CGI Studio Connector

- Overview
- Connector
  - Welcome Screen
  - New Project
  - Modules
  - Interfaces
  - Edit Menu
  - Build Menu
  - Project Settings
  - Builds
- Complex Types
  - XML Definition
  - Converter
  - Editor
- Native Functions
  - XML Definition
  - Implementation
  - Editor
- Build Projects
  - XML Definition

    - Sample
    - Editor
    - Default variables
    - Command Line Interface

- Interface
    - Interface-Buffer
    - Action-Context
    - Scene-Context
    - Data-Model

- Tools
    - Interface-Code Generator
    - Model-Code Generator
    - Build Command Line Interface

- Tool Support
    - Overview
    - Generic
    - MATLAB

Overview

The CGI Studio Connector is an additional tool for CGI Studio which makes it easy to connect a CGI Studio Courier application to other systems. It provides a visual editor to define the data, actions and events which are received and send by the application.

Further it provides additional tools to integrate more deeply with specific 3rd party tools and interfaces.

Connector

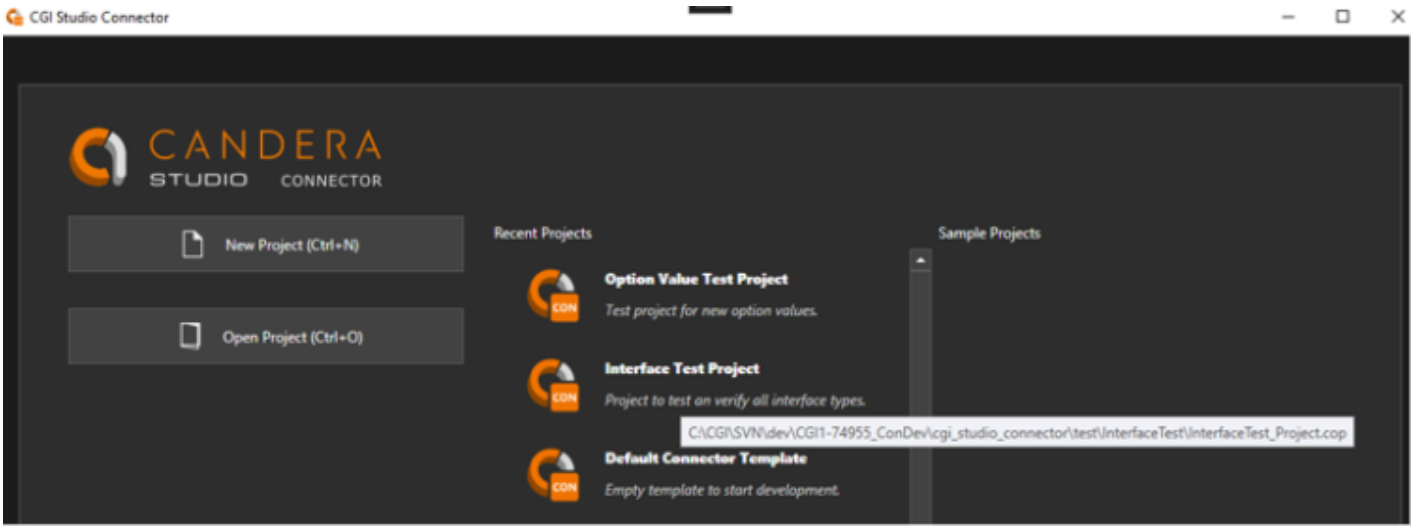
This is the central tool which allows the user to define the interfaces to interact with the other system.

Welcome Screen

The welcome screen allows the user to create a new (Ctrl + N) or open an existing (Ctrl + O) CGI Connector Project.

Additionally, it also shows a list of the recently used projects to provide quick access. The exact location of the project can be seen in the tooltip of the individual projects.

It will also display a list of sample projects that can be used as references.



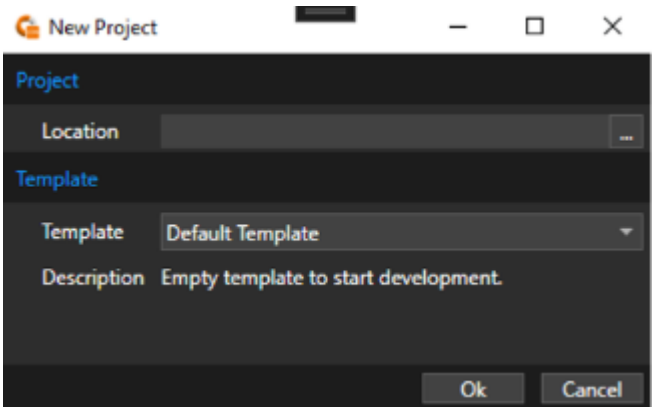
New Project

A new project can be created from the Welcome Screen or via the menu bar (File > New Project) or via the shortcut “Ctrl + N”. This will open additional windows to help setting up the new project.

Template-Selection

In the Template-Selection screen the user needs to first select the location where the new project shall be stored and select which template shall be used as a starting point for the project.

A template usually contains premade application code, build scripts, tooling related files and predefined project settings.



Setup

On the next screen the user can configure the most important project settings.

General

Property Name	Description
Name	Name of the project.
Description	Description of the project.

Tools

Property Name	Description
Generic	Enables generic support. This will provide a plain C/C++ interface.

MATLAB	Enables support for MATLAB. This will provide additional options/setting related to MathWorks MATLAB support

CGI Studio

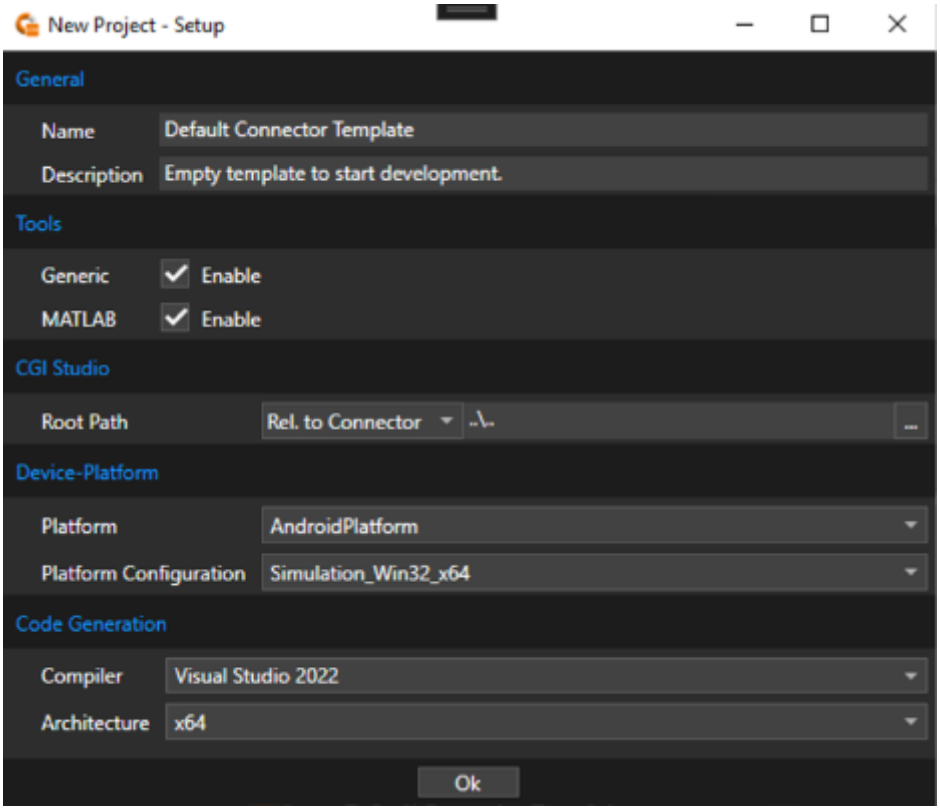
Property Name	Description
Root Path	Path to CGI Studio package. The path can be either absolute or relative to CGI Connector/Project paths.

Device-Platform

Property Name	Description
Platform	Name of the CGI Studio Platform (as found in CMake)
Platform Configuration	Name of the CGI Studio Platform configuration (as found in CMake)

Code Generation

Property Name	Description
Compiler	Compiler used to build code.
Architecture	Architecture used to build the code.



> These, and more, settings can later be changed. For more information, please refer to the “Project Settings” chapter.

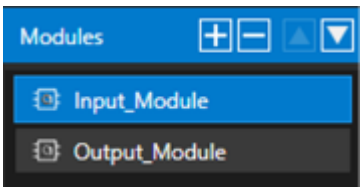
Custom Templates

It is possible to add custom templates to the CGI Connector. This can be useful to apply customizations from a project to other projects.
To create a template, pack all needed content into a ZIP file. Make sure that it contains the CGI Connector-Project file (.cop) in the root. Then add an entry to the “Templates.xml” in “cgi_studio_connector/bin” where the name, description and file location needs to be specified. Note that the file location is relative to the CGI Connector binaries.

Modules

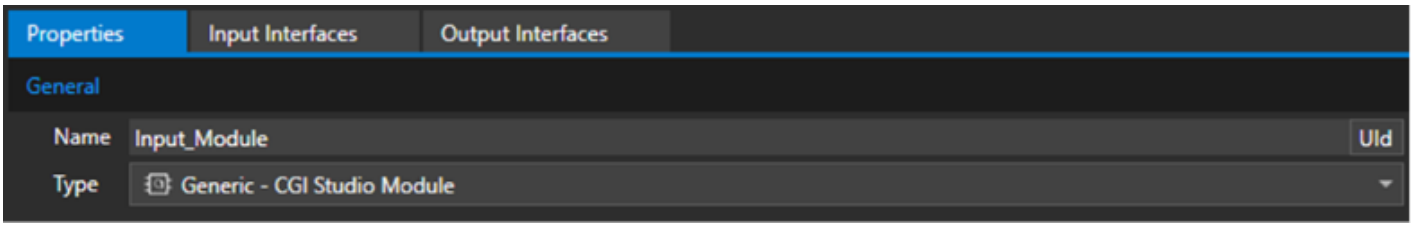
Modules are used to group a set of interfaces which usually belong logically together.

It is possible to add, remove and reorder them using the corresponding buttons at the top of the module list.



Properties

In general, these offer a few basic options. However, depending on the targeted tool, different tool-specific options might be available. For more information about tool specific options and limitation refer to the specific target tool chapter.

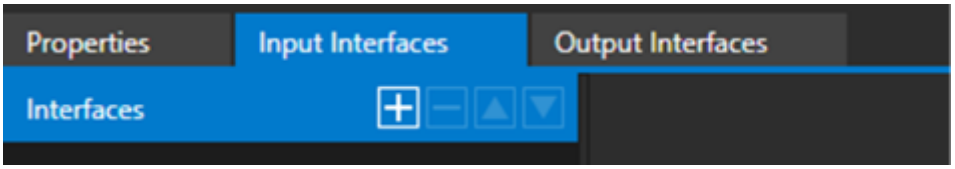


Property Name	Description
Name	Name of the module. Only characters from ‘a’-‘z’, ‘A’-Z’, ‘0’-‘9’, ‘ ‘ , ‘-‘ and ‘_’ are allowed, though the name must start with a character.
UId	The UId is used to identify the module. It can be seen as the tooltip of the ‘UId’ button. When the button is clicked a new UId will be generated.
Type	Defines the type of module. The available options depend on the specific target tool.

Interfaces

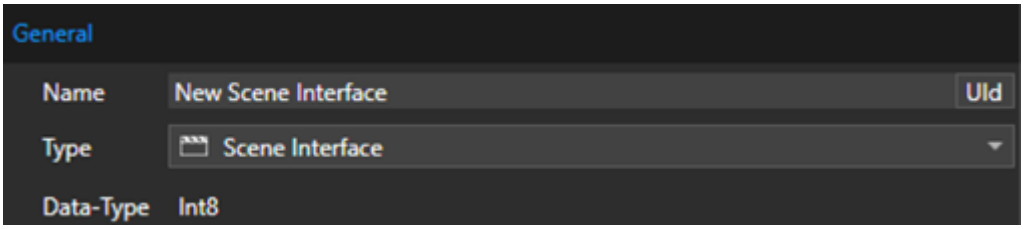
Interfaces define the inputs and outputs of the CGI Application. Depending on the module type either one or both types of interfaces are available.

By selecting the corresponding tab, interfaces can be added, removed or rearranged using the buttons at the top of the interface list.



General Properties

The following general properties are available for all interfaces.



Property Name	Description
Name	Name of the interface. Only characters from ‘a’-‘z’, ‘A’-Z’, ‘0’-‘9’, ‘ ‘, ‘-’ and ‘_’ are allowed, though the name must start with a character.
UId	The UId is used to identify the interface. It can be seen as the tooltip of the ‘UId’ button. When the button is clicked a new UId will be generated.
Type	Defines the type of interface. The available options depend on the specific target tool.
Data-Type	The used basic data type of the interface to transfer data. This value is automatically calculated depending on the used interface type and set properties. For arrays the length is indicated in brackets.

Types

The following general interface types are available.

Input:

Name
Scene
Scene-Table
Data-Input
Function Call (Action)
Message (Action)

Output:

Name
Data-Output
Generic Event
Function Call (Event)
Message (Event)

> Depending on the used module type, some interface types might not be available or special interface types might be supported. Please refer to the target tool specific section for more details.

