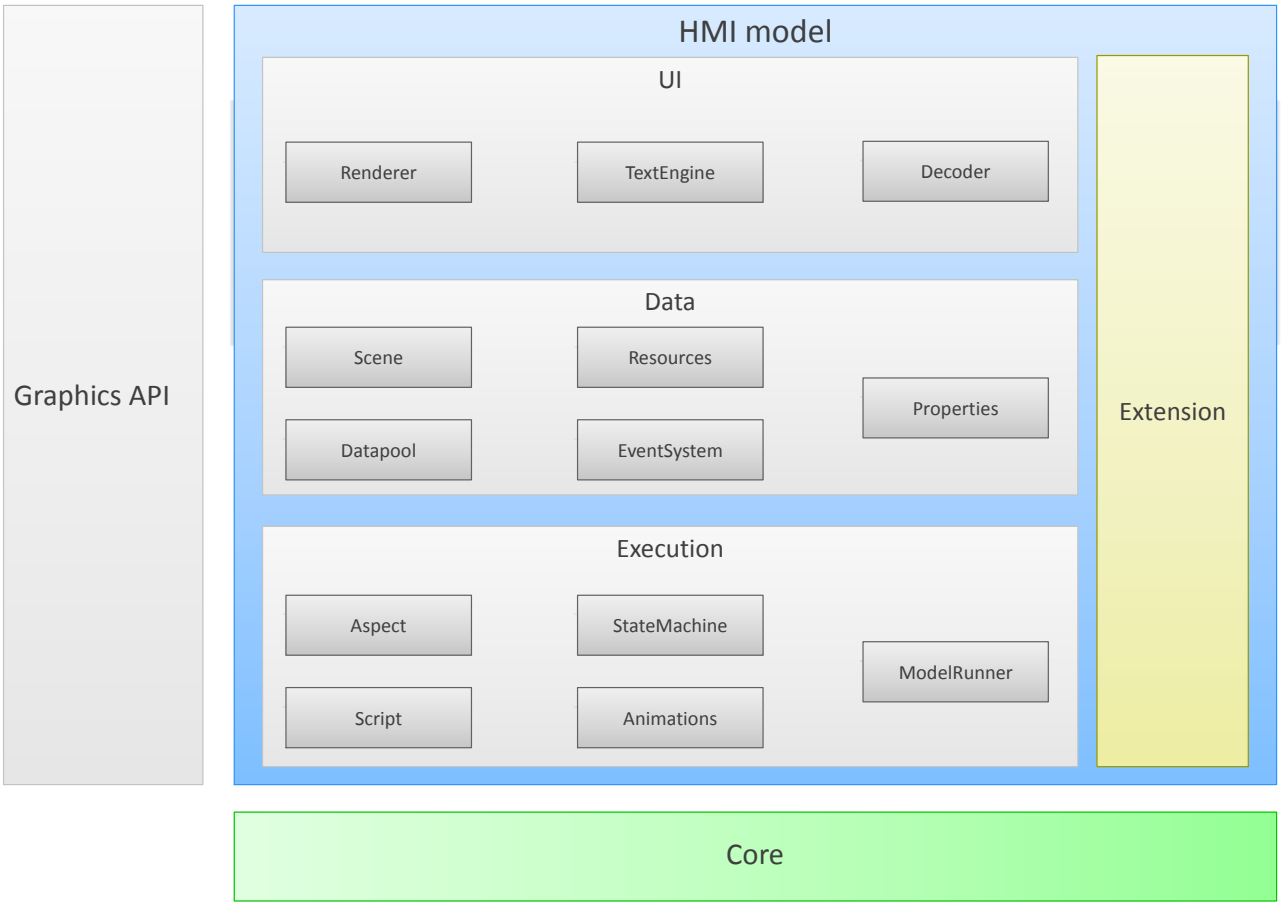


Figure 5.6. HMI model components

The execution components take care of the dynamic behavior of an EB GUIDE model. This means the execution of the following:

- State machine
- EB GUIDE Script that is part of the datapool, state machine or widget tree
- Animations that describe property changes within the widget tree
- Parts of an EB GUIDE model that are described using languages and skins
- EB GUIDE models for which the `ModelRunner` is responsible

The data components take care of data definition and management.

EB GUIDE GTF considers the following items as data items of an EB GUIDE model:

- Datapool items
- Events that are managed using `EventSystem`
- Resources
- Scenes built up from views and widget trees

- Widget properties

The user interface components take care of functions required to support graphical user interfaces. These components enable EB GUIDE GTF to perform the following:

- The `TextEngine` component supports processing and rendering of internationalized and localized texts.
- The `Decoder` component loads and decodes various types of resources, for example images as `.png`, `.jpeg`.
- The `Renderer` component creates the visual representation using the available EB GUIDE GTF graphics abstraction layer.

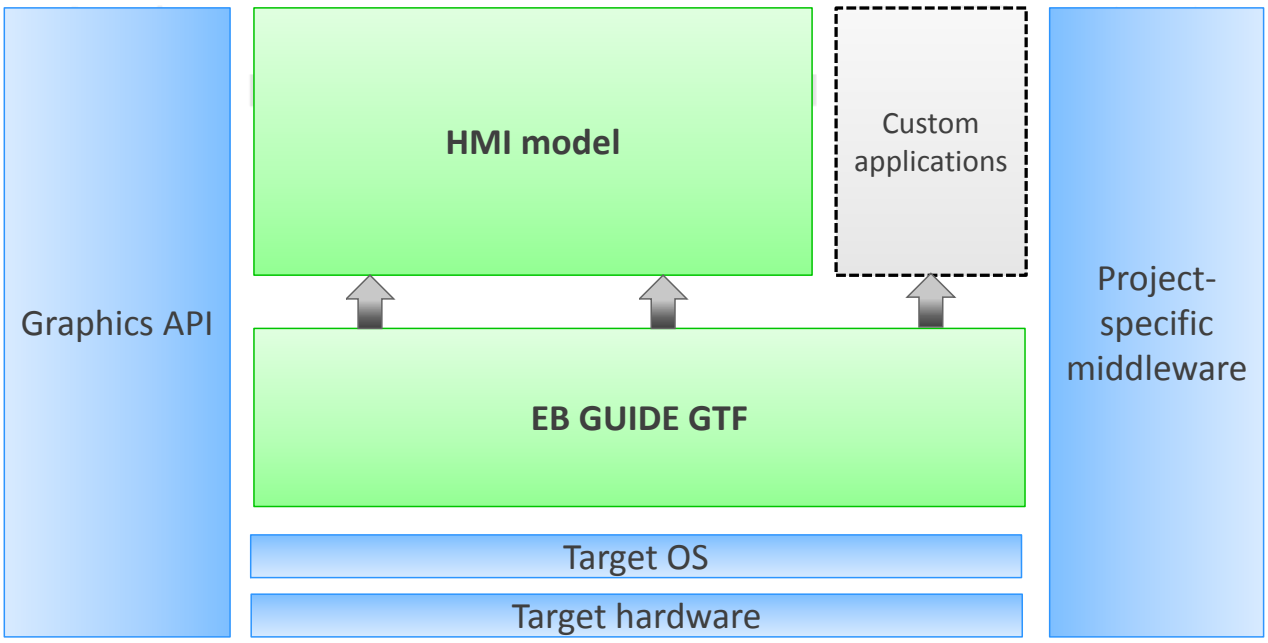


Figure 5.7. Overview Architecture

5.7. Deployment of EB GUIDE GTF

EB GUIDE GTF consists of `GtfStartup.exe` and several optional assemblies. The default delivery of EB GUIDE GTF runs on operating systems that support shared libraries, for example Windows 10, Linux or QNX. EB GUIDE GTF is divided into the following components to fit most customer projects out of the box:

- `GtfStartup.exe`

  The executable file that contains platform-specific start-up code and interprets the configuration files. `GtfStartup.exe` is configurable with parameters. For more information, see section 7.7.1, “Run EB GUIDE GTF with `GtfStartup.exe`”.

- `GtfCore`

  Shared library that contains all mandatory modules for each GUI project based on EB GUIDE Studio and EB GUIDE GTF.

- `GtfRuntime`

  Shared library that contains all mandatory functionality for EB GUIDE GTF based projects.

- `GtfGui`

  Shared library that is responsible for text and resource handling and is required for EB GUIDE GTF based GUI projects.

- `GtfGuiOpenGLES20`

  Shared library that contains the OpenGL ES 2.0 API renderer implementation.

- `GtfGuiOpenGLES3`

  Shared library that contains the OpenGL ES 3.0 API renderer implementation.

- `GtfGuiDirectX11`

  Shared library that contains the DirectX 11 API renderer implementation.

- `GtfService`

  Shared library that is required to establish TCP connection to an EB GUIDE GTF instance, used by for example EB GUIDE Monitor or `GtfIpc`.

- `GtfIpc`

  Shared library that extends `GtfService` and provides possibility for the inter-process communication (IPC).

- `GtfFileOutput`

  Shared library that is used by the trace logging system for the file output on all target platforms.

- `GtfProfilingOutput`

  Shared library that is used by the trace logging system for the output of the profiling data on all target platforms.

- `GtfVsDebugOutput`

  Shared library that is used by the trace logging system for the output inside of the IDE on Win64 target platforms.

- `GtfKernelEventOutput`

  Shared library that is used by the trace logging system for the output using Kernel event traces on QNX target platforms.

6. General concepts

6.1. Non-blocking strategy

EB GUIDE GTF does not block while waiting for changes. To implement the non-blocking strategy, use callbacks that are registered using delegates.

6.1.1. Observer patterns and callbacks

To track when a subject changes its state, for example if the property of a widget is changed, EB GUIDE GTF uses the observer pattern. The observer pattern happens through a callback function that you register using `gtf::delegate`. This function is invoked for each modification of the subject's state. For more information, see section 6.1.2, “Delegates”.

Example 6.1.
Delegate example

The event system provides an event queue which purpose is to control the point in time when event processing happens. The event system provides a subscribe method which takes a callback as parameter. The callback is processed when the event occurs.

```
gtf::utils::Delegate<void(gtf::eventsystem::EventHandle<const char*>)>
const subscriberCallback =
gtf::utils::Delegate<void(gtf::eventsystem::global::EventHandle)>::create
<EventApplication, &EventApplication::processEvent>(this);
    eventQueue->subscribe(gtf::eventsystem::SubscriptionType::Event
        , GLOBAL_EVENT_GROUP_GLOBAL
        , GLOBAL_EVENT_MSG_ID_WHATTODO
        , this
        , subscriberCallback);
```

6.1.2. Delegates

A delegate is a data type that represents references to methods with a particular parameter list and return type. To pass methods as arguments to other methods, use delegates. Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either a static or an instance method. Also it can be used on constant methods.

To store callbacks, use the delegate templates. Empty delegates are not allowed.

The delegate class provides the `gtf::utils::Delegate<R(Params)>::create <Class, F&> ()` method that returns `gtf::utils::Delegate<R(Params)>`. This method expects a specialization of the delegate class, which takes as template arguments the return type of the function and the type of the parameters. The create method expects as template arguments the class name and a reference to the actual method from that class. As parameter, the instance of the specific class is provided.

Create delegates as follows:

- Create a delegate for an object method:

```
gtf::utils::Delegate<R(Params)> delegate =
  gtf::utils::Delegate<Class, R(Params)>::create <Class,F&> (instance);
```

- Create a delegate for a static method or a function:

```
gtf::utils::Delegate<R(Params)> delegate =
  gtf::utils::Delegate<R(Params)>::create <F>();
```

  In case of static methods or functions, do not specify neither the class name as template argument nor the instance as parameter.

- Create a delegate for `const` function or `const` arguments:

  There is no difference in creating a delegate for a constant or non-constant argument.

You can also initialize a delegate using the `bind` method:

```
gtf::utils::Delegate<R(Params)> delegate;
delegate.bind<&F>();
```

NOTE **Change of function**

If you declared a delegate and it has a function assigned, the bind method changes the function with another function if it has the same return type and arguments list.

7. Core concepts

7.1. Core life cycle stages

The life cycle consists of stages. Stages are ordered and the start-up is done according to the order. The shutdown is done in the reverse order. There are separate life cycle stages for the core and for the EB GUIDE model. For the EB GUIDE model life cycle, see section 8.2, “Model life cycle stages”.

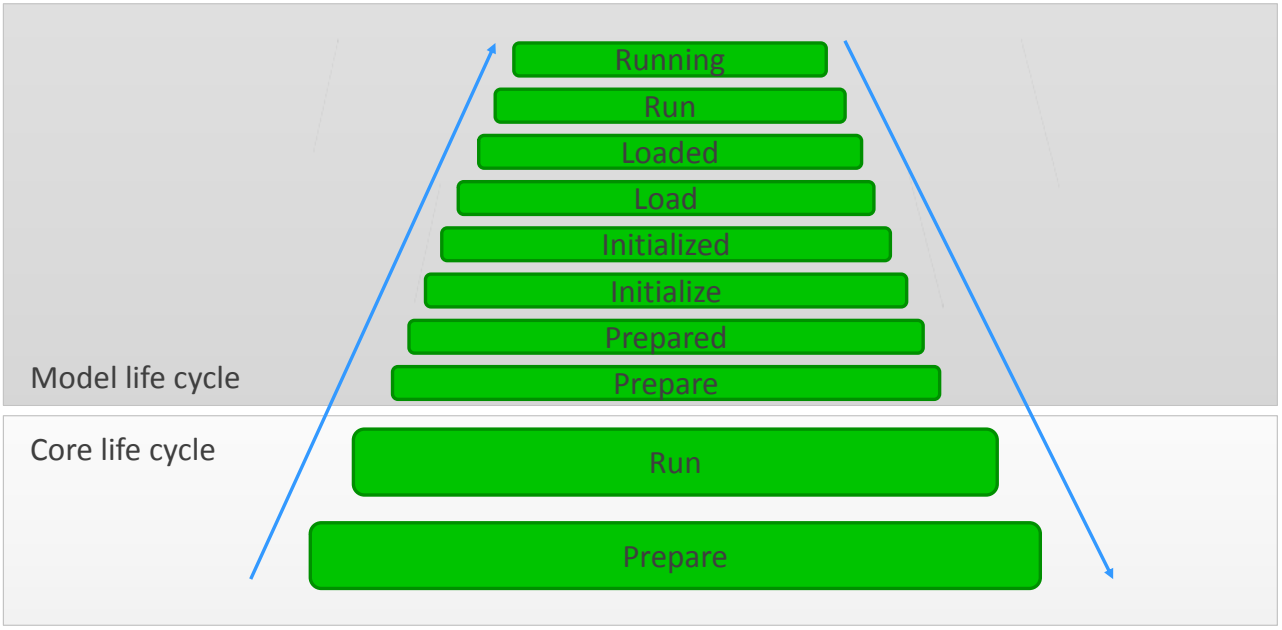


Figure 7.1. Overview of the EB GUIDE GTF life cycle stages

A stage is a named interface only. There may be multiple implementations for each of the stages. On start-up all implementations of one stage are created, before the next stage is entered. On shutdown all instances of one stage are destroyed, before the next stage is left.

You can customize the life cycle of EB GUIDE GTF by implementing predefined stages or by adding and implementing custom stages.

One and the same `gtf::dependencyresolver::DependencyContainer` instance is provided to each core stage `create` function. This instance is dedicated to the core and differs from the instance provided to stages of the EB GUIDE model life cycle.

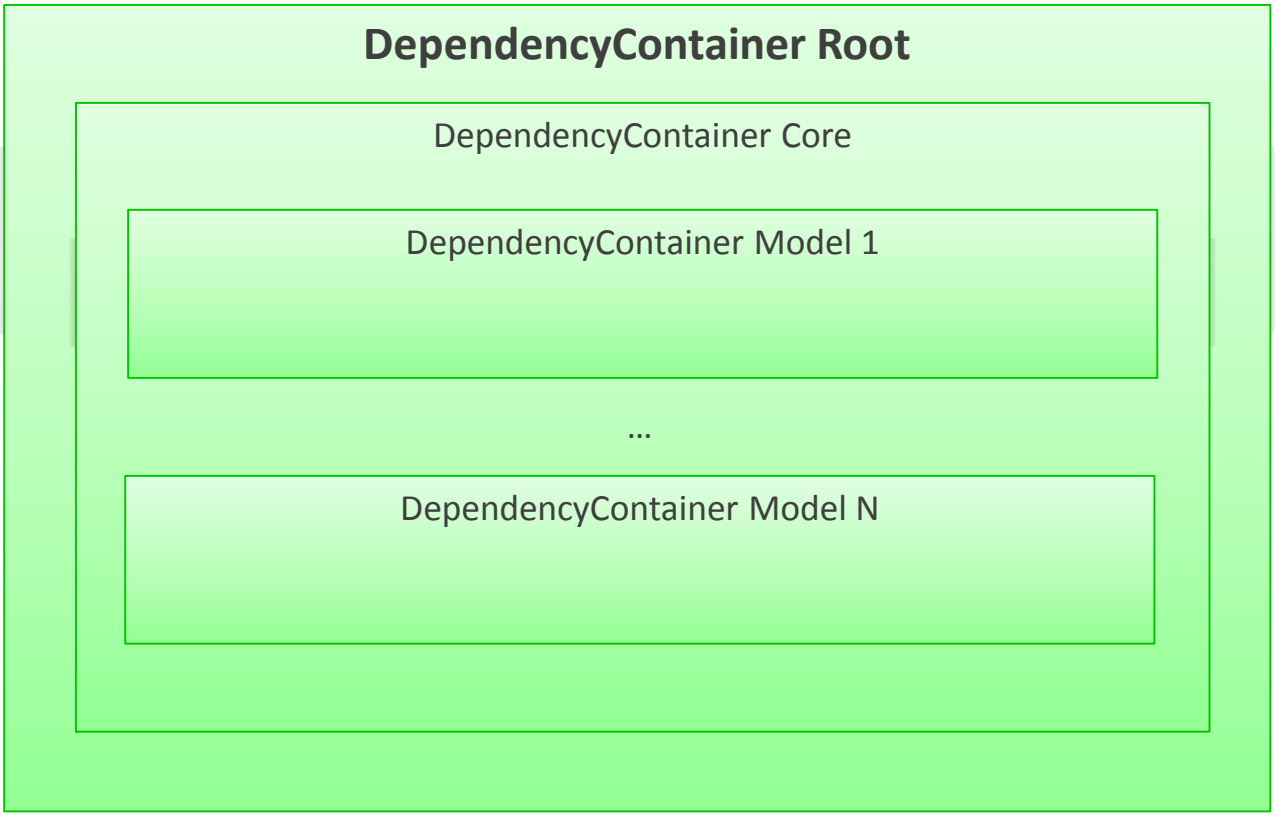


Figure 7.2. Integration of the core stages

The following two core stages are predefined:

`PrepareStage`
: Use this stage to prepare your application.

  All EB GUIDE GTF plugins, listed in the `platform.json`, are loaded. Configuration is ready and dependencies of the EB GUIDE GTF core functionality are resolved.

`RunStage`
: Use this stage to start your application.

  At this stage EB GUIDE GTF starts:

  - EB GUIDE model
  - EB GUIDE Monitor service
  - Custom applications

To implement predefined stages, see stage classes `gtf::launcher::PrepareStage` or `gtf::launcher::RunStage`. Register the stage at `gtf::dependencyresolver::DependencyCatalog`.

Derive custom core stages from `gtf::launcher::Stage`. You also need to create your own custom `StageProvider` that returns the custom stage name in the `gtf::launcher::StageProvider::stage()` function. Derive your `CustomStageProvider` from `gtf::launcher::StageProvider`. Register the stage at `gtf::dependencyresolver::DependencyCatalog`.

For an example on how to add a custom stage, see the EB GUIDE SDK example `CoreStageExample`. 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.

7.2. Tracing

With EB GUIDE GTF you can use traces on your platform. In the default EB GUIDE GTF delivery, traces are enabled. To customize the usage of traces on your platform, you need to create and configure a `tracing.json` configuration file. Depending on your configuration, you can provide output not only for your traces, but also for the EB GUIDE GTF traces. For each level of traces you can define the severity.

NOTE **Traces**

If you enable traces on your platform, the system may respond with delays.

For instructions on how to create, configure, and use traces, see chapter 13, “Tracing logging messages“.

For more information on the configuration items of the `tracing.json` configuration file, see section 19.11, “`tracing.json` configuration file”.

For an example, see section 19.11.2, “Example `tracing.json`”.

For an example of the `tracing.json` configuration file, see the EB GUIDE SDK example `TraceOutputExample`. 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.

EB GUIDE GTF provides the possibility of logging messages from an application using the `Logging` interface.

Based on the message severity, you can use the following trace macros from the `$GTF_INSTALL_PATH\win64\include\gtf\tracing\Logging.h` header file.

Table 7.1. Trace macros

Trace macro	Description
`GTF_DEBUG(channelName, MSG)`	The debug level provides fine-grained informational messages to debug an application.
`GTF_INFO(channelName, MSG)`	The information level provides informational messages that highlight the progress of the application at the high level.
`GTF_NOTICE(channelName, MSG)`	The notice level provides informational messages that highlight the progress of the application at the highest level.
`GTF_WARNING(channelName, MSG)`	The warning level provides potentially harmful situations or interfaces that are not used as expected.
`GTF_ERROR(channelName, MSG)`	The error level provides errors that might still allow the application to continue running.
`GTF_FATAL(channelName, MSG)`	The fatal level provides very severe errors that may lead the application to abort.

Tracing output is configured using the `tracing.json` configuration file.

The following table contains the available tracing output plugins.

Table 7.2. Available tracing output plugins

Plugin	Platform compatibility	Description
`GtfFileOutput`	All platforms	Writes the traces to the `log.txt` file. `log.txt` is located in the same directory as `tracing.json`.
`GtfKernelEventOutput`	QNX	For debugging with kernel event traces
`GtfVsDebugOutput`	Windows	If connected with the Visual Studio debugger, forwards the traces to Visual Studio debugger console

In case no plugins are configured in `tracing.json`, the default outputs in the following table are used, depending on the platform.

Table 7.3. Integrated tracing outputs

Output	Platform compatibility	Description
`stderr/stdout`	All platforms except Android	Forwards the traces to `stderr` and `stdout` streams:

Output	Platform compatibility	Description
		▶ `Warning`, `Error` and `Fatal` are traced to `stderr` ▶ `Debug`, `Info` and `Notice` are traced to `stdout`
`LogCat`	Android	Forwards the traces to `LogCat` The severities are mapped to the Android log levels as follows: ▶ `None` is mapped to `ANDROID_LOG_UNKNOWN` ▶ `Debug` is mapped to `ANDROID_LOG_DEBUG` ▶ `Info` and `Notice` are mapped to `ANDROID_LOG_INFO` ▶ `Warning` is mapped to `ANDROID_LOG_WARN` ▶ `Error` is mapped to `ANDROID_LOG_ERROR` ▶ `Fatal` is mapped to `ANDROID_LOG_FATAL`
EB GUIDE Monitor	Windows	Traces having severity `Notice` and above are displayed in the EB GUIDE Monitor **Logger** component.

For more information on the configuration items of the `tracing.json` configuration file, see section 19.11, "`tracing.json` configuration file".

7.3. Error handling

If an error occurs, EB GUIDE GTF produces tracing messages. The corresponding tracing severity is used based on the level of the occurred error. For more information about the tracing severities, see section 7.2, "Tracing".

The following table contains error levels that are managed by EB GUIDE GTF.

Table 7.4. Error levels in EB GUIDE GTF

Error level	Description	Example
Critical	EB GUIDE GTF shuts down, if possible, or produces a crash. The trace message is printed out using the fatal severity.	The system is out of memory, or an unsupported renderer plugin is used.
Controllable	EB GUIDE GTF tries to initialize an ordered, i.e. normal, shut down.	The model binary directory contains invalid information.
Minor	EB GUIDE GTF execution is not affected by the error, trace messages are generated.	An EB GUIDE GTF plugin is not valid and cannot be loaded.

7.3.1. Critical errors

To indicate critical errors, use the `GTF_OSAL_CRITICAL_ERROR` macro in custom applications and extensions. It is recommended to use `GTF_OSAL_CRITICAL_ERROR` when you start EB GUIDE GTF in an already existing process.

For an example on how to manage critical errors, 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.

To set callbacks to handle errors, use the `gtf_osal_set_assert_handlers` method.

7.3.2. Return value of `GtfStartup.exe`

If EB GUIDE GTF has detected an error, the `GtfStartup.exe` executable file returns `EXIT_FAILURE`. Otherwise it returns `EXIT_SUCCESS`.

7.4. Interface dependencies

EB GUIDE GTF provides a mechanism to manage the hierarchy of interfaces.

This mechanism is implemented by means of the `DependencyResolver` module. It enforces loose coupling between modules and is thread safe.

The interfaces of `DependencyResolver` module are found in the namespace `gtf::dependencyresolver`:

- `DependencyContainer`: Main interface that is used to register and retrieve instances.

- `DependencyObject`: Helper class that is used for adding data to a catalog.
- `Interface`: Parent for all interfaces that are registered in `DependencyContainer`.

For more information, see API documentation module DependencyResolver.

For instructions how to register interfaces and retrieve implementations of registered interfaces, see chapter 11, “Resolving interface dependencies“.

For an example on `DependencyResolver`, see the EB GUIDE SDK example `DependencyResolverExample`. 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.

7.5. Configuration of EB GUIDE GTF

EB GUIDE GTF has the possibility to read the configuration items that are included in the `platform.json` and `model.json` files. The `.json` files are part of the exported EB GUIDE model and contain configuration items that were specified in EB GUIDE Studio.

- To configure EB GUIDE GTF globally for all EB GUIDE models, use `platform.json` which contains core-related information.
- To configure a single EB GUIDE model, use `model.json`, which contains configuration items like model identifier, display size, or color mode.

For more information on these `.json` files, see section 19.6, “`model.json` configuration file” and section 19.8, “`platform.json` configuration file”.

For more information about JSON format, see http://www.json.org.

The `Configuration` module gives you the possibility to build, add and retrieve configuration items. User-defined applications or extensions can use these configuration items as well as retrieve them from the `Configuration` module using the `Settings` interface. The `Settings` interface is the main interface of the `Configuration` module.

Besides the standard `.json` files that `EB GUIDE GTF` uses, you can use your own `.json` files to register custom configuration items. To register the custom configuration items, use the `ConfigurationImporter` interface that parses a `.json` file and adds its elements into the `Configuration` module. For instructions, see section 12.6, “Using a custom `.json` file to define configuration items.”. You can also create your own `Settings` instances. For instructions, see section 12.7, “Creating new `Settings`”.

EB GUIDE GTF stores the items that were read from the `model.json` file at the model level `Settings` and the items that were read from the `platform.json` file at the core level `Settings`. However, since the model

level `Settings` interface is a child of the core level `Settings`, the model level `Settings` also has access to the configuration items stored at the core level. Note that this is not valid in the opposite direction. This means that each `Settings` instance has access to the configuration items of its ancestors, but has no knowledge about the configuration items of its children. For more information, see section 12.7, “Creating new `Settings`”.

A configuration item is a key-value pair. The key must be unique. Configuration items can have the values of the following types:

- `BooleanScalar`
- `IntegerScalar`
- `FloatScalar`
- `StringScalar`
- `BooleanList`
- `IntegerList`
- `FloatList`
- `StringList`
- `Object`
- `ObjectList`

The key naming structure is `namespace1.namespace2.[...].namespaceN.itemName`, for example `gtf.core.pluginstoload` or `gtf.core.servicemapper.port`.

NOTE

Thread safety of the `Configuration` API

Adding and retrieving operations of the `Configuration` module are thread-safe.

For instructions on how to use the `Configuration` module, see chapter 12, “Configuring EB GUIDE GTF“.

For more information, see API documentation: `Configuration` module.

7.6. EB GUIDE GTF plugin concept

Plugins are external libraries that EB GUIDE GTF can load and thus gain additional functionalities. EB GUIDE GTF differentiates between two types of plugins:

- Applications

  For more information, see section 5.3, “Interactions between EB GUIDE GTF and application”.
- Extensions

For more information, see section 5.2, “Interactions between EB GUIDE GTF and extensions”.

For instructions, see chapter 10, “Using the EB GUIDE GTF plugin mechanism“.

7.7. Run modes of EB GUIDE GTF

7.7.1. Run EB GUIDE GTF with `GtfStartup.exe`

On most target platforms EB GUIDE GTF is started by executing the `GtfStartup.exe` executable file. With this approach EB GUIDE GTF starts its own process. You can add own project-specific extensions and applications using EB GUIDE GTF plugins.

For more information, see section 7.6, “EB GUIDE GTF plugin concept”.

The `GtfStartup.exe` executable file provides platform-specific start-up code and interprets the `model.json` and `platform.json` configuration files. Additional functionality is available for specific platforms, for example command line parameter handling or detection of other EB GUIDE GTF instances.

7.7.1.1. Command line options

The following table lists command line options available in EB GUIDE GTF for `GtfStartup.exe` and explains their meaning. Undefined commands are ignored.

The general syntax for a command line is as follows: `GtfStartup.exe <option>`

Table 7.5. Command line options

Option	Description
`--version`	Optional parameter. Displays the EB GUIDE GTF version.
`--romfs $ROMFS_FILE_PATH $ROMFS_ROOT_FOLDER`	Optional parameter. In order to run, EB GUIDE GTF needs a file system. Therefore, if the target device has no file system available, read-only memory file system (RomFS) is used. If specified, the given RomFS is loaded. `$ROMFS_FILE_PATH` is the path to the RomFS container file and `$ROMFS_ROOT_FOLDER` is the root location in the RomFS file system.
`$MODEL_JSON_PATH`	Path to the `model.json` configuration file You can specify either of the following:

Option	Description
	▶ Only the directory where the file is stored. In this case the file with the name `model.json` is used. ▶ Only the file name, if the file is stored in the directory in which you called `GtfStartup.exe`. ▶ The directory where the file is stored and the file name, if the file name is not `model.json`.
`$PLATFORM_JSON_PATH`	Path to the `platform.json` configuration file You can specify either of the following: ▶ Only the directory where the file is stored. In this case the file with the name `platform.json` is used. ▶ Only the file name, if the file is stored in the directory in which you called `GtfStartup.exe`. ▶ The directory where the file is stored and the file name, if the file name is not `platform.json`.

For examples on how to redirect `stdout` and `stderr` messages to a file, see the table below.

Table 7.6. Redirecting `stdout` and `stderr` messages to a file

Option	Description
`> <output_file.txt>`	Redirects the output of `stdout` to a `.txt` file. The output file is created in the current working directory.
`> <output_file.txt> 2>&1`	Redirects `stderr` to `stdout` and then `stdout` to a `.txt` file. The output file is created in the current working directory.

Note that the output messages are not displayed after you entered a command line option. To see the output messages, use the `>` redirect option.

For more information about usage of `stdout` and `stderr` with the `Tracing` module, see section 7.2, “Tracing”.

Example 7.1.
Usage of command line options

To start an EB GUIDE model with the specified configuration files `model.json` and `platform.json`, enter the following:

```
GtfStartup.exe $MODEL_JSON_PATH $PLATFORM_JSON_PATH
```

Note the order: first you specify the path to `model.json`, second you specify the path to `platform.json`.

7.7.2. Run EB GUIDE GTF in an existing process

Another approach is to run EB GUIDE GTF within an already existing process. To do so, call the `gtf::launcher::main(...)` method. The related C++ header file `gtf/launcher/main.h` is part of the EB GUIDE SDK.

The `gtf::launcher::main(...)` method blocks the calling thread for the run-time of the started EB GUIDE GTF instance.

You can provide the following additions to the `gtf::launcher::main(...)` method:

- String parameters, processed like the command line parameters of the `GtfStartup.exe` executable file. For more information, see section 7.7.1.1, “Command line options”.
- Optional `gtf::dependencyresolver::DependencyContainer` instance that is used as root. You can provide extensions and applications to the EB GUIDE GTF instance using this root container.
  - You can provide interfaces without using an EB GUIDE GTF plugin.
  - You can provide interface instances, which are already available when calling the `gtf::launcher::main(...)` method.
- Optional ROM file system (RomFS) containers in the process address space. The ROM file system containers may contain configuration files, EB GUIDE GTF plugins, or exported EB GUIDE models. They are added before the EB GUIDE GTF start-up. Command line parameters provided in the string parameters can refer to the files in these ROM file system containers. For more information, see section 7.8, “Read-only memory file system support”.

7.8. Read-only memory file system support

A read-only memory file system (RomFS) is a read-only data structure representing a file system. The data structure can be provided to EB GUIDE GTF as follows:

- In a container file of the underlying operating system (OS)
- In a container file of an already provided RomFS
- As an address in the process address space at the EB GUIDE GTF start-up

For more details, see section 7.7.1, “Run EB GUIDE GTF with `GtfStartup.exe`” and section 7.7.2, “Run EB GUIDE GTF in an existing process”.

You can also use a RomFS to run EB GUIDE GTF on an embedded systems without the OS file system support.

The EB GUIDE GTF RomFS support is implemented in user space and does not depend on the underlying OS.

EB GUIDE provides a tool for creating RomFS containers, called `RomFsCreate`, which contains for example the exported EB GUIDE Studio model. Find `RomFsCreate` in the `$GUIDE_INSTALL_PATH/tools/RomFs-Tools` directory.

When you execute `RomFsCreate.exe` from the command line, the following options are available.

Table 7.7. Command line options

Option	Description
`--create-c-file BASE_NAME`	Creates a `.c` source file that contains the RomFS data and a `.h` header file. The files are named `BASE_NAME.c` and `BASE_NAME.h`.
`--output-dir TARGET_DIRECTORY`	Defines the location were the `.romfs` file is be created
`--max-size N`	Specifies the maximum size of the container
`-h or --help`	Displays the help

Use command line to perform the following RomFS-related tasks:

Create a RomFS container
: `RomFsCreate.exe romfs_root_directory` creates the file `romfs_root_directory.romfs`. This file contains `romfs_root_directory`.

Create a RomFS container and specify the name of the resulting file
: `RomFsCreate.exe romfs_root_directory image` creates the file `image.romfs`. This file contains `romfs_root_directory`.

Limit the size of the resulting container
: Specify `--max-size N` on the command line. If the size limit you specify is exceeded, `RomFsCreate` sends an error message and stops putting files into the container. The maximum size `max-size` is defined in bytes.

Create a RomFS container and put it ready-to-use in a C-array
: `RomFsCreate.exe romfs_root_directory --create-c-file c_array` creates the `romfs_root_directory.romfs` file. This file contains `romfs_root_directory`.

  Content is put in the file `c_array.c` as `const unsigned char romfs_root_directory[] = "...";`. `"..."` is the content of the container encoded in C hexadecimal literals.

  Additionally the `c_array.h` header file is created. The header file has an `extern const unsigned char romfs_root_directory[N];` forward declaration which you can include and use in your code.

  The `--max-size N` parameter is respected.

Create a RomFS container, specify the name of the resulting file and put it, ready to use, in a C-array

`RomFsCreate.exe romfs_root_directory image --create-c-file c_array` creates the file `image.romfs`. This file contains `romfs_root_directory`. Content is put in file `c_array.c` as `const unsigned char romfs_root_directory[] = "...";`.

`"..."` is the content of the container encoded in C heximal literals.

Additionally a `c_array.h` header file is created. This header file has an `extern const unsigned char romfs_root_directory[N];` forward declaration, which you can include and use in your code.

The `--max-size N` parameter is respected.

For instructions of how to run EB GUIDE GTF from a RomFS container, see section 9.4, “Running EB GUIDE GTF with a read-only memory file system (RomFS) container”.

8. Model concepts

8.1. Configure EB GUIDE GTF with an exported EB GUIDE model

EB GUIDE Studio provides input to EB GUIDE GTF in form of an EB GUIDE model that is generated during export. EB GUIDE GTF runs the exported EB GUIDE model in the following use cases:

- During the simulation in EB GUIDE Studio
- On a dedicated target platform

The EB GUIDE model consists of the following parts:

- The binary model description files:
  - `eventMap.gtf` is used to configure the event system.
  - `datapool.gtf` is used to configure the datapool.
  - `model.bin` is used to configure the state machines.
  - `types.bin` is used to configure the type system.
  - `scenes.gtf` is used to configure the scenes and views.
  - `resources.gtf` is used to configure the resource system.
- The text-based configuration files:
  - `model.json` contains configuration items for the EB GUIDE model.
  - `platform.json` is an optional configuration file in the exported directory. If this file is not available in the exported directory, the `platform.json` file in the `$GUIDE_INSTALL_PATH` directory is used.
  - `tracing.json` is an optional configuration file and is not generated by EB GUIDE Studio. If you want to use the tracing mechanism, you have to create this file manually. The possibility to overwrite the default configuration of the tracing system exists. For more information, see table 7.1, “Trace macros”.
  - `monitor.cfg` is a configuration file for EB GUIDE Monitor.
- The `$GUIDE_INSTALL_PATH\platform\win64\include` directory with `C++` headers that contain the `define` macros to address the datapool items, events, languages and skins defined in the EB GUIDE Studio model.
- The `resources` directory is a copy of the `resources` directory of the EB GUIDE Studio project that is also visible in the **Assets** component of EB GUIDE Studio.

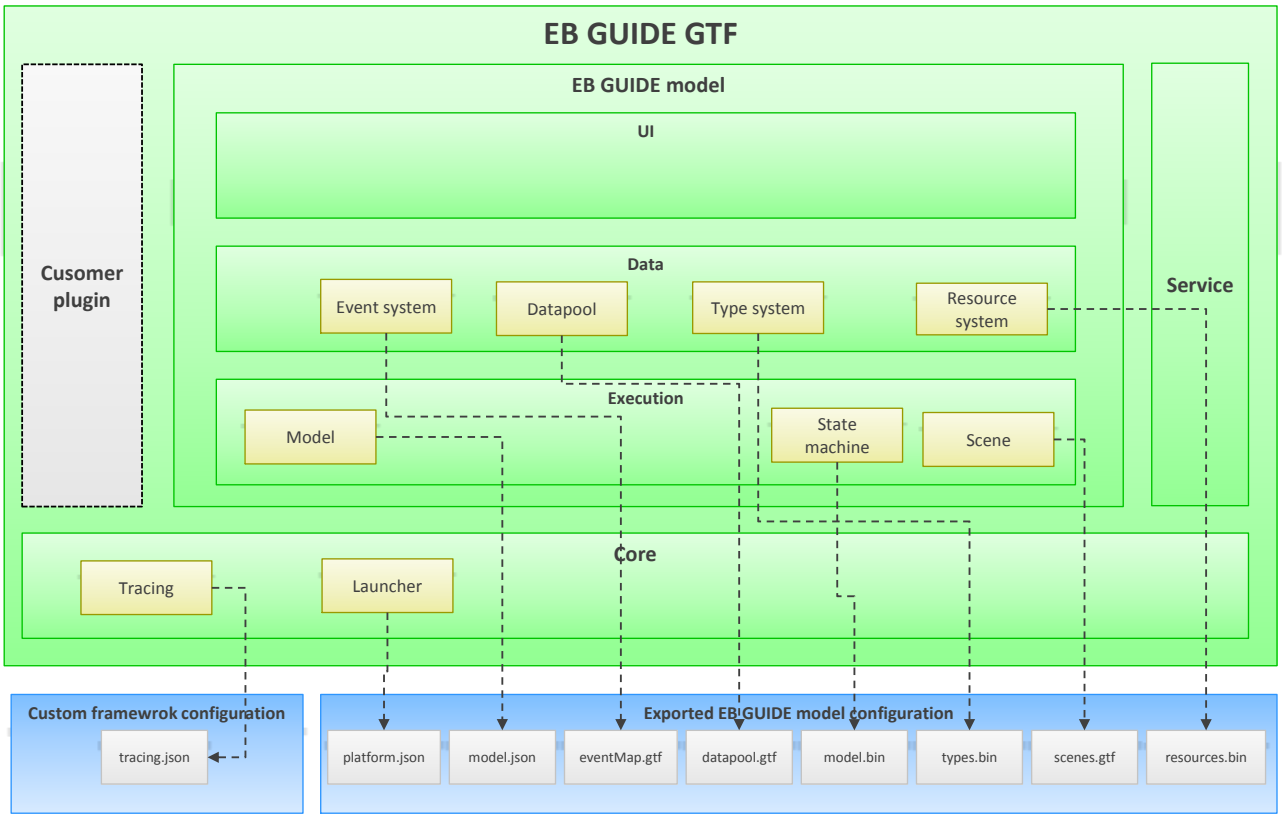


Figure 8.1. Configuration files overview

8.2. Model life cycle stages

Like the life cycle of the core the stages of the model life cycle are defined in an order that is followed when starting a model. For the core life cycle, see section 7.1, “Core life cycle stages”.

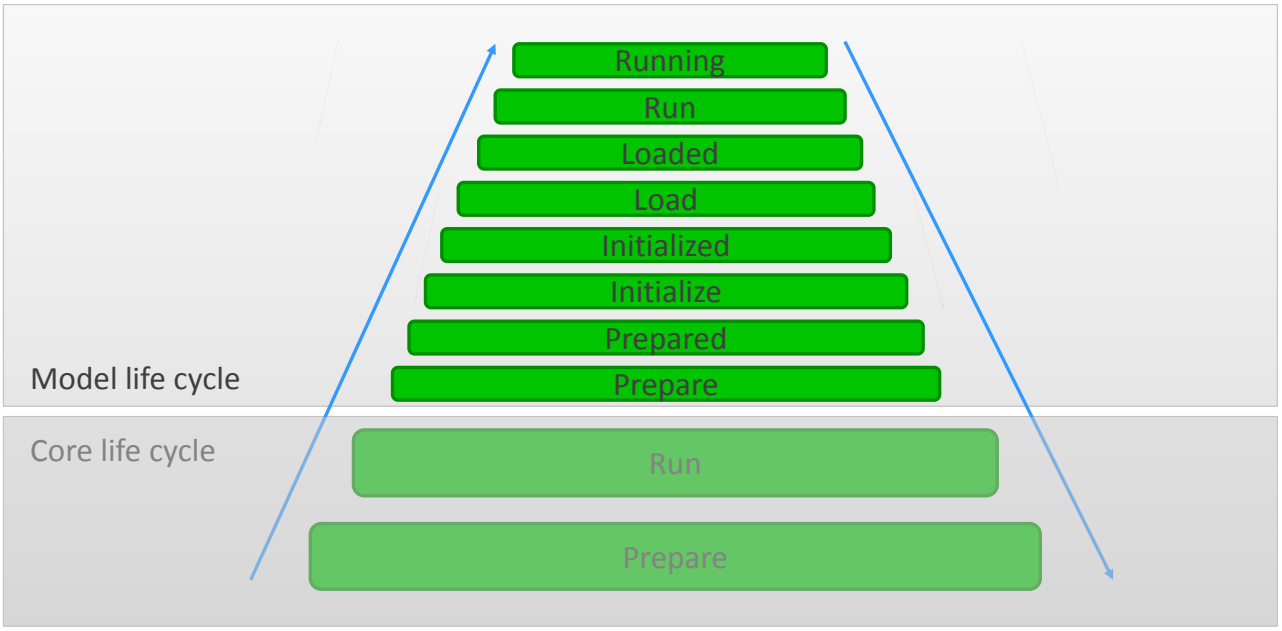


Figure 8.2. Overview of the EB GUIDE GTF life cycle stages

As depicted above, the model life starts during the run stage of the EB GUIDE GTF core. This means the model life cycle is not a direct successor of the core life cycle as there is always one core but there can be many models living on it in parallel.

Each EB GUIDE model has its own `gtf::dependencyresolver::DependencyContainer` instance, which is provided to the model stage `create` functions. This instance is dedicated to one model only and also differs from the instance provided to the core stages of the EB GUIDE core life cycle.

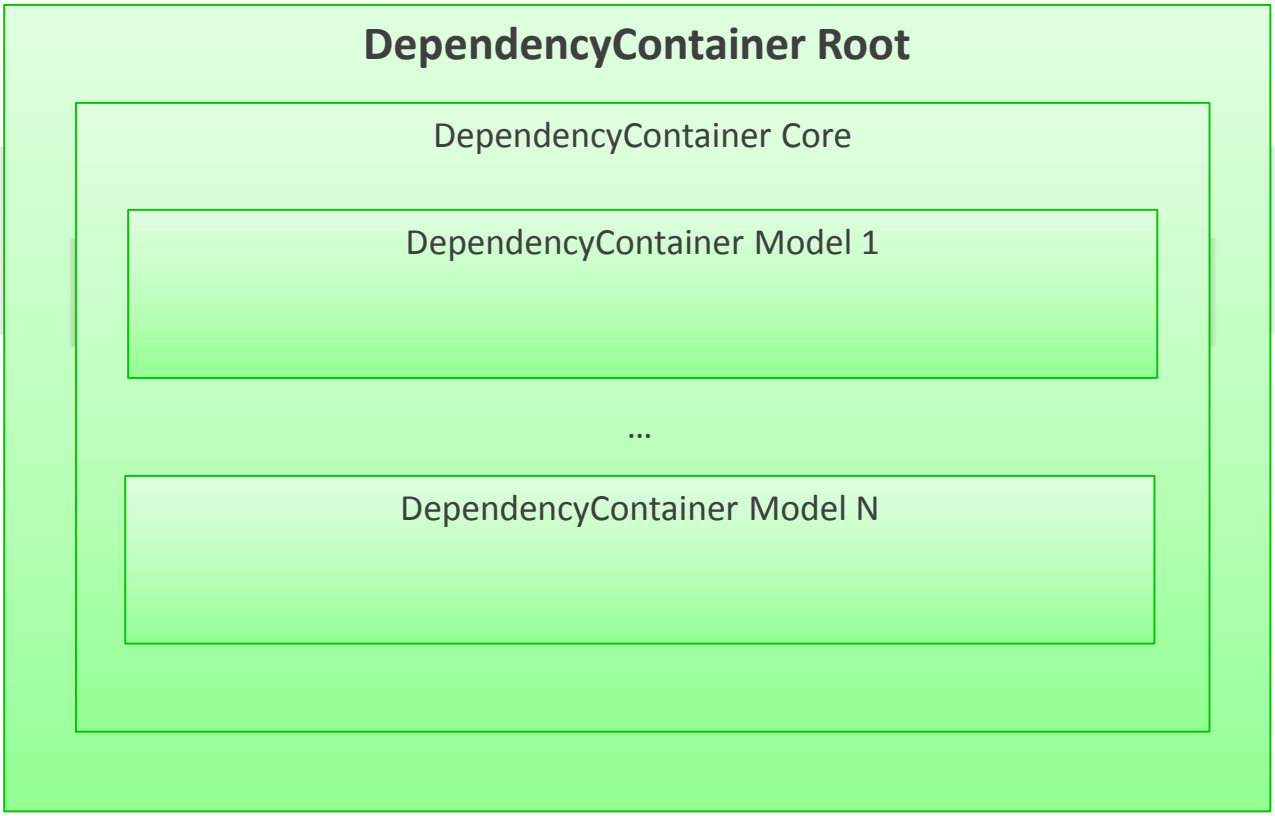


Figure 8.3. Integration of the model stages

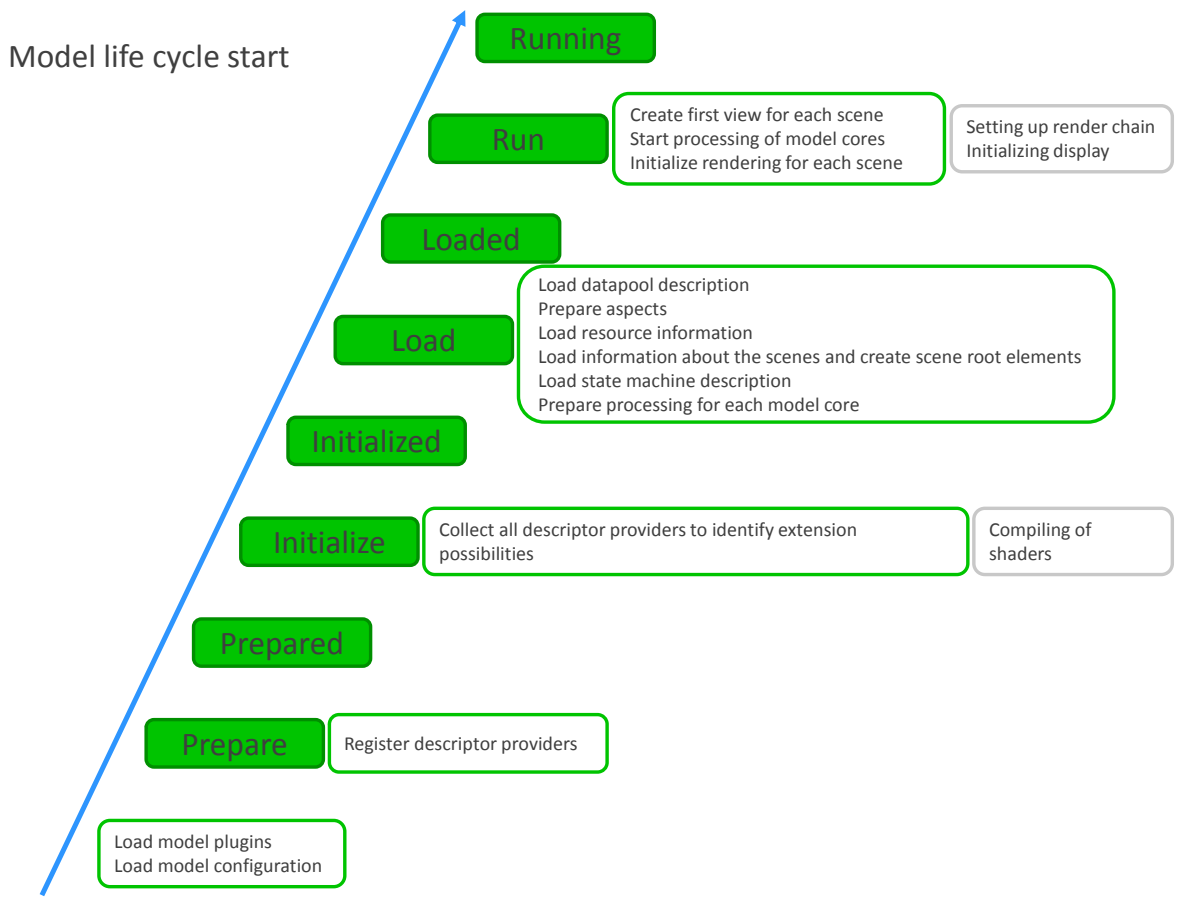


Figure 8.4. Overview of the EB GUIDE GTF model startup sequence

Each loaded EB GUIDE model has its own life cycle. As depicted in the picture above, the model life cycle consists of eight successive predefined stages. It may contain custom stages. The stages Prepare, Initialize, Load and Run are designated for the internal EB GUIDE GTF startup steps. The stages Prepared, Initialized, Loaded and Running offer the possibility of reacting on the startup steps and extending the EB GUIDE GTF startup sequence with additional steps:

- `PreparedStage`

  Within this stage you add descriptors for your extensions.

- `InitializedStage`

  All descriptors are published and available. Scene configuration is loaded.

- `LoadedStage`

  Datapool and state machines are available and initialized. The view description is loaded. Previously announced EB GUIDE Script functions are registered. Core run-time framework is created. Everything is ready to start.

- `RunningStage`

The renderer starts and the window is created, the state machines and threads are started. EB GUIDE model runs.

You can customize the EB GUIDE model life cycle by modifying and adding stages. You can add custom stages at any point before, after, or between the predefined stages. You can define several instances per stage.

To implement a predefined stage, derive it from a predefined stage class and register it at `gtf::dependencyresolver::DependencyCatalog`. The following are the predefined stage classes:

- `gtf::model::PrepareStage`
- `gtf::model::PreparedStage`
- `gtf::model::InitializeStage`
- `gtf::model::InitializedStage`
- `gtf::model::LoadStage`
- `gtf::model::LoadedStage`
- `gtf::model::RunStage`
- `gtf::model::RunningStage`

To define a custom model stage, derive it from `gtf::model::Stage`. You have to create an own custom `StageProvider` that returns the custom stage name in the `gtf::model::StageProvider::stage` function. `CustomStageProvider` is derived from `gtf::model::StageProvider` for custom model stages. You need to register both in `gtf::dependencyresolver::DependencyCatalog`.

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

For an example on how to add a custom stage, see the EB GUIDE SDK example `ModelStageExample`. 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.

When a model shuts down, the EB GUIDE GTF stays active until no model is running inside of the core anymore. When no models are running, the core initiates the shutdown sequence. Shutdown sequence is carried out in the opposite order of the start sequence.