Scene Interface

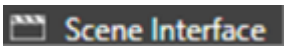
The Scene interface allows to control the state of a scene or group of scenes. Depending on the set value the scene/scenes will change to a different state. By default, the values have the following meaning:

Property Name	Description
0	Destroyed (No data loaded)
1	Loaded (Scene data is loaded to system memory)
2	Uploaded (Scene data is uploaded to VRAM)

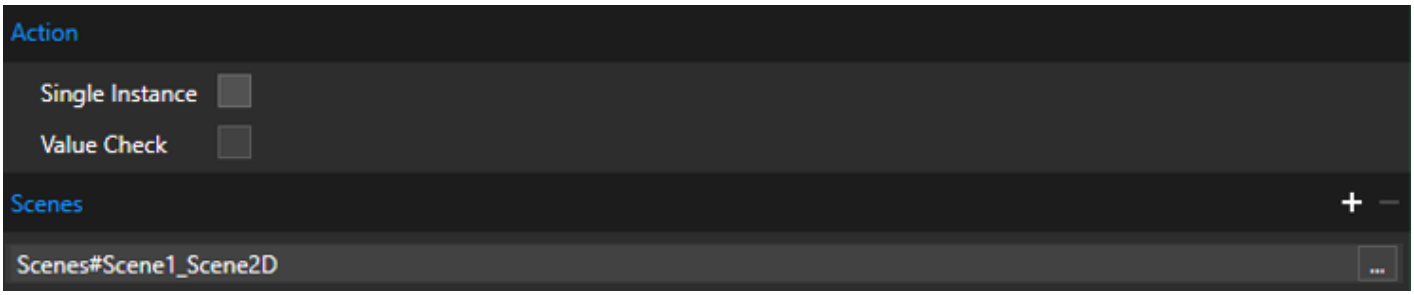
3	Active (Scene and related behaviors are updated)
4	Rendering (Scene is drawing)

> The actual meaning of each value is controlled by the application specific implementation.

Icon:



Properties:



Property Name	Description
Single Instance	When selected, only a single instance of this action will be queue. If the action is triggered again before the previous instance is executed it will overwrite the previous action in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.
Scenes	Scene controlled by the interface. A scene is defined by the following expression: [FOLDERNAME#….#]SCENENAME Clicking on the '…' button next to the field opens a dialog window which lists possible scenes from the referenced Scene Composer solution. Multiple scenes can be specified by adding or removing additional lines with the '+' and '-' buttons.

Scene Table Interface

The Scene-Table signal allows to specify a collection of scenes and assigns them a certain Id. The interface value controls this Id and therefore selects the next entry to be activated. If the new entry contains a scene which was not 'active' before, it will be 'activated'. Similar if the new entry is missing a scene which was 'active' before it will be 'deactivated'. If a scene was part of the previous set of scenes and is still part of the new set, no action will be performed.

Additionally, “Hints” can be specified which can affect how the change between different sets happens. These are primarily used to control the transition animation.

> The actual behavior is controlled by the application specific implementation.

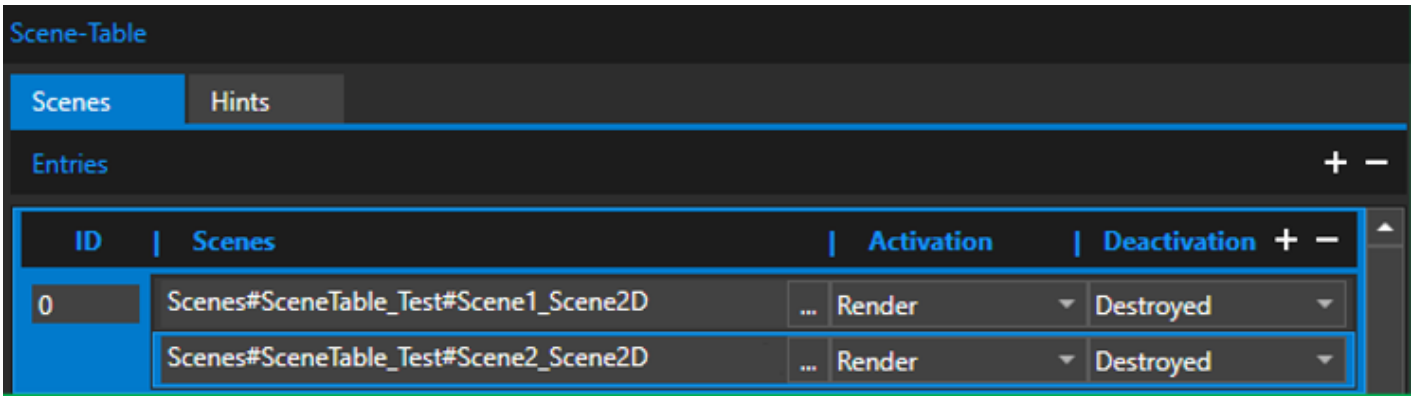
Icon:



Properties:

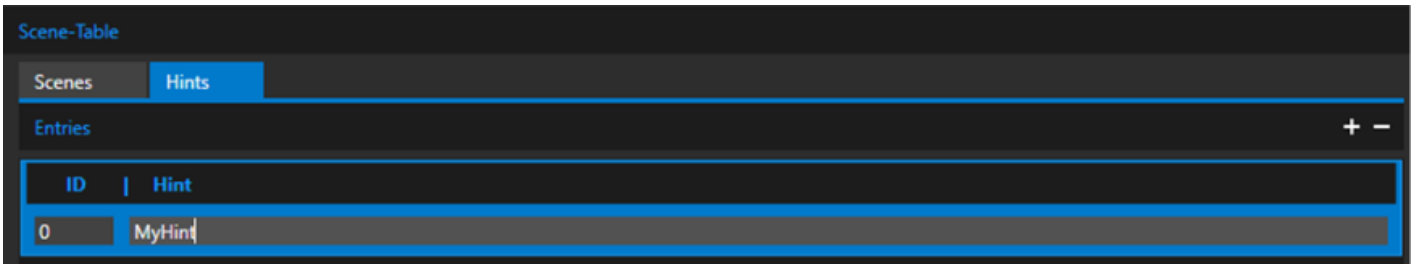


Property Name	Description
Single Instance	When selected, only a single instance of this action will be queue. If the action is triggered again before the previous instance is executed it will overwrite the previous action in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.



Property Name	Description
(Scenes) Entries	Collection of Scenes represented by an Id. Multiple collections can be specified by adding or removing additional lines with the ‘+’ and ‘-‘ buttons.
(Scenes) Entries ID	The Id representing this set of Scenes.

(Scenes) Entries Scenes	Scenes which belong to this entry. A scene is defined by the following expression: [FOLDERNAME#….#]SCENENAME Clicking on the '…' button next to the field opens a dialog window which lists possible scenes from the referenced Scene Composer solution. Multiple scenes can be specified by adding or removing additional lines with the '+' and '-' buttons.
(Scenes) Entries Activation	State of the scene to be applied when scene is activated.
(Scenes) Entries Deactivation	State of the scene to be applied when scene is deactivated.

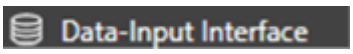


Property Name	Description
(Hints) Entries	Collection of Hints represented by an Id. Multiple collections can be specified by adding or removing additional lines with the '+' and '-' buttons.
(Hints) Entries ID	The Id representing this Hint.
(Hints) Entries Hint	Hint which belongs to this entry.

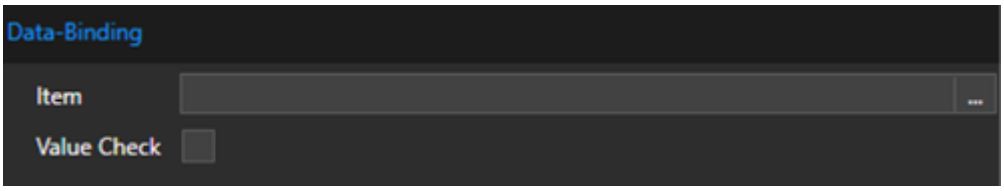
Data-Input Interface

The Data-Input interface is used to send data values from external to the CGI Studio application. Only data-items provided by Courier Data-Binding can be set as destination. For more information about Data-Binding please refer to the Courier documentation.

Icon:



Properties:



Property Name	Description

Item	Data-Item used as a destination for the data value transferred via this signal. Clicking on the '...' button next to the field opens a dialog window which lists all possible data items defined for this application.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.

Function Call (Action) Interface

The Function-Call (Action) interface allows to trigger a user defined function. To learn how to define functions to be used in the Connector refer to the Native Functions chapter.

Icon:



Properties:



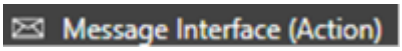
Property Name	Description
Single Instance	When selected, only a single instance of this action will be queue. If the action is triggered again before the previous instance is executed it will overwrite the previous action in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.
Name	Name of the function to execute. Clicking on the '...' button next to the field opens a dialog window which lists all possible functions defined for this application.
Input Argument	When enabled, a single argument of the function can be defined as Input-Argument. The value of the argument will then depend directly on the data sent when triggering the interface.

Arguments	List of arguments for this function (excluding the Input-Argument if specified). Here the values for the arguments that will be used to execute the function are set. Depending on the type of the argument, the [...] button will be displayed next to the field. When clicking it, it will open a dialog window which lists possible objects from the referenced Scene Composer solution that can be used as a value.

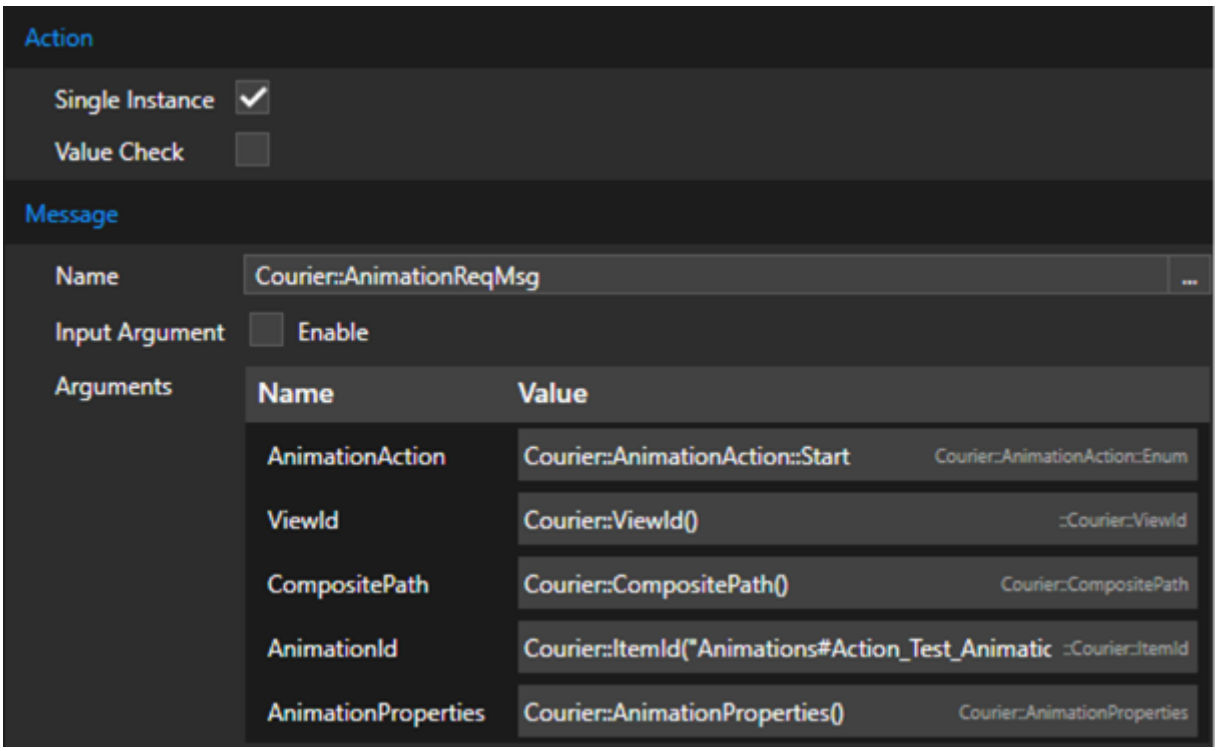
Message (Action) Interface

The Message (Action) interface allows to trigger the sending of a Courier message. For more information about Messaging in Courier please refer to the Courier documentation.

Icon:



Properties:



Property Name	Description
Single Instance	When selected, only a single instance of this action will be queue. If the action is triggered again before the previous instance is executed it will overwrite the previous action in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.

Name	Name of the message to send. Clicking on the '…' button next to the field opens a dialog window which lists all possible messages defined for this application.
Input Argument	When enabled, a single argument of the message can be defined as Input-Argument. The value of the argument will then depend directly on the data sent when triggering the interface.
Arguments	List of arguments for this message (excluding the Input-Argument if specified). Here the values for the arguments that will be used to send the message are set. Depending on the type of the argument, the […] button will be displayed next to the field. When clicking it, it will open a dialog window which lists possible objects from the referenced Scene Composer solution that can be used as a value.

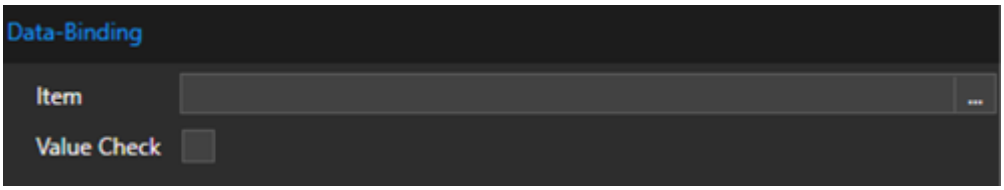
Data-Output Interface

The Data-Output interface is used to receive data values from CGI Studio application. Only data-items provided by Courier Data-Binding can be set as source. For more information about Data-Binding please refer to the Courier documentation.

Icon:



Properties:



Property Name	Description
Item	Data-Item used as a destination for the data value transferred via this signal. Clicking on the '…' button next to the field opens a dialog window which lists all possible data items defined for this application.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.

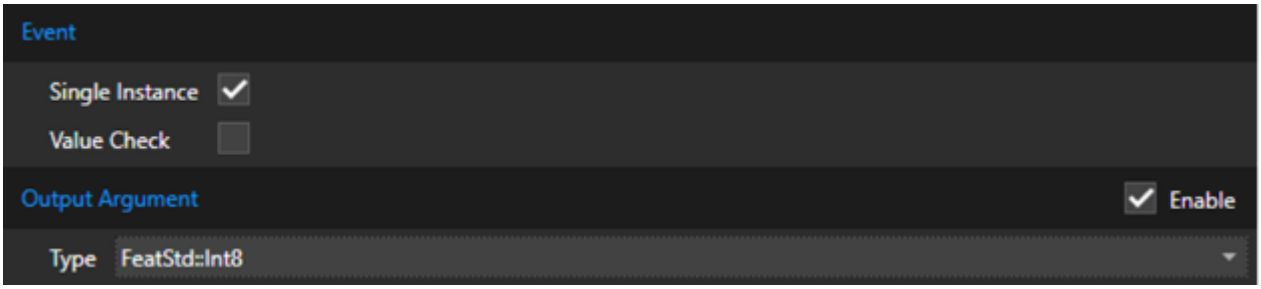
Generic Event Interface

The Generic Event interface is used to signal an event. Instead of being automatically triggered, this interface must be manually triggered from the CGI Application side. For more information on how to do this refer to the TriggerGenericEvent article.

Icon:



Properties:



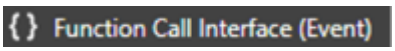
Property Name	Description
Single Instance	When selected, only a single instance of this event will be queue. If the event is triggered again before the previous instance is executed it will overwrite the previous event in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.
Output Argument	If enabled the event will be accompanied by an output value.
Type	Data type of the value.

Function Call Interface (Event)

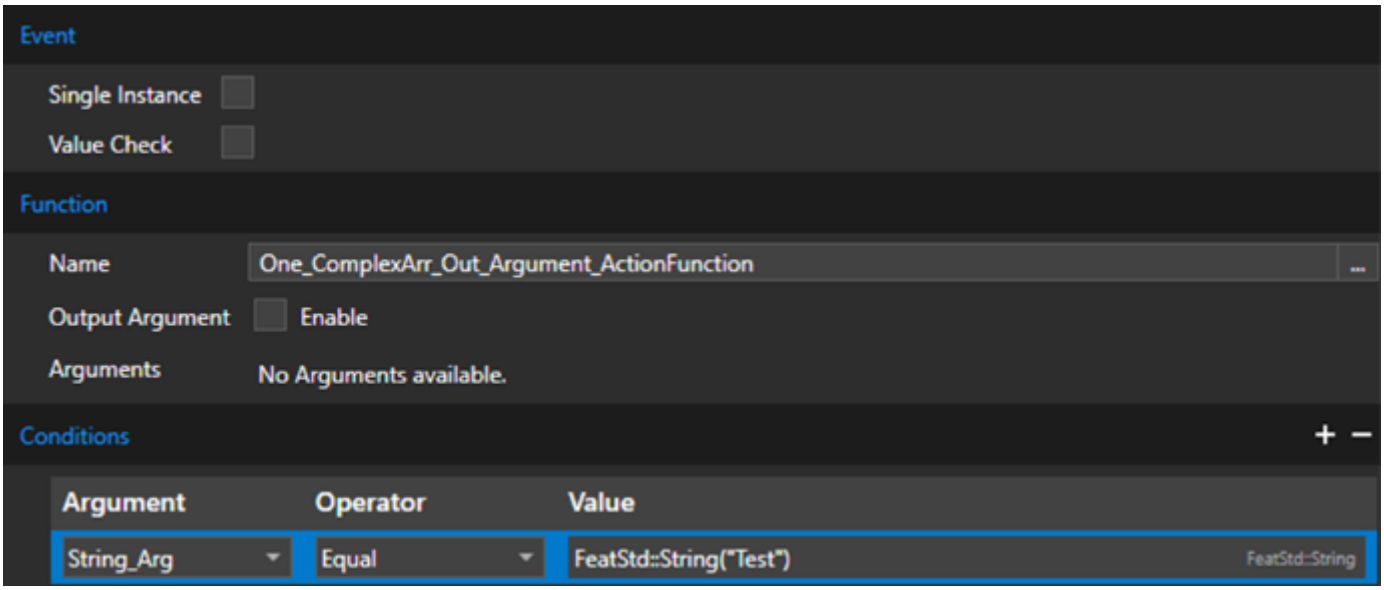
The Function-Call (Event) interface will execute a native function to check if an event has been triggered. Depending on the result of the function call (return value must be true) and specified conditions, an event will be queued.

To learn how to define functions to be used in the Connector refer to the Native Functions chapter.

Icon:



Properties:



Property Name	Description
Single Instance	When selected, only a single instance of this event will be queue. If the event is triggered again before the previous instance is executed it will overwrite the previous event in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.
Name	Name of the function which triggers the event. Clicking on the '...' button next to the field opens a dialog window which lists all possible messages defined for this application.
Output Argument	If enabled, the event will be accompanied by an output value. This value can be selected from the available members of the related message.
Conditions	Conditions allow to specify additional conditions which the function must fulfill before the event is triggered. If multiple conditions are set, all conditions must be fulfilled. Conditions can be added and removed with the '+' and '-' buttons.
Argument	Name of the argument of the function this condition is applied to
Operator	Operator used to check the condition
Value	Value of the condition. Depending on the type of the argument, the [...] button will be displayed next to the field. When clicking it, it will open a dialog window which lists possible objects from the referenced Scene Composer solution that can be used as a value.

Message Interface (Event)

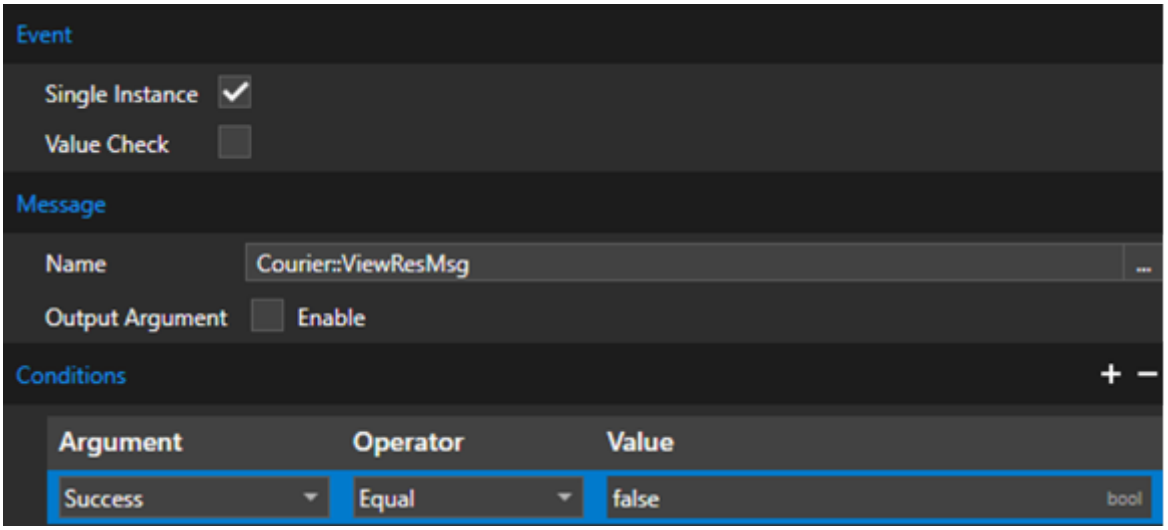
The Message (Event) interface allows to react to incoming Courier messages. If a matching message is received and all conditions are fulfilled an event will be queued.

For more information about Messaging in Courier please refer to the Courier documentation.

Icon:



Properties:

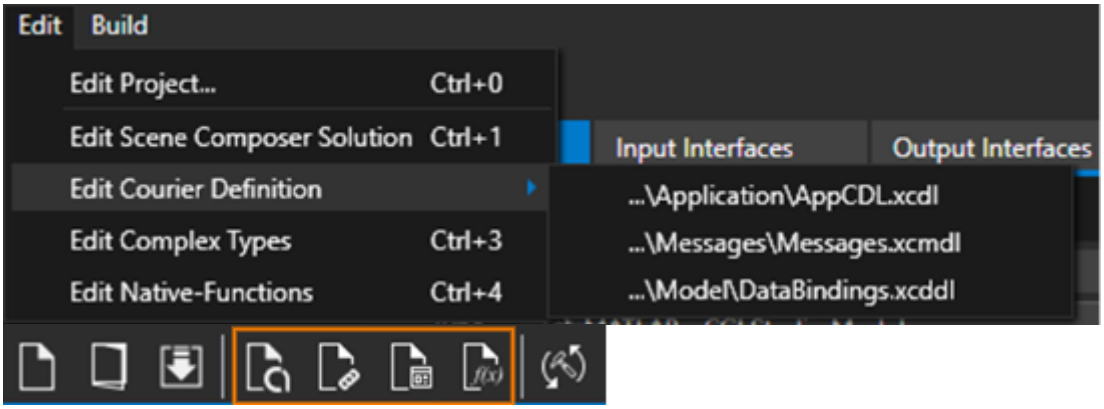


Property Name	Description
Single Instance	When selected, only a single instance of this event will be queue. If the event is triggered again before the previous instance is executed it will overwrite the previous event in the queue instead.
Value Check	If enabled, the sent value will be compared to the previous value and processing will be skipped if it has not changed. This can improve performance if the sent value is often the same/different.
Name	Name of the message which triggers the event. Clicking on the '...' button next to the field opens a dialog window which lists all possible messages defined for this application.
Output Argument	If enabled, the event will be accompanied by an output value. This value can be selected from the available members of the related message.
Conditions	Conditions allow to specify additional conditions which the message must fulfill before the event is triggered. If multiple conditions are set, all conditions must be fulfilled. Conditions can be added and removed with the '+' and '-' buttons.

Argument	Name of the member of the message this condition is applied to
Operator	Operator used to check the condition
Value	Value of the condition. Depending on the type of the argument, the […] button will be displayed next to the field. When clicking it, it will open a dialog window which lists possible objects from the referenced Scene Composer solution that can be used as a value.

Edit Menu

It is possible to quickly edit external resources with via the edit menu or the toolbar.

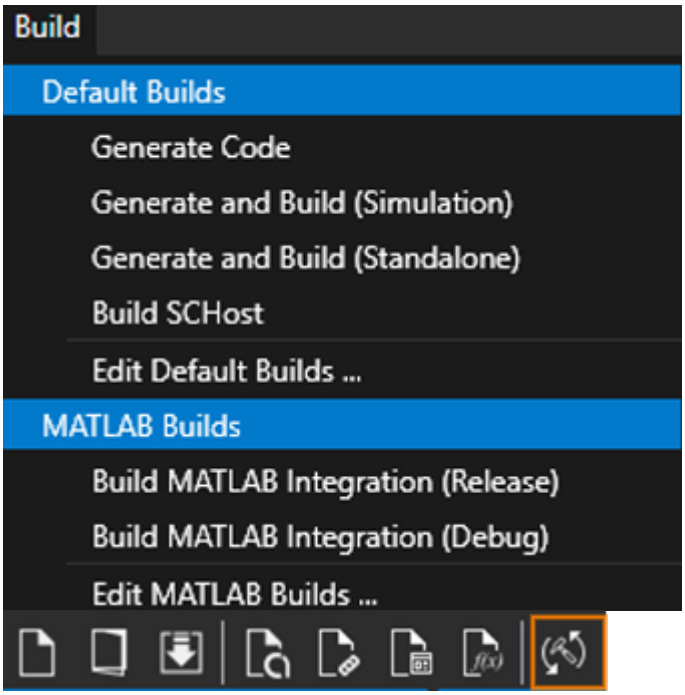


Property Name	Description
Edit Project	Opens the Project configuration window.
Edit Scene Composer Solution	Edits the configured Scene Composer solution.
Edit Courier Definition	Displays all Courier definition files based on the configured XCDL file and allows to quickly edit them using the Courier Editor.
Edit Complex Types	Edits the configured Complex-Types definition file.
Edit Native-Functions	Edits the configured Native-Functions definition file.

> If a resource or related script file is not available, the function is disabled.

Build Menu

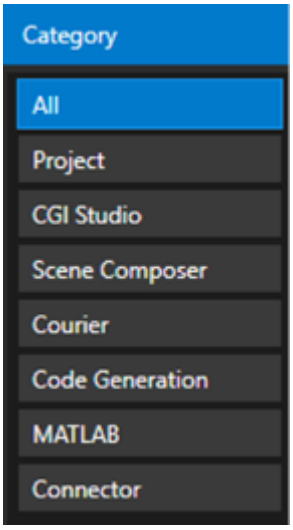
The build menu allows to quickly trigger a build from the specified Build-Projects. It lists all available default builds and, depending on the tool configuration, special tool specific builds. Same can be done via the toolbar.



Additionally, the related Build-Projects can be directly edited from the menu (not available via the Toolbar)

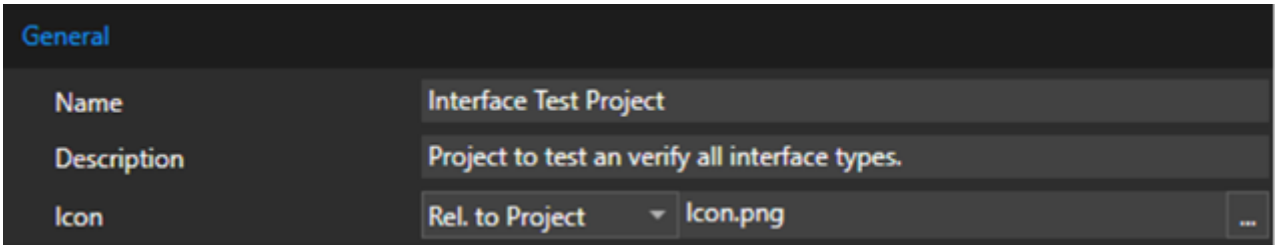
Project Settings

Several project related configurations can be performed in the project settings window which is accessible var the “Edit” menu or the shortcut “Ctrl + 0”.