8.3. Model structure

8.3.1. Model MVC

Model–view–controller (MVC) is an architectural pattern used in the application development. MVC consists of the following three parts:

- The model part represents the data.
- The view part is the visual representation of the model.
- The controller part links the user and the system.

MVC decouples the views and models by establishing a subscribe and notify protocol.

EB GUIDE GTF uses the MVC-pattern on different layers:

- The first layer is the model layer, see figure 8.5, “MVC model layer”.
- The second layer is the widget layer, which is described in section 8.5.1, “Widget MVC”.

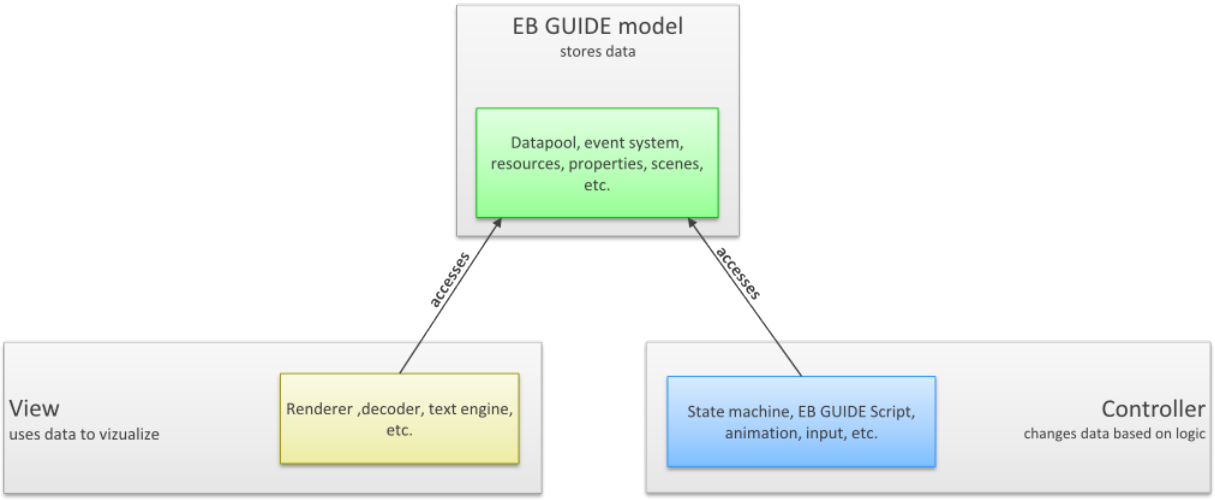


Figure 8.5. MVC model layer

8.3.2. Model runtime structure

An EB GUIDE model consists of several connected pieces as shown in figure 8.6, “Model structure”. Depending on the configuration a single EB GUIDE model can contain several so called `CoreRuntimes`.

A `CoreRuntime` is responsible for processing changes within all scenes connected to it as well as notifications about changes from the outside, like datapool changes, events or user input.

In order to react on these changes, each `CoreRuntime` owns own instances of a `Datapool`, an `EventSystem` and observers for scene changes.

For details on how this processing is done, see section 8.6, “Update processing in EB GUIDE GTF”.

For details about scene and property handling, see section 8.5, “Scenes and properties”.

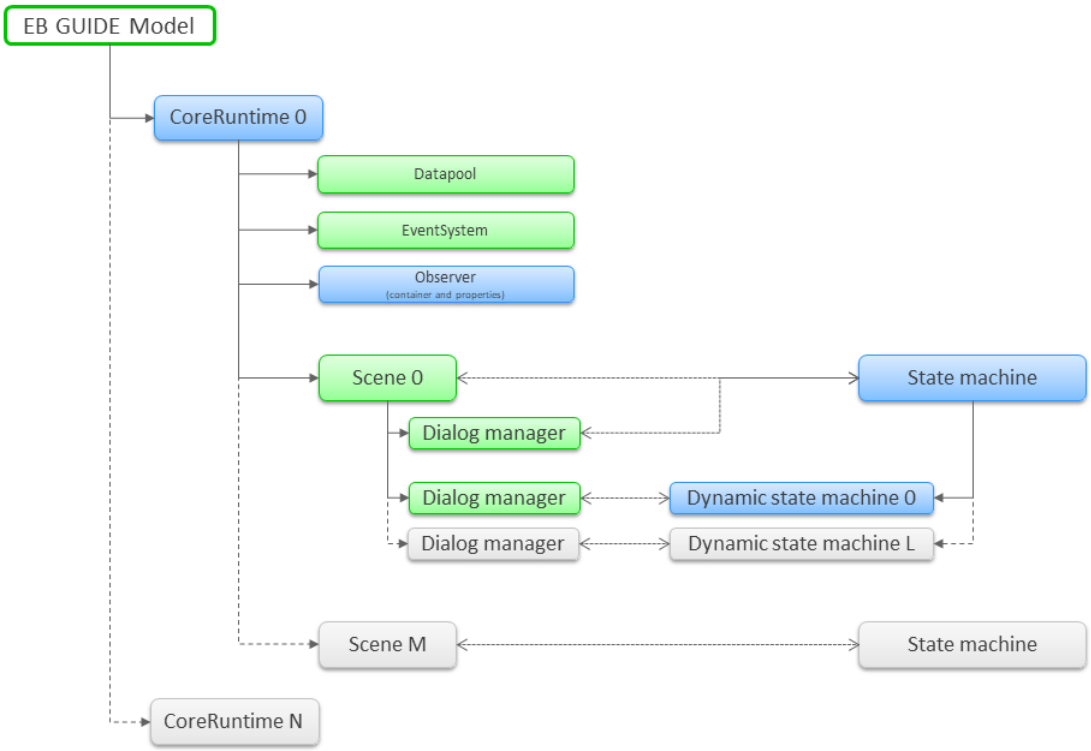


Figure 8.6. Model structure

Each of these `CoreRuntimes` is represented in the model configuration by a name in the configuration item "gtf.model.coreNames" and a configuration item describing the attributes of the `CoreRuntime`.

Such attributes are as follows:

- An identifier within the model for this `CoreRuntime`
- A flag determining if `CoreRuntime` uses an own thread for processing

```
"gtf": {
    "model": {
      "coreNames": [
        "HMI",    // CoreRuntime 0
                  // ...
        "Hud"     // CoreRuntime N
      ],

      "cores": {
        "HMI": {
          "id": 0,
          "ownThread": false
        },
```

```
        "Hud": {
          "id": 1,
          "ownThread": true
        }

      },
```

Each `CoreRuntime` can have several scenes attached. The processing of each attached scene is done during the update processing cycle of `CoreRuntime`. The model configuration contains two configuration items defining which scenes EB GUIDE GTF shall load and what their distinct attributes are. As shown in the following example, the configuration item `"gtf.model.sceneNames"` contains the names of all configured scenes and for each scene a configuration item describing its attributes.

```
"sceneNames": [
        "Main",     // Scene 0
                    // ...
        "Hud-SM"    // Scene M
      ],

      "scenes": {
        "Main": {
          "name": "Main",   // connected to statemachine with the name "Main"   (Scene 0)
          // ...
          "context": 0      // connected to CoreRuntime 0
        },
        "Hud-SM": {
          "name": "Hud-SM", // connected to statemachine with the name "Hud-SM" (Scene M)
          // ...
          "context": 1      // connected to CoreRuntime N (1 in this example)
        }
      },
```

The following attributes of the scene define the runtime structure:

- The `"name"` attribute defines that this scene will be connected to a state machine with this name.
- The `"context"` attribute defines that the scene will be part of `CoreRuntime` with the same identifier.

Each scene creates `DialogManager` for the state machine it is attached to and for all dynamic state machines, that are part of dynamic state lists within the attached state machine. `DialogManager` takes care of dialogs, foe example the current view or talk, and their dynamic change, for example view transition animations.

8.3.3. Scene structure

In an EB GUIDE model there can be several scenes, for example for different displays on a target platform, cluster, and HU-displays.

As shown in figure 8.5, “MVC model layer”, each scene owns a `Dialog manager` for each active state machine.

`Dialog manager` contains the following:

- The `current` dialog, which is for example the current shown view or talk of that state machine
- An `upcoming` dialog, which is the next view of that state machine during an animated view transition

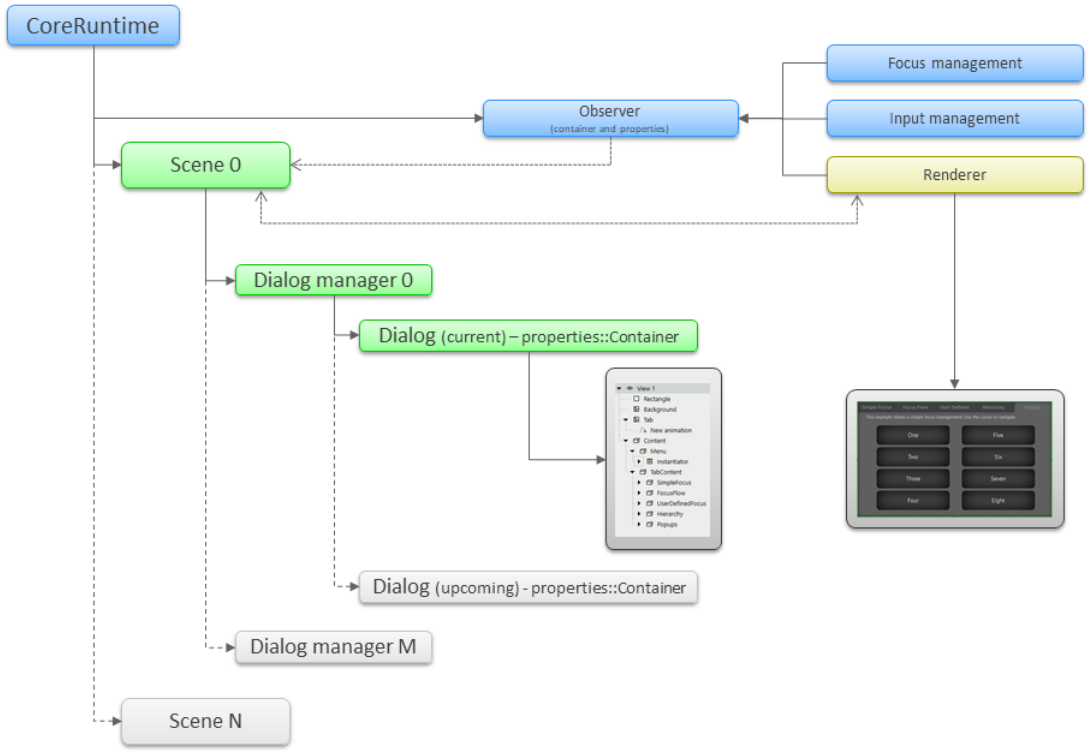


Figure 8.7. Runtime structure

As described in figure 8.7, “Runtime structure”, the scene is an instance managing the structure of all dialogs contained in the state machines that are configured for it. >This means the scene is represented as the root element of the scene element tree. Such a scene element tree is a rooted tree structured build from `properties::Container` instances, shown in section 8.5, “Scenes and properties”.

The scene is observed for structural and property changes using `Observer` mechanisms, which are used by input and focus management.

The renderer also makes use of these `Observers` in order to evaluate possible changes that need to be reflected through a change of the visualization.

For every scene configured for a haptic state machine there is always one exclusive renderer instance. This instance will evaluate the result of the scene update processing to provide a fitting visualization. The evaluation as well as the process of rendering is executed on the thread, the scene is attached to through its `CoreRuntime`.

8.4. Extensions description

Some model element types that are available in EB GUIDE Studio are provided by EB GUIDE GTF and its extensions. Besides this EB GUIDE GTF provides interfaces for adding own implementations of the following:

- Widgets
- Widget features
- EB GUIDE Script functions

The newly added model elements are represented as follows:

- As an abstract description that defines what will be available, for example which properties a widget might have
- As an optional implementation that is registered at a later stage, for example registering a concrete implemented C++ function for a EB GUIDE Script function

The description is introduced with a registration of an implementation of the `DescriptorProvider` interface, at the latest during the `Prepared` stage of the EB GUIDE model start-up phase.

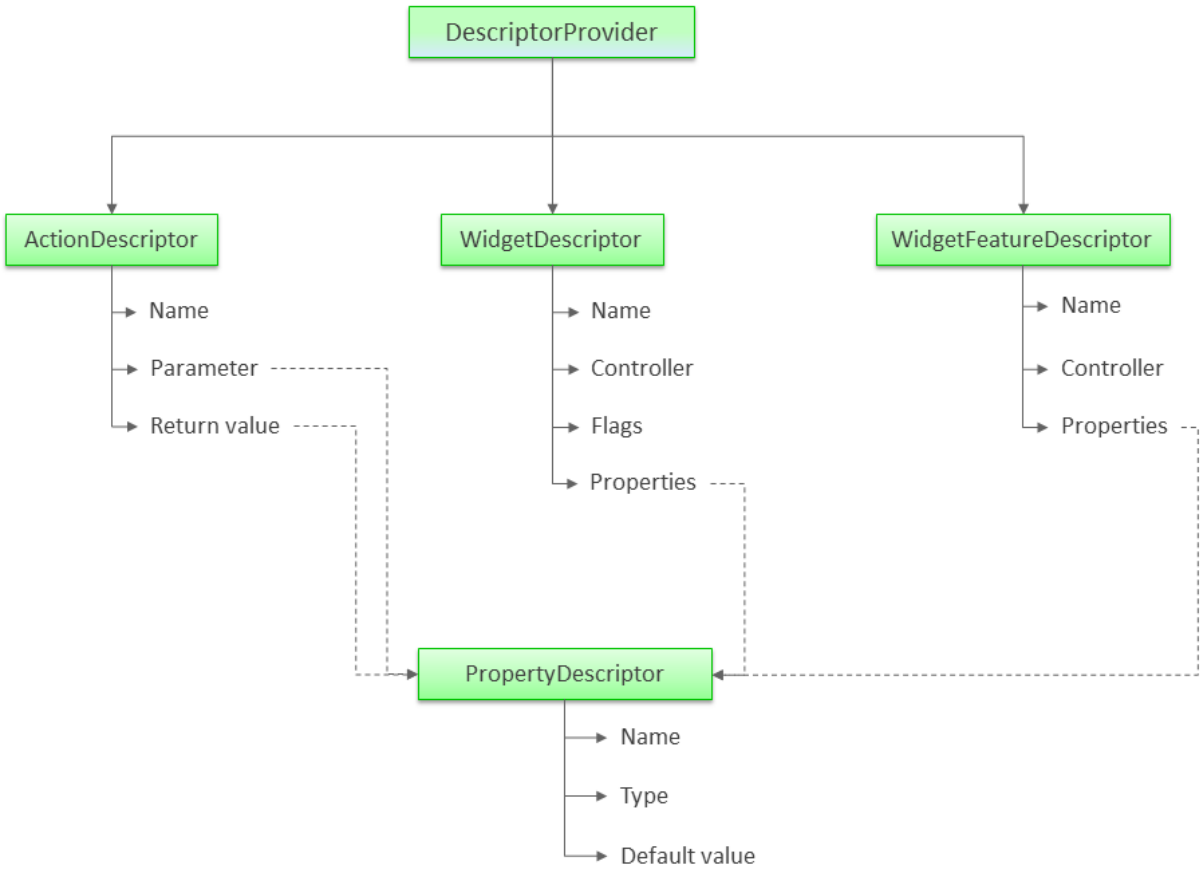


Figure 8.8. `DescriptorProvider`

As shown in figure 8.8, "`DescriptorProvider`", such a `DescriptorProvider` interface retrieves the information about the following added model elements:

- Widgets, represented as `WidgetDescriptor`
- Widget features, represented as `WidgetFeatureDescriptor`
- EB GUIDE Script functions, represented as `ActionDescriptor`

These model elements use `PropertyDescriptor` for describing typed value information, like widget properties or the parameter of an EB GUIDE Script function. In order to register an implementation, the `WidgetDescriptor` and `WidgetFeatureDescriptor` interfaces can specify `InterfaceName` of a `scene::element::Controller` implementation.

For more information on how the scene element tree construction utilizes this information of the `WidgetDescriptor` and `WidgetFeatureDescriptor` instances, see section 8.5.2, "Construction and decomposition of a scene element tree".

To register an implementation of an EB GUIDE Script function, EB GUIDE GTF needs to know this EB GUIDE Script function. This means that only the implementations can be attached to EB GUIDE Script functions that were included in the exported EB GUIDE model and registered by `DescriptorProvider`.

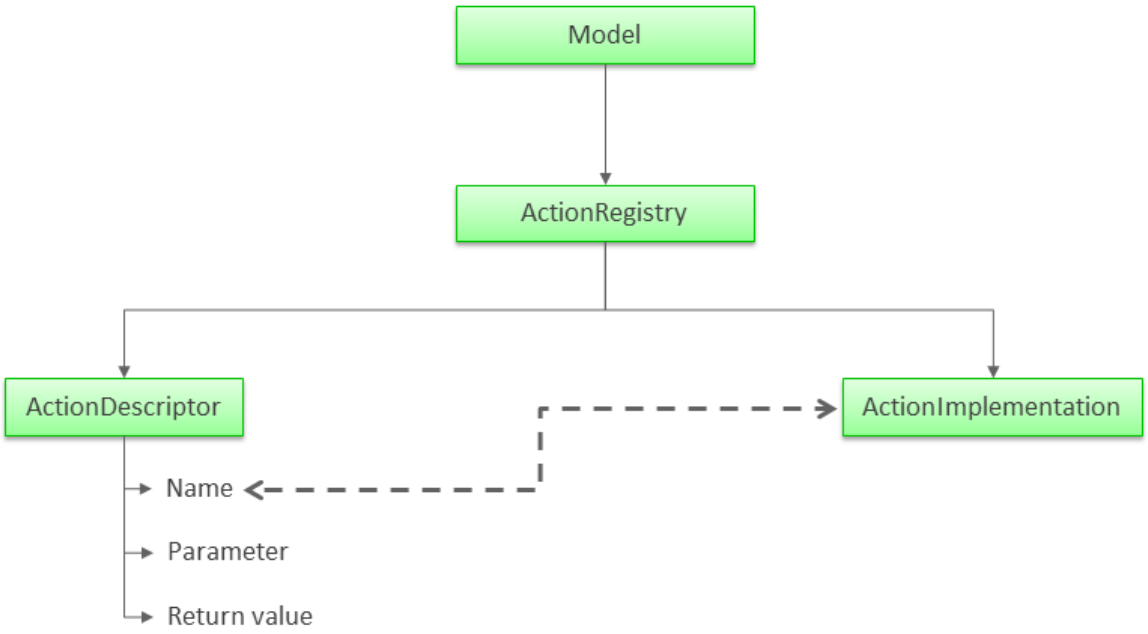


Figure 8.9. `DescriptorProvider`

As shown in figure 8.9, "`DescriptorProvider`", every EB GUIDE model contains `ActionRegistry` that manages the registered EB GUIDE Script function descriptions and their implementations. At this instance an EB GUIDE model specific identifier of an EB GUIDE Script function can be acquired for which an implementation can be registered. When EB GUIDE Script is executing a script containing such a function identifier, the registry will be asked and the respective function will be called. If the function is not available, then the script will be stopped immediately and an error message is provided using the EB GUIDE GTF trace mechanisms, for example logged in EB GUIDE Monitor.

8.5. Scenes and properties

The central parts for the scene handling inside EB GUIDE GTF are the `Scene` and `Properties` components.

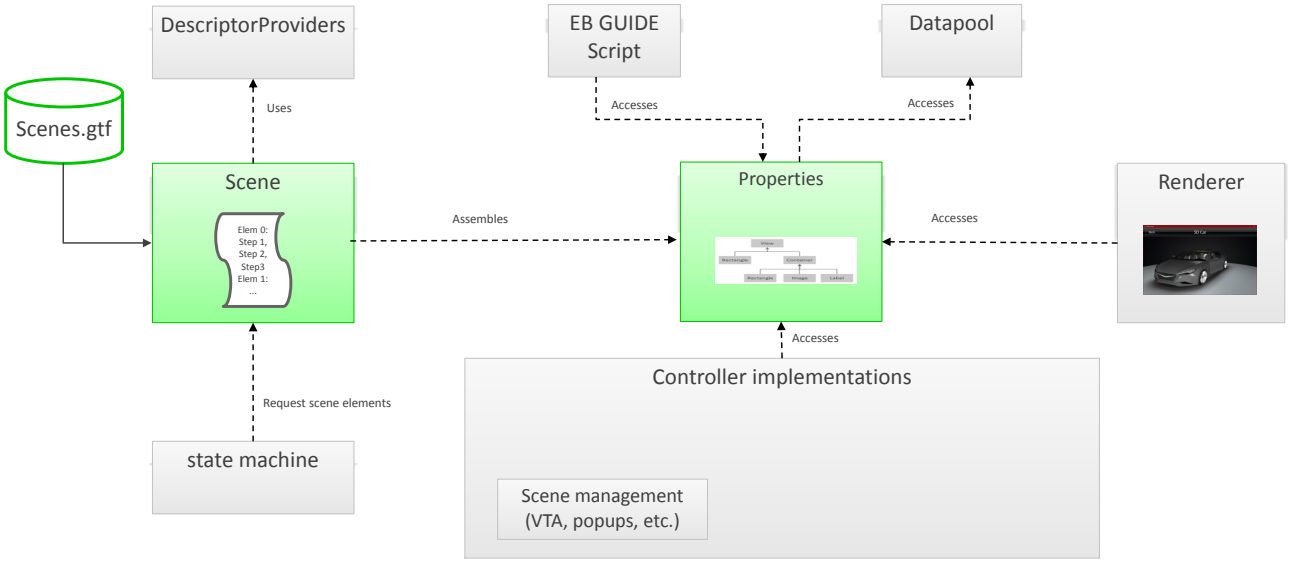


Figure 8.10. Scene and property handling

As shown in figure 8.10, “Scene and property handling”, the data about the exported scene elements and the descriptor data are used to assemble the scene element tree. This is represented by a root tree of `properties::Container` instances. The `properties::Container` instances contain all properties of the widgets modelled as part of the EB GUIDE model as well as the properties of all attached widget features and user-defined properties. The `properties::Container` instances are accessed by `scene::element::Controller` implementations, like the layout widget features or the instantiator widget.

As properties can also be scripted values or accessed by such, they can also be accessed and manipulated by EB GUIDE Script.

The renderer also observes and accesses the properties in order to create a fitting visualization. This is also reflected in the representation of a widget. For more information, see section 8.5.1, “Widget MVC”.

8.5.1. Widget MVC

Like the overall model structure, a widget can also be represented as an MVC-pattern.

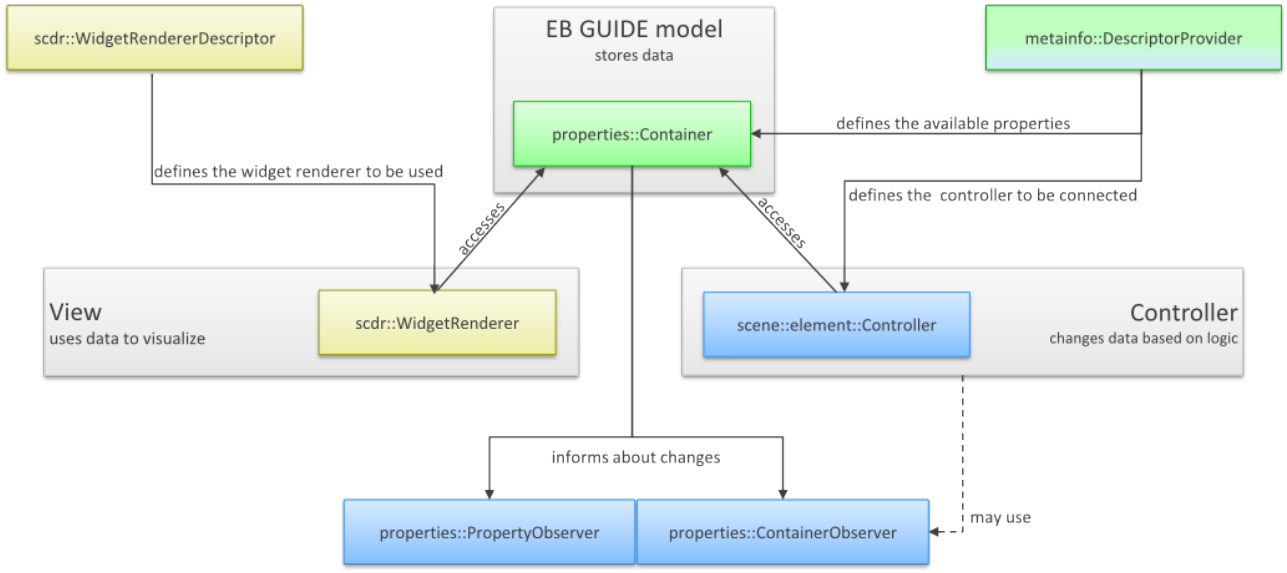


Figure 8.11. MVC widget layer

The model compartment hereby is represented through the `properties::Container` that stores all properties and therefore the data of the widget.

The view compartment is represented through an implementation of the `scdr::WidgetRenderer` interface, which evaluates the properties and tree structure of the scene element tree in order to draw it.

The controller compartment is represented by the `scene::element::Controller` implementations, which evaluate and manipulate the properties or the tree structure of the scene element tree.

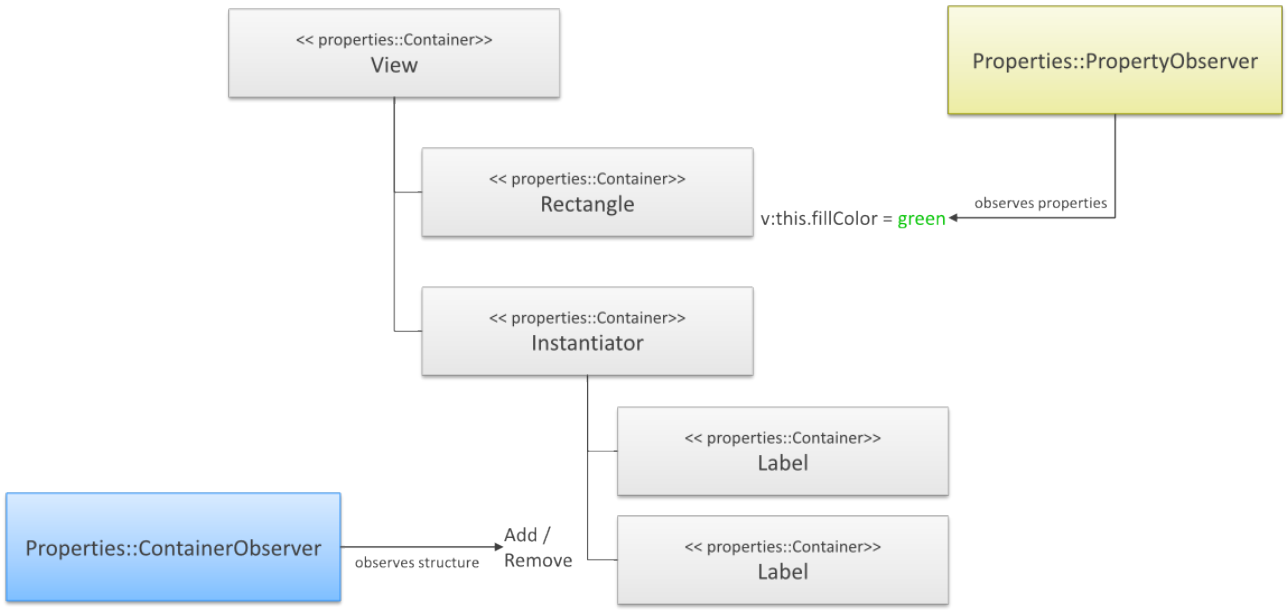


Figure 8.12. Subscribe and notify protocol

EB GUIDE GTF provides additional to the classic MVC the following observer mechanisms:

- `Properties::PropertyObsever` notifies when a property value changes.
- `Properties::ContainerObserver` notifies the structure of the scene element tree changes.

8.5.2. Construction and decomposition of a scene element tree

In order to construct a scene, or parts of the scene like a view or an instantiator child, `scene::Factory` is using information that the exported EB GUIDE model and the `metainfo::DescriptorProvider` contain.

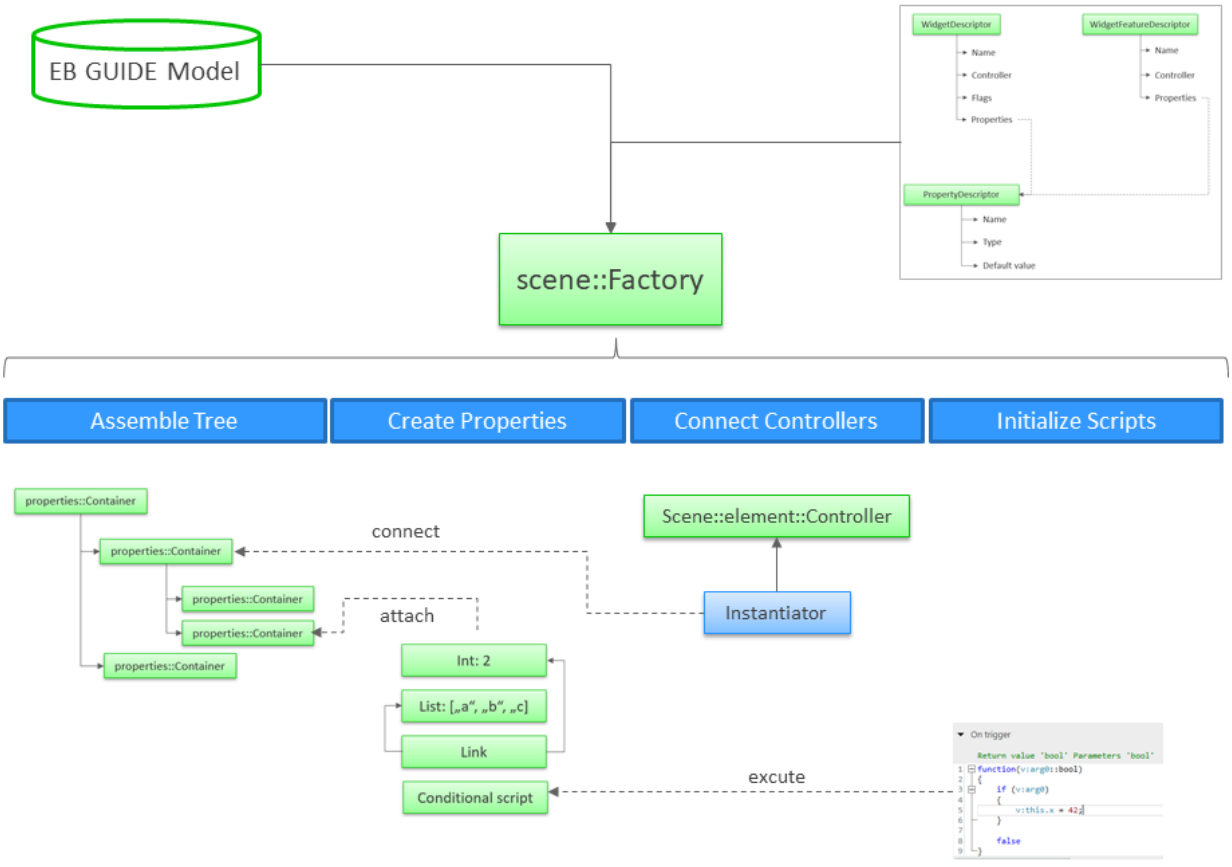


Figure 8.13. Creating a scene

Whenever a single scene element or even a fairly complex scene is created, `scene::Factory` runs through the following phases:

1. Assembling the tree means that the instances of `properties::Container` are created and connected.

2. Creating properties means that the properties are created and their values are loaded from the scenes description file.
3. Controller classes are connected means that the `scene::element::Controller` implementations are created and connected to the `properties::Container` instances.
4. EB GUIDE Script functions are initialized means that the conditional scripts are initially executed.

After manipulating the scene element tree structure every observing entity, like for example renderers are informed so that they can react on this.

The counter action to the creation of a scene is its decomposition. The decomposition is done in the following phases:

1. Decoupling the root means the sub-tree that needs to be decomposed looses its connection to the parent.
2. Information about the upcoming destruction is forwarded to the observer of the structure.
3. Controller classes are disconnected.
4. Final deletion of all `properties::Controller` and their property handles takes place.

8.5.3. Renderer

Renderer modules are sets of classes that implement the rendering algorithms for a particular widget. The `IWidgetRenderer` is the starting point for all renderer modules. It implements the actual drawing algorithm for the particular type of widget that is being implemented. It is also responsible for drawing all widgets of that particular type in the widget tree.

In order to be able to draw an instance of a widget, `IWidgetRenderer` requires data that is specific to an instance of the corresponding widget. For that, `IWidgetRenderer` creates instances of the `RenderObject` interface. An instance of a widget and `RenderObject` always have a one-to-one correspondence. Implement this interface in order to store your widget specific data that you require for drawing, for example, the GPU resources.

EB GUIDE GTF invokes `IWidgetRenderer` at multiple points in its execution sequence. The three most important phases are as follows:

Traversal phase
: During the traversal phase, `IWidgetRenderer` is responsible for loading parameters from the widget tree that are specific for the type of widget being rendered. Further, `IWidgetRenderer` is responsible for keeping track of the `RenderObject` instances being visited during the traversal phase so that it can update all resources required in the following update phase. After the update phase the next phase is the render phase. In this phase `IWidgetRenderer` executes the drawing algorithms.

Update phase
: The update phase is designed to issue the GPU-relevant code, for example GPU resource updates or draw calls using OpenGL ES.

Render phase

The render phase, as well as the update phase, is also designed to issue the GPU-relevant code, for example GPU resource updates or draw calls using OpenGL ES.

This kind of GPU programming code must be executed in instances of the `Command` interface. To execute instances of the `Command` interface, the renderer must enqueue `Command` in the relevant queues of `CommandBuffer` that is provided during the update and render phases.

8.5.4. Animations and view transition animations

The animation system describes the change of properties over time and consists of animation containers and curves. An animation container defines the processing of a sequence of curves. A curve defines the change of a single property. Each animation curve is created as a scene element controller. The controller is triggered for evaluating the property value changes that depend on the type of the curve, for example, linear interpolation curve where the relation between the start and end property value has to be evaluated. The animation system is based on property containers, scene element controllers, and the observer mechanism.

View transition animation consists of the following steps:

1. The state machine enters a new view state and triggers the building the new view.
2. The view transition animations are executed.
3. The old view is destructed.

8.5.5. Focus policy

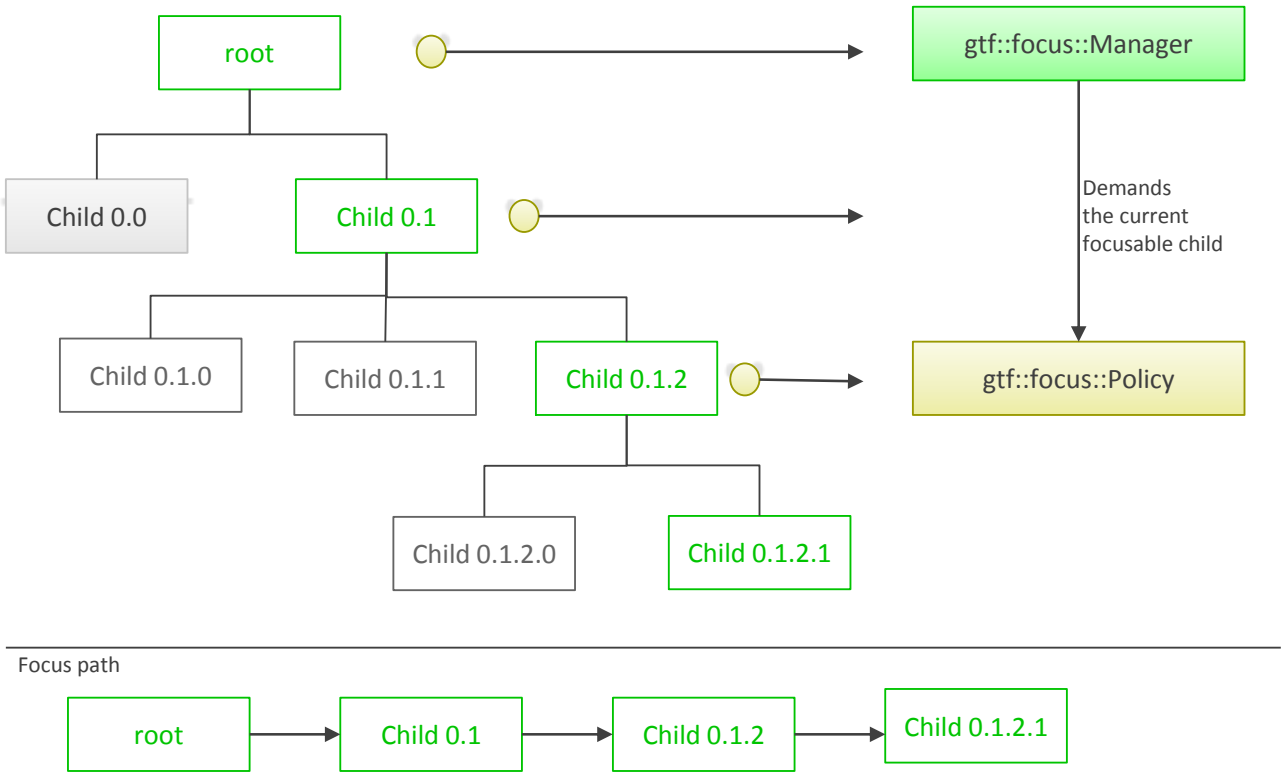


Figure 8.14. Focus policy in EB GUIDE TF

Focus manager traverses the scene and searches for `gtf::focus::policy`. Each policy on the focus path is asked for the next valid element of the path. If there is no further policy, the last leaf will be the current focused element. Custom focus policies can be added with the `gtf::focus::policy` interface. For more information, see EB GUIDE GTF API.

8.6. Update processing in EB GUIDE GTF

EB GUIDE GTF reacts and updates on several occurrences:

External occurrences, for example as follows:

- Operating system events
- Input devices
- Applications sending events or modifying datapool items
- External switching of languages and skins

- Interaction with other local or remote connected EB GUIDE GTF instances
- Interaction with a connected EB GUIDE Monitor

Internal occurrences, for example as follows:

- Delayed events
- Animations
- Issues in the model, for example cyclic dependency and mutual modification of model elements

Extensions

Extensions may need to execute operation that is synchronized with the update processing of EB GUIDE GTF. EB GUIDE GTF provides dedicated EB GUIDE SDK interface methods for this purpose.

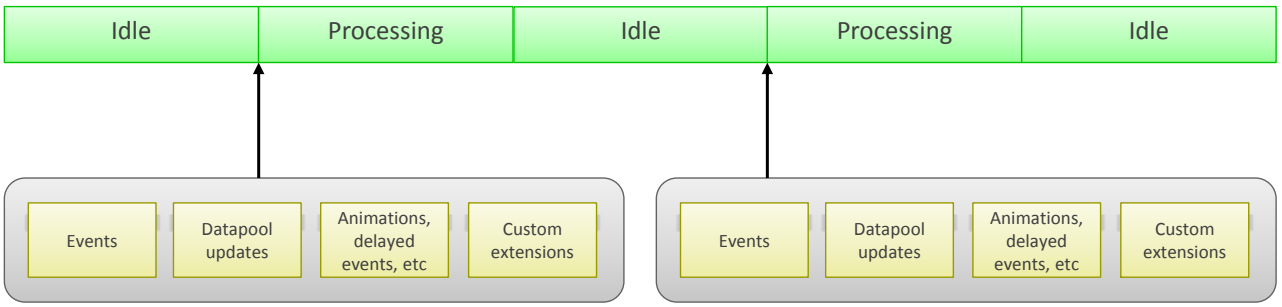


Figure 8.15. Processing in EB GUIDE GTF

It is a general concept of EB GUIDE GTF to process all these external and internal occurrences in an asynchronous manner. This is done by performing update processing cycles.

There are two kinds of update processing cycles. A major cycle takes care of the external occurrences and includes a limited number of minor processing cycles. These minor cycles are responsible for processing of the model internal occurrences. EB GUIDE GTF tries to reach stable states for its internal software units during this phase, for example state machines, layout, or property values.

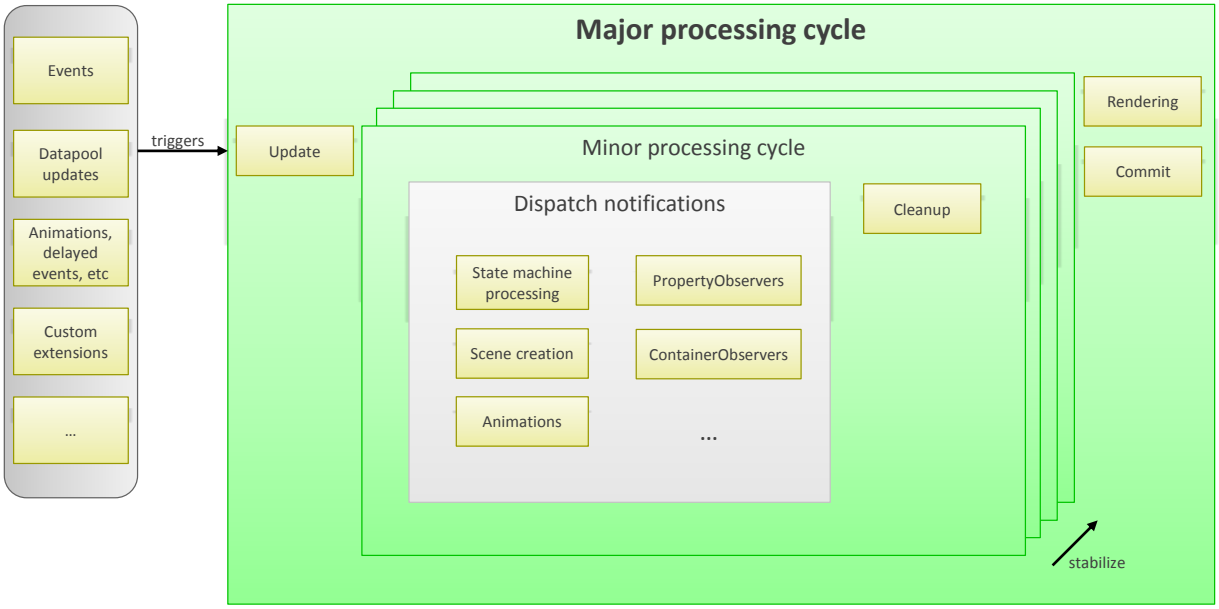


Figure 8.16. Update processing performed by EB GUIDE GTF

When a stable state is reached or the maximum number of minor processing cycles is exceeded, the stabilization phase is aborted. EB GUIDE GTF renders the scenes and provides the processing results, for example commits datapool changes and sends events in the global event system.

The above mentioned update processing happens separately in each `gtf::coreruntime::CoreRuntime` instance. One `gtf::coreruntime::CoreRuntime` instance is responsible for all parts of one EB GUIDE model that run in one and the same communication context.

To achieve this functionality independent from other `gtf::coreruntime::CoreRuntime` instances running in parallel, each `gtf::coreruntime::CoreRuntime` has own dedicated instances:

- `gtf::dependencyresolver::DependencyContainer`
- `gtf::datapool::Datapool`
- `gtf::eventsystem::local::EventSystem`
- `gtf::eventsystem::local::EventQueue`

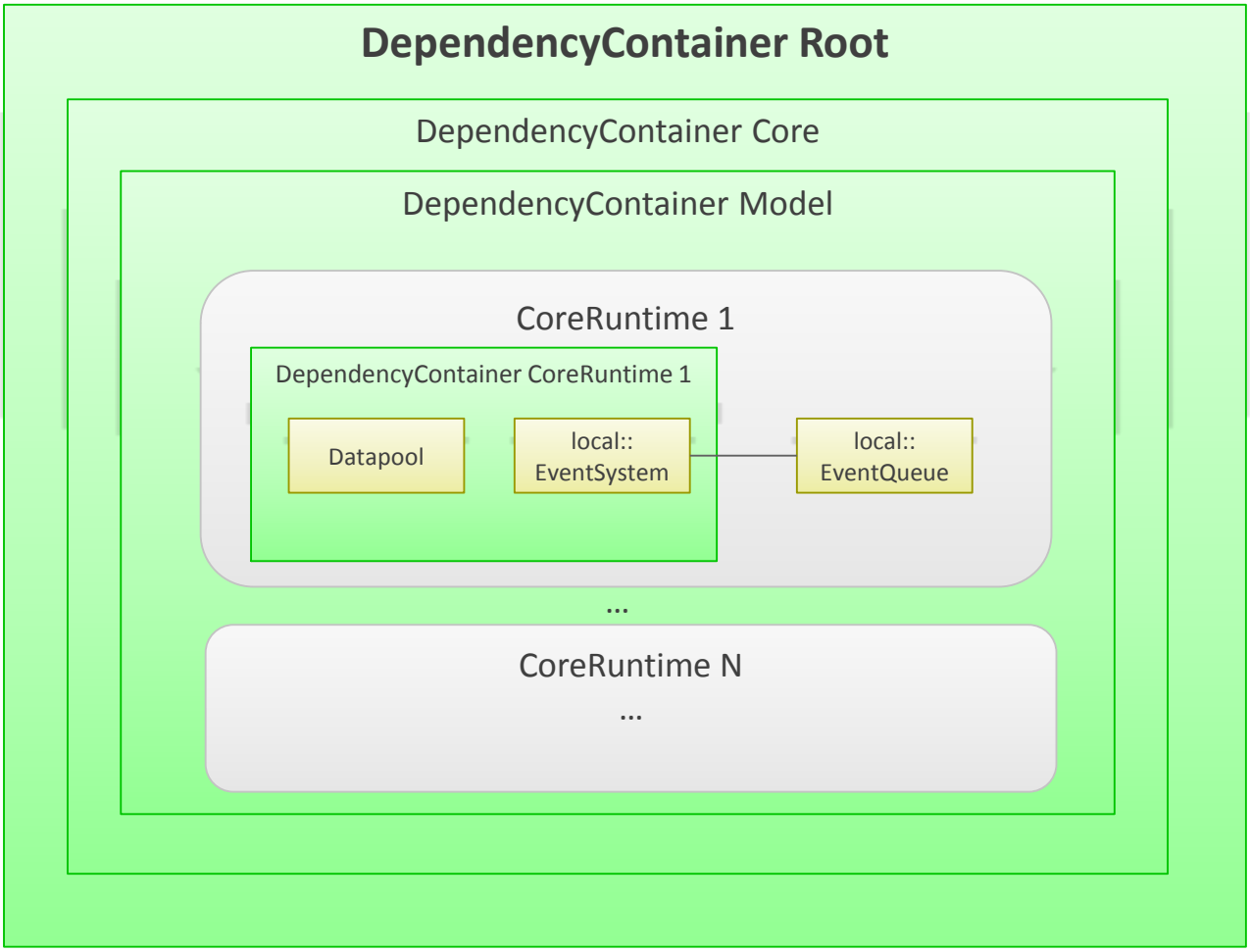


Figure 8.17. Integration of CoreRuntime

All these facilities are available for EB GUIDE GTF extension developers by using the `gtf::coreruntime::CoreRuntime` EB GUIDE SDK interface. In addition `gtf::datapool::Datapool` and `gtf::eventsystem::local::EventSystem` are also registered at the respective `gtf::dependencyresolver::DependencyContainer` instance.

8.7. Event system

An event system provides an asynchronous communication mechanism based on events. Events are delivered in exactly the same order they were sent. An event is added to all event queues subscribed to this event. Events can transport values, for example integers, strings or resources.

An event queue is a facility used by a receiver to control, when to process the arrived events. An event queue may limit the number of unprocessed events. New events are ignored by this queue until the number of unprocessed events does not exceed the limit anymore.

EB GUIDE GTF uses multiple event systems for different purposes. The most important one for application developers is the global event system. It provides communication between applications and EB GUIDE models. Another event system is used for local communication within each `gtf::coreruntime::CoreRuntime` of an EB GUIDE model.

NOTE

Limited event transition between global and local event systems

Only events of user-specific event groups are transferred between global event system and local event systems and vice versa.

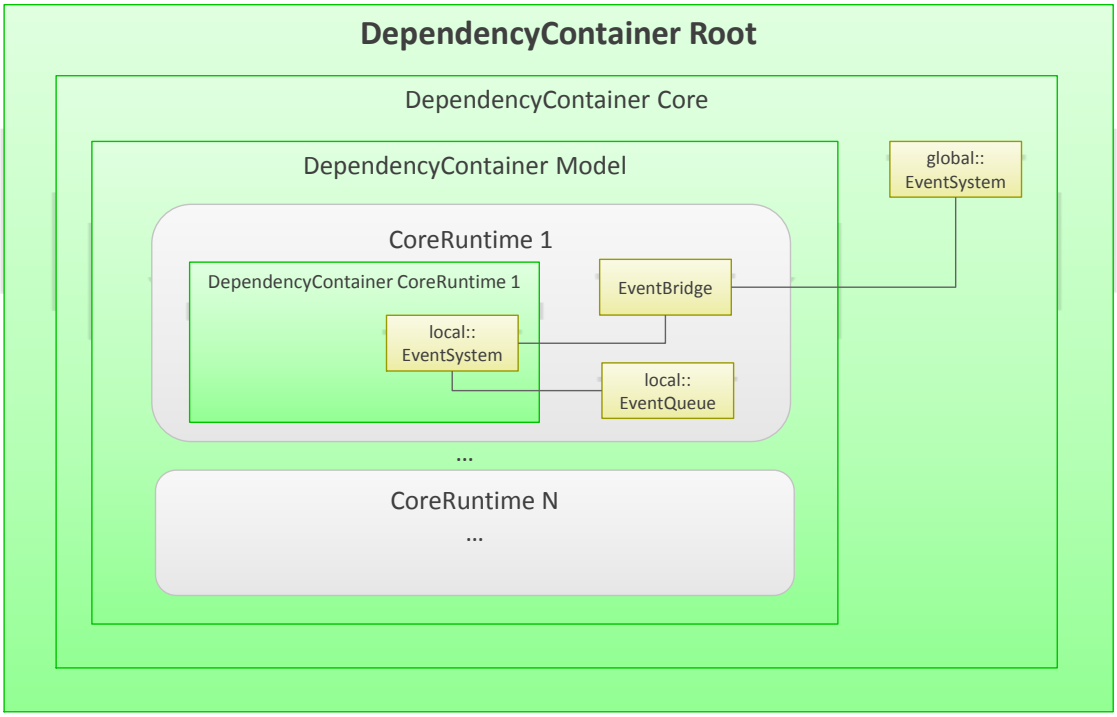


Figure 8.18. Integration of the event system in EB GUIDE GTF

In the global event system, string-based identifiers are used. They are created from the EB GUIDE model during export and are unique for all models.

In the local event system, numeric identifiers are used. They are also created from the EB GUIDE model during export, but are unique in one model only.

Both types of identifiers are exported to C++ header files and can be used by application developers to interact with an EB GUIDE model.

8.7.1. Event publication

The following steps show the general procedure of event publication:

1. Create a new event at the event system or at an event queue and provide the identifiers for event and group.
2. Optionally add parameter values.
3. Send the event to the event system or to the event queue.

   To identify the sender, you can provide an optional send parameter. This may be useful if you want to identify your own events.

After you send an event, the event is read-only. You cannot change parameter values anymore.

NOTE **The event API is not thread-safe**

Do not access one event from concurrent threads before it was sent, because the API is not thread-safe. After you send an event, the event is read-only and can be used by different threads.

8.7.2. Event receipt

The following steps show the general procedure of event receipt:

1. Create an event queue.
2. Register an invoking callback method at the event queue.
3. To subscribe to required events, provide a callback method for specific events, groups of events or all events.
4. After a new event arrives, the invoking callback method is executed. This method should invoke the worker task. To dispatch and process events which are currently in the event queue, use method `dispatch()` within the task. After dispatching the event queue is empty.

NOTE **Ensure asynchronous event processing**

It is not allowed to dispatch and process events directly in the registered callback method. Event dispatching and processing has to be done asynchronously, even if the whole system runs in one and the same thread. The callback method must only invoke an asynchronous worker task.

NOTE **The event queue API is not thread-safe**

Do not access one event queue instance from concurrent threads, because the API is not thread-safe.

Only synchronization between event system and event queue provides thread-safety. Different threads must use different event queue instances.