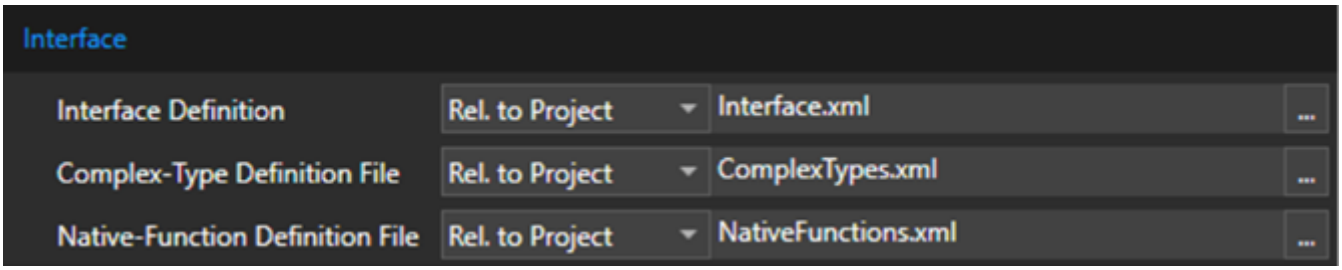
Via the List on the left the displayed options can be filtered to make it faster to find a related setting.
All project settings (with a few additions) are available in any build that is executed. For more details about this refer to Builds chapter.

General



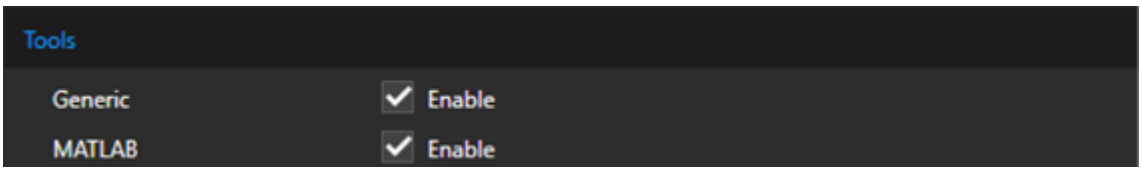
Property Name	Description
Name	Name of the project.
Description	Description of the project.
Icon	Custom icon used for the project.

Interface



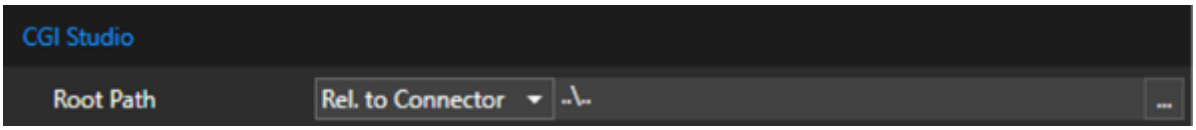
Property Name	Description
Interface Definition	File containing the Interface definition. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Complex-Type Definition	File containing the Complex-Type definition. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Native-Function Definition	File containing the Native-Function definition. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.

Tools



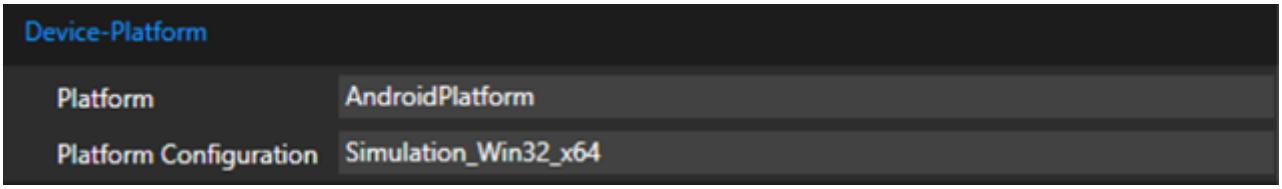
Property Name	Description
Generic	Enables generic support. This will provide a plain C/C++ interface.
MATLAB	Enables support for MATLAB. This will provide additional options/setting related to Mathworks MATLAB support

CGI Studio



Property Name	Description
Root Path	Path to CGI Studio package. The path can be either absolute or relative to CGI Connector/Project path.

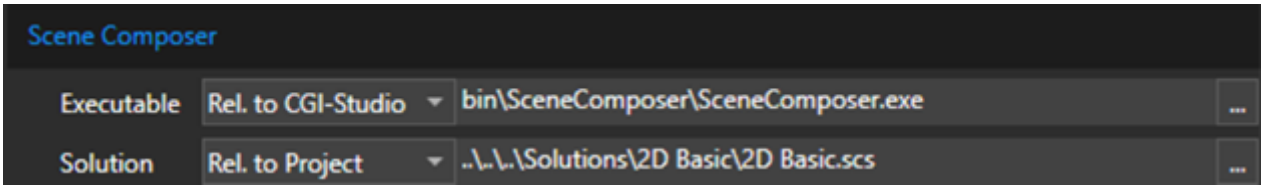
Device-Platform



Property Name	Description

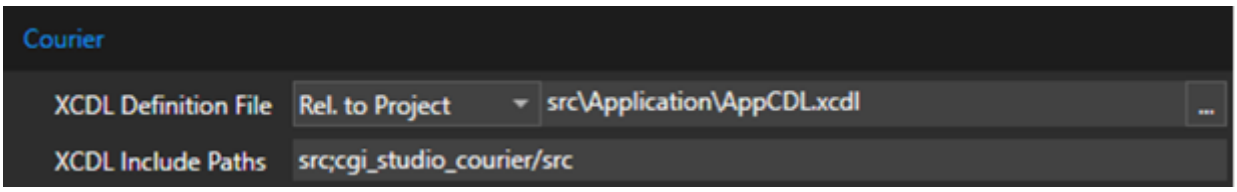
Platform	Name of the CGI Studio Platform (as found in CMake)
Platform Configuration	Name of the CGI Studio Platform configuration (as found in CMake)

Scene Composer



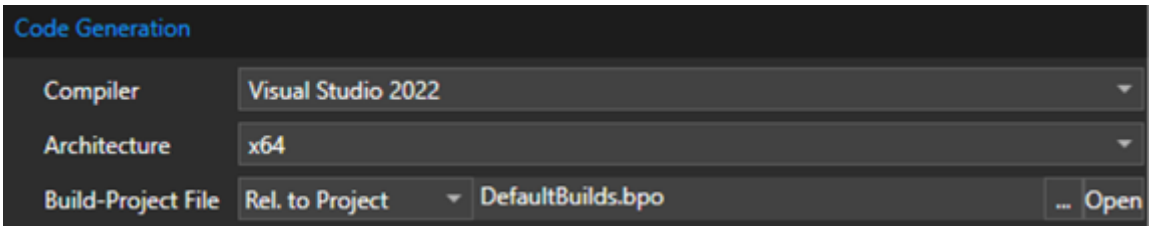
Property Name	Description
Executable	Executable used to edit the Scene Composer solution file.
Solution	Scene Composer solution file. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.

Courier



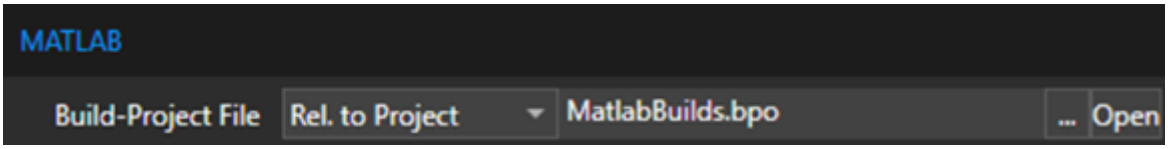
Property Name	Description
XCDL Definition File	File containing the Courier definition. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
XCDL Include Path	Additional include path required to by the XCDL definition. Multiple path must be separated by a ‘;’. Paths must be relative to the project file or CGI Studio root path.

Code Generation



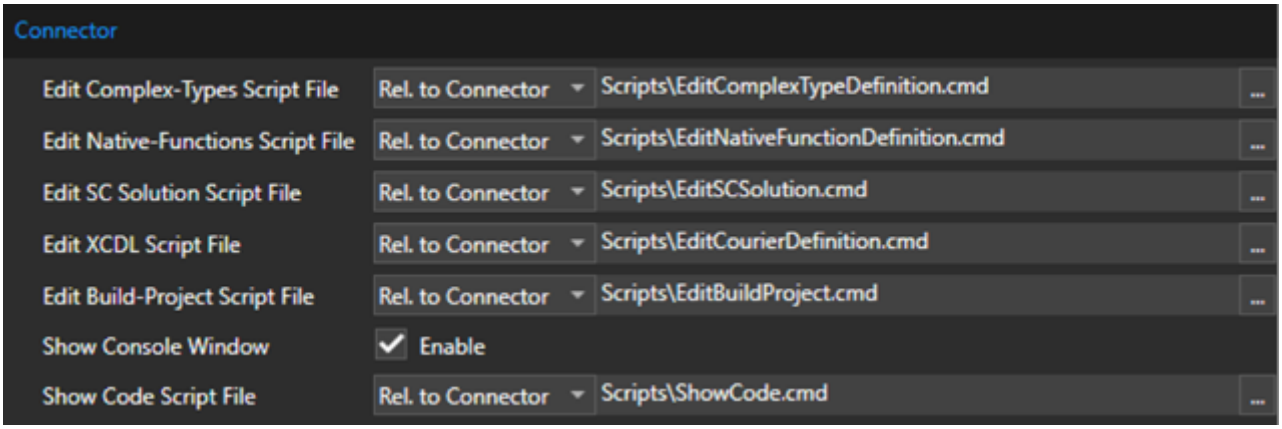
Property Name	Description
Compiler	Compiler used to build code.
Architecture	Architecture used to build the code.
Build-Project File	File containing default builds. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.

MATLAB



Property Name	Description
Build-Project File	File containing MATLAB related builds. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.

Connector

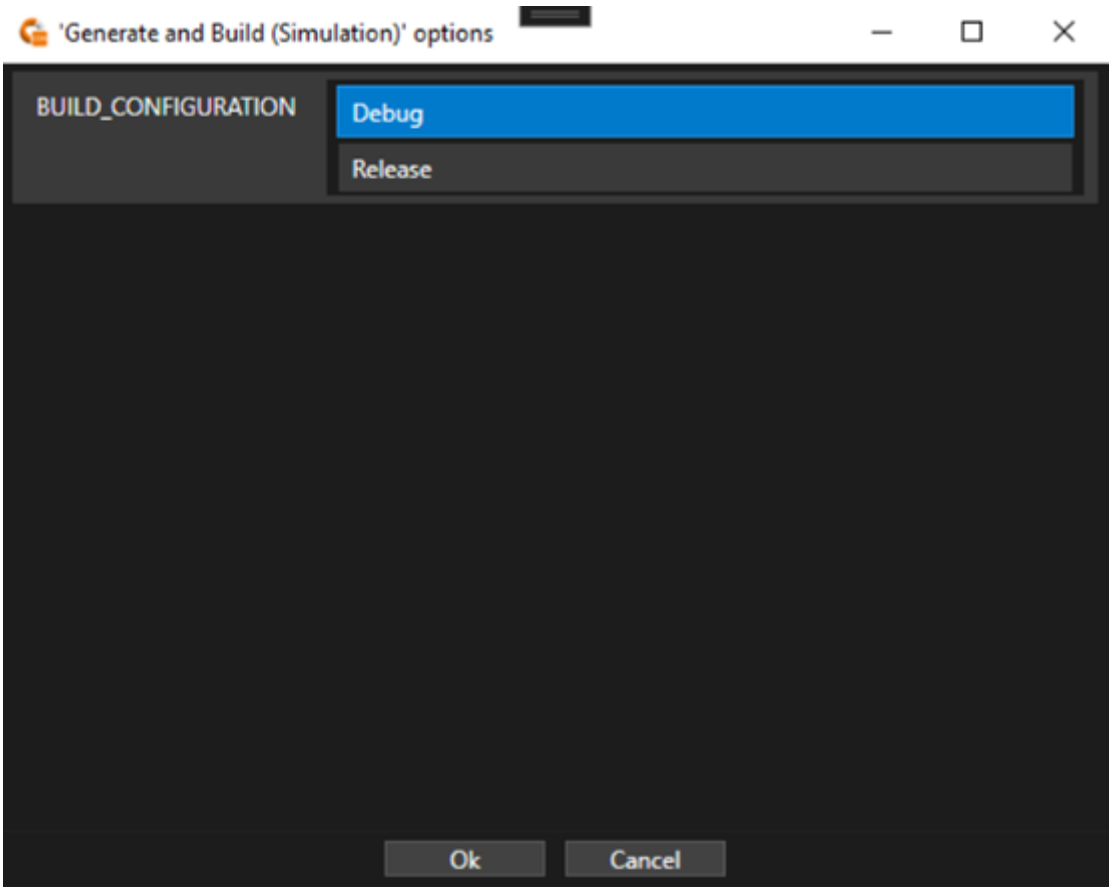


Property Name	Description
Edit Complex-Types Script File	Script file executed when editing Complex-Types. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Edit Native-Functions Script File	Script file executed when editing Native-Functions. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Edit SC Solution Script File	Script file executed when editing SC Solution. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Edit XCDL Script File	Script file executed when editing Courier definition. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Edit Build-Project Script File	Script file executed when editing a Build-Project. The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.
Show Console Window	When enabled, shows a command line window executing scripts. This can be used to debug scripts.
Show Code Script File	Script file executed when editing code (via the log window). The path can be either absolute or relative to CGI Connector/Project/CGI Studio path.