8.8. External input events

External input events can be sent to a `CoreRuntime` of an EB GUIDE model. This can be done using the `ExternalInput` interface, which is a helper interface based on the functionality of a local event system. Each `CoreRuntime` uses a separate local event system. The following are the types of external input events that can be created and sent to the local event system:

- Key
- Touch
- Rotary

For more information about events and `EventSystem`, see section 8.7, “Event system”.

The events are then processed by the input management during the processing round. For more information on processing, see section 8.6, “Update processing in EB GUIDE GTF”.

During the input management it is decided whether a touch event belongs to a sequence resulting in, for example, gestures. Besides that the input management decides to which of the focused widgets a key or a rotary input event will be delivered. For more information on touch input, see EB GUIDE Studio user guide. Events are delivered in exactly the same order they were sent.

In case of a touch input event the interaction is done with a widget at a certain location on the display so the input is clear and the system knows to which widget the input event should be delivered. In case of a key or rotary input event, it will be delivered to the currently active display widget. This is typically highlighted so the user knows with which widget he will interact next. This highlighting feature can be modeled in EB GUIDE Studio using the **Focused** widget feature so that the widget that is currently focused is marked with a colored border.

NOTE

Process hierarchy

If a widget cannot handle a key, rotary, or touch input, the input is processed along the modeled widget tree, towards the root widget. If one of the parents of the widget handles the input, it is considered as processed.

For instructions on how to create external input events, see chapter 17, “Using external input events“.

For examples of external input events, see the EB GUIDE SDK examples `ExternalKeyInputExample` and `ExternalTouchInputExample`. 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.

8.9. Datapool

The datapool provides an asynchronous communication mechanism based on datapool items. Datapool items can be of scalar, list, or project specific resource types, for example string lists or image lists.

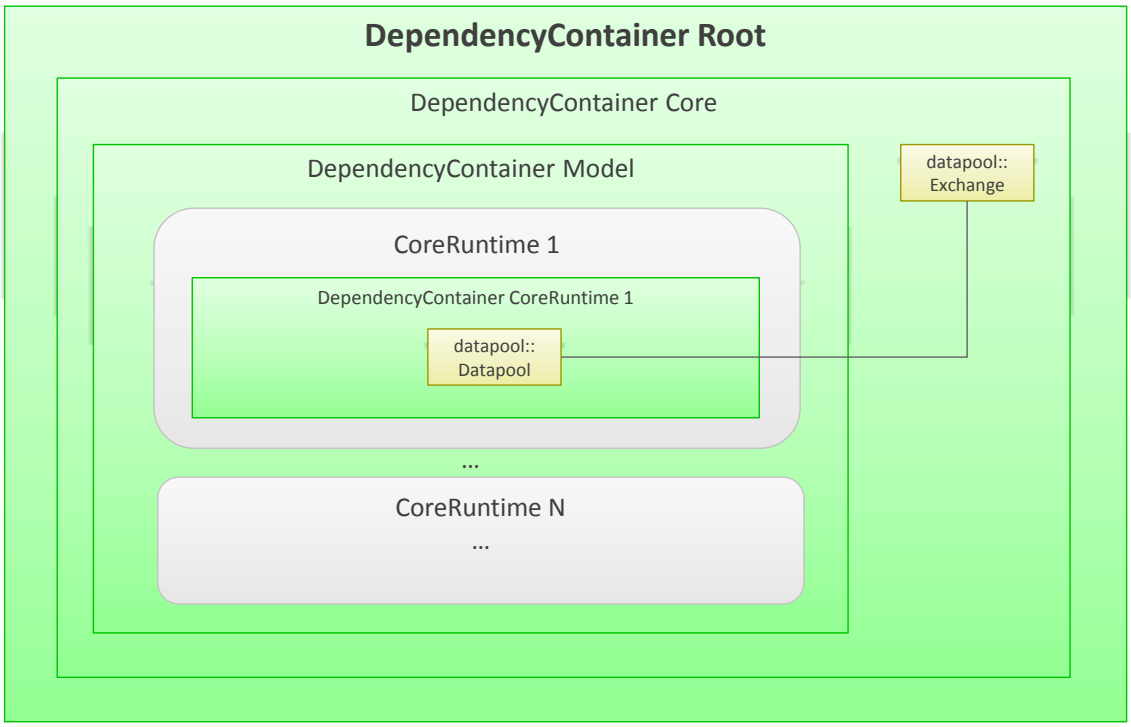


Figure 8.19. Integration of the datapool

At EB GUIDE GTF run-time each application and each `gtf::runtime::CoreRuntime` of an EB GUIDE model is working on its own datapool instance. A datapool instance is empty by default. The owner of a datapool can load items and their initial values from a description file. This datapool description file is part of the exported EB GUIDE model.

EB GUIDE GTF synchronizes the values of all datapool items that are shared by different datapool instances. Explicit calls to the API methods of the datapool control the synchronization.

NOTE **The datapool API is not thread-safe**

Do not access one datapool instance from concurrent threads, because the API is not thread-safe. The synchronization between datapool instances provides thread safety. Different datapool instances can be used in different threads.

The common use case is that one datapool user modifies a datapool item and one or more users read the changed values.

NOTE **Possible race conditions on competing modifications in different datapool instances**

If one datapool item is modified in several datapool instances, race conditions may occur.

Datapool items can change as follows:

- Datapool items use the API of this datapool instance.
- The datapool items change during the synchronization of the changes done in other datapool instances.

NOTE **No order of change notifications**

The change notifications do not depend on the order of modifications.

8.9.1. Identifiers of datapool items

The datapool API supports two types of datapool item identifiers: a string-based identifier, and a numeric one. During the export of the EB GUIDE model, datapool item identifiers are created and written to a C++ header file. The string-based identifier is unique for all EB GUIDE models, the numeric identifier is unique in one EB GUIDE model only.

8.9.2. Synchronization of datapool items

The following steps show the general procedure for synchronization of datapool items:

- Register an invoking callback method.
- The API method `commit()` provides all changed values of one datapool instance to all other datapool instances.
- All datapool instances sharing a sub-set of the changed datapool items are invoked by calling the registered invoking callback method.
- The API method `update()` applies the latest available values to the specific datapool instance only.

`commit()` and `update()` affect all changed datapool items at once.

NOTE **Possible loss of intermediate values**

The datapool provides access to the latest available value only.

NOTE **Ensure asynchronous processing of changed datapool items**

It is not allowed to process updates or notifications directly in the registered invoking callback method. Updates and notifications must be processed asynchronously, even if the whole system runs in the same thread.

The callback method must only invoke an asynchronous worker task.

8.9.3. Windowed lists

Windowed list is a data type that is available only in the datapool.

Windowed lists support two operating modes. The default operating mode is based on one data element per list element. The second operating mode provides the concept of windowed list. Windowed lists are useful to save system resources in case of very big list with thousand or millions of elements, or in cases where access to the data source is very slow.

List access is possible only if list elements are covered by at least one window. If the window position or window size is changed, the newly covered list elements are uninitialized until the application writes the list element value for the first time. Read access fails for all uninitialized list elements. Some operations are not supported for windowed lists and will fail, for example insert, append or remove.

An application can switch between both operation modes.

In a typical use case the EB GUIDE model tells the application about the needed parts of a list by using events or datapool items. The application creates windows that cover these parts and provides the requested data for these parts only. To avoid missing data in the EB GUIDE model, define a pre-loading strategy.

8.10. Aspect

You can define run-time related aspects, languages or skins, in an EB GUIDE model. Aspects modify the appearance of the HMI.

The EB GUIDE SDK provides interfaces to provide, request and change aspects. Applications and EB GUIDE GTF extensions can register as observers to get notified about new aspects or aspect changes.

NOTE

Ensure asynchronous processing of changed aspects

It is not allowed to process changed aspects directly in the registered invoking callback method. Updates must be processed asynchronously, even if the whole system runs in the same thread.

The callback method must only invoke an asynchronous worker task.

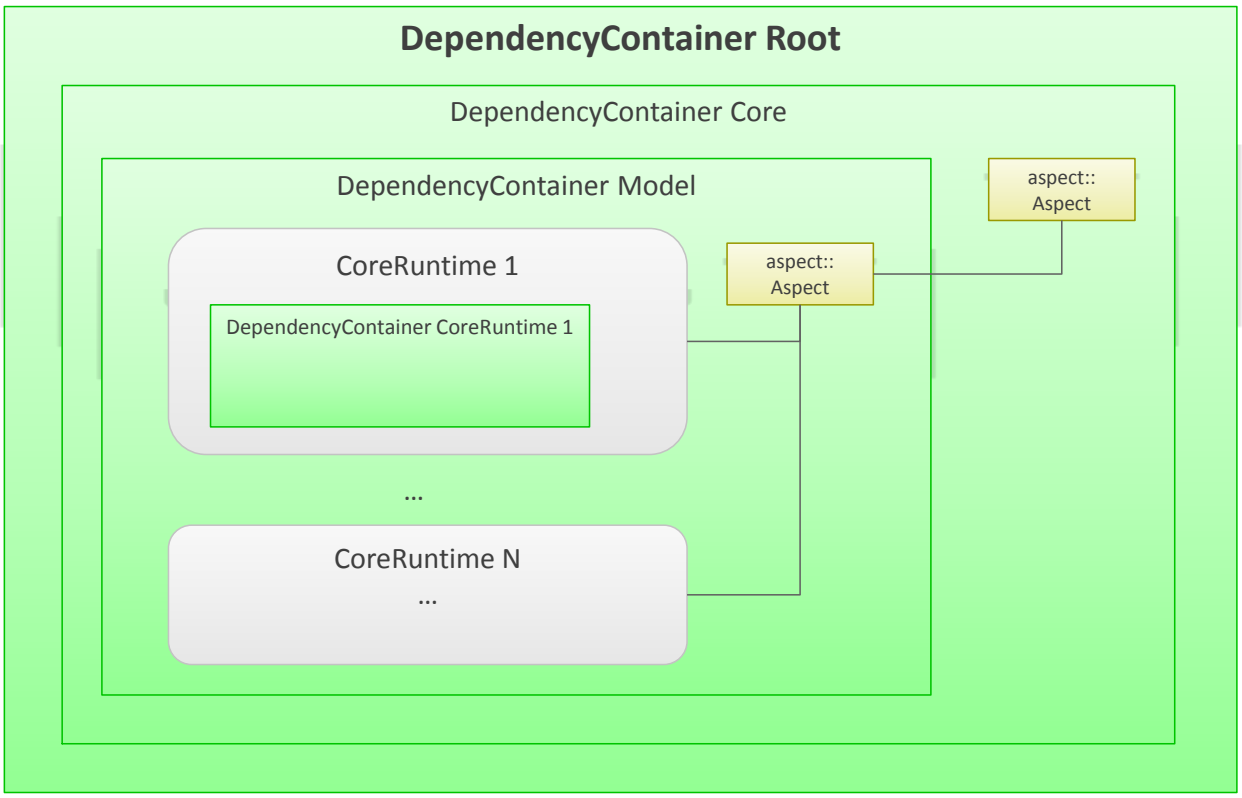


Figure 8.20. Integration of the aspect

Aspects on the model level try to get synchronized with the aspects on the core level and vice versa.

8.11. Resource management

Resources are content that is not created within EB GUIDE but can be used in your EB GUIDE projects. Examples for resources are fonts, images, meshes for 3D graphics.

For more information about resource handling in EB GUIDE, see the EB GUIDE Studio documentation, chapter "Resource management".

Each resource has a certain format, for example `.jpeg` or `.png` for images.

After the export of an EB GUIDE project, the following two items serve as input to configure and run the EB GUIDE GTF resource management system:

- `resources.gtf`

  The `resources.gtf` binary file is used to configure the resource management system.

- `resources` folder

  This `resources` folder contains all resource files available in the EB GUIDE model, for example images and fonts.

For more information about the exported EB GUIDE model, see section 8.1, “Configure EB GUIDE GTF with an exported EB GUIDE model”.

The EB GUIDE GTF resource management consists of the following three modules:

- `ResourceLoader` module

  Responsible for loading the resources within one EB GUIDE model and setting the initial format data

- `Decoder` module

  Responsible for converting one format to another and also provides the possibility of adding own decoders

- `ResourceSystem` module

  Responsible for managing the decoders and triggering them to retrieve the required format and the attached format data

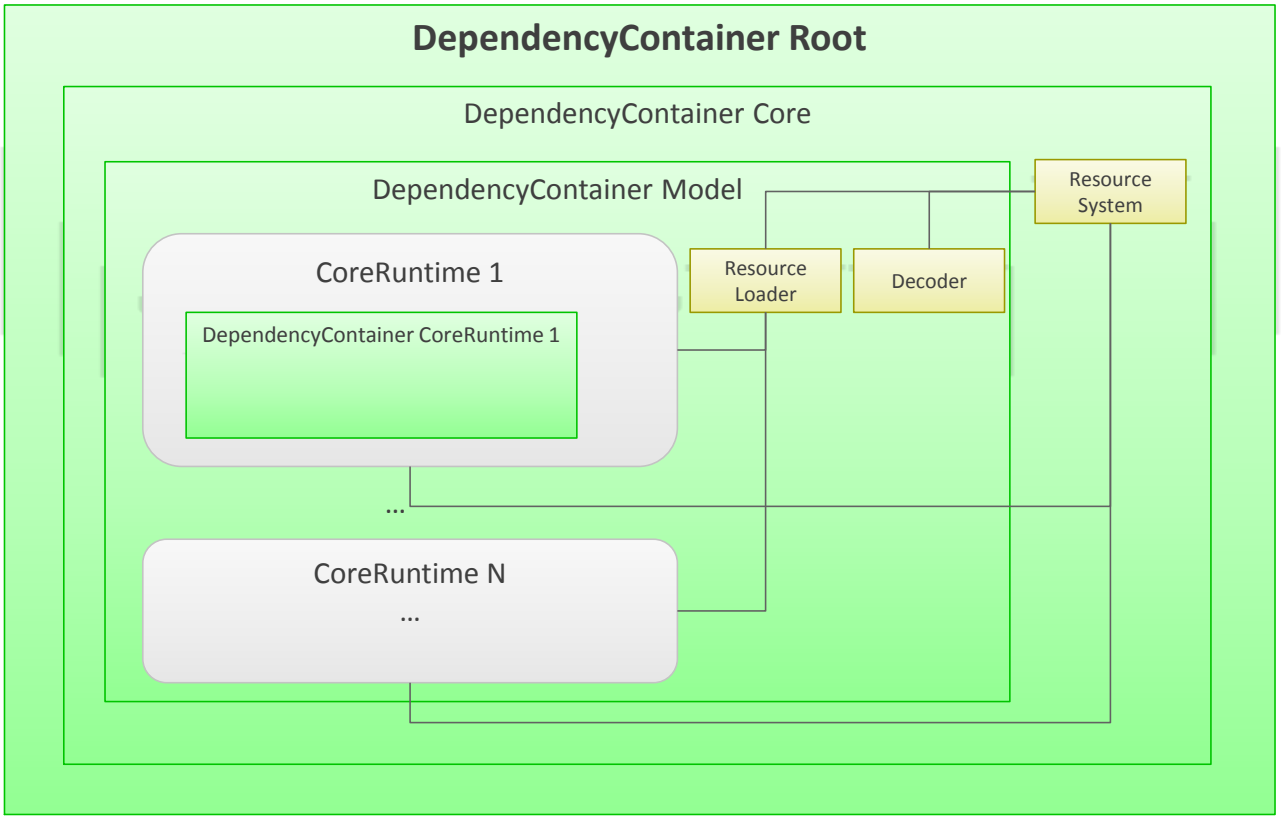


Figure 8.21. Integration of the resource management system

Table 8.1. Default resource formats of EB GUIDE GTF

Name	Accessor in EB GUIDE SDK	Description
Font formats		
`gtf::decoder::FontDescriptor`	no	EB GUIDE GTF internal intermediate format that is used to read information contained in `resources.bin`
`gtf::decoder::FontData`	yes	Set of font information, i.e. pairs of font file path and size For more details, ee `gtf/decoder/FontData.h`.
IBL formats		
`gtf::decoder::IBLDescriptor`	no	EB GUIDE GTF internal intermediate format that is used to decode information contained in `resources.bin`
`gtf::decoder::IBLPath`	yes	Path of an IBL file

Name	Accessor in EB GUIDE SDK	Description
		For more details, see `gtf/decoder/IBLPath.h`.
`gtf::decoder::EncodedIBLData`	yes	Path of the original file and a buffer providing the raw IBL data in memory For more detials, see `gtf/decoder/EncodedIBLData.h`. Note: Decoders of this format may require to set suitable file path with file ending to decide if they support the format or not. The path is also used to print error messages.
Image formats		
`gtf::decoder::ImageDescriptor`	no	EB GUIDE GTF internal intermediate format that is used to read information contained in `resources.bin`
`gtf::decoder::ImagePath`	yes	Path of an image file For more details, see `gtf/decoder/ImagePath.h`.
`gtf::decoder::EncodedMemImage`	yes	Path of the original file and a buffer providing the raw image data in memory For more details, see `gtf/decoder/EncodedMemImage.h`. Note: Decoders of this format may require to set suitable file path with file ending to decide if they support the format or not. The path is also used to print error messages.
`gtf::decoder::MemImage`	yes	Path of the original file and the set of plain image information For more detials, see `gtf/decoder/MemImage.h`. Note: Decoders of this format may require to set suitable file path with file ending to

Name	Accessor in EB GUIDE SDK	Description
		decide if they support the format or not. The path is also used to print error messages.
`gtf::scdr::decoder::Texture`	yes	Renderer texture data For more details, see `gtf/scdr/resources/TextureResource.h`.
Mesh formats		
`gtf::decoder::MeshDescriptor`	no	EB GUIDE GTF internal intermediate format that is used to read information contained in `resources.bin`
`gtf::decoder::MeshPath`	yes	Path of a mesh file For more details, see `gtf/decoder/MeshPath.h`.
`gtf::decoder::EncodedMeshData`	yes	Path of the original file and a buffer providing the raw mesh data in memory For more details, see `gtf/decoder/EncodedMeshData.h`. Note: Decoders of this format may require to set suitable file path with file ending to decide if they support the format or not. The path is also used to print error messages.

Table 8.2. Default decoder chains of EB GUIDE GTF

EB GUIDE Studio type	Default decoder chain in EB GUIDE GTF
Font	`gtf::decoder::FontDescriptor` → `gtf::decoder::FontData`
IBL	`gtf::decoder::IBLDescriptor` → `gtf::decoder::IBLPath` → `gtf::decoder::EncodedIBLData` → EB GUIDE GTF renderer internal formats
Image	`gtf::decoder::ImageDescriptor` → `gtf::decoder::ImagePath` → `gtf::decoder::EncodedMemImage` → `gtf::decoder::MemImage` → EB

EB GUIDE Studio type	Default decoder chain in EB GUIDE GTF
	GUIDE GTF renderer internal formats → `gtf::scdr::decoder::Texture`
Mesh	`gtf::decoder::MeshDescriptor` → `gtf::decoder::MeshPath` → `gtf::decoder::EncodedMeshData` → EB GUIDE GTF renderer internal formats

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

8.12. Inter-process communication

The `GtfIpc` plugin extends the EB GUIDE GTF with the inter-process communication (IPC) mechanism that connects multiple EB GUIDE GTF instances and exchanges information, for example datapool items and events.

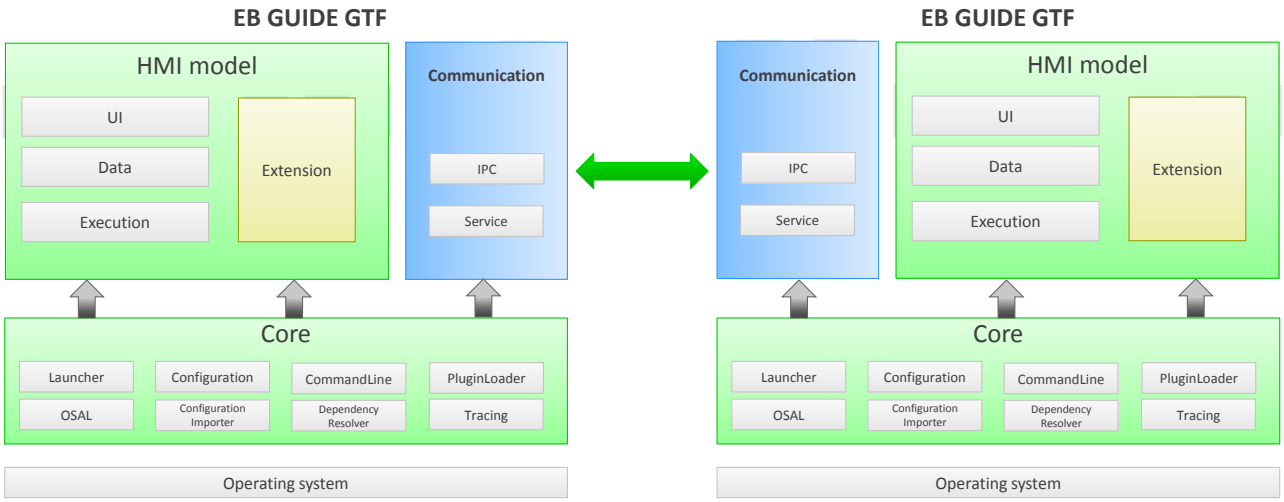


Figure 8.22. Connecting EB GUIDE GTF instances with IPC

The IPC has a server-client architecture. The implementation supports several clients. Clients and server can be restarted. If a server is restarted, it will be initialized with the default datapool item values and update the clients with the same values.

The server instance is responsible for the following:

- Connecting clients
- Mastering datapool data

Once the IPC network is established, the server forwards the changes in the exchange information to the other clients.

Example 8.1.
IPC

An IPC network is created by connecting a server and several clients. All these connected nodes contain the current time information in a datapool item. If a client updates the datapool item and therefore the time, all the other nodes present in the IPC network will get updated with the new value.

By default, all datapool items and events contained in a nodes' model are shared. You can configure the items which are shared as follows:

- Set the `"gtf.datapool.descriptionfile"` configuration item.

  If the configuration item is set, the IPC initializes its datapool with the file set in this configuration item.
- Set the start-up parameter that refers to the `model.json` path.

Data exchange within the IPC has the default datapool and event system behavior. For more information, see section 8.9, “Datapool” and section 8.7, “Event system”.

In case of datapool, if one node changes a datapool item e.g. five times, other nodes will only get the latest updated value of the datapool item.

In case of events, if one node fires an event e.g. five times, other nodes will have the event triggered also five times.

8.12.1. Connection modes

EB GUIDE GTF supports three types of connection modes:

- Broadcast
- Multicast
- Direct

NOTE **Structure of configuration items**

Basic structure of an item is as follows: `<configuration_item> : <value>`

String values must be enclosed with `" "`.

For more information on `platform.json`, see section 19.8, “`platform.json` configuration file”.

8.12.1.1. Broadcast

The following tables show the examples for the configuration of the broadcast connection mode.

Values in the tables below are just example values. Adapt the values according to your project's needs.

Table 8.3. Server configuration

Configuration item and value	Description
`"gtf.ipc.role": "server"`	The role in the IPC network
`"gtf.ipc.discovery.mode": "broadcast"`	The connection mode
`"gtf.ipc.discovery.network": "255.255.-255.255"`	The broadcast network address
`"gtf.ipc.discovery.port": 4712`	The broadcast port
`"gtf.ipc.datapool.config": "ipc_dat-apool.gtf"`	The configuration file containing the datapool items that should be part of the IPC communication

Table 8.4. Client configuration

Configuration item and value	Description
`"gtf.ipc.role": "client"`	The role in the IPC network
`"gtf.ipc.discovery.mode": "broadcast"`	The connection mode
`"gtf.ipc.discovery.network": "255.255.-255.255"`	The broadcast network address
`"gtf.ipc.discovery.port": 4712`	The broadcast port
`"gtf.ipc.datapool.config": "ipc_dat-apool.gtf"`	The configuration file containing the datapool items that should be part of the IPC communication
`"gtf.ipc.client.timeout": 5000`	The time interval in milliseconds between the connection retry attempts of the client

8.12.1.2. Multicast

The following tables show the examples for the configuration of the multicast connection mode.

Values in the tables below are just example values. Adapt the values according to your project's needs.

Table 8.5. Server configuration

Configuration item and value	Description
`"gtf.ipc.role": "server"`	The role in the IPC network
`"gtf.ipc.discovery.mode": "multicast"`	The connection mode
`"gtf.ipc.discovery.network": "230.0.0.-1"`	The multicast network address
`"gtf.ipc.discovery.port": 4712`	The multicast port

Configuration item and value	Description
`"gtf.ipc.datapool.config": "ipc_datapool.gtf"`	The configuration file containing the datapool items that should be part of the IPC communication

Table 8.6. Client configuration

Configuration item and value	Description
`"gtf.ipc.role": "client"`	The role in the IPC network
`"gtf.ipc.discovery.mode": "multicast"`	The connection mode
`"gtf.ipc.discovery.network": "230.0.0.-1"`	The multicast network address
`"gtf.ipc.discovery.port": 4712`	The multicast port
`"gtf.ipc.datapool.config": "ipc_datapool.gtf"`	The configuration file containing the datapool items that should be part of the IPC communication
`"gtf.ipc.client.timeout": 5000`	The time interval in milliseconds between the connection retry attempts of the client

8.12.1.3. Direct

The following tables show the examples for the configuration of the direct connection mode.

Values in the tables below are just example values. Adapt the values according to your project's needs.

Table 8.7. Server configuration

Configuration item and value	Description
`"gtf.ipc.role": "server"`	The role in the IPC network
`"gtf.ipc.discovery.mode": "direct"`	The connection mode
`"gtf.ipc.discovery.network": "127.0.0.-1"`	Localhost
`"gtf.ipc.datapool.config": "ipc_datapool.gtf"`	The configuration file containing the datapool items that should be part of the IPC communication
`"gtf.servicemapper.port": 60001`	If the configuration is generated, the port might already be set. You need to set the value only if it does not yet exist.

Table 8.8. Client configuration

Configuration item and value	Description
`"gtf.ipc.role": "client"`	The role in the IPC network

Configuration item and value	Description
`"gtf.ipc.discovery.mode": "direct"`	The connection mode
`"gtf.ipc.discovery.network": "x.x.x.x"`	The network address of the server
`"gtf.ipc.discovery.port": 60001`	The port should match the `"gtf.servicemapper.port"` value of the server
`"gtf.ipc.datapool.config": "ipc_datapool.gtf"`	The configuration file containing the datapool items that should be part of the IPC communication
`"gtf.ipc.client.timeout": 5000`	The time interval in milliseconds between the connection retry attempts of the client

9. Running EB GUIDE GTF

9.1. Configuring an EB GUIDE model for running on a target platform

EB GUIDE Studio offers the possibility to create different profiles for an EB GUIDE model. In profiles you set up a configuration for a specific target platform, for example Windows, Linux or QNX. During export the two EB GUIDE GTF start-up configuration files `platform.json` and `model.json` are generated from profiles.