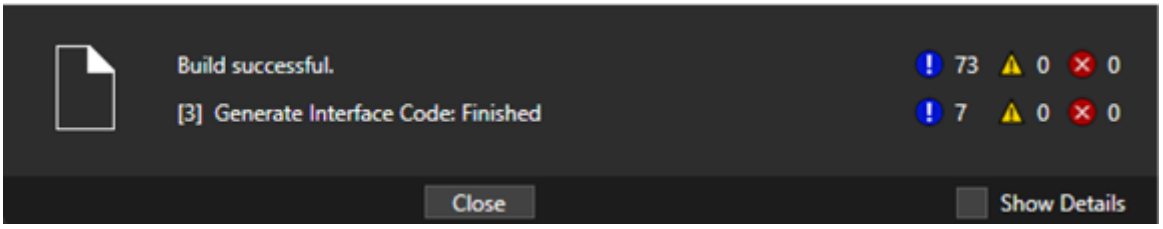
Builds

When a build is started the Build-Window will be shown and the build will be automatically started.

If the build contains some Option-Values an additional dialog will be displayed before the actual Build-Window, which allows the user to configure the build. Depending on the option, the user can either select from a set of predefined values or needs to enter a custom value.



A build is commonly separated into multiple steps each corresponding to a separate script file that is executed from the specified build-script folder.



The top row shows the current build state. It indicates the progress or if the build has finished/failed. On the right side the combined information about all build steps is shown. It displays the total amount of info, warning and error messages from the output. The second row provides information about the current executed step.

When enabling the details, by checking the option in the bottom right, the output of the build scripts can be seen, grouped by the step. This output is also saved into a log-file into the specified build folder.

Build Settings

When a build is started, the builds specific options defined in the related Build Project are set (See Build Projects for more information). Further the following settings are applied by the Connector.

Root-Paths:

Argument	Description
CONNECTOR_ROOT	Path to the Connector binary
PROJECT_ROOT	Path to the Connector project file
CGI_ROOT	Path to the CGI packages

Variables:

Argument	Description
CONNECTOR_EXECUTABLE_PATH	Path to the Connector binary
CONNECTOR_PROJECT_FILE	Connector project file
CONNECTOR_PROJECT_PATH	Path to the Connector project file
CONNECTOR_PROJECT_NAME	Connector project Name
CONNECTOR_PROJECT_DESCRIPTION	Connector project Description
CONNECTOR_PROJECT_ICON_FILE	Connector project icon file
CONNECTOR_INTERFACE_DEFINITION_FILE	Interface definition file
CONNECTOR_COMPLEXTYPE_DEFINITION_FILE	Complex-Type definition file
CONNECTOR_NATIVEFUNCTION_DEFINITION_FILE	Native-Function definition file
CONNECTOR_ENABLE_GENERIC_TOOL	Indicates if ‘Generic Tool’ is enabled
CONNECTOR_ENABLE_MATLAB_TOOL	Indicates if ‘MATLAB Tool’ is enabled
CONNECTOR_CGISTUDIO_ROOT_PATH	Path to CGI Studio root folder
CONNECTOR_CGISTUDIO_PLATFORM	CGI Studio Platform (e.g. Android, RCar,..)

CONNECTOR_CGISTUDIO_PLATFORM_CONFIG	CGI Studio Platform Configuration (x86/x64)
CONNECTOR_SCENECOMPOSER_EXECUTABLE_FILE	Scene-Composer executable file
CONNECTOR_SCENECOMPOSER_EXECUTABLE_PATH	Scene-Composer executable path
CONNECTOR_SCENECOMPOSER_SOLUTION_FILE	Scene-Composer solution file
CONNECTOR_COURIER_XCDL_DEFINITION_FILE	XCDL root definition file
CONNECTOR_COURIER_XCDL_INCLUDE_PATHS	XCDL include paths (separated by ';')
CONNECTOR_COURIER_XCDL_RESOLVED_INCLUDE_PATHS	Resolved XCDL include paths using available root paths
CONNECTOR_CODE_GENERATION_COMPILER	Compiler used for code generation/build
CONNECTOR_CODE_GENERATION_ARCHITECTURE	Architecture used for code generation/build
CONNECTOR_CODE_GENERATION_BUILD_PROJECT_FILE	Default Build-Project file
CONNECTOR_MATLAB_ENABLE_LEGACY_OPTIONS	Indicates if MATLAB legacy options are enabled
CONNECTOR_MATLAB_BUILD_PROJECT_FILE	MATLAB Build-Project file
CONNECTOR_EDIT_COMPLEX_TYPES_SCRIPT_FILE	Script for 'Edit Complex-Type definition'
CONNECTOR_EDIT_NATIVE_FUNTIONS_SCRIPT_FILE	Script for 'Edit Native-Function definition'
CONNECTOR_EDIT_SC_SOLUTION_SCRIPT_FILE	Script for 'Edit Scene Composer solution'
CONNECTOR_EDIT_XCDL_SCRIPT_FILE	Script for 'Edit Courier Definition'
CONNECTOR_EDIT_BUILD_PROJECT_SCRIPT_FILE	Script for 'Edit Build-Project'
CONNECTOR_SHOW_CONSOLE_WINDOW	Indicates if console window is shown when running scripts.
CONNECTOR_SHOW_CODE_SCRIPT_FILE	Script for triggering editor from Build details

The same environment variables are also set when any other script is executed (such as 'Edit Scene-Composer solution')

Close on Success

If a build contains the CONNECTOR_BUILD_CLOSE_ON_SUCCESS variable and sets a value of '1' then the Build-Window will be automatically closed when the build is finished successfully.

This can feature is very useful if Builds are used to just perform simple tasks or open other programs.

> Note that some programs do ‘block’ the command script execution until closed even when started using the START command. To workaround this problem the provided ‘ProcRun’ tool found in ‘cgi_studio_connector\BuildScripts\Common\ProcRun‘ can be used. The syntax to use it is:
> ProcRun <Executable> [“<Arguments>”]

Complex Types

Besides integral types, such as various integer and floating-point types, also complex types are supported. This is achieved by converting the complex type into an array of basic types. These can be static or dynamic in size.

To make this possible, the code generator must know the underlying basic-types for a specific complex type which is done by providing an optional Complex-Type definition file. Further the user must implement matching conversion functions between the specified basic and complex types.

Note: Complex-Types using mixed data types can be supported by treating them as simple byte-arrays

XML Definition

Complex Types are defined in an XML Schema file which is located at “cgi_studio_connector/schema/ComplexType.xsd”. This file can be used to validate xml files and with some editors (e.g. Visual-Studio) and provide code-completion when writing the XML file.

ComplexTypeDefinition-Element

Root Element. List of Complex types.

Attributes:
None

Children:

Argument	Occurrence	Description
ComplexType	0 - *	Complex type

ComplexType-Element

Defines a specific Complex type.

Attributes:

Argument	Occurrence	Description
native_type	Yes	Native data type (e.g. Candera::Vector3)
basic_type	Yes	Corresponding basic type. Valid values are: - Double - Single - Int8 - UInt8 - Int16 - UInt16 - Int32 - UInt32 - Int64 - UInt64 - Bool
basic_type_length	Yes	Length of basic type -1: Dynamic size >0: Fixed size

default_value	Yes	Default value of type

Children:
None

Sample

Below a sample XML file is shown

```
<?xml version="1.0"?>
<ComplexTypeDefinition xsi:noNamespaceSchemaLocation="../../tools/Core/Schema/ComplexType.xsd"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ComplexType native_type="Candera::Vector2" basic_type="Single" basic_type_length="2"
                        default_value="Candera::Vector2(0.0F, 0.0F)"/>
   <ComplexType native_type="Candera::Vector3" basic_type="Single" basic_type_length="3"
                        default_value="Candera::Vector3(0.0F, 0.0F, 0.0F)"/>
   <ComplexType native_type="FeatStd::String" basic_type="UInt8" basic_type_length="-1"
                        default_value="FeatStd::String(&amp;quot;&amp;quot;)"/>
   <ComplexType native_type="Custom::ByteArray" basic_type="UInt8" basic_type_length="-1"
                        default_value=""/>
</ComplexTypeDefinition>
```

Converter

For conversion between the basic type and actual complex type, conversion functions needs to be available. This function must be implemented inside the “ComplexType” namespace (additionally to the default namespace for the Interface) and within the “ComplexType.h/cpp” files.

For fixed size data types the conversion looks like the following:

```
void Convert(FeatStd::Float* from, Candera::Vector3* to)
{
   to->SetX(from[0]);
   to->SetY(from[1]);
   to->SetZ(from[2]);
}

void Convert(Candera::Vector3* from, FeatStd::Float* to)
{
   to[0] = from->GetX();
   to[1] = from->GetY();
   to[2] = from->GetZ();
}
```

For login/tracing purposes a “PrintValue_CUSTOM_TYPE” function is also required. This function’s purpose is to print the given value to the provided buffer. The provided value is the basic type not the final complex type (e.g. float[3] instead of Candera::Vector3). The function must return the bytes written to the buffer.

```
FeatStd::Int PrintValue_Candera_Vector3(void* buffer, FeatStd::UInt32 bufferLength, const void* value)
{
   const Candera::Float* floatValues = (const Candera::Float *)value;
   return FeatStd::Internal::String::StringPrintf((char*)buffer, bufferLength,
                  "[%.2f, %.2f, %.2f]", floatValues[0], floatValues[1], floatValues[2]);
}
```

For dynamic arrays (such as used for strings) the “DataBuffer” class will be used when the type is used in Data-Interfaces and the “Queue” class will be used if this type is used in actions or events. Additionally, a function that converts a void* with size information to the target type is required in both cases.

```
void Convert(FeatStd::String* from, Internal::DataBuffer* to)
{
   to->CopyFrom(from->GetCString(), from->GetCharCount());
}

void Convert(FeatStd::String* from, Internal::Queue* to)
{
   to->WriteSizeData(from->GetCString(), from->GetCharCount());
}

void Convert(const void* data, FeatStd::UInt32 dataSize, FeatStd::String* to)
{
   FEATSTD_UNUSED(dataSize);

   *to = FeatStd::String((FeatStd::TChar*)(data));
}

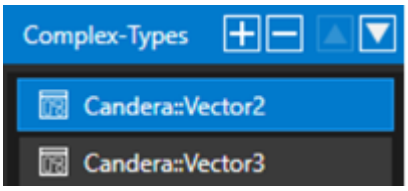
FeatStd::Int PrintValue_FeatStd_String(void* buffer, FeatStd::SizeType bufferLength,
                                       const void* value, FeatStd::UInt32 valueSize)
{
   return FeatStd::Internal::String::StringPrintf((char*)buffer, bufferLength, "%s", (const char*)value);
}
```

Editor

An editor for the Native-Function definition can be found here:
“cgi_studio_connector/bin/ComplexTypeEditor.exe”

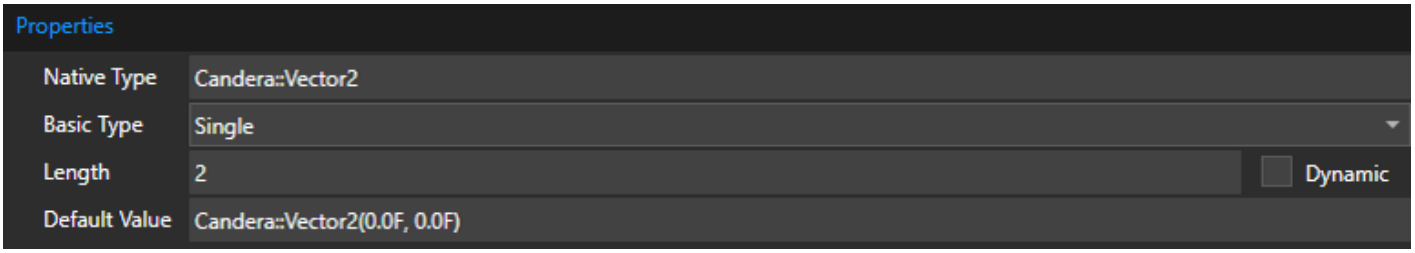
If called with a file as first argument the editor will try to open the specified file.

Complex-Types



On the left side, Complex-Types can be added/removed and reordered using the respective buttons.

Properties:



Property Name	Description
Native Type	C/C++ name of the native type
Basic Type	Basic type used to represent the Complex-Type. This type will be used when transferring data via the Interface.
Length	Length of the Complex-Type based on the Basic-Type. To use to a dynamic length (e.g. an array) the “Dynamic” checkbox can be used.
Default Value	Default value used for this Complex-Type.

Native Functions

Native Functions allow to execute custom code directly from the interface or receive customizable events. To provide the necessary information to the tool they must be specified in an XML file and finally be implemented in the Action-Context class used by interface.

XML Definition

Native Functions are defined in an XML Schema file which is located at “cgi_studio_connector/schema/NativeFunction.xsd”. This file can be used to validate xml files and with some editors (e.g. Visual-Studio) and provide code-completion when writing the XML file.

NativeFunctionDefinition-Element

Root Element. List of Function types.

Attributes:
None

Children:

Argument	Occurrence	Description
Function	0 - *	A native function definition

Function-Element

Definition of a native Function.

Attributes:

Argument	Required	Description
name	Yes	Name of the function.
uid	Yes	Unique Id of function.
has_result	Yes	Indicates if function has a result. If set, the function needs to return a bool value and can be called for checking events. In this case an event is triggered if the function returns true and all conditions are valid.
description	No	Description of the function.

Children:

Argument	Occurrence	Description
Argument	0 - *	Argument of the function

Argument-Element

Definition of an argument of the function.

Attributes:

Argument	Required	Description
name	Yes	Name of the argument.
uid	Yes	Unique Id of argument.
type	Yes	Type of the argument. This can be any basic or complex type if intended to be used for dynamic input/output. For static values any type is supported.
description	No	Description of the argument.
is_output	No	If set, this argument is passed as a reference with the intend to use it as an output value or in a condition.

Children:

None

Implementation

The actual native functions must be implemented as part of the Action-Context class which will then be passed to the Interface to trigger them.

The implementation must match the name of the specified function in the XML as well as the names, types and order of the arguments. Note that Complex-Types must be passed as const references.

```
<?xml version="1.0"?>
<NativeFunctionDefinition xsi:noNamespaceSchemaLocation="
../../tools/Core/NativeFunction/Schema/NativeFunction.xsd" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" version="1">
   <Function name="FloatTestFunction" uid="test11" has_result="false">
      <Argument name="value" type="FeatStd::Float" uid="arg11" is_output="false"/>
   </Function>
   <Function name="StringTestFunction" uid="test12" has_result="false">
      <Argument name="value" type="FeatStd::String" uid="arg11" is_output="false"/>
   </Function>
   <Function name="MixedArgumentFunction" uid="test13" has_result="true">
      <Argument name="floatValue" type="FeatStd::Float" uid="arg11" is_output="false"/>
      <Argument name="intValue" type="FeatStd::Int32" uid="arg12" is_output="false"/>
   </Function>
   <Function name="EventTestFunction" uid="test26" has_result="true">
      <Argument name="conditionArg" type="FeatStd::Int32" uid="arg11" is_output="true"/>
   </Function>
</NativeFunctionDefinition>
```

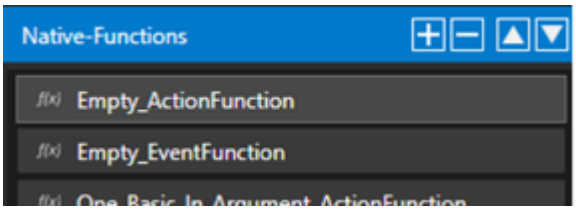
```
namespace Action {
   class ActionContext
   {
      public:
...
         // User Functions
         void FloatTestFunction(FeatStd::Float value);
         void Vector3TestFunction(const Candera::Vector3& value);
         void StringTestFunction(const FeatStd::String& value);
         void MixedArgumentTestFunction(FeatStd::Float floatValue, FeatStd::Int32 intValue);
         bool EventTestFunction(FeatStd::Int32& conditionArg);
```

Editor

An editor for the Native-Function definition can be found here:
“cgi_studio_connector/bin/NativeFunctionEditor.exe”

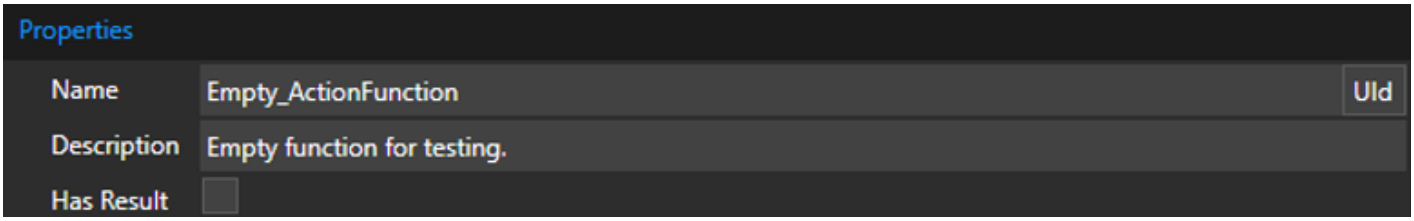
If called with a file as first argument the editor will try to open the specified file.

Native-Functions



On the left side, Native-Functions can be added/removed and reordered using the respective buttons.

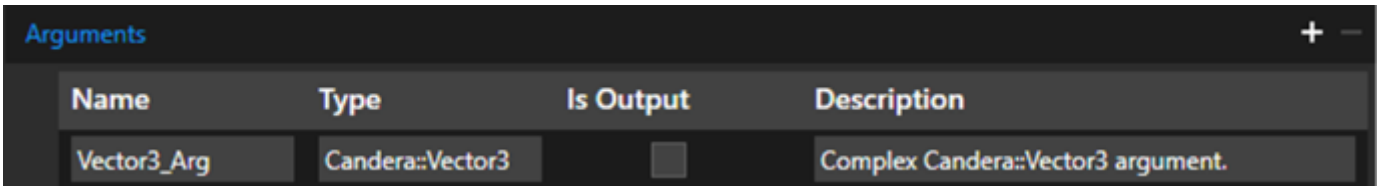
Properties:



Property Name	Description
Name	Name of the function. This name must be a valid C/C++ name and must be implemented in the Action-Context.
UId	The UId is used to identify the native function. It can be seen as the tooltip of the ‘UId’ button. When the button is clicked a new UId will be generated.
Description	Description of the function.
Has Result	If enabled, the function must return a bool value. This indicates that the function is used for Function-Call Events. The return value will indicate if the related event has been raised or not.

Arguments:

A function can have arguments. Arguments can be added/removed with the respective buttons.



Property Name	Description
Name	Name of argument. This name must be a valid C/C++ identifier.
Type	Data type of the argument. In principal all types can be used here. However if the argument is used for a dynamic input/output the type must either be a compatible basic FeatStd type or a supported Complex-Type.
Is Output	Indicates if the argument is used as an output. This can then be used to apply additional conditions when checking for an event.
Description	Description of the argument.

Build Projects

Build-Projects contain so called builds which help to automate the process of generating code and/or building the application. They allow to specify a series of script files which are to be executed and provide arguments which can be configured by the user.

XML Definition

Builds are defined in an XML Schema file which is located at “cgi_studio_connector/schema/BuildProject.xsd”. This file can be used to validate xml files and with some editors (e.g. Visual-Studio) and provide code-completion when writing the XML file.

BuildProject-Element

Root Element. Contains list of Root-Paths, global Variables and Builds.

Attributes:

Argument	Required	Description
version	Yes	Version of the Build-Project file.

Children:

Argument	Occurrence	Description
RootPaths	1	List of Root-Paths
Variables	1	List of global Variables. These Variables can be used in any Build.
Builds	1	List of Builds

RootPaths-Element

Defines a list of Root-Paths.

Attributes:
None

Children:

Argument	Occurrence	Description
RootPath	0 - *	Root-Path

Variables-Element

Defines a list of Variables.

Attributes:
None

Children:

Argument	Occurrence	Description
Variable	0 - *	Variable

Builds-Element

Defines a list of Builds.

Attributes:
None

Children:

Argument	Occurrence	Description
Build	0 - *	Build

RootPath-Element

Defines a Root-Path. Root-Paths can be referenced by variables to make them relative to a common path.

Attributes:

Argument	Required	Description
name	Yes	Name of the Root-Path.
description	No	Description of the argument.

Children:
None

Variable-Element

A variable for the build which will be applied for all executed scripts.

Attributes:

Argument	Required	Description

name	Yes	Name of the variable.
type	Yes	Type of the variable. Valid values are ‘Value’, ‘Option’, ‘Path’ and ‘File’
root_path_key	No	If set, paths (or files) will be relative to the given Root-Path.
value	Yes	Value of the variable. If the type is ‘Option’, valid values can be specified as a list of values separated by semicolons. If left blank, any value is considered valid.
description	Yes	Description of the variable.

Children:
None

Build-Element

A build which can be executed.

Attributes:

Argument	Required	Description
name	Yes	Name of the build.
description	Yes	Description of the build.

Children:

Argument	Occurrence	Description
ScriptPath	1	Path to the script files which should be executed for this build.
OutputPath	1	Output-Path for this build.
Variable	0 - *	Variable for this build

ScriptPath-Element

Path to the script files which are executed when this build is started. The script files should start with a number to indicate their execution order.

Attributes:

Argument	Required	Description
value	Yes	Relative path.
root_path	No	If set, path will be relative to the given Root-Path.

Children:
None

OutputPath-Element

Main output path for this build. Even if the build doesn't produce any artifacts or generates output into other directories, at least the log file will be saved here.

Attributes:

Argument	Required	Description
value	Yes	Relative path.
root_path	No	If set, path will be relative to the given Root-Path.

Children:
None

Sample

Below a sample XML file can be seen

```
<?xml version="1.0"?>
<BuildProject xsi:noNamespaceSchemaLocation="../../tools/ConnectorApp/Build/Schema/BuildProject.xsd"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    version="1">
   <RootPaths>
      <RootPath name="CONNECTOR_ROOT" description="Path to CGI Connector executable." />
      <RootPath name="PROJECT_ROOT" description="Path to CGI Connector project file." />
      <RootPath name="CGI_ROOT" description="Path to CGI root folder" />
   </RootPaths>
   <Variables>
      <Variable name="INTERFACE_GENERATOR_OUTPUT_PATH" type="Path" value="src/Interface" root_path="PROJECT_ROOT" description="
Path for generated output files." />
      <Variable name="INTERFACE_NAMESPACE" type="Value" value="Application::Interface" description="Namespace for generated Interface."
/>
      <Variable name="INTERFACE_DATAMODEL_INCLUDE" type="Value" value="Model/!DATAMODEL_CLASSNAME!.h" description="Include for
Data-Model." />

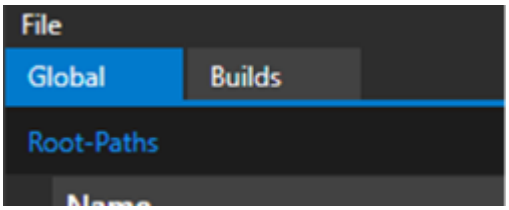
      <Variable name="BUILD_CGISTUDIO_PLATFORM" type="Value" value="!CONNECTOR_CGISTUDIO_PLATFORM!" description="CGI-Studio
Platform/Device." />
      <Variable name="BUILD_CGISTUDIO_PLATFORM_CONFIG" type="Value" value="!CONNECTOR_CGISTUDIO_PLATFORM_CONFIG!" description
="CGI-Studio Platform/Device configuration." />
   </Variables>
   <Builds>
      <Build name="Generate Code" description="Generates only code">
         <ScriptPath value="../BuildScripts/GenerateCode" root_path="CONNECTOR_ROOT" />
         <OutputPath value="" root_path="PROJECT_ROOT"/>
      </Build>
      <Build name="Generate and Build (Simulation)" description="Generates code and builds the application.">
         <ScriptPath value="../BuildScripts/BuildApplication" root_path="CONNECTOR_ROOT" />
         <OutputPath value="build" root_path="PROJECT_ROOT"/>
         <Variable name="BUILD_TARGET" type="Value" value="Application" description="Build target" />
         <Variable name="BUILD_CONFIGURATION" type="Option" value="Release;Debug" description="Build configuration/>
         <Variable name="BUILD_CMAKE_CUSTOM_FLAGS" type="Value" value="-DCOURIER_ADD_SCHOST_PROJECT=1 -
DAPPLICATION_IPC_ENABLED -DAPPLICATION_IPC_INTERFACE_ENABLED -DAPPLICATION_CONNECTOR_SIMULATION=1" description="Additional
CMake flags." />
      </Build>
   </Builds>
</BuildProject>
```

Editor

An editor for the Build-Projects definition can be found here:
“cgi_studio_connector/bin/BuildProjectEditor.exe”

If called with a file as first argument the editor will try to open the specified file.

Global



On this tab, global settings can be configured.

Root-Paths
Root-Paths allow the user to make path/file variables relative to a specified path. When a build is started these Root-paths must be set by the user beforehand.

They can be added and removed via the respective buttons on the right of the group.



Properties:

Property Name	Description
Name	Name of the Root-Path
Description	Description of the Root-Path.

Variables

Here, global variables can be set which apply to every build. However, each build can override the value of a variable simply by respecifying it directly in the build.

They can be added and removed via the respective buttons on the right of the group.



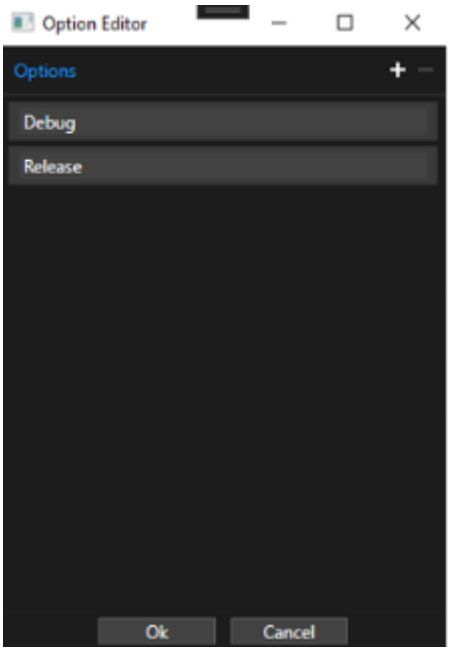
Propertise:

Property Name	Description
Name	Name of the Variable. This is the same name that is to be used in the scripts.
Description	Description of the Variable.
Type	Type of the Variable. The following options are available: - Value: Simple value type - Option: Value that must be set at run time. - Path: A path (folder) - File: Full file path
Root-Path	If the variable type is Path or File, this is used to reference a specified Root-Path. Doing so will treat the path as a relative path.
Value	Value of the variable. In case of 'Option' type values, the valid choices can be restricted by defining multiple values separated with a semicolon.

If the type is set to 'Option' the 'Option Editor' can be opened by clicking on the '...' button next to the Value field.



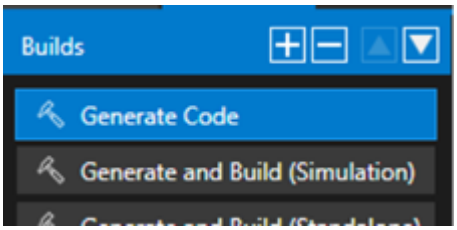
In the Option Editor window individual options can be easily added and removed using the 'plus' and 'minus' buttons on the right side.