Use profiles to do the following:

- Add custom configuration items
- Configure internal and user-defined EB GUIDE GTF plugins to load
- Configure a scene
- Configure a renderer

There are two default profiles: **Edit** and **Simulation**.

For more information, see section 7.7, “Run modes of EB GUIDE GTF”.

For details on how to configure profiles, see *Configuring profiles* in the EB GUIDE Studio user guide.

9.2. Exporting an EB GUIDE model

For details on how to export an EB GUIDE model, see *Exporting an EB GUIDE model* in the EB GUIDE Studio user guide.

9.3. Configuring and starting EB GUIDE GTF

Configuring the system start

Prerequisite:

- Profiles are configured according to the project requirements.

- An EB GUIDE model is exported.

Step 1
Copy the platform specific binaries and all project-specific EB GUIDE GTF plugins that are required by your `model.json` and `platform.json` configuration to your target platform.

Step 2
Copy the exported EB GUIDE model. Make sure the relative paths in `model.json` refer to the exported EB GUIDE model path.

Step 3
Start EB GUIDE GTF on the target device.

For details on how to start EB GUIDE GTF from the command line, see section 7.7.1.1, “Command line options”.

You have configured and started EB GUIDE GTF for your target platform.

9.4. Running EB GUIDE GTF with a read-only memory file system (RomFS) container

Running EB GUIDE GTF with a read-only memory file system (RomFS) container

The directory you create serves as root directory in the RomFS. It is referred to as `"/"` on the POSIX platforms and as `"C:/"` on the Microsoft Windows platforms.

Prerequisite:

- An EB GUIDE model is created and exported.

Step 1
Navigate to `$GUIDE_INSTALL_PATH/tools/RomFsTools` and open a command line tool.

Step 2
To create the RomFS container for the exported model, in the command line tool enter:

```
RomFsCreate.exe EXPORTED_MODEL_FOLDER ROMFS_FILE
```

Step 3
Navigate to `$GUIDE_INSTALL_PATH/platform/win64/bin` and open the command line tool.

Step 4
To start EB GUIDE GTF using the created RomFS container, in the command line tool enter:

```
GtfStartup.exe --romfs ROMFS_FILE_PATH ROMFS_ROOT_FOLDER
```

10. Using the EB GUIDE GTF plugin mechanism

10.1. Creating an EB GUIDE GTF plugin

Two types of EB GUIDE GTF plugins exist: extensions and applications. The following instructions are valid for both types.

Creating an EB GUIDE GTF plugin

The following section explains the general workflow for integrating EB GUIDE GTF into your build system on Windows platforms. Find the instructions for each step in the sections below.

Step 1
Write an EB GUIDE GTF plugin. For details, see section 10.2, “Writing an EB GUIDE GTF plugin”.

Step 2
Copy the resulting `.dll` file. For details, see section 10.3, “Copying the resulting `.dll` file”.

Step 3
Adjust the `model.json` to load the EB GUIDE GTF plugin. For details, see section 10.4, “Adding an EB GUIDE GTF plugin”.

Step 4
Start the simulation with `GtfStartup.exe`. For details, see section 10.5, “Starting the simulation with `GtfStartup.exe`”.

10.2. Writing an EB GUIDE GTF plugin

Writing an EB GUIDE GTF plugin

In order to access an example, visit our website https://www.elektrobit.com/ebguide/learn/resources/. Download the `EB_GUIDE_Examples` file and open the `BasicExample` example.

Prerequisite:

- An EB GUIDE model is exported.

- The `model.json` file is adapted.
- A new directory is created, for example `C:/plugin/myplugin`.

Step 1
Navigate to the directory you prepared for the plugin, for example `C:/plugin/myplugin`.

Step 2
Create a file named `myplugin.cpp`.

Step 3
Open the `myplugin.cpp` file and write an EB GUIDE GTF plugin.

Find a description of all relevant classes and methods in the EB GUIDE GTF API.

Step 4
Compile `myplugin.cpp`.

The result is a `.dll` file `myplugin.dll`.

10.3. Copying the resulting `.dll` file

Copying the resulting `.dll` file

Prerequisite:

- An EB GUIDE model is exported.
- The `model.json` file is adapted.
- A new directory is created, for example `C:/plugin/myplugin`.
- A compiled EB GUIDE GTF plugin is created.

Step 1
Navigate to the directory where you saved the `myplugin.dll` file, for example `C:/plugin/myplugin`.

Step 2
Copy `C:/plugin/myplugin` to the directory where you exported the EB GUIDE model, for example `C:/projects/example_project`.

10.4. Adding an EB GUIDE GTF plugin

To add an EB GUIDE GTF plugin, you can do either of the following:

- Add the EB GUIDE GTF plugin to an EB GUIDE model in EB GUIDE Studio.

- Modify the `model.json` file of an exported EB GUIDE model.

Adding an EB GUIDE GTF plugin to an EB GUIDE model

In order to use a newly written EB GUIDE GTF plugin such as a widget or a widget feature in an EB GUIDE model, it is necessary to add the EB GUIDE GTF plugin to the EB GUIDE model.

Prerequisite:

- An EB GUIDE GTF plugin exists.

Step 1
Compile the EB GUIDE GTF plugin.

Step 2
Copy the resulting file to the `$GUIDE_PROJECT_PATH/resources` directory.

Step 3
Start EB GUIDE Studio and open an EB GUIDE Studio project.

Step 4
Go to the project center and click **Configure** > **Profiles**.

Step 5
Select the required profile.

Step 6
Open the **Model** tab.

Step 7
To load your EB GUIDE GTF plugin, integrate the following code into the `model.json`:

```
{
 "gtf": {
     "model": {
      "pluginstoload": [
             "resources/myplugin"
         ]
     }
 }
}
```

`myplugin` is the name of the example EB GUIDE GTF plugin.

Step 8
Save the project and close EB GUIDE Studio.

Step 9
Open the project again.

You have added an EB GUIDE GTF plugin to an EB GUIDE model.

Step 10
Export the EB GUIDE model.

Adding an EB GUIDE GTF plugin without EB GUIDE Studio

The following instruction shows you how to modify the `model.json` file directly so that it loads an EB GUIDE GTF plugin.

Prerequisite:

- An EB GUIDE model is exported.

Step 1
Navigate to the exported EB GUIDE model.

Step 2
Open the `model.json` file with a text editor.

Step 3
To load your EB GUIDE GTF plugin, include the following code:

```
{
 "gtf": {
     "model": {
      "pluginstoload": [
             "myplugin"
         ]
     }
 }
}
```

`myplugin` is the name of the example EB GUIDE GTF plugin.

Step 4
Save the `model.json` file.

10.5. Starting the simulation with `GtfStartup.exe`

Starting the simulation with `GtfStartup.exe`

On Windows platforms you can start the simulation directly using `GtfStartup.exe`.

Prerequisite:

- An EB GUIDE model is exported.
- The `model.json` file is adapted.
- A new directory is created, for example `C:/plugin/myplugin`.
- A compiled EB GUIDE GTF plugin is created.
- The resulting `.dll` file is available in the directory of the exported EB GUIDE model.

Step 1
Navigate to `$GUIDE_INSTALL_PATH/platform/win64/bin`.

Step 2
Execute `GtfStartup.exe` with the complete path to `model.json` as the first argument. Enter the following command line:

```
GtfStartup.exe C:/projects/example_project
```

The framework opens a window which displays the start view.

11. Resolving interface dependencies

When you use the dependency resolving mechanism, note the following:

- All interfaces that have to be registered in the container must inherit from the public interface `dependencyresolver::Interface`.
- The macro `GTF_DEFINE_INTERFACE_NAME(InterfaceName)` must be added in the class header file.

  For example:

```
class UserInterface : public dependencyresolver::Interface
public:
 virtual void fooBar() = 0;
};
typedef gtf::smartptr::RefCountedPtr<UserInterface> UserInterfaceHandle;
GTF_DEFINE_INTERFACE_NAME(UserInterface);
```

For more information, see API documentation module DependencyResolver.

For background information, see section 7.4, “Interface dependencies”.

For an example on interface dependencies, see the EB GUIDE SDK example `DependencyResolverExample`. 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.

11.1. Retrieving an item from `DependencyContainer`

Retrieving an item from `DependecyContainer`

Prerequisite:

- A handle to an item is registered to the container.

Step 1
Declare a handle variable.

Example: `UserInterfaceHandle value;`

Step 2
The call to the `get` method returns a handle. Store this handle in the variable.

Example: `value = container->get<UserInterface>();`

If the item does not exist in the container, `get` returns `NULL`.

NOTE

Lookup scope

If the lookup scopes are not specified, EB GUIDE GTF is going to search for an instance of the requested interface in the hierarchy of the dependency containers.

If EB GUIDE GTF does not find an instance, EB GUIDE GTF is going to search for a `create` function in the hierarchy of the dependency containers. If the `create` function is available, EB GUIDE GTF if going to call this function to create a new instance.

The method's parameters control the lookup scope for instances and functions separately.

11.2. Retrieving all instances registered to an interface

Retrieving all instances registered to an interface

Step 1
Declare an array of handles.

Example: `gtf::container::Array<UserInterfaceHandle> instances;`

Step 2
To store the specified instances from the container, use the `getAll` method.

Example: `container->getAll(instances);`

Step 3
To navigate through the array, use an iterator .

Example:

```
for (gtf::container::Array<UserInterfacelHandle>::Iterator it = instances.begin();
    it != instances.end(); ++it)
{
//process data here...
}
```

If no items are found, `getAll` returns `FALSE` and the array is empty.

NOTE

Lookup scope

If the lookup scopes are not specified, EB GUIDE GTF is going to search for an instance of the requested interface in the hierarchy of the dependency containers.

If EB GUIDE GTF does not find an instance, EB GUIDE GTF is going to search for a `create` function in the hierarchy of the dependency containers. If the `create` function is available, EB GUIDE GTF is going to call this function to create a new instance.

The method's parameters control the lookup scope for instances and functions separately.

11.3. Registering an instance to the container

Registering an instance to the container

Prerequisite:

- A valid handle to an item exists.

Step 1
To register an instance of an interface, add the new instance to the container.

Example: `container->registerInstance<UserInterfaceHandle>(instance);`

Step 2
To register the same instance for multiple interfaces, add the same instance for each interface to the container.

Example:

```
class InterfaceA : public Interface
{
};
class InterfaceB : public InterfaceA
{
};
typedef gtf::smartptr::RefCountedPtr<InterfaceB> InterfaceBHandle;
InterfaceBHandle interfaceB= new InterfaceB();
container->registerInstance<InterfaceA>( interfaceB);
container->registerInstance<InterfaceB>( interfaceB);
```

If the item cannot be registered, `registerInstance` returns `FALSE`.

NOTE **Registring an instance**

It is not possible to register the same instance with the same interface twice.

Registering a `create` function

Step 1

Create a function that returns a handle to the instance that you want to register.

Example:

```
static dependencyresolver::UserInterfaceHandle createInstance
(const dependencyresolver::DependencyContainerHandle & container)
{
    if (!container.valid())
    {
        return dependencyresolver::InterfaceHandle();
    }
    UserInterfaceHandle instance = new UserInterfaceImplementation();
    return instance;
}
```

Step 2

Create a catalog instance.

Example: `gtf::dependecyresolver::DependencyCatalog catalog;`

Step 3

Extend the catalog with the previous function.

```
Dependencyresolver::extendCatalog<UserInterface>(catalog, createInstance);
```

NOTE

`DependencyObjects`

A catalog is an array of `DependencyObjects`.

For more information, see API documentation module DependencyResolver.

11.4. Unregistering an instance

Unregistering an instance

Prerequisite:

- A handle to an item was registered to the container.

Step 1
Unregister the instance from the container.

Example: `container->unregisterInstance<UserInterface>(instance);`

If the item cannot be unregistered, `unregisterInstance` returns `FALSE`.

11.5. Registering a catalog

The following instruction shows you how to add the content of a catalog to a container.

Registering a catalog

Step 1
Declare a catalog.

Example: `Example: gtf::dependecyresolver::DependencyCatalog catalog;.`

Step 2
Register the catalog in the container.

Example: `container->registerCatalog(catalog);`

If the call to `registerCatalog` fails, it returns `FALSE`.

11.6. Unregistering a catalog

Unregistering a catalog

Step 1
Unregister the catalog from the container.

Example: `container->unregisterCatalog(catalog);`

If the call to `unregisterCatalog` fails, it returns `FALSE`.

11.7. Creating a container

Creating a container

Prerequisite:

- A handle to an item was registered to the container.

Step 1
Create a container.

Example: `DependencyContainerHandle childContainer = container->createContainer();`

`get` and `getAll` use `retrieve` and `retrieveAll` internally.

12. Configuring EB GUIDE GTF

NOTE

Retrieving a handle

To retrieve a `Settings` instance from the `DependencyContainer` interface, use `get`.

Example: `gtf::configuration::SettingsHandle settings = container->get<gtf::configuration::Settings>();`

Make sure that the retrieved settings handle is valid. For more information about `DependencyContainer`, see section 7.4, “Interface dependencies”.

12.1. Adding a scalar item to `Configuration`

NOTE

Item types

The following instructions describe the steps for adding an item of type `StringScalar`. The same approach can be used for `BooleanScalar`, `IntegerScalar`, or `FloatScalar` types.

Adding a scalar item to `Configuration`

Step 1

To retrieve a scalar builder from the `Settings` interface, use `getBuilder`.

Example: `gtf::configuration::StringScalarBuilderHandle scalarBuilder = settings->getBuilder<gtf::configuration::StringScalarBuilder>();`

Step 2

To set the value of the scalar item that you want to add, use `scalarBuilder`.

Example: `scalarBuilder->set("some_string");`

Step 3

To create an item that you want to add, use `scalarBuilder`. This item has the value that was specified in Step 2.

Example: `gtf::configuration::StringScalarHandle item = scalarBuilder->build();`

Step 4

Create a key for your item.

Example: `const char* key = "some_key";`

Step 5

To add the scalar item into the configuration of EB GUIDE GTF, use `addItem`.

Example: `settings->addItem(item, key);`

If the key already exists or the handle (`item`) is null, `addItem` returns false.

12.2. Adding a list item to `Configuration`

NOTE **Item types**

The following instructions describe the steps for adding a list item of type `FloatList`. The same approach can be used for `BooleanList`, `IntegerList`, `StringList`, or `ObjectList` types.

Adding a list item to `Configuration`

Step 1
To retrieve a list builder from the `Settings` interface, use `getBuilder`.

Example: `gtf::configuration::FloatListBuilderHandle listBuilder = settings->getBuilder<gtf::configuration::FloatListBuilder>();`

Step 2
To add values to the list, use `add`.

Example: `listBuilder->add(1.0f); listBuilder->add(2.0f); listBuilder->add(3.0f);`

Step 3
To create a list item that you want to add, use the `listBuilder` . The elements of this list have the values that were added in Step 2.

Example: `gtf::configuration::FloatListHandle listItem = listBuilder->build();`

Step 4
Create a key for your list item.

Example: `const char* key = "some_key";`

Step 5
To add the list item into the configuration of the EB GUIDE GTF, use `addItem`.

Example: `settings->addItem(listItem, key);`

If the key already exists or the handle (`listItem`) is null, `addItem` returns false.

12.3. Adding an object item in `Configuration`

Adding an object item in `Configuration`

Step 1
To retrieve an `objectBuilder` from the `Settings` interface, use `getBuilder`.

Example: `gtf::configuration::ObjectBuilderHandle objectBuilder = settings->getBuilder<gtf::configuration::ObjectBuilder>();`

Step 2
Build the items that you want to add to the object item. For a list of all configuration item types, see section 7.5, “Configuration of EB GUIDE GTF”.

For example:

```
gtf::configuration::StringScalarBuilder stringScalarBuilder =
    settings->getBuilder<gtf::configuration::StringScalarBuilder>();
stringScalarBuilder->set("some_string_scalar");
gtf::configuration::StringScalarHandle stringScalar = stringScalarBuilder->build();
gtf::configuration::IntegerScalarBuilder integerScalarBuilder =
    settings->getBuilder<gtf::configuration::IntegerScalarBuilder>();
integerScalarBuilder->set(22);
gtf::configuration::IntegerScalarHandle integerScalar = integerScalarBuilder->build();
```

Step 3
To add the values, use `add`.

Example: `objectBuilder->add(stringScalar, "string.scalar.key"); objectBuilder->add(integerScalar, "integer.scalar.key");`

Step 4
To create an object item that you want to add, use `objectBuilder`.

Example: `gtf::configuration::ObjectValueHandle objectItem = objectBuilder->build();`

Step 5
To add the object item into the configuration of EB GUIDE GTF, use `addItem`.

Example: `settings->addItem(objectItem, "object_key");`

If the key already exists or the handle (`objectItem`) is null, `addItem` returns false.

12.4. Retrieving an item from `Configuration`

NOTE

Item types

The following instructions describe the steps for retrieving an item of `StringScalar` type. Use the same approach for all configuration item types. For a list of all configuration item types, see section 7.5, “Configuration of EB GUIDE GTF”.

Retrieving an item from `Configuration`

Prerequisite:

- You know the data type of the configuration item that you want to retrieve, for example `StringScalar`.

Step 1
Declare a handle for storing the retrieved value.

Example: `gtf::configuration::StringScalarHandle retrievedValue;`

Step 2
To retrieve the value, use `getItem`.

Example: `retrievedValue = settings->getItem<gtf::configuration::StringScalar>(key);`

The template parameter must be of the same type as the handle where you save the result (`retrievedValue`).

The key argument represents the key of the configuration item.

NOTE **Configuration items**

If the item does not exist in the current settings, EB GUIDE GTF searches for it recursively in all ancestors of the current settings. If the item can not be found in any of the ancestors, `getItem` returns null. For more information, see section 7.5, “Configuration of EB GUIDE GTF”.

12.5. Creating a path value using `Configuration`

Creating a path value using `Configuration`

Prerequisite:

- A `StringScalar` item is added in `Configuration`. For more information, see section 12.1, “Adding a scalar item to `Configuration`”.

Step 1
Declare a handle to the `PathValue` variable.

Example: `configuration::PathValueHandle pathHandle;`

Step 2
Retrieve the previously added `StringScalar` as `PathValue`.

Example: `pathHandle = settings->getItem<configuration::PathValue>("stringItemName");`

Step 3
Specify the base value for the path as `const char*`.

Example: `pathHandle->setBasePath("basePath");`

Step 4
Retrieve the resulting value.

Example: `const char* stringPath = pathHandle->get();`

12.6. Using a custom `.json` file to define configuration items.

Using a custom `.json` file to define configuration items.

Prerequisite:

- To retrieve a `ConfigurationImporter` instance from the `DependencyContainer` interface, use `get`.
- Example: `gtf::configurationimporter::ConfigurationImporterHandle configurationImporter = container->get<gtf::configurationimporter::ConfigurationImporter>();`
- Make sure that the retrieved `configurationImporter` handle is valid.

Step 1
To add the elements of the `.json` file into the `Configuration`, use `parse`.

Example: `bool result = configurationImporter->parse($JSON_PATH);`

If the file is missing or invalid, `parse` returns `false`.

Example 12.1. example.json

```
{
    "platform":
    {
        "users":
        [{
            "name": "Michael",
            "age": 22,
            "joined":
                {
                    "month": "March",
                    "year": 2017
                }
         }
        ],
        "version": 3.0
    }
}
```

Parsing of `example.json` adds the following items into `Configuration`:

Type	Key	Value
`ObjectValueHandle`	`"platform"`	Platform object
`ObjectListHandle`	`"platform.users"`	List of users
`FloatScalarHandle`	`"platform.version"`	3.0

for an example on how to define configuration items, see the EB GUIDE SDK example `ConfigurationImporter`.

12.7. Creating new `Settings`

Creating new `Settings`

Step 1
Declare a handle to a `Settings` variable.

Example: `gtf::configuration::SettingsHandle childSettings;`

Step 2
Store the result of the `createSettings` method.

Example: `childSettings = settings->createSettings();`

NOTE

The `createSettings` method sets settings as a parent for `childSettings`. This is useful for lookup when it retrieves a configuration item. For instructions, see section 12.4, “Retrieving an item from `Configuration`”.

13. Tracing logging messages

Tracing logging messages

For more information on traces, see section 7.2, “Tracing” and section 19.11, “`tracing.json` configuration file”.

Prerequisite:

- An EB GUIDE GTF extension or application is available.
- `tracing.json` is configured and available in one of the following directories:
  - The exported EB GUIDE model directory
  - The EB GUIDE project directory
  - The binaries directory

Step 1
Add the `Tracing.h` file to the source directory of your extension or application.

Step 2
In the `Tracing.h` file, include the `Logging.h` and define all the channels which are used inside the application:

```
#include <gtf/tracing/Logging.h>

#define GTF_MyChannel
```

Step 3
In your source file, include `Tracing.h` and add the required log macros, for example as follows:

```
GTF_WARNING(GTF_MyChannel, “My warning message”)
```

As a result the `log.txt` file is added and the defined traces are included.

NOTE

`log.txt` log file

Note that the `log.txt` file is always added next to `tracing.json`:

- If `tracing.json` is in the EB GUIDE project directory, the created `log.txt` contains only traces from the edit mode.
- If `tracing.json` is in the binaries directory, the created `log.txt` contains traces both from the edit mode and the simulation mode.
- If `tracing.json` is in both the EB GUIDE project directory and the binaries directory, the created `log.txt` in the EB GUIDE project directory contains traces from the edit mode, and the created `log.txt` in the binaries directory contains traces from the simulation mode.

For an example on how to add a custom trace output, see the EB GUIDE SDK example `TraceOutputExample`. 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.

14. Extending EB GUIDE Script with foreign functions

For an example on how to implement an EB GUIDE Script function that returns the sum of two integers, see the EB GUIDE SDK example `ScriptFunction`. 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.

15. Using the Properties module

For background information on the Properties module, see section 8.5, “Scenes and properties”.

15.1. Using Container interface

Obtaining a Container instance

To obtain an already created Container instance, use the DependencyResolver module. For instructions, see chapter 11, “Resolving interface dependencies“.

To create a new Container instance, use the Factory interface.

Step 1
To create an instance, use the createContainer method of the Factory interface.

Example:

```
using gtf::properties;
ContainerHandle container = factory->createContainer(type);
```

Accessing children of Container

Step 1
To access children of Container, use the getChildren method.

Example:

```
using gtf::properties;
Children& children = container->getChildren();
```

Accessing properties of Container

Step 1
To access properties of Container, use the getProperties method.

Example:

```
using gtf::properties;
Properties& properties = container->getProperties();
```

Accessing features of Container

Step 1
To access features of Container, use the getFeatures method.

Example:

```
using gtf::properties;
Features& features = container->getFeatures();
```

Accessing parent of Container

Step 1
To access the parent of Container, use the getParent method.

Example:

```
using gtf::properties;
ContainerHandle parent = container->getParent();
```

Setting parent of Container

Step 1
To set the parent of Container, use the setParent method.

Example:

```
using gtf::properties;
container->setParent(parentContainer);
```

Setting type of Container

Step 1
To set the type of Container, use the setType method.

Example:

```
using gtf::properties;
container->setType(type);
```

Getting type of Container

Step 1
To get the type of Container, use the getType method.

Example:

```
using gtf::properties;
Type containerType = container->getType();
```

15.2. Using Children interface

Accessing a Children container

Step 1
Access the Children container.

Example:

```
using gtf::properties;
Children& children = container->getChildren();
```

Retrieving a child from a specific index

Step 1
To retrieve a child, use the get method.

Example:

```
using gtf::properties;
ContainerHandle child = children.get(0); // Get the child at index 0
```

Inserting a child at a specified index

Step 1
To insert a child, use the get method.

Example:

```
using gtf::properties;
children.insert(0, containerHandle1); // Insert the child at index 0
```

Removing a child from a specified index

Step 1
To remove a child, use the remove method.

Example:

```
using gtf::properties;
```

```
children.remove(0); // Remove the child from index 0
```

Getting the number of children from a container

Step 1
To get the number of children, use the `count` method.

Example:

```
using gtf::properties;
uint321_t numberOfChildren = children.count();
```

15.3. Using `Property` interface

Accessing `Property` of a `Container` instance

Step 1
Get `Property` of the `Container` instance.

Example:

```
using gtf::properties;
Properties& properties = container->getProperties();
uint32_t index = 0;
```

Getting the number of elements in a list

Step 1
To get the number of elements, use the `count` method.

Example:

```
using gtf::properties;
uint32_t count = properties.count();
```

Getting a property from a specified index

Prerequisite:

- You know the type of the property that you want to retrieve.
- The property is stored in a handle of the variable type.

- The property index is provided by the `TypeResolver` in the `Scene` module.

Step 1
To retrieve a property, use the `get` method.

Example:

```
using gtf::properties;
IntegerScalarHandle intProp = properties.get<IntegerValue>(index);
```

Getting a property from a feature

Prerequisite:

- You have a feature identifier `Features::Feature` that is provided by `TypeResolver` from the `Scene` module.

Step 1
To retrieve a property from a feature, use the `get` method.

Example:

```
using gtf::properties;
Features::Feature feature;
IntegerScalarHandle intProp = preperties.get<IntegerValue>(feature, index);
```

Getting a property value

Step 1
To retrieve a scalar value, use the `get` method.

Example:

```
using gtf::properties;
IntegerScalarHandle intProp = properties.get<IntegerValue>(index);
int value = intProp->get();
int value1;
intProp->get(value1);
```

Step 2
To retrieve a list element value, use the `getElement` method.

Example:

```
using gtf::properties;
IntegerListHandle intProp = properties.get<IntegerList>(index);
int value = intProp->getElement(index);
```

```
int value1;
intProp->getElement(index, value1);
```

Editing a property

Prerequisite:

- To edit a property, you must first receive it from the Properties interface.

Step 1
To edit a scalar property, use the set method.

Example:

```
using gtf::properties;
IntegerScalarHandle intProp = properties.get<IntegerValue>(index);
intProp->set(7);
```

Step 2
To edit a list property, use the setElement method.

Example:

```
using gtf::properties;
IntegerListHandle intListProp = properties.get<IntegerList>(index);
intListProp->setElement(index, value);
```

Step 3
To insert an element before an index, use the insert method.

Example:

```
intListProp->insert(index, value);
```

Step 4
To append an element to a list, use the append method.

Example:

```
intListProp->append(value);
```

Observing structural changes of a widget

Step 1
To observe structural changes of a widget, like adding or removing elements, use the subscribe method from ContainerObserver.

Example:

```
using gtf::properties;
Listener listener; ❶
ContainerObserverHandle containerObserver = resolver->get<ContainerObserver>();
containerObserver->subscribe(container, listener,
    ContainerObserver::Callback::create<Listener,
    &Listener::processStructureChanged>(listener),
    ContainerObserver::Recursive::None); ❷ ❸
```

❶ The `Listener` class is a user-defined class in this example. The `Listener` class must contain a method that is called after the container structure has been changed. The method must have the following signature:

```
void processStructureChanged(const ContainerHandle& container_,
    container::Array<SubscriberNotificationData>& data_);
```

❷ If `ContainerObserver::Recursive::None` is used, only the notifications for the current container changes are received.

❸ If `ContainerObserver::Recursive::All` is used, the notifications for the current container changes as well as the notifications for its children are received.

Example 15.1.
`ContainerObserver`

A view contains an instantiator.

If you subscribe using `ContainerObserver::Recursive::None`, you will only be notified for the changes affecting the view itself.

If you subscribe using `ContainerObserver::Recursive::All`, you will also be notified for the changes affecting the view and its' children recursively. If an instantiator element is removed, you will be notified about it. If the instantiator widget is removed, you will be notified only of this removal and not of the removal of its children, because they are part of the instantiator.

Step 2
To stop receiving notifications from a container, use the `unsubscribe` method.

Example:

```
using gtf::properties;
containerObserver->unsubscribe(container, listener);
```

Observing properties of a `Property` container

Step 1
To observe properties of a `Property` container, use the `subscribe` method from `PropertyObserver`.

Example:

```
using gtf::properties;
Listener listener; ❶
PropertyKey key(container, 3); // "3" represents the property index
PropertyObserverHandle propertyObserver = resolver->get<PropertyObserver>();
propertyObserver->subscribe(key, listener, PropertyObserver::Callback::create<Listener,
    &Listener::processPropertyChanged>(listener));
```

❶ The `Listener` class is a user-defined class in this example. The `Listener` class must contain a method that is called after the value of a property has been changed. The method must have the following signature:

```
void processPropertyChanged(const ContainerHandle& container_,
    container::Array<SubscriberNotificationData>& data_);
```

Step 2

To stop receiving notifications from a container's properties, use the `unsubscribe` method.

Example:

```
using gtf::properties;
propertyObserver->unsubscribe(key, listener);
```

NOTE **Unsubscribing**

Always make sure to unsubscribe from all items that you subscribed to. For example, in case of a widget controller, unsubscribe at the latest when you disconnect.

16. Adding widgets and widget features

This section provides information about EB GUIDE GTF extension examples such as widgets and widget features.

For an example on how to add a widget, see the EB GUIDE SDK example `ExtendedContainerWidget`. 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.

This example implements a container widget that controls the visibility of its child widgets. The `displayStatus` property defines which child widgets are displayed: all, none, or only the first.

For an example on how to add a widget feature, see the EB GUIDE SDK example `FocusedWidgetFeature`. 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.

This example adds a widget feature to a rectangle widget. The example widget feature is called `focusRectColor` and has a property `focusedColor`. The purpose of the widget feature is to change the background color of a rectangle to the value of the `focusedColor` property when the rectangle is focused. To enable the focus widget feature, add the widget feature **State focused**.

For background information on the `Properties` module, see section 8.5, “Scenes and properties”.

For instructions on how to use the `Properties` module, see chapter 15, “Using the `Properties` module“.

17. Using external input events

17.1. Getting the `ExternalInput` interface

Getting the `ExternalInput` interface

Key, touch or rotary events can be created, sent and read using the `ExternalInput` interface. The `ExternalInput` is a help interface based on the functionality of a local event system.

For more information about local event system, see section 8.7, “Event system”.

For more information about `DependencyContainer`, see section 7.4, “Interface dependencies”.

Find the description of all relevant classes and methods in the EB GUIDE GTF API.

Step 1
To retrieve an `ExternalInput` interface from the `DependencyContainer`, use the `get` method.

Example:

```
gtf::inputmapper::ExternalInputHandle externalInput;
gtf::container::Array<gtf::coreruntime::CoreRuntimeHandle> runtimes;
modelContainer->getAll<gtf::coreruntime::CoreRuntime>(runtimes);

for (uint32_t i = 0, e = runtimes.size(); i != e; ++i)
{
    if (runtimes[i]->getContextId() == EXPECTED_RUNTIME_ID)
    {
        externalInput = runtimes[i]->getDependencyContainer()
                    ->get<gtf::inputmapper::ExternalInput>();
        break;
    }
}

if (externalInput.valid()) ❶
{
    //...
}
```

❶ Make sure that the retrieved interface instance is valid before you use it.

NOTE

Scope of `ExternalInput`

One EB GUIDE GTF instance may run multiple models. And each model may run multiple `CoreRuntime` instances. Each `CoreRuntime` uses a separate local event system. External input events that are send in one local event system are not available in other local event systems of other `CoreRuntime` instances.

To retrieve the `ExternalInput` interface, use the `DependencyContainer` of the respective `CoreRuntime` instance with which you want to interact.

NOTE

Identifiers of EB GUIDE model, `CoreRuntime` and scene

Each exported EB GUIDE model, as well as each `CoreRuntime` instance of a model, as well as each scene of one `CoreRuntime` instance can be identified. The identifiers are part of EB GUIDE model and exported to `model.json`. To retrieve this information, use the `gtf::configuration::Settings` interface.

- `CoreRuntime` is identified by the `id` of the `gtf.model.cores` objects in `model.json`
- A scene is identified by the `sceneId` of the `gtf.model.scenes` objects in `model.json`

17.2. Creating and sending a key event

Creating and sending a key event

Prerequisite:

- You retrieved an `ExternalInput` interface from `DependencyContainer`.

Step 1
Set the ID of the scene that should receive the key input event.

Example: `uint32_t sceneId = 0;`

NOTE

A scene is identified by the `sceneId` of the `gtf.model.scenes` objects in `model.json`

Step 2
Set the event ID of the key.

Example: `uint32_t keyEventId = 5;`

Step 3
Set the key status that caused the key event.

Example: `gtf::inputmapper::ExternalInput::KeyStatus::Value status = gtf::inputmapper::ExternalInput::KeyStatus::KeyPress;`

Find the description of the relevant values in the EB GUIDE GTF API.

Step 4
To create the key event, use the `createKeyEvent` method of the `ExternalInput` interface.

Example: `gtf::eventsystem::local::EventHandle keyInputEvent = externalInput->createKeyEvent(sceneId, keyEventId, status);`

Step 5
To send the newly created key event to the model and invoke processing, use the `send` method.

Example: `externalInput->send(keyInputEvent);`

17.3. Creating and sending a touch event

Creating and sending a touch event

Prerequisite:

- You retrieved an `ExternalInput` interface from `DependencyContainer`.

Step 1
Set the ID of the scene that should receive the touch input event.

Example: `uint32_t sceneId = 1;`

NOTE

A scene is identified by the `sceneId` of the `gtf.model.scenes` objects in `model.json`

Step 2
Set the event ID of the touch screen.

Example: `uint32_t touchEventId = 0;`

Step 3
Set the touch status.

Example: `gtf::inputmapper::ExternalInput::TouchStatus::Value status = gtf::inputmapper::ExternalInput::TouchStatus::TouchPress;`

Find the description of the relevant values in the EB GUIDE GTF API.

Step 4
Set the coordinates of the touch event.

Example: `uint32_t x = 100; uint32_t y = 50;`

Step 5
Set the finger ID.

Example: `uint32_t fingerId = 0;`

NOTE `fingerId` is used to track multiple parallel touch positions for multi-touch support.

Step 6
To create the touch event, use the `createTouchEvent` method of the `ExternalInput` interface.

Example: `gtf::eventsystem::local::EventHandle touchInputEvent = externalInput->createTouchEvent(sceneId, touchEventId, status, x, y, fingerId);`

Step 7
To send the newly created touch event to the model and invoke processing, use the `send` method.

Example: `externalInput->send(touchInputEvent);`

17.4. Creating and sending a rotary event

Creating and sending a rotary event

Prerequisite:

- You retrieved an `ExternalInput` interface from `DependencyContainer`.

Step 1
Set the ID of the scene that should receive the rotary input event.

Example: `uint32_t sceneId = 1;`

NOTE A scene is identified by the `sceneId` of the `gtf.model.scenes` objects in `model.json`

Step 2
Set the event ID of the rotary knob.

Example: `uint32_t rotaryEventId = 2;`

Step 3
Set the rotary increment or decrement value.

Example: `int32_t rotaryIncrement = 1;`

NOTE A positive value means rotation to the right while a negative value means a rotation to the left.

Step 4
To create the rotary event, use the `createRotaryEvent` method of the `ExternalInput` interface.

Example: `gtf::eventsystem::local::EventHandle rotaryInputEvent = externalInput->createRotaryEvent(sceneId, rotaryEventId, rotaryIncrement);`

Step 5
To send the newly created rotary event to the model and invoke processing, use the `send` method .

Example: `externalInput->send(rotaryInputEvent);`

17.5. Reading a key event

Reading a key event

Prerequisite:

- You retrieved an `ExternalInput` interface from `DependencyContainer`.
- You received an external input event.

Step 1
To read the key event, use the `readKeyEvent` method of the `ExternalInput` interface.

Example: `externalInput->readKeyEvent(keyInputEvent, outputSceneId, outputKeyEventId, outputKeyStatus);`

The `keyInputEvent` is read and it's parameter values are stored in the output variables.

17.6. Reading a touch event

Reading a touch event

Prerequisite:

- You retrieved an `ExternalInput` interface from `DependencyContainer`.
- You received an external input event.

Step 1
To read the touch event, use the `readTouchEvent` method of the `ExternalInput` interface.

Example: `externalInput->readTouchEvent(touchInputEvent, outputSceneId, outputTouchEventId, outputTouchStatus, outputX, outputY, outputFingerId);`

The `touchInputEvent` is read and it's parameters are placed in the output variables.

17.7. Reading a rotary event

Reading a rotary event

Prerequisite:

- You retrieved an `ExternalInput` interface from `DependencyContainer`.
- You received an external input event.

Step 1
Use the `readRotaryEvent` method of the `ExternalInput` interface to read the rotary event.

Example: `externalInput->readRotaryEvent(rotaryInputEvent, outputSceneId, outputRotaryEventId, outputRotaryIncrement);`

The `rotaryInputEvent` are read and it's parameters are placed in the output variables.

18. Monitoring memory usage

Monitoring memory usage helps you to debug the system and the EB GUIDE model. During the run-time, EB GUIDE GTF can continuously print out memory information that it dynamically manages.

The memory report is printed to the standard output streams. On the Microsoft Windows operating systems, outputs to console are deactivated. To see the stream outputs, pipe the streams into files. EB GUIDE Monitor also shows the memory report in the **Logger** component. On other operating systems that are POSIX compatible, for example Linux, the report is shown on console.

Configuring a memory report

You configure a memory report by adding a configuration message to the `model.json` configuration file.

Prerequisite:

- An EB GUIDE Studio project is opened.
- The project center is displayed.

Step 1
In the navigation area, select **Configure** > **Profiles**.

Step 2
Select the **Simulation** profile.

Step 3
Select the **Platform** tab.

Step 4
Enter the following code:

```
{
 "gtf": {
     "diagnostic": {
      "memory": {
             "interval":5000
         }
     }
 }
}
```

You configured the memory report to be printed with an interval of 5 seconds.

19. References

The following chapter provides you with lists and tables for example parameters, properties, and identifiers.

19.1. Android events

Android events belong to the `SystemNotifications` event group and have event group ID 13.

Table 19.1. 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

19.2. Datapool items

Table 19.2. Properties of a datapool item

Property name	Description
`Value`	The initial value of the datapool item

19.3. 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.

19.3.1. 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 19.3.12, “List”.

19.3.2. 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 19.3.12, “List”.

19.3.3. 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.

19.3.4. 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 19.3.12, “List”.

19.3.5. 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 19.3.12, “List”.

19.3.6. `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.

19.3.7. 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 19.3.12, “List”.

19.3.8. 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 19.3.12, “List”.

19.3.9. 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 19.3.12, “List”.

19.3.10. 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 19.3.12, “List”.

19.3.11. 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 19.3.12, “List”.

19.3.12. 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: ([])

19.4. EB GUIDE Script

19.4.1. 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

19.4.2. 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 19.3. 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

19.4.3. EB GUIDE Script standard library

The following chapter provides a description of all EB GUIDE Script functions.

19.4.3.1. EB GUIDE Script functions A - B

19.4.3.1.1. `abs`

The function returns the absolute value of the integer number `x`.

Table 19.4. Parameters of `abs`

Parameter	Type	Description
`x`	integer	The number to return the absolute value from
`<return>`	integer	The return value

19.4.3.1.2. `absf`

The function returns the absolute value of the float number `x`.

Table 19.5. Parameters of `absf`

Parameter	Type	Description
`x`	float	The number to return the absolute value from
`<return>`	float	The return value

19.4.3.1.3. `acosf`

The function returns the principal value of the arc cosine of `x`.

Table 19.6. Parameters of `acosf`

Parameter	Type	Description
`x`	float	The number to return the arc cosine from
`<return>`	float	The return value

19.4.3.1.4. `animation_before`

The function checks if a running animation has passed a given point in time.

Table 19.7. 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.

19.4.3.1.5. `animation_beyond`

The function checks if a running animation has passed a given point in time.

Table 19.8. 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.

19.4.3.1.6. `animation_cancel`

The function cancels an animation and leaves edited properties in the current state.

Table 19.9. Parameters of `animation_cancel`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

19.4.3.1.7. `animation_cancel_end`

The function cancels an animation and sets edited properties to the end state where possible.

Table 19.10. Parameters of `animation_cancel_end`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

19.4.3.1.8. `animation_cancel_reset`

The function cancels an animation and resets edited properties to the initial state where possible.

Table 19.11. Parameters of `animation_cancel_reset`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

19.4.3.1.9. `animation_pause`

The function pauses an animation.

Table 19.12. Parameters of `animation_pause`

Parameter	Type	Description
`animation`	GtfTypeRecord	The animation to manipulate
`<return>`	boolean	If true, the function succeeded.

19.4.3.1.10. `animation_play`

The function starts or continues an animation.

Table 19.13. 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.

19.4.3.1.11. animation_reverse

The function plays an animation backwards.

Table 19.14. Parameters of animation_reverse

Parameter	Type	Description
animation	GtfTypeRecord	The animation to manipulate
<return>	boolean	If true, the animation is not running yet.

19.4.3.1.12. animation_running

The function checks if an animation is currently running.

Table 19.15. Parameters of animation_running

Parameter	Type	Description
animation	GtfTypeRecord	The animation to manipulate
<return>	boolean	If true, the animation is running.

19.4.3.1.13. animation_set_time

The function sets the current time of an animation, can be used to skip or replay an animation.

Table 19.16. Parameters of animation_set_time

Parameter	Type	Description
animation	GtfTypeRecord	The animation to manipulate
time	integer	time
<return>	boolean	If true, the function succeeded.

19.4.3.1.14. asinf

The functions calculates the principal value of the arc sine of x.

Table 19.17. Parameters of `asinf`

Parameter	Type	Description
`x`	float	The number to return the arc sine from
`<return>`	float	The return value

19.4.3.1.15. `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 19.18. Parameters of `atan2f`

Parameter	Type	Description
`y`	float	Argument `y`
`x`	float	Argument `x`
`<return>`	float	The return value

19.4.3.1.16. `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 19.19. Parameters of `atan2i`

Parameter	Type	Description
`y`	integer	Argument `y`
`x`	integer	Argument `x`
`<return>`	float	The return value

19.4.3.1.17. `atanf`

The function calculates the principal value of the arc tangent of `x`.

Table 19.20. Parameters of `atanf`

Parameter	Type	Description
`x`	float	The number to return the arc tangent from

Parameter	Type	Description
`<return>`	float	The return value

19.4.3.1.18. `bool2string`

The function converts a boolean variable to either the string `true` or `false`.

Table 19.21. 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

19.4.3.2. EB GUIDE Script functions C - H

19.4.3.2.1. `ceil`

The function returns the smallest integral value that is not less than the argument.

Table 19.22. Parameters of `ceil`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

19.4.3.2.2. `changeDynamicStateMachinePriority`

The function changes the priority of a dynamic state machine.

Table 19.23. 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

19.4.3.2.3. `character2unicode`

The function returns the Unicode value of the first character in a string.

Table 19.24. Parameters of `character2unicode`

Parameter	Type	Description
`str`	string	The input string
`<return>`	integer	The character as Unicode value 0 in case of errors

19.4.3.2.4. `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 19.25. 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

19.4.3.2.5. `clampi`

The function clamps an integer value to a defined range `[xmin, xmax]`, this means the function computes `max (xmin, min (xmax, x))`.

Table 19.26. 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

19.4.3.2.6. `clearAllDynamicStateMachines`

The function removes all dynamic state machines from the dynamic state machine list.

Table 19.27. Parameters of `clearAllDynamicStateMachines`

Parameter	Type	Description
`state`		The state with the dynamic state machine list

19.4.3.2.7. `color2string`

The function converts a color to eight hexadecimal values.

Table 19.28. 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`.

19.4.3.2.8. `cosf`

The function returns the cosine of `x`, where `x` is given in radians.

Table 19.29. Parameters of `cosf`

Parameter	Type	Description
`x`	float	The number to return the cosine from
`<return>`	float	The return value

19.4.3.2.9. `deg2rad`

The function converts an angle from degrees to radians.

Table 19.30. Parameters of `deg2rad`

Parameter	Type	Description
`x`	float	The angle to convert from degrees to radians

Parameter	Type	Description
<return>	float	The return value

19.4.3.2.10. expf

The function returns the value of e, the base of natural logarithms, raised to the power of x.

Table 19.31. Parameters of expf

Parameter	Type	Description
x	float	The exponent
<return>	float	The return value

19.4.3.2.11. float2string

The function converts simple float to string.

Table 19.32. Parameters of float2string

Parameter	Type	Description
value	float	The value to convert to string
<return>	string	The float value, formatted as string

19.4.3.2.12. floor

The function returns the largest integral value not greater than the parameter value.

Table 19.33. Parameters of floor

Parameter	Type	Description
value	float	The value to round
<return>	integer	The rounded value

19.4.3.2.13. fmod

The function computes the remainder of the floating-point division x/y.

Table 19.34. 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`

19.4.3.2.14. `focusMoveTo`

The function forces the focus manager to forward the focus to a dedicated focusable element.

Table 19.35. Parameters of `focusMoveTo`

Parameter	Type	Description
`widget`	widget	The widget on which the focus is moved.
`<return>`	void	

19.4.3.2.15. `focusNext`

The function forces the focus manager to forward the focus to the next focusable element.

Table 19.36. Parameters of `focusNext`

Parameter	Type	Description
`<return>`	void	

19.4.3.2.16. `focusPrevious`

The function forces the focus manager to return the focus to the previous focusable element.

Table 19.37. Parameters of `focusPrevious`

Parameter	Type	Description
`<return>`	void	

19.4.3.2.17. `format_float`

The function formats a float value.

Table 19.38. 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.

19.4.3.2.18. `format_int`

The function formats an integer value.

Table 19.39. 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.

19.4.3.2.19. `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 19.40. Parameters of `frac`

Parameter	Type	Description
`x`	float	The floating point value
`<return>`	float	The fractional part of the floating-point value.

19.4.3.2.20. `getConfigItem`

The function fills a datapool item with a configuration item value.

Table 19.41. 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

19.4.3.2.21. `getFontAscender`

The function returns the ascender of the font passed as parameter.

Table 19.42. 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

19.4.3.2.22. `getFontDescender`

The function returns the descender of the font passed as parameter.

Table 19.43. 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

19.4.3.2.23. `getFontLineGap`

The function returns the line gap of the font passed as parameter.

Table 19.44. 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

19.4.3.2.24. `getImageHeight`

The function returns the height in pixels of an image passed as parameter.

Table 19.45. Parameters of `getImageHeight`

Parameter	Type	Description
`x`	image widget	The widget to evaluate
`<return>`	integer	The height in pixels of an image

19.4.3.2.25. `getImageWidth`

The function returns the width in pixels of an image passed as parameter.

Table 19.46. Parameters of `getImageWidth`

Parameter	Type	Description
`x`	image widget	The widget to be evaluated
`<return>`	integer	The width in pixels of an image

19.4.3.2.26. `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 19.47. 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

19.4.3.2.27. getLabelTextWidth

The function returns the width of the longest line of a label's text.

Table 19.48. 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

19.4.3.2.28. getLineCount

The function returns the number of lines of a label's text.

Table 19.49. Parameters of getLineCount

Parameter	Type	Description
widget	label widget	The widget to be evaluated
<return>	integer	The number of lines of the text

19.4.3.2.29. getLineHeight

The function returns the height of a line written with the font passed as parameter.

Table 19.50. 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

19.4.3.2.30. getProductString

The function returns a string with the product name of EB GUIDE GTF.

Table 19.51. Parameters of `getProductString`

Parameter	Type	Description
`<return>`	string	The product name

19.4.3.2.31. `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 19.52. 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.

19.4.3.2.32. `getTextLength`

The function returns the number of characters in a text.

Table 19.53. 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`.

19.4.3.2.33. `getTextWidth`

The function returns the width of a text with regard to its font resource.

Table 19.54. 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.

19.4.3.2.34. `getVersionString`

The function returns a string with the version number of EB GUIDE GTF.

Table 19.55. Parameters of `getVersionString`

Parameter	Type	Description
`<return>`	string	The version string

19.4.3.2.35. `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 19.56. 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.

19.4.3.2.36. `hsba2color`

The function converts an HSB/HSV color to an EB GUIDE GTF color.

Table 19.57. 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

19.4.3.3. EB GUIDE Script functions I - R

19.4.3.3.1. `int2float`

The function returns the integer value converted to a float point value.

Table 19.58. Parameters of `int2float`

Parameter	Type	Description
`value`	integer	The value to convert to float
`<return>`	float	The integer value, converted to float

19.4.3.3.2. `int2string`

The function converts a simple integer to string.

Table 19.59. 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

19.4.3.3.3. `isDynamicStateMachineActive`

The function checks if the state with the dynamic state machine list is active.

Table 19.60. Parameters of `isDynamicStateMachineActive`

Parameter	Type	Description
`state`		The state with the dynamic state machine list
`sm`	integer	The dynamic state machine

19.4.3.3.4. `isWidgetOnActiveStatemachine`

The function checks if the widget belongs to an active state machine.

Table 19.61. 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

19.4.3.3.5. `language`

The function switches the language of all datapool items. This operation is performed asynchronously.

Table 19.62. Parameters of `language`

Parameter	Type	Description
`language`	languageType	The language to switch to, for example `f:language(l:German)`
`<return>`	void	

19.4.3.3.6. `lerp`

The function calculates the linear interpolation of two values x and y using the formula `(1-s) * x + s * y`

Table 19.63. 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`

19.4.3.3.7. `localtime_day`

The function extracts the day [1:31] in local time from a system time value.

Table 19.64. Parameters of `localtime_day`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted day

19.4.3.3.8. `localtime_hour`

The function extracts the hours from the local time of a system time value.

Table 19.65. Parameters of `localtime_hour`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted hour

19.4.3.3.9. `localtime_minute`

The function extracts the minutes from the local time of a system time value.

Table 19.66. Parameters of `localtime_minute`

Parameter	Type	Description
`time`	integer	A time stamp as returned by system time
`<return>`	integer	The extracted minute

19.4.3.3.10. `localtime_month`

The function extracts the month [0:11] from the local time of a system time value.

Table 19.67. 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

19.4.3.3.11. localtime_second

The function extracts the seconds from the local time of a system time value.

Table 19.68. Parameters of localtime_second

Parameter	Type	Description
time	integer	A time stamp as returned by system time
<return>	integer	The extracted second

19.4.3.3.12. localtime_weekday

The function extracts the week day [0:6] from the local time of a system time value. 0 is Sunday.

Table 19.69. Parameters of localtime_weekday

Parameter	Type	Description
time	integer	A time stamp as returned by system time
<return>	integer	The extracted weekday

19.4.3.3.13. localtime_year

The function extracts the year from the local time of a system time value.

Table 19.70. Parameters of localtime_year

Parameter	Type	Description
time	integer	A time stamp as returned by system time
<return>	integer	The extracted year

19.4.3.3.14. log10f

The function returns the base 10 logarithm of x.

Table 19.71. Parameters of `log10f`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

19.4.3.3.15. `logf`

The function returns the natural logarithm of `x`.

Table 19.72. Parameters of `logf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

19.4.3.3.16. `maxf`

The function computes the maximum of two floating-point values.

Table 19.73. Parameters of `maxf`

Parameter	Type	Description
`x`	float	The first value
`y`	float	The second value
`<return>`	float	The maximum of x and y

19.4.3.3.17. `maxi`

The function computes the maximum of two integer values.

Table 19.74. 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

19.4.3.3.18. minf

The function computes the minimum of two floating-point values.

Table 19.75. Parameters of minf

Parameter	Type	Description
x	float	The first value
y	float	The second value
<return>	float	The minimum of x and y

19.4.3.3.19. mini

The function computes the minimum of two integer values.

Table 19.76. Parameters of mini

Parameter	Type	Description
x	int	The first value
y	int	The second value
<return>	int	The minimum of x and y

19.4.3.3.20. nearbyint

The function rounds to nearest integer.

Table 19.77. Parameters of nearbyint

Parameter	Type	Description
value	float	The value to round
<return>	integer	The rounded value

19.4.3.3.21. popDynamicStateMachine

The function removes the specified dynamic state machine from the priority queue.

Table 19.78. Parameters of `popDynamicStateMachine`

Parameter	Type	Description
`state`		The state with the dynamic state machine list
`sm`	string	The dynamic state machine

19.4.3.3.22. `powf`

The function returns the value of `x` raised to the power of `y`.