Builds

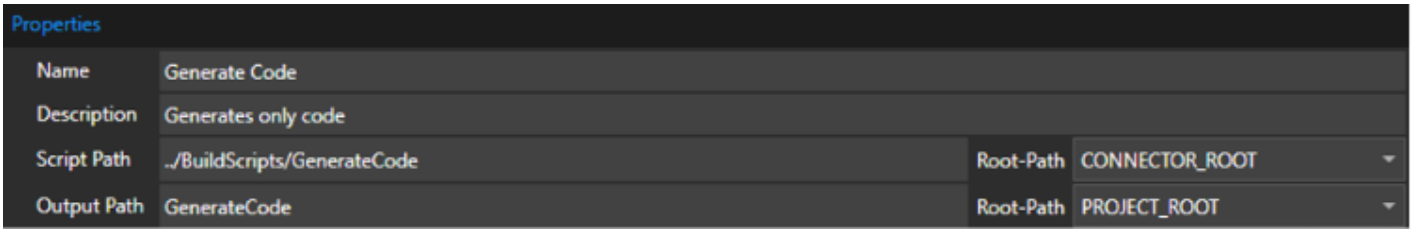
On this tab, individual builds can be configured.

On the left side, Native-Functions can be added/removed and reordered using the respective buttons.



Properties

Builds contain several general properties.



Property Name	Description
Name	Name of the build.
Description	Description of the build.
Script Path	Path to the script file which should be executed for this build. All scripts (.bat/.cmd) inside the specified folder are executed in alphabetical order so it is recommended to begin the filename with the number of the execution step.

Root-Path (Script Path)	Root-Path of the Script Path.
Output Path	Main output path for this build. Even if the build doesn't produce any artifacts or generates output in other directories, at least the log file will be saved here.
Root-Path (Output Path)	Root-Path of the Output Path.

Variables

Variables for this build can be set here.

They can be added and removed via the respective buttons on the right of the group.



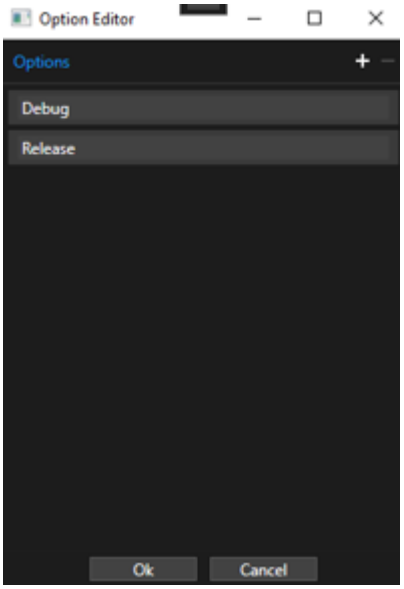
Properties:

Property Name	Description
Name	Name of the Variable. This is the same name that is to be used in the scripts.
Description	Description of the Variable.
Type	Type of the Variable. The following options are available: - Value: Simple value type - Option: Value that must be set at run time. - Path: A path (folder) - File: Full file path
Root-Path	If the variable type is Path or File, this is used to reference a specified Root-Path. Doing so will treat the path as a relative path.
Value	Value of the variable. In case of 'Option' type values, the valid choices can be restricted by defining multiple values separated with a semicolon.

If the type is set to 'Option' the 'Option Editor' can be opened by clicking on the '...' button next to the Value field.



In the Option Editor window individual options can be easily added and removed using the ‘plus’ and ‘minus’ buttons on the right side.



Default variables

Besides user configured variables the following default environment variables will be set.

Name	Description
BUILD_NAME	Name of the build
BUILD_DESCRIPTION	Description of the build
BUILD_SCRIPT_PATH	Path to build script files
BUILD_OUTPUT_PATH	Output path of the build

Command Line Interface

It is possible to start a build from the Command Line via the Build-CLI tool. For more information, please refer to the “Tools” chapter.

Interface

The main purpose of the CGI Studio Connector is to create an interface that allows to easily connect an external system, such as 3rd party Statemachines, custom middleware etc, to a CGI Studio Courier based application. Via this interface it is possible to share data as well as trigger actions and receive events.
To achieve this, multiple classes are generated by the tool which are to be integrated into the CGI-Studio application. These are further supported by multiple behavioral classes which can be implemented/modified by the user to define the exact behavior of the interface.

Interface-Buffer

The main class of the generated interface is called “Interface-Buffer”. This class is responsible to handle the data, actions and events from both sides and provide them to the other side.

The Interface-Buffer class contains the following methods:

Init

Initializes the Interface-Buffer.

Argument Name	Type	Description
dataModel	DataModel	The generated Data-Model. Used to initialize data-values of the Interface-Buffer
defaultActionQueueSize	FeatStd::UInt32	Size of the Action-Queue in bytes. The action queue is used to store actions for later execution. By pre-initializing it with a proper size, costly runtime memory allocations can be avoided.
defaultEventQueueSize	FeatStd::UInt32	Size of the Event-Queue in bytes. The event queue is used to store event for later handling. By pre-initializing it with a proper size, costly runtime memory allocations can be avoided.

SetDataValue

Sets data values of the Interface-Buffer. This method is intended to be called by the external system to send data to the CGI application.

Argument Name	Type	Description
interfaceId	InputInterfaceId::Enum	Id of the interface to set the value.
data	void*	Pointer to the data to be set.
dataSize	FeatStd::UInt32	Size of the data to be shared in bytes. This value will be ignored for static sized data unless INTERFACE_DEBUG_SIZECHECK_ENABLED is defined.

Result: True if setting the data-value is successful. False otherwise.

GetDataValue

Gets data values of the Interface-Buffer. This method is intended to be called by the external system to send data to the CGI application.

Argument Name	Type	Description
interfaceId	OutputInterfaceId::Enum	Id of the interface to get the value.
target	void*	Pointer where the value should be written.
maxTargetSize	FeatStd::UInt32	Maximum size of in bytes that the 'target' can store. This value will be ignored for static sized data unless INTERFACE_DEBUG_SIZECHECK_ENABLED is defined.
dataSize	FeatStd::UInt32*	Actual size of the data to be written.

Result: True if getting the data-value is successful. False otherwise. If the return value is false and the dataSize > maxTargetSize the cause for the failure was that not enough memory was available to store the output value.

TriggerAction

Triggers the specified action.

Argument Name	Type	Description
interfaceId	InputInterfaceId::Enum	Id of the action to trigger.
data	void*	Pointer to data needed for the action.
dataSize	FeatStd::UInt32	Actual size of the data to be written. This value will be ignored for static sized data unless INTERFACE_DEBUG_SIZECHECK_ENABLED is defined.

Result: True if getting action was successfully triggered. False otherwise.

SwapActionQueue

Swaps the action queue.

ProcessEvents

Gets events of the Interface-Buffer. This method is intended to be called by the external system to send data to the CGI application.

Argument Name	Type	Description
eventHandler	Internal::EventHandler*	Pointer to the Event-Handler object. This will be called for every event in the queue.
maxCount	FeatStd::Int32	Maximum count of events to handle. Use '-1' for unlimited amount.

The Event-Callback function must have the following arguments:

Argument Name	Type	Description
interfaceId	FeatStd::UInt32	Id of the event
data	void*	Pointer to the event data if available.
dataSize	FeatStd::UInt32	Size of the event data if available.

ProcessInputDataValues

Synchronizes the Interface-Buffer input data values with the applications Data-Model. This method is intended to be called by the CGI application.

Argument Name	Type	Description
dataModel	DataModel*	The generated Data-Model.

ProcessOutputDataValues

Synchronizes the Interface-Buffer output data values with the applications Data-Model. This method is intended to be called by the CGI application.

Argument Name	Type	Description
dataModel	DataModel*	The generated Data-Model.

ProcessActions

Executes all queued actions. This method is intended to be called by the CGI application.

Argument Name	Type	Description
actionContext	Action::ActionContext	Action-Context used to execute the action functions and basic actions (e.g. Message-Actions)

sceneContext	Scene::SceneContext	Scene-Context used to execute the scene related actions (e.g. Scene-Interface or Scene-Table Interface)
maxCount	FeatStd::Int32	Maximum count of actions to handle. Use '-1' for unlimited amount.

TriggerGenericEvent

Raises a generic event. This method is intended to be called by the CGI application.

Argument Name	Type	Description
interfaceId	OutputInterfaceId::Enum	Id of the event to raise.
data	void*	Pointer to the data of the event if available.
dataSize	FeatStd::UInt32	Size of the event data if available.

Result: True if an event was raised (considering additional conditions). False otherwise.

TriggerMessageEvent

Raises a message event. This method is intended to be called by the CGI application.

Argument Name	Type	Description
message	Courier::Message*	Message to check if an event is to be raised.

Result: True if an event was raised (considering additional conditions). False otherwise.

TriggerFunctionCallEvents

Raises Function-Call events. This method is intended to be called by the CGI application.

Argument Name	Type	Description
actionContext	Action::ActionContext*	Action-Context used to call related Event-Functions.

Result: True if an event was raised (considering additional conditions). False otherwise.

SwapEventQueue

Swaps the event queue.

Interface

Action-Context

The action context is used whenever any actions (except scene related actions) are executed. Besides some basic methods it must also implement all native-functions that the user wants to call via the interface.

Required Methods

There are some required methods that need to be implemented in the Action-Context by default.

ProcessMessage

Handles messages that are meant to be sent to the CGI application from the Interface-Buffer.

Argument Name	Type	Description
message	Courier::Message*	Messages to be sent by the method.

Interface

Scene-Context

The Scene-Context is used whenever scene related actions are executed. Here the user can provide his own implementation on how these actions are to be performed. Besides the Scene-Context itself there are also secondary classes (e.g. Scene/SceneTable) what the user can adapt to fit certain requirements.

Required Methods

There are some required methods that need to be implemented in the Scene-Context by default.

SetSceneState

Called when a Scene action is triggered, and the state of a scene is to be changed.

Argument Name	Type	Description
scene	Scene&	Scene to be manipulated
state	FeatStd::Int8	State of the scene to set.

SetSceneTableState

Called when a Scene-Table action is triggered, and the state of a Scene-Table is to be changed.

Argument Name	Type	Description
sceneTable	SceneTable&	Scene-Table to be manipulated
entryId	FeatStd::Int32	Id of the Scene-Table entry to be activated.
hint	const char*	Optional hint string.

Data-Model

The basic Data-Model used by the Interface-Buffer is completely generated. However, the user can still override the handling of update requests if needed.

Tools

There are multiple tools include in the CGI Connector which also can be used directly without interfacing with the main Connector UI. These tools will be explained here.

Interface-Code Generator

This tool is located at “cgi_studio_connector/bin/InterfaceCodeGenerator.exe” and is the main tool to generate the interface code. It has the following arguments:

It has the following arguments:

Full Name	Short name	Optional	Description
-definitionFile	-d	No	File containing interface definition
-outputPath	-o	Yes	Path for generated output files
-dataModelClass	-dmc	Yes	Fully qualified name of Data-Model
-dataModelInclude	-dmi	Yes	Include path for Data-Model
-namespace	-n	Yes	Namespace of generated code
-xcdlNamespace	-xn	Yes	Namespace of code generated by XCDL files
-complexTypes	-ct	Yes	Path of Complex-Type definition file

Model-Code Generator

This tool is located at “cgi_studio_connector/bin/ModelCodeGenerator.exe” and generates a basic Data-Model based on the Courier-XCDL definition. This model is generated in such a way that the Interface-Buffer can easily interact with it.

It has the following arguments:

Full Name	Short name	Optional	Description
-xcdlDefinitionFile	-xd	No	File containing XCDL definition
-xcdlIncludePaths	-xi	No	XCDL include paths separated by ';'
-className	-c	No	Output class and filename of generated files
-namespace	-n	Yes	Namespace of generated code
-outputPath	-o	Yes	Path for generated output files
-includeFiles	-i	No	Include files separated by ';'. Must at least include root XCDL file.

Build Command Line Interface

This tool is located at “cgi_studio_connector/bin/BuildCLI.exe” and allows the user to execute a build via the command line. The main purpose of this is to automate builds.

It has the following arguments:

Full Name	Short name	Optional	Description
-connectorProject	-cp	No	Connector Project file
-buildProject	-bp	No	Build Project file
-buildName	-bn	No	Name of the Build to run
-buildOptions	-bo	Yes	Options for this build provided as 'NAME=VALUE' pairs separated by a semicolon

Tool Support

Overview

The CGI Connector offers support for multiple ‘tools’ for which it can generate specific code to make integration easier.

Currently the following tools are supported:

Name	Description
Generic	Standard generic C/C++ interface for integration into other C/C++ based software.
MATLAB	Integration for MathWorks MATLAB

Generic

The ‘Generic’ tool provides a plain C/C++ based interface which can be used to integrate the CGI application into any C/C++ based software.

This generic interface is also the basis for the integration of other tools and will therefore always be present.

Special Features

There are no tool specific features.

Limitations

There are no limitations when using the ‘Generic’ tool.

Generators

This tool does not use any special code generator tools.

MATLAB

The ‘MATLAB’ tool offers special features to make integration into MathWorks MATLAB easier.

Requirements

The following requirements must be fulfilled to be able to use the CGI Connector with MATLAB:

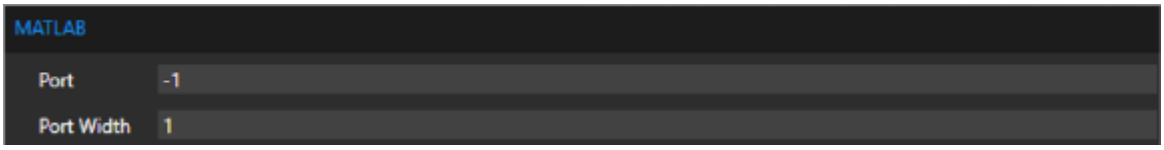
MATLAB Version:
The current version is tested to run with MATLAB 2021b. However newer versions are also expected to be compatible.

MEX Compiler:
It is necessary to configure the compiler that is used by MATLAB to build a MEX File. This can be achieved by calling “mex -setup" and “mex -setup c++” from within MATLAB. The configured compiler must match with the compiler used to build the CGI Studio application.