Table 19.79. Parameters of `powf`

Parameter	Type	Description
`x`	float	The argument `x`
`y`	float	The argument `y`
`<return>`	float	The return value

19.4.3.3.23. `pushDynamicStateMachine`

The function inserts the dynamic state machine in a priority queue.

Table 19.80. 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`.

19.4.3.3.24. `rad2deg`

The function converts an angle form radians to degree.

Table 19.81. Parameters of `rad2deg`

Parameter	Type	Description
`x`	float	The argument

Parameter	Type	Description
`<return>`	float	The return value

19.4.3.3.25. `rand`

The function gets a random value between 0 and 2^{31}-1.

Table 19.82. Parameters of `rand`

Parameter	Type	Description
`<return>`	integer	A random number between 0 and 2^{31}-1

19.4.3.3.26. `rgba2color`

The function converts from RGB color space to EB GUIDE GTF color.

Table 19.83. 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

19.4.3.3.27. `round`

The function rounds to nearest integer, but rounds halfway cases away from zero.

Table 19.84. Parameters of `round`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

19.4.3.4. EB GUIDE Script functions S - W

19.4.3.4.1. `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 19.85. Parameters of `saturate`

Parameter	Type	Description
`x`	float	The value to clamp
`<return>`	float	The `x` value clamped to the [0, 1] range

19.4.3.4.2. `seed_rand`

The function sets the seed of the random number generator.

Table 19.86. Parameters of `seed_rand`

Parameter	Type	Description
`seed`	integer	The value to seed the random number generator
`<return>`	void	

19.4.3.4.3. `shutdown`

The function requests the framework to shutdown the program.

19.4.3.4.4. `sinf`

The function returns the sine of `x`, where `x` is given in radians.

Table 19.87. Parameters of `sinf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

19.4.3.4.5. `skin`

The function switches the skin of all datapool items. This operation is performed asynchronously.

Table 19.88. Parameters of `skin`

Parameter	Type	Description
`skin`	skinType	The skin to switch to, for example `f:skin(s:Standard)`
`<return>`	void	

19.4.3.4.6. `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 19.89. 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)`

19.4.3.4.7. `sqrtf`

The function returns the non-negative square root of `x`.

Table 19.90. Parameters of `sqrtf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

19.4.3.4.8. `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 19.91. Parameters of `string2float`

Parameter	Type	Description
`str`	string	The string value
`<return>`	float	The return value

19.4.3.4.9. `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 19.92. Parameters of `string2int`

Parameter	Type	Description
`str`	string	The string value
`<return>`	integer	The return value

19.4.3.4.10. `string2string`

The function is used to truncate a string to a given number of characters.

Table 19.93. 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

19.4.3.4.11. `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 19.94. 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

19.4.3.4.12. `system_time`

The function gets the current system time in seconds. The result is intended to be passed to the `localtime_*` functions.

Table 19.95. Parameters of `system_time`

Parameter	Type	Description
`<return>`	integer	The system time in seconds

19.4.3.4.13. `system_time_ms`

The function gets the current system time in milliseconds.

Table 19.96. Parameters of `system_time_ms`

Parameter	Type	Description
`<return>`	integer	The system time in milliseconds

19.4.3.4.14. `tanf`

The function returns the tangent of `x`, where `x` is given in radians.

Table 19.97. Parameters of `tanf`

Parameter	Type	Description
`x`	float	The argument
`<return>`	float	The return value

19.4.3.4.15. `trace_dp`

The function writes debugging information about a datapool item to the trace log and the connection log.

Table 19.98. Parameters of `trace_dp`

Parameter	Type	Description
`itemId`	dp_id	The datapool ID of the item to trace debug information about
`<return>`	void	

19.4.3.4.16. `trace_string`

The function writes a string to the trace log and the connection log.

Table 19.99. Parameters of `trace_string`

Parameter	Type	Description
`str`	string	The text to trace
`<return>`	void	

19.4.3.4.17. `transformToScreenX`

The function takes a widget and a local coordinate and returns x-position in the screen-relative world coordinate system.

Table 19.100. 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

19.4.3.4.18. 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 19.101. 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

19.4.3.4.19. 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 19.102. 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

19.4.3.4.20. 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 19.103. 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

19.4.3.4.21. `trunc`

The function rounds to the nearest integer value, always towards zero.

Table 19.104. Parameters of `trunc`

Parameter	Type	Description
`value`	float	The value to round
`<return>`	integer	The rounded value

19.4.3.4.22. `widgetGetChildCount`

The function obtains the number of child widgets of the given widget.

Table 19.105. 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

19.5. Events

Table 19.106. 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.

19.5.1. Decimal codes for key events

Table 19.107. 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 19.108. 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 19.109. 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

19.6. `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 19.6.1, “Example `model.json` in EB GUIDE Studio”.

For more information about the `JSON` format, see http://www.json.org.

Table 19.110. 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 19.111. 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 19.112. 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 19.113. 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 19.114. 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 traversal 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 19.115. 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 19.116. TextEngine configuration items

Configuration item	Type	Description	Default value
gtf.model.textengine.replacementGlyph	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 19.116, “`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 19.117. 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 19.118. 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`

19.6.1. Example `model.json` in EB GUIDE Studio

Example 19.1. `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"
      ]
    }
  }
}
```

19.7. 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 19.119. 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

19.8. 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 19.8.1, “Example `platform.json` in EB GUIDE Studio”.

For more information about the `JSON` format, see http://www.json.org.

Table 19.120. 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 19.8.1, “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 19.8.1, “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`

19.8.1. Example `platform.json` in EB GUIDE Studio

Example 19.2.
`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"
        ]
      }
    }
  }
```

19.9. Scenes

Table 19.121. 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.

19.10. Touch screen types supported by EB GUIDE GTF

The supported types depend on the target device.

Table 19.122. 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

19.11. `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 19.11.2, “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 19.123. 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 19.11.1, “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. `gtf-fileoutput`, `gtfker-neleventoutput`, `std-output`, `gtfvsdebu-goutput`.	
`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`

19.11.1. Severity levels

The following table lists the trace severity levels that you can use.

Table 19.124. 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 19.124, “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.

19.11.2. Example `tracing.json`

Example 19.3.
`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.

19.12. Widgets

19.12.1. View

Table 19.125. 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 19.126. 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`

19.12.2. 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.

19.12.2.1. 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 19.127. 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.

19.12.2.2. 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 19.128. 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`

19.12.2.2.1. 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 19.129. 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

19.12.2.2.2. 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 19.130. 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

19.12.2.2.3. 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 19.131. 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

19.12.2.2.4. 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 19.132. 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

19.12.2.2.5. 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 19.133. 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

19.12.2.2.6. 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 19.134. 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

19.12.2.2.7. 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 19.135. 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

19.12.2.2.8. 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 19.136. 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

19.12.2.3. Container

A container holds several widgets as child widgets and thus groups the widgets.

Table 19.137. 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

19.12.2.4. 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 19.138. 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

19.12.2.5. Image

An image places a picture into a view.

Table 19.139. 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.

19.12.2.6. 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 19.140. 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

19.12.2.7. Label

A label places text into a view.

Table 19.141. 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

19.12.2.8. Rectangle

A rectangle draws a colored rectangle with the dimensions and coordinates of the widget into a view.

Table 19.142. 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

19.12.3. 3D widgets

19.12.3.1. 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 19.143. 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

19.12.3.2. 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 19.144. 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.

19.12.3.3. Directional light

A directional light illuminates the scene from one direction.

Table 19.145. 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

19.12.3.4. 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 19.146. 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

19.12.3.5. Material

A material defines the visual appearance of the mesh surface using the Phong reflection model.

Table 19.147. 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.

19.12.3.6. Mesh

A mesh defines the shape of the 3D object.

Table 19.148. 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

19.12.3.7. PBR GGX material

A PBR GGX material defines the visual appearance of the mesh surface using the physically correct Cook-Torrance model.

Table 19.149. 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`.

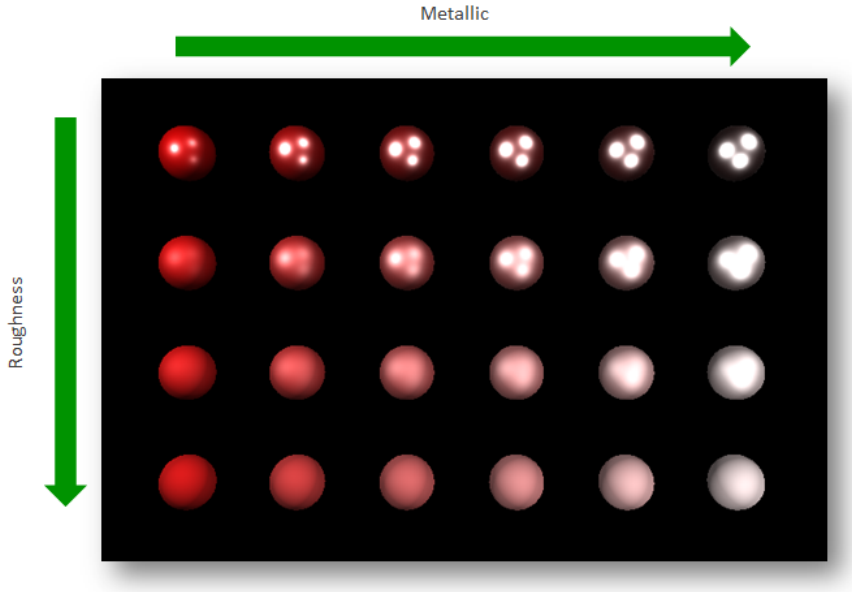


Figure 19.1. Example for a physically-based material

19.12.3.8. 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 19.150. 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`.

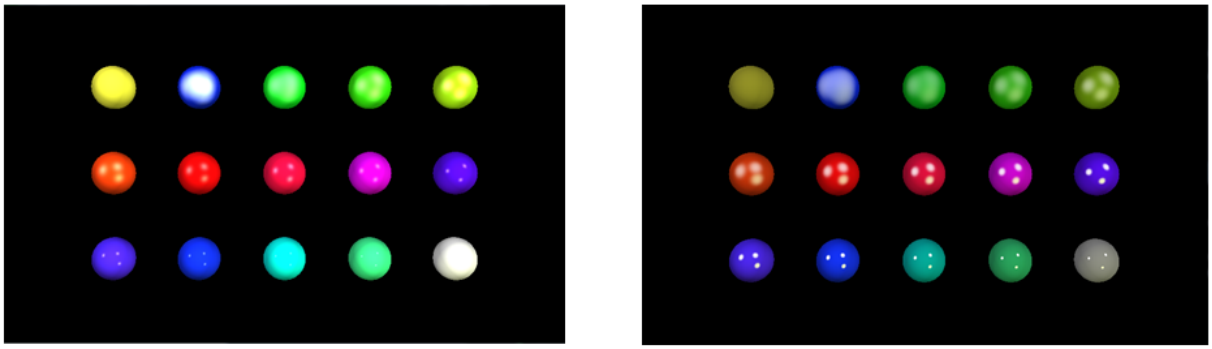


Figure 19.2. Example for a non-normalized material (left) and a normalized material (right)

19.12.3.9. Point light

A point light adds a light to the scene that emits light in all directions like a light bulb.

Table 19.151. 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.

19.12.3.10. Scene graph

A scene graph places a 3D object into a view.

Table 19.152. 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.

19.12.3.11. 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 19.153. 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

19.12.3.12. Spot light

A spot light adds a light which restricts illumination to a cone of influence.

Table 19.154. 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

19.13. 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.

19.13.1. Common

19.13.1.1. 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 19.155. 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.

19.13.1.2. Enabled

The **Enabled** widget feature adds an `enabled` property to a widget.

Table 19.156. Properties of the **Enabled** widget feature

Property name	Description
`enabled`	If true, the widget reacts on touch and press input

19.13.1.3. Focused

The **Focused** widget feature enables a widget to have input focus.

Table 19.157. 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

19.13.1.4. Multiple lines

The **Multiple lines** widget feature enables line breaks.

Restrictions:

- The **Multiple lines** widget feature is only available for the label widget.

Table 19.158. 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 19.4.3.2.28, “`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 ' '.

19.13.1.5. 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 19.159. 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.

19.13.1.6. 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 19.160. Properties of the **Selected** widget feature

Property name	Description
`selected`	If true, the widget is selected

19.13.1.7. 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 19.161. 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.

19.13.1.8. 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 19.162. 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.

19.13.1.9. 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 19.163. 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

19.13.1.10. Touched

The **Touched** widget feature enables a widget to react to touch input.

Table 19.164. 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)`.

19.13.2. Effect

19.13.2.1. 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 19.165. 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

19.13.2.2. Coloration

The **Coloration** widget feature colors the widget and its widget subtree. It also affects transparency if the alpha value is not opaque.

Example 19.4.
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 19.166. Properties of the **Coloration** widget feature

Property name	Description
`colorationEnabled`	If true, coloration is used
`colorationColor`	The color used for the coloration

19.13.2.3. Stroke

The **Stroke** widget feature activates a configurable text outline, i.e. a label border.

Restrictions:

- The widget feature is available for labels.

Table 19.167. 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

19.13.3. Focus

The **Focus** widget feature category provides the widget features relating to focus management.

19.13.3.1. 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 19.168. 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.

19.13.3.2. 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 19.169. 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

19.13.4. Gestures

19.13.4.1. 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 19.170. 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

19.13.4.2. 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 19.171. 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

19.13.4.3. 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 19.172. 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

19.13.4.4. 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 19.173. 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

19.13.4.4.1. 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 19.174. 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

19.13.4.5. 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 19.175. 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

19.13.4.6. 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 19.176. 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

19.13.5. Input handling

19.13.5.1. 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.

19.13.5.2. 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 19.177. 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

19.13.5.3. 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 19.178. 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

19.13.5.4. 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 19.179. 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

19.13.5.5. 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 19.180. 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

19.13.5.6. 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 19.181. 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

19.13.5.7. 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 19.182. 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

19.13.5.8. 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 19.183. 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

19.13.5.9. 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 19.184. 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)

19.13.5.10. 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 19.185. 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

19.13.5.11. 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 19.186. 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

19.13.5.12. 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 19.187. 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

19.13.5.13. 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 19.188. 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

19.13.5.14. 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 19.189. 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

19.13.5.15. 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 19.190. 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

19.13.6. Layout

19.13.6.1. 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 19.191. 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.

19.13.6.2. 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 19.192. 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)`

19.13.6.3. 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 19.193. 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.

19.13.6.4. 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 19.194. 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

19.13.6.5. 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 19.195. 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

19.13.6.6. 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 19.13.7.2, “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 19.196. 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.

19.13.6.7. 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 19.197. 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`

19.13.7. List management

19.13.7.1. 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 19.198. Properties of the **Line index** widget feature

Property name	Description
`lineIndex`	The index of the current line in a table

19.13.7.2. 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 19.199. Properties of the **List index** widget feature

Property name	Description
`listIndex`	The index of the current widget in a list

19.13.7.3. 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 19.200. Properties of the **Template index** widget feature

Property name	Description
`lineTemplateIndex`	The index of the used line template

19.13.7.4. 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 19.201. 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

19.13.8. 3D

Widget features in the **3D** category are only available for 3D widgets.

19.13.8.1. 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 19.202. 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

19.13.8.2. 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 19.203. 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)`.

19.13.8.3. 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 19.204. 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)`.

19.13.8.4. 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 19.205. 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)`.

19.13.8.5. 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 19.206. Properties of the **Light map texture** widget feature

Property name	Description
`lightMapTexture`	The file name of the texture
`lightMapTextureAddressModeU`	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.
`lightMapTextureAddressModeV`	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.

19.13.8.6. 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 19.207. 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.

19.13.8.7. 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 19.208. Properties of the **Normal map** widget feature

Property name	Description
`normalMapTexture`	The file name of the texture

Property name	Description
`normalMapTextureAddressModeU`	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.
`normalMapTextureAddressModeV`	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.

19.13.8.8. 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 19.209. 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.

19.13.8.9. 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 19.210. 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.

19.13.8.10. 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 19.211. 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.
`roughnessTextureAddressModeV`	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.

19.13.8.11. 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 19.212. 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
`shininessTextureAddressModeU`	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.
`shininessTextureAddressModeV`	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.

19.13.8.12. 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 19.213. Properties of the **Specular texture** widget feature

Property name	Description
`specularTexture`	The file name of the texture
`specularTextureAddressModeU`	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.
`specularTextureAddressModeV`	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)`.

19.13.8.13. Texture coordinate transformation

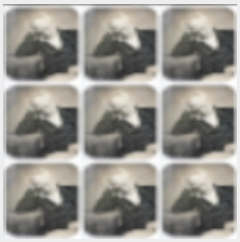
The **Texture coordinate transformation** widget feature makes it possible to modify the coordinates of the material texture. This feature is available for material, PBR Phong material, and PBR GGX material.

Table 19.214. 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 19.5.
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
				

19.13.8.14. 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 19.215. 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.

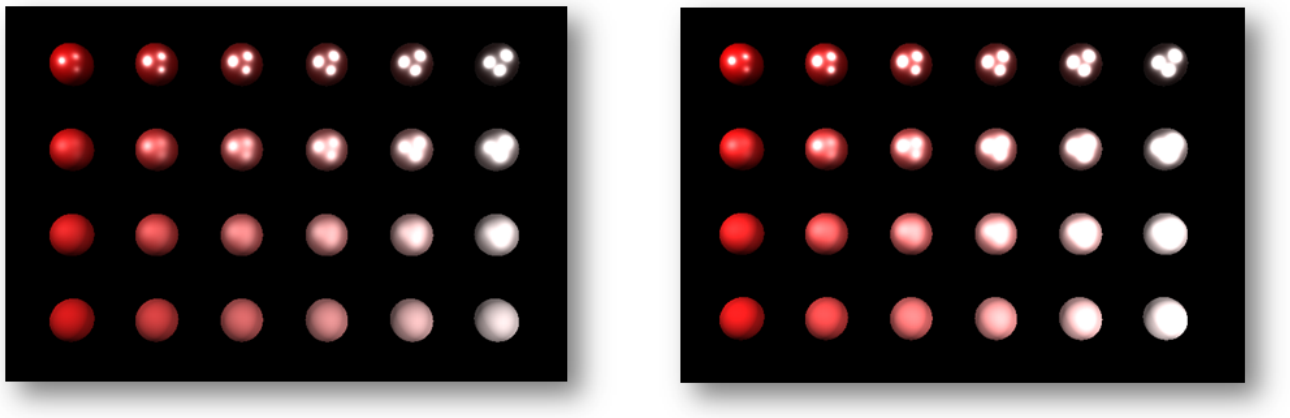


Figure 19.3. Example for image without tone mapping (left) and with tone mapping (right)

[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

19.13.9. 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

19.13.9.1. 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 19.216. 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

19.13.9.2. Rotation

The **Rotation** widget feature is used to rotate the widget and its subtree.

Table 19.217. 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

19.13.9.3. Scaling

The **Scaling** widget feature is used to scale the widget and its subtree.

Table 19.218. 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

19.13.9.4. Shearing

The **Shearing** widget feature is used to distort the widget and its subtree.

Table 19.219. 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

19.13.9.5. Translation

The **Translation** widget feature is used to translate the widget and its subtree. It moves widgets in x, y and z directions.

Table 19.220. 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

Appendix A. EB GUIDE TF APK

A.1. Installation of EB GUIDE TF on Android

The Android application package (APK) file format is used to distribute and install applications and other middleware on Android devices.

A.1.1. System requirements

The Android APK version that is currently released for EB GUIDE TF is designed to run on a wide range of Android devices.

Table A.1. Minimal requirements

Architecture	ARMv7
Platform	EB GUIDE TF: Android 5.0 (API Level 21)

A.1.2. Features of the EB GUIDE TF APK

Table A.2. Features of the EB GUIDE TF APK

Feature	**Description**
Life cycle management	EB GUIDE TF supports Android life cycle management.
Multi-touch support	EB GUIDE TF supports up to ten fingers for multi-touch. The number of supported fingers may be limited by the Android device.
Key handling	EB GUIDE TF processes 16-bit UTF key mapping codes.
Interaction with the Java API	EB GUIDE TF can be started and controlled by the Android activity. Example code and a template implementation are provided by the application. A native activity is not necessary.
Android layout handling	The exported EB GUIDE model is informed through events if the layout of the visible screen area changes. That way you can handle a virtual keyboard or changes in rotation.

A.1.3. Description of the EB GUIDE TF APK files

- `EB GUIDE Launcher.apk`

  The EB GUIDE Launcher starts EB GUIDE TF and displays the exported EB GUIDE model.

  If you start the EB GUIDE Launcher, it displays the exported EB GUIDE model that was selected last by EB GUIDE Model Chooser.

- `EB GUIDE Model Chooser.apk`

  The EB GUIDE Model Chooser provides a user interface to select an exported EB GUIDE model that is executed on the Android device.

  By selecting an exported EB GUIDE model, the EB GUIDE Launcher is started with the corresponding model.

  Clicking the **Info** button displays the directory where exported EB GUIDE models are stored for the EB GUIDE Model Chooser, and a list with device-related details. For information about the location of the exported EB GUIDE models in the file system, see section A.1.5, “Directory for EB GUIDE models”.

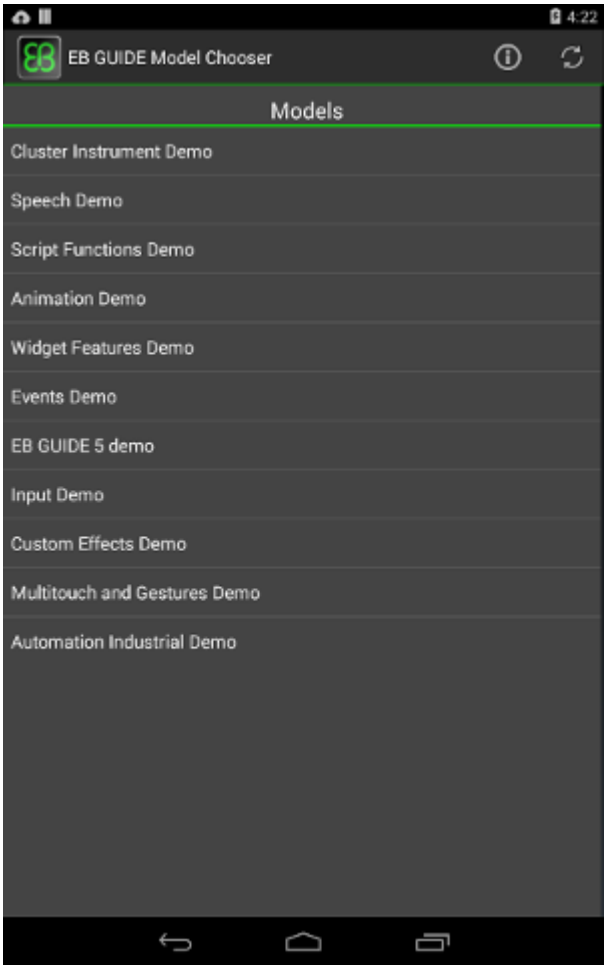


Figure A.1. EB GUIDE Model Chooser

A.1.3.1. Released APK and custom APK

EB GUIDE TF is delivered and installed as an APK. Use either a pre-built released APK of the released version or create a custom version based on the delivered Android binaries and the APK template in the SDK.

The following lists help you to decide whether or not you need a custom APK.

If the following applies to your project, use the released APK:

- Your project contains EB GUIDE functionality or feature demonstrations with no further extensions.
- Your project contains project-specific extensions, for example EB GUIDE GTF extensions, to be added to the exported EB GUIDE model.
- Standard access rights are sufficient for your project. The standard access rights are read or write to the external storage of the device, network access `android.permission.INTERNET`, record audio, and modify audio settings.

If the following applies to your project, use the delivered APK template:

- You need additional access rights that are not granted by the released APK version, for example `CALL_PHONE`.
- You require a customer-specific APK, for example a customer signature for APK verification or icons.
- You use Android framework features that are not accessible in the stable API of the native development kit (NDK). The NDK contains only a small subset of features and functionality which you can use with the Java API.
- You need additional Android application functionalities that require modifications to Java-related code pieces, for example activities, services, skins, intents, or compositing.

A.1.3.2. Restrictions

The Android APK that is currently released for EB GUIDE TF has the following restrictions:

- The exported EB GUIDE model is informed about rotation changes and layout changes, for example an incoming virtual keyboard on the display. The exported EB GUIDE model must handle these events.
- If the system uses Android layout handling, the Android flag `SOFT_INPUT_ADJUST_NOTHING` must not be set in the configuration of the Android activity.

A.1.4. Android life cycle management

The Android life cycle management is an optimization implemented by the Android operating system. If an application moves to the background, Android releases all graphics resources and makes the resources available for the application that moves to the foreground. An application is responsible for recreating the resources when it moves to the foreground.

A.1.5. Directory for EB GUIDE models

EB GUIDE models are stored in the `com.elektrobit.guide_model_chooser/files` directory that is located on the primary external storage directory. Application-related files are stored there permanently. One directory is required per EB GUIDE model.

NOTE

`com.elektrobit.guide_model_chooser/files` directory

After you installed EB GUIDE Model Chooser and EB GUIDE Launcher the `com.elektrobit.guide_model_chooser/files` directory does not yet exist. The directory is created when you start EB GUIDE Model Chooser for the first time. Alternatively, you can also add the directory manually.

The EB GUIDE Model Chooser searches only in the primary external storage directory. Usually Android devices have their primary external storage on a portion of the internal storage. Make sure you copy the files to the correct place.

Examples for primary external storage directory:

- For a Samsung Galaxy S3 device with Android 5.0 that is connected to a PC with Windows 7, the directory is `Computer/GT-I9300/Phone/Android/data/com.elektrobit.guide_model_chooser/files`.
- For a Nexus 7 device with Android 5.0 that is connected to a PC with Windows 7, the directory is `Computer/Nexus 7/Internal storage/Android/data/com.elektrobit.guide_model_chooser/files`.

On start-up or refresh, EB GUIDE Model Chooser recursively scans the directory for the EB GUIDE TF configuration file `model.json`. The parent directory for each start-up configuration is displayed as the EB GUIDE model name.

A.1.6. Android layout handling

Android is designed for mobile devices. On a mobile device, some characteristics concerning the layout of the visible screen area need to be considered.

EB GUIDE provides events that indicate layout changes in the visible screen area.

Example A.1.
Examples for layout handling

- When a mobile device is rotated, the GUI has to adapt according to the rotation.
- When a virtual keyboard is displayed on the screen, the GUI has to adapt to the new element.

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

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W