The “mex” command must also be executable from the command line. This can be done by adding the correct path in the systems environment variables. Alternatively, the build scripts can be modified to point directly to the mex executable.

Special Features

Interfaces defined in a MATLAB module have the following tool specific properties.



Property Name	Description
Port	Defines the port index of the interface for the generated MATLAB S-Function block. When specifying ‘-1’ the port index will be calculated automatically during code generation.
Port Width	Specifies the MATLAB port width. This option is only visible if the interface uses a dynamic length datatype.

Limitations

Currently the following limitation apply:

- Interfaces that use dynamic sized data need to specify a fixed sized for MATLAB side. See "Special Features" for more details.

Generators

This tool is located at "cgi_studio_connector/bin/MatlabCodeGenerator.exe" and generates files to be used in MathWorks MATLAB.

It has the following arguments:

Full Name	Short Name	Optional	Description
-definitionFile	-d	No	File containing interface definition
-complexTypes	-ct	Yes	Path of Complex-Type definition file
-outputPath	-o	Yes	Path for generated output files

Usage

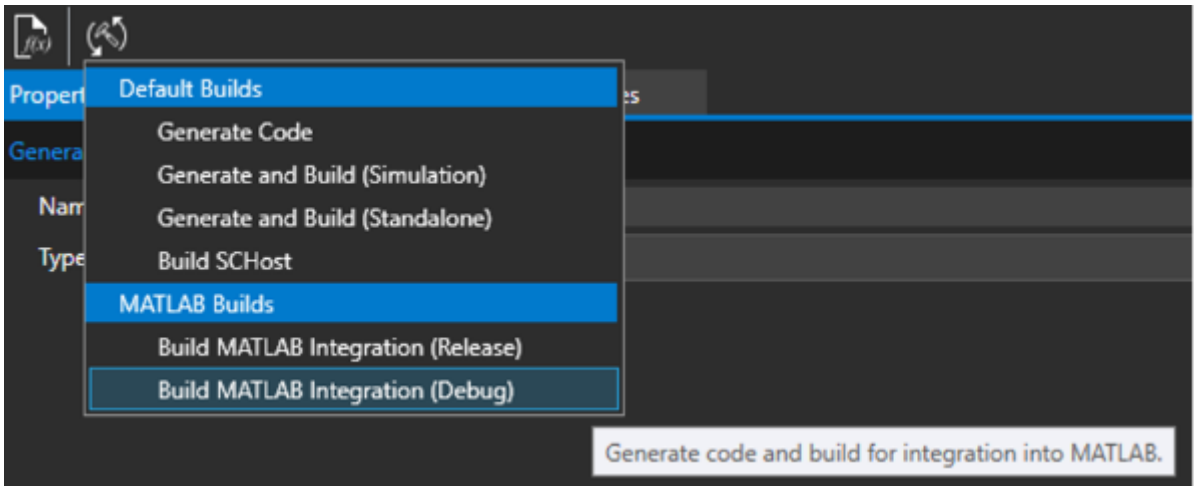
To use the MATLAB integration the following steps are required.

1. **Build CGI Application**
   As the integration into MATLAB simulation uses TCP/IP to send data between MATLAB and the CGI Application, this feature needs to be enabled. To do this the 'APPLICATION_IPC_ENABLED' and 'APPLICATION_IPC_INTERFACE_ENABLED' CMake flags must be set. This is already done when building the application using the default builds from within the Connector.

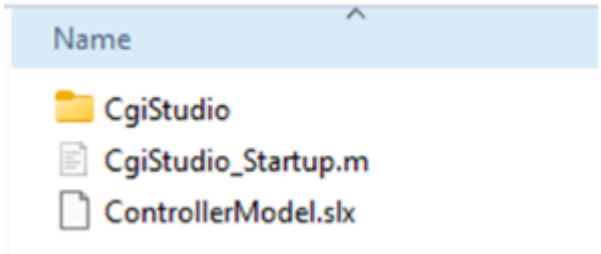
   When running the generated MATLAB code after integration into the application code the IPC communication is however not required anymore.

2. **Build Integration Modules**
   After the application is built, the 'MATLAB Integration' can be built. The integration must be built matching the Release/Debug configuration of the CGI Application.

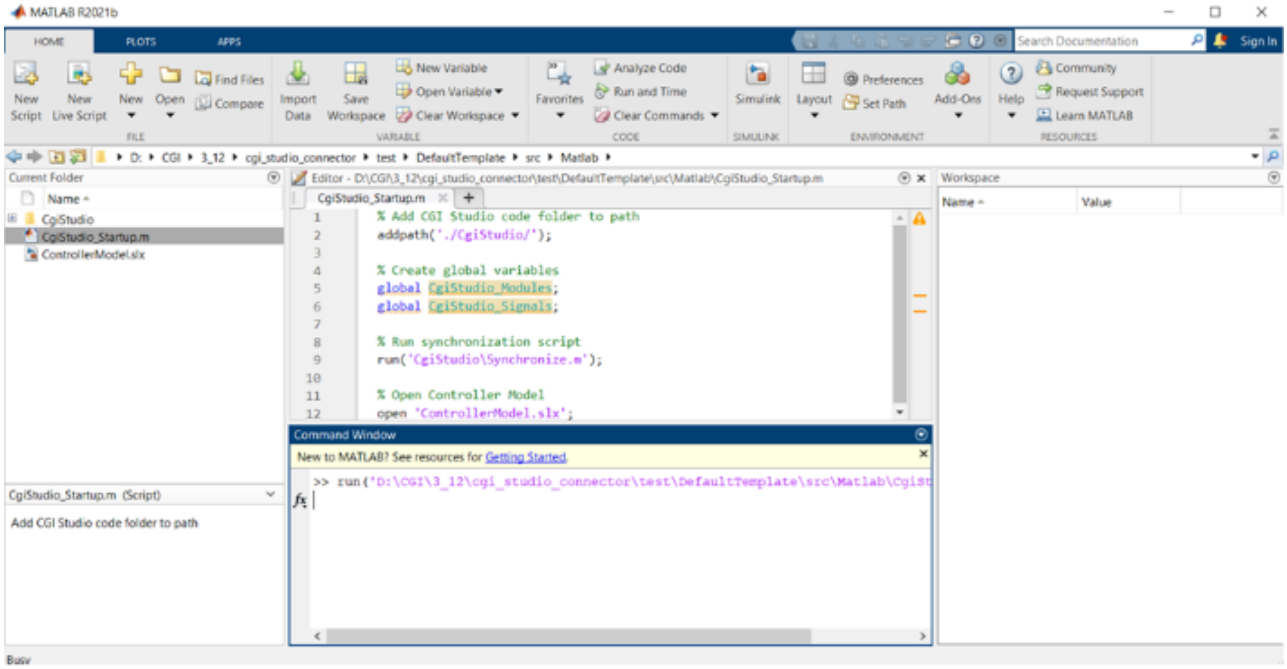
   If the MATLAB application is already opened and the Model has been started before, it is required to unload the S-Function from MATLAB before building a new version. This can be done by running 'clear mex' command in MATLAB.



3. **Startup MATLAB model**
   After the build is finished, the MATLAB application can be opened. The default template includes all related MATLAB component in the ‘src\Matlab’ folder.

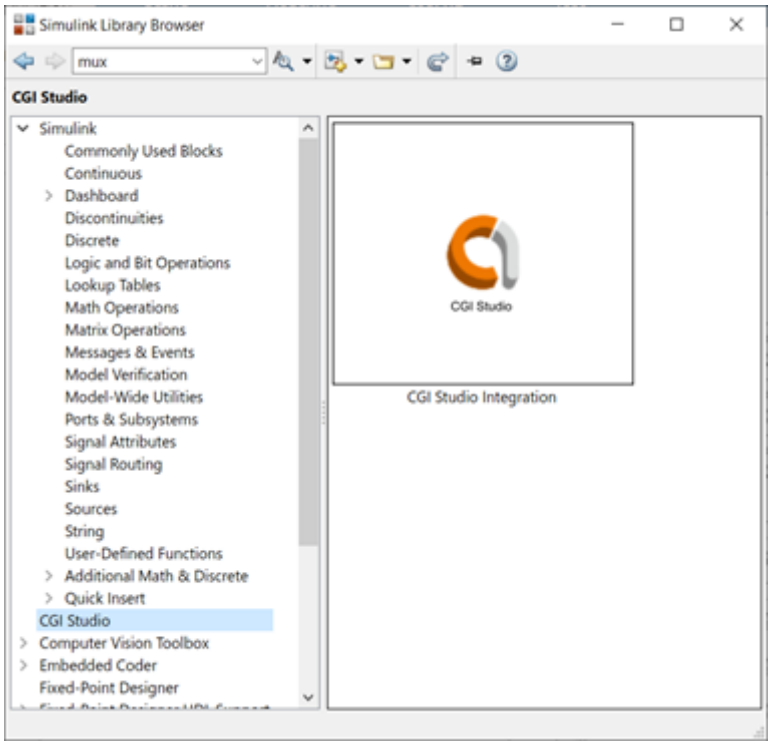


   The ‘CgiStudio_Startup.m’ script needs to be run in MATLAB to include the ‘CgiStudio’ code folder and files.



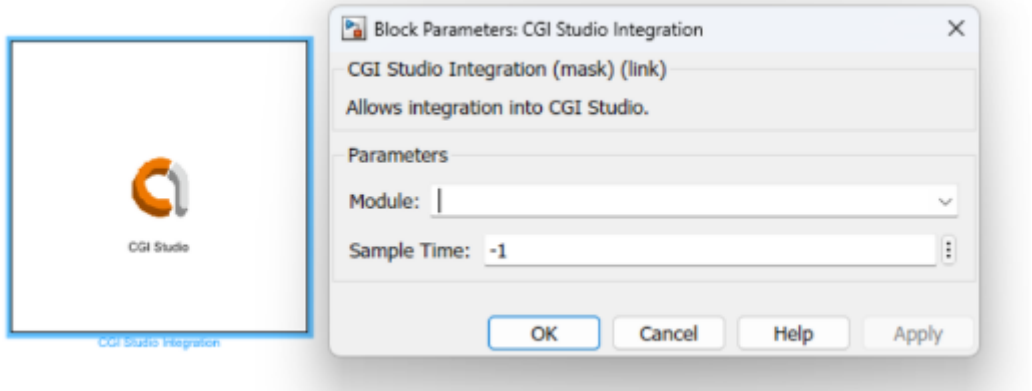
   If already run before, it is sufficient to only run the ‘CgiStudio/Synchronize.m’ script.

4. **Add CGI Studio block**
   The integration provides a ‘CGI Studio’ library which contains the ‘CGI Studio Integration’ block.



This block can be added to the Model to represent a Module defined with the CGI Connector. The provided ‘ControllerModule’ model already contains one such block per default.

5. **Configure Module**
   After a ‘CGI Studio Integration’ block has been added, it can be configured by double clicking it. This will open a dialog which allows the user to select the module this block should represent. Additionally, the Sample Time for the block can be configured.



After the module has been selected, the block will show all configured input- and output interfaces as ports.



> It might be required to update the model (CTRL + D) to show the ports.

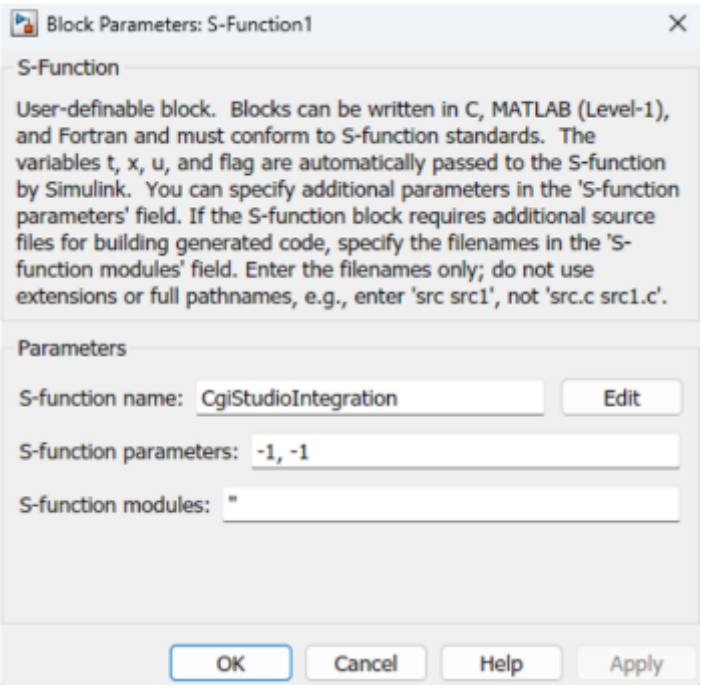
6. **Running the Simulation**
   When running the simulation, the CGI Application must be started beforehand. Similar after stopping the simulation the CGI Application should be closed afterwards.

Additional Information

Actions/Events: Actions are triggered by any flank change. If an action has additional data an additional port with the suffix '_Data' is generated. The same applies also to events.

Module Changes: If a module can't be found in the generated code, the block will indicate this by adding '[Not found]' to the Module name. If the containing MATLAB model is opened newly no ports will be generated and connected signals will be disconnected. If the model has already been opened before, the latest generated ports will be kept (until it is reopened). Therefore, if the name of a module is changed, it is recommended to open the MATLAB model before generating the new integration code. Doing so will keep all signals connected.

S-Function: It is possible to directly use the generated S-Function without making use of the CGI Studio block, by using a standard S-Function block.



The name of the S-Function is “CgiStudioIntegration”, and the first parameter indicates the MATLAB-Id of the module the block should represent. This Id can be found in the Interface-Definition XML file or the generated MATLAB code (e.g. in ‘[PROJECT]srcMatlab\CgiStudio\Modules.xml’ called MID). The second parameter refers to the sample time of the S-Function.

In this case no named ports will be generated for the module.